Flint Documentation

Release 3.1.0-dev

The Flint development team

Jan 26, 2024
## Generic rings

3.1 **gr.h** – generic structures and their elements

3.1.1 Introduction

3.1.2 Context operations

3.1.3 Element operations

3.2 **gr.h (continued)** – implementing rings

3.2.1 Example

3.2.2 Method table

3.2.3 Placeholder and trivial methods

3.2.4 Required methods

3.2.5 Testing rings

3.3 **gr.h (continued)** – builtin domains and types

3.3.1 Coercions

3.3.2 Domain properties

3.3.3 Groups

3.3.4 Base rings and fields

3.3.5 Extended number sets

3.3.6 Floating-point arithmetic

3.3.7 Vectors

3.3.8 Matrices

3.3.9 Polynomial rings

3.3.10 Fraction fields

3.3.11 Symbolic expressions

3.4 **gr_generic.h** – basic algorithms and fallback implementations for generic elements

3.4.1 Generic string parsing

3.4.2 Generic arithmetic

3.4.3 Generic special functions

3.4.4 Generic vector methods
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td><strong>gr_special.h</strong> – special arithmetic and transcendental functions</td>
<td>61</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Mathematical constants</td>
<td>61</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Elementary functions</td>
<td>61</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Factorials and gamma functions</td>
<td>62</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Combinatorial numbers</td>
<td>63</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Error function and exponential integrals</td>
<td>64</td>
</tr>
<tr>
<td>3.5.6</td>
<td>Orthogonal polynomials</td>
<td>64</td>
</tr>
<tr>
<td>3.5.7</td>
<td>Bessel, Airy and Coulomb functions</td>
<td>65</td>
</tr>
<tr>
<td>3.5.8</td>
<td>Hypergeometric functions</td>
<td>65</td>
</tr>
<tr>
<td>3.5.9</td>
<td>Riemann zeta, polylogarithms and Dirichlet L-functions</td>
<td>66</td>
</tr>
<tr>
<td>3.5.10</td>
<td>Elliptic integrals</td>
<td>66</td>
</tr>
<tr>
<td>3.5.11</td>
<td>Elliptic, modular and theta functions</td>
<td>67</td>
</tr>
<tr>
<td>3.6</td>
<td><strong>gr_vec.h</strong> – vectors over generic rings</td>
<td>68</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Types and basic operations</td>
<td>68</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Arithmetic</td>
<td>69</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Sums and products</td>
<td>71</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Dot products</td>
<td>71</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Other functions</td>
<td>72</td>
</tr>
<tr>
<td>3.7</td>
<td><strong>gr_mat.h</strong> – dense matrices over generic rings</td>
<td>73</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Type compatibility</td>
<td>73</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Types, macros and constants</td>
<td>73</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Memory management</td>
<td>74</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Window matrices</td>
<td>74</td>
</tr>
<tr>
<td>3.7.5</td>
<td>Input and output</td>
<td>74</td>
</tr>
<tr>
<td>3.7.6</td>
<td>Comparisons</td>
<td>74</td>
</tr>
<tr>
<td>3.7.7</td>
<td>Assignment and special values</td>
<td>74</td>
</tr>
<tr>
<td>3.7.8</td>
<td>Basic row, column and entry operations</td>
<td>75</td>
</tr>
<tr>
<td>3.7.9</td>
<td>Arithmetic</td>
<td>76</td>
</tr>
<tr>
<td>3.7.10</td>
<td>Diagonal and triangular matrices</td>
<td>76</td>
</tr>
<tr>
<td>3.7.11</td>
<td>Gaussian elimination</td>
<td>77</td>
</tr>
<tr>
<td>3.7.12</td>
<td>Solving</td>
<td>77</td>
</tr>
<tr>
<td>3.7.13</td>
<td>Determinant and trace</td>
<td>78</td>
</tr>
<tr>
<td>3.7.14</td>
<td>Rank</td>
<td>79</td>
</tr>
<tr>
<td>3.7.15</td>
<td>Row echelon form</td>
<td>79</td>
</tr>
<tr>
<td>3.7.16</td>
<td>Nullspace</td>
<td>79</td>
</tr>
<tr>
<td>3.7.17</td>
<td>Inverse and adjugate</td>
<td>80</td>
</tr>
<tr>
<td>3.7.18</td>
<td>Characteristic polynomial</td>
<td>80</td>
</tr>
<tr>
<td>3.7.19</td>
<td>Minimal polynomial</td>
<td>81</td>
</tr>
<tr>
<td>3.7.20</td>
<td>Similarity transformations</td>
<td>81</td>
</tr>
<tr>
<td>3.7.21</td>
<td>Eigenvalues</td>
<td>81</td>
</tr>
<tr>
<td>3.7.22</td>
<td>Jordan decomposition</td>
<td>82</td>
</tr>
<tr>
<td>3.7.23</td>
<td>Matrix functions</td>
<td>82</td>
</tr>
<tr>
<td>3.7.24</td>
<td>Hessenberg form</td>
<td>82</td>
</tr>
<tr>
<td>3.7.25</td>
<td>Random matrices</td>
<td>83</td>
</tr>
<tr>
<td>3.7.26</td>
<td>Special matrices</td>
<td>83</td>
</tr>
<tr>
<td>3.7.27</td>
<td>Helper functions for reduction</td>
<td>84</td>
</tr>
<tr>
<td>3.8</td>
<td><strong>gr_poly.h</strong> – dense univariate polynomials over generic rings</td>
<td>85</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Type compatibility</td>
<td>85</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Weak normalization</td>
<td>85</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Types, macros and constants</td>
<td>85</td>
</tr>
<tr>
<td>3.8.4</td>
<td>Memory management</td>
<td>85</td>
</tr>
<tr>
<td>3.8.5</td>
<td>Basic manipulation</td>
<td>86</td>
</tr>
<tr>
<td>3.8.6</td>
<td>Arithmetic</td>
<td>87</td>
</tr>
<tr>
<td>3.8.7</td>
<td>Powering</td>
<td>87</td>
</tr>
<tr>
<td>3.8.8</td>
<td>Shifting</td>
<td>88</td>
</tr>
<tr>
<td>3.8.9</td>
<td>Scalar division</td>
<td>88</td>
</tr>
<tr>
<td>3.8.10</td>
<td>Division with remainder</td>
<td>88</td>
</tr>
<tr>
<td>3.8.11</td>
<td>Power series division</td>
<td>89</td>
</tr>
</tbody>
</table>
4 Integers .......................................................... 103
  4.1 ulong_extras.h – arithmetic and number-theoretic functions for single-word integers .... 103
    4.1.1 Simple example ........................................ 103
    4.1.2 Random functions ...................................... 104
    4.1.3 Basic arithmetic ....................................... 105
    4.1.4 Miscellaneous ......................................... 105
    4.1.5 Basic arithmetic with precomputed inverses ....... 105
    4.1.6 Greatest common divisor ............................. 108
    4.1.7 Jacobi and Kronecker symbols ...................... 108
    4.1.8 Modular Arithmetic .................................... 109
    4.1.9 Divisibility testing .................................. 110
    4.1.10 Prime number generation and counting ............ 110
    4.1.11 Primality testing .................................... 112
    4.1.12 Chinese remaindering ............................... 114
    4.1.13 Square root and perfect power testing ........... 114
    4.1.14 Factorisation ........................................ 116
    4.1.15 Arithmetic functions ................................ 119
    4.1.16 Factorials .......................................... 119
    4.1.17 Primitive Roots and Discrete Logarithms ......... 120
    4.1.18 Elliptic curve method for factorization of mp_limb_t .......................... 120
  4.2 fmpz.h – integers ........................................ 121
    4.2.1 Simple example ........................................ 122
    4.2.2 Types, macros and constants ....................... 122
    4.2.3 Memory management ................................... 123
    4.2.4 Random generation ................................... 124
    4.2.5 Conversion ........................................... 124
    4.2.6 Input and output ..................................... 127
    4.2.7 Basic properties and manipulation ................. 128
    4.2.8 Comparison .......................................... 129
    4.2.9 Basic arithmetic ..................................... 129
    4.2.10 Greatest common divisor ......................... 133
    4.2.11 Modular arithmetic ................................. 135
    4.2.12 Bit packing and unpacking ......................... 135
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.13</td>
<td>Logic Operations</td>
<td>136</td>
</tr>
<tr>
<td>4.2.14</td>
<td>Chinese remaindering</td>
<td>136</td>
</tr>
<tr>
<td>4.2.15</td>
<td>Primality testing</td>
<td>138</td>
</tr>
<tr>
<td>4.2.16</td>
<td>Special functions</td>
<td>141</td>
</tr>
<tr>
<td>4.3</td>
<td>fmpz_vec.h – vectors of integers</td>
<td>141</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Memory management</td>
<td>141</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Randomisation</td>
<td>141</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Bit sizes and norms</td>
<td>142</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Input and output</td>
<td>142</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Conversions</td>
<td>143</td>
</tr>
<tr>
<td>4.3.6</td>
<td>Assignment and basic manipulation</td>
<td>143</td>
</tr>
<tr>
<td>4.3.7</td>
<td>Comparison</td>
<td>143</td>
</tr>
<tr>
<td>4.3.8</td>
<td>Sorting</td>
<td>144</td>
</tr>
<tr>
<td>4.3.9</td>
<td>Addition and subtraction</td>
<td>144</td>
</tr>
<tr>
<td>4.3.10</td>
<td>Scalar multiplication and division</td>
<td>144</td>
</tr>
<tr>
<td>4.3.11</td>
<td>Sums and products</td>
<td>145</td>
</tr>
<tr>
<td>4.3.12</td>
<td>Reduction mod ( p )</td>
<td>145</td>
</tr>
<tr>
<td>4.3.13</td>
<td>Gaussian content</td>
<td>146</td>
</tr>
<tr>
<td>4.3.14</td>
<td>Dot product</td>
<td>146</td>
</tr>
<tr>
<td>4.4</td>
<td>fmpz_factor.h – integer factorisation</td>
<td>146</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Types, macros and constants</td>
<td>146</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Factoring integers</td>
<td>146</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Input and output</td>
<td>148</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Elliptic curve (ECM) method</td>
<td>149</td>
</tr>
<tr>
<td>4.5</td>
<td>fmpz_mat.h – matrices over the integers</td>
<td>150</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Simple example</td>
<td>151</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Types, macros and constants</td>
<td>151</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Memory management</td>
<td>151</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Basic assignment and manipulation</td>
<td>152</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Window</td>
<td>152</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Random matrix generation</td>
<td>153</td>
</tr>
<tr>
<td>4.5.7</td>
<td>Input and output</td>
<td>154</td>
</tr>
<tr>
<td>4.5.8</td>
<td>Comparison</td>
<td>154</td>
</tr>
<tr>
<td>4.5.9</td>
<td>Transpose</td>
<td>155</td>
</tr>
<tr>
<td>4.5.10</td>
<td>Concatenate</td>
<td>155</td>
</tr>
<tr>
<td>4.5.11</td>
<td>Modular reduction and reconstruction</td>
<td>155</td>
</tr>
<tr>
<td>4.5.12</td>
<td>Addition and subtraction</td>
<td>156</td>
</tr>
<tr>
<td>4.5.13</td>
<td>Matrix-scalar arithmetic</td>
<td>156</td>
</tr>
<tr>
<td>4.5.14</td>
<td>Matrix multiplication</td>
<td>157</td>
</tr>
<tr>
<td>4.5.15</td>
<td>Inverse</td>
<td>159</td>
</tr>
<tr>
<td>4.5.16</td>
<td>Kronecker product</td>
<td>159</td>
</tr>
<tr>
<td>4.5.17</td>
<td>Content</td>
<td>159</td>
</tr>
<tr>
<td>4.5.18</td>
<td>Trace</td>
<td>159</td>
</tr>
<tr>
<td>4.5.19</td>
<td>Determinant</td>
<td>159</td>
</tr>
<tr>
<td>4.5.20</td>
<td>Transforms</td>
<td>160</td>
</tr>
<tr>
<td>4.5.21</td>
<td>Characteristic polynomial</td>
<td>160</td>
</tr>
<tr>
<td>4.5.22</td>
<td>Minimal polynomial</td>
<td>161</td>
</tr>
<tr>
<td>4.5.23</td>
<td>Rank</td>
<td>161</td>
</tr>
<tr>
<td>4.5.24</td>
<td>Column partitioning</td>
<td>161</td>
</tr>
<tr>
<td>4.5.25</td>
<td>Nonsingular solving</td>
<td>161</td>
</tr>
<tr>
<td>4.5.26</td>
<td>Row reduction</td>
<td>163</td>
</tr>
<tr>
<td>4.5.27</td>
<td>Modular gaussian elimination</td>
<td>164</td>
</tr>
<tr>
<td>4.5.28</td>
<td>Strong echelon form and Howell form</td>
<td>164</td>
</tr>
<tr>
<td>4.5.29</td>
<td>Nullspace</td>
<td>165</td>
</tr>
<tr>
<td>4.5.30</td>
<td>Echelon form</td>
<td>165</td>
</tr>
<tr>
<td>4.5.31</td>
<td>Hermite normal form</td>
<td>165</td>
</tr>
<tr>
<td>4.5.32</td>
<td>Smith normal form</td>
<td>166</td>
</tr>
<tr>
<td>4.5.33</td>
<td>Special matrices</td>
<td>167</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>4.5.34</td>
<td>Conversions</td>
<td>167</td>
</tr>
<tr>
<td>4.5.35</td>
<td>Cholesky Decomposition</td>
<td>167</td>
</tr>
<tr>
<td>4.5.36</td>
<td>LLL</td>
<td>167</td>
</tr>
<tr>
<td>4.5.37</td>
<td>Classical LLL</td>
<td>168</td>
</tr>
<tr>
<td>4.5.38</td>
<td>Modified LLL</td>
<td>168</td>
</tr>
<tr>
<td>4.6</td>
<td>fmpz_illus.h – LLL reduction</td>
<td>168</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Parameter manipulation</td>
<td>168</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Random parameter generation</td>
<td>169</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Heuristic dot product</td>
<td>169</td>
</tr>
<tr>
<td>4.6.4</td>
<td>The various Babai's</td>
<td>169</td>
</tr>
<tr>
<td>4.6.5</td>
<td>Shift</td>
<td>170</td>
</tr>
<tr>
<td>4.6.6</td>
<td>Varieties of LLL</td>
<td>170</td>
</tr>
<tr>
<td>4.6.7</td>
<td>ULLL</td>
<td>172</td>
</tr>
<tr>
<td>4.6.8</td>
<td>LLL-reducedness</td>
<td>172</td>
</tr>
<tr>
<td>4.6.9</td>
<td>Modified ULLL</td>
<td>172</td>
</tr>
<tr>
<td>4.6.10</td>
<td>Main LLL functions</td>
<td>173</td>
</tr>
<tr>
<td>4.7</td>
<td>fmpz_poly.h – univariate polynomials over the integers</td>
<td>173</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Introduction</td>
<td>173</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Simple example</td>
<td>173</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Definition of the fmpz_poly_t type</td>
<td>174</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Types, macros and constants</td>
<td>174</td>
</tr>
<tr>
<td>4.7.5</td>
<td>Memory management</td>
<td>174</td>
</tr>
<tr>
<td>4.7.6</td>
<td>Polynomial parameters</td>
<td>175</td>
</tr>
<tr>
<td>4.7.7</td>
<td>Assignment and basic manipulation</td>
<td>175</td>
</tr>
<tr>
<td>4.7.8</td>
<td>Randomisation</td>
<td>177</td>
</tr>
<tr>
<td>4.7.9</td>
<td>Getting and setting coefficients</td>
<td>177</td>
</tr>
<tr>
<td>4.7.10</td>
<td>Comparison</td>
<td>178</td>
</tr>
<tr>
<td>4.7.11</td>
<td>Addition and subtraction</td>
<td>178</td>
</tr>
<tr>
<td>4.7.12</td>
<td>Scalar absolute value, multiplication and division</td>
<td>179</td>
</tr>
<tr>
<td>4.7.13</td>
<td>Bit packing</td>
<td>180</td>
</tr>
<tr>
<td>4.7.14</td>
<td>Multiplication</td>
<td>181</td>
</tr>
<tr>
<td>4.7.15</td>
<td>FFT precached multiplication</td>
<td>183</td>
</tr>
<tr>
<td>4.7.16</td>
<td>Squaring</td>
<td>184</td>
</tr>
<tr>
<td>4.7.17</td>
<td>Powering</td>
<td>184</td>
</tr>
<tr>
<td>4.7.18</td>
<td>Shifting</td>
<td>185</td>
</tr>
<tr>
<td>4.7.19</td>
<td>Bit sizes and norms</td>
<td>187</td>
</tr>
<tr>
<td>4.7.20</td>
<td>Greatest common divisor</td>
<td>188</td>
</tr>
<tr>
<td>4.7.21</td>
<td>Discriminant</td>
<td>190</td>
</tr>
<tr>
<td>4.7.22</td>
<td>Gaussian content</td>
<td>191</td>
</tr>
<tr>
<td>4.7.23</td>
<td>Square-free</td>
<td>191</td>
</tr>
<tr>
<td>4.7.24</td>
<td>Euclidean division</td>
<td>191</td>
</tr>
<tr>
<td>4.7.25</td>
<td>Division with precomputed inverse</td>
<td>195</td>
</tr>
<tr>
<td>4.7.26</td>
<td>Divisibility testing</td>
<td>196</td>
</tr>
<tr>
<td>4.7.27</td>
<td>Division mod p</td>
<td>197</td>
</tr>
<tr>
<td>4.7.28</td>
<td>Power series division</td>
<td>197</td>
</tr>
<tr>
<td>4.7.29</td>
<td>Pseudo division</td>
<td>198</td>
</tr>
<tr>
<td>4.7.30</td>
<td>Derivative</td>
<td>200</td>
</tr>
<tr>
<td>4.7.31</td>
<td>Evaluation</td>
<td>200</td>
</tr>
<tr>
<td>4.7.32</td>
<td>Newton basis</td>
<td>202</td>
</tr>
<tr>
<td>4.7.33</td>
<td>Interpolation</td>
<td>202</td>
</tr>
<tr>
<td>4.7.34</td>
<td>Composition</td>
<td>202</td>
</tr>
<tr>
<td>4.7.35</td>
<td>Inflation and deflation</td>
<td>203</td>
</tr>
<tr>
<td>4.7.36</td>
<td>Taylor shift</td>
<td>203</td>
</tr>
<tr>
<td>4.7.37</td>
<td>Power series composition</td>
<td>204</td>
</tr>
<tr>
<td>4.7.38</td>
<td>Power series reversion</td>
<td>205</td>
</tr>
<tr>
<td>4.7.39</td>
<td>Square root</td>
<td>205</td>
</tr>
<tr>
<td>4.7.40</td>
<td>Power sums</td>
<td>206</td>
</tr>
<tr>
<td>4.7.41</td>
<td>Signature</td>
<td>207</td>
</tr>
</tbody>
</table>
4.7.42 Hensel lifting .......................................................... 207
4.7.43 Input and output ...................................................... 209
4.7.44 Modular reduction and reconstruction .......................... 211
4.7.45 Products ............................................................... 211
4.7.46 Roots ................................................................. 212
4.7.47 Minimal polynomials ................................................. 212
4.7.48 Orthogonal polynomials ............................................ 213
4.7.49 Fibonacci polynomials .............................................. 214
4.7.50 Eulerian numbers and polynomials ............................. 214
4.7.51 Modular forms and q-series ....................................... 215
4.7.52 CLD bounds .......................................................... 215
4.8 fmpz_poly_mat.h – matrices of polynomials over the integers 215
  4.8.1 Simple example ...................................................... 215
  4.8.2 Types, macros and constants ..................................... 216
  4.8.3 Memory management .............................................. 216
  4.8.4 Basic properties .................................................. 216
  4.8.5 Basic assignment and manipulation ............................ 217
  4.8.6 Input and output .................................................. 217
  4.8.7 Random matrix generation ...................................... 217
  4.8.8 Special matrices .................................................. 217
  4.8.9 Basic comparison and properties .............................. 218
  4.8.10 Norms ............................................................. 218
  4.8.11 Transpose ......................................................... 218
  4.8.12 Evaluation ........................................................ 218
  4.8.13 Arithmetic ........................................................ 218
  4.8.14 Row reduction .................................................... 220
  4.8.15 Trace ............................................................. 220
  4.8.16 Determinant and rank ......................................... 221
  4.8.17 Inverse ........................................................... 221
  4.8.18 Nullspace ........................................................ 221
  4.8.19 Solving ........................................................... 221
4.9 fmpz_poly_factor.h – factorisation of polynomials over the integers 222
  4.9.1 Types, macros and constants ..................................... 222
  4.9.2 Memory management .............................................. 222
  4.9.3 Manipulating factors .............................................. 222
  4.9.4 Input and output .................................................. 223
  4.9.5 Factoring algorithms ............................................. 223
4.10 fmpz_multpoly.h – multivariate polynomials over the integers 224
  4.10.1 Types, macros and constants ..................................... 224
  4.10.2 Context object .................................................... 224
  4.10.3 Memory management ............................................. 224
  4.10.4 Input/Output ...................................................... 225
  4.10.5 Basic manipulation .............................................. 225
  4.10.6 Constants ........................................................ 226
  4.10.7 Degrees ........................................................... 226
  4.10.8 Coefficients ...................................................... 227
  4.10.9 Comparison ........................................................ 228
  4.10.10 Conversion ...................................................... 228
  4.10.11 Container operations .......................................... 228
  4.10.12 Random generation ............................................ 230
  4.10.13 Addition/Subtraction ......................................... 230
  4.10.14 Scalar operations .............................................. 231
  4.10.15 Differentiation/Integration ................................... 232
  4.10.16 Evaluation ....................................................... 232
  4.10.17 Multiplication ................................................... 233
  4.10.18 Powering ......................................................... 233
  4.10.19 Division .......................................................... 233
  4.10.20 Greatest Common Divisor .................................... 234
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.10.21</td>
<td>Square Root</td>
</tr>
<tr>
<td>4.10.22</td>
<td>Univariate Functions</td>
</tr>
<tr>
<td>4.10.23</td>
<td>Internal Functions</td>
</tr>
<tr>
<td>4.10.24</td>
<td>Vectors</td>
</tr>
<tr>
<td>4.10.25</td>
<td>Ideals and Gröbner bases</td>
</tr>
<tr>
<td>4.10.26</td>
<td>Special polynomials</td>
</tr>
<tr>
<td>4.11</td>
<td>fmpz_mpoly_factor.h – factorisation of multivariate polynomials over the integers</td>
</tr>
<tr>
<td>4.11.1</td>
<td>Types, macros and constants</td>
</tr>
<tr>
<td>4.11.2</td>
<td>Memory management</td>
</tr>
<tr>
<td>4.11.3</td>
<td>Basic manipulation</td>
</tr>
<tr>
<td>4.11.4</td>
<td>Factorisation</td>
</tr>
<tr>
<td>4.12</td>
<td>long_extras.h – support functions for signed word arithmetic</td>
</tr>
<tr>
<td>4.12.1</td>
<td>Properties</td>
</tr>
<tr>
<td>4.12.2</td>
<td>Checked Arithmetic</td>
</tr>
<tr>
<td>4.12.3</td>
<td>Random functions</td>
</tr>
<tr>
<td>4.12.4</td>
<td>Modular arithmetic</td>
</tr>
<tr>
<td>4.13</td>
<td>longlong.h – support functions for multi-word arithmetic</td>
</tr>
<tr>
<td>4.13.1</td>
<td>Auxiliary asm macros</td>
</tr>
<tr>
<td>4.14</td>
<td>mpn_extras.h – support functions for limb arrays</td>
</tr>
<tr>
<td>4.14.1</td>
<td>Macros</td>
</tr>
<tr>
<td>4.14.2</td>
<td>Utility functions</td>
</tr>
<tr>
<td>4.14.3</td>
<td>Multiplication</td>
</tr>
<tr>
<td>4.14.4</td>
<td>Divisibility</td>
</tr>
<tr>
<td>4.14.5</td>
<td>Division</td>
</tr>
<tr>
<td>4.14.6</td>
<td>GCD</td>
</tr>
<tr>
<td>4.14.7</td>
<td>Random Number Generation</td>
</tr>
<tr>
<td>4.15</td>
<td>aprcl.h – APRCL primality testing</td>
</tr>
<tr>
<td>4.15.1</td>
<td>Primality test functions</td>
</tr>
<tr>
<td>4.15.2</td>
<td>Configuration functions</td>
</tr>
<tr>
<td>4.15.3</td>
<td>Cyclotomic arithmetic</td>
</tr>
<tr>
<td>4.16</td>
<td>arith.h – arithmetic and special functions</td>
</tr>
<tr>
<td>4.16.1</td>
<td>Primorials</td>
</tr>
<tr>
<td>4.16.2</td>
<td>Harmonic numbers</td>
</tr>
<tr>
<td>4.16.3</td>
<td>Stirling numbers</td>
</tr>
<tr>
<td>4.16.4</td>
<td>Bell numbers</td>
</tr>
<tr>
<td>4.16.5</td>
<td>Bernoulli numbers and polynomials</td>
</tr>
<tr>
<td>4.16.6</td>
<td>Euler numbers and polynomials</td>
</tr>
<tr>
<td>4.16.7</td>
<td>Multiplicative functions</td>
</tr>
<tr>
<td>4.16.8</td>
<td>Landau’s function</td>
</tr>
<tr>
<td>4.16.9</td>
<td>Dedekind sums</td>
</tr>
<tr>
<td>4.16.10</td>
<td>Number of partitions</td>
</tr>
<tr>
<td>4.16.11</td>
<td>Sums of squares</td>
</tr>
<tr>
<td>4.17</td>
<td>fft.h – Schoenhage-Strassen FFT</td>
</tr>
<tr>
<td>4.17.1</td>
<td>Split/combine FFT coefficients</td>
</tr>
<tr>
<td>4.17.2</td>
<td>Test helper functions</td>
</tr>
<tr>
<td>4.17.3</td>
<td>Arithmetic modulo a generalised Fermat number</td>
</tr>
<tr>
<td>4.17.4</td>
<td>Generic butterflies</td>
</tr>
<tr>
<td>4.17.5</td>
<td>Radix 2 transforms</td>
</tr>
<tr>
<td>4.17.6</td>
<td>Matrix Fourier Transforms</td>
</tr>
<tr>
<td>4.17.7</td>
<td>Negacyclic multiplication</td>
</tr>
<tr>
<td>4.17.8</td>
<td>Integer multiplication</td>
</tr>
<tr>
<td>4.17.9</td>
<td>Convolution</td>
</tr>
<tr>
<td>4.17.10</td>
<td>FFT Precaching</td>
</tr>
<tr>
<td>4.18</td>
<td>fft_small.h – FFT modulo word-size primes</td>
</tr>
<tr>
<td>4.18.1</td>
<td>Integer multiplication</td>
</tr>
<tr>
<td>4.18.2</td>
<td>Polynomial arithmetic</td>
</tr>
<tr>
<td>4.18.3</td>
<td>Preconditioned polynomial arithmetic</td>
</tr>
<tr>
<td>4.19</td>
<td>qsieve.h – Quadratic sieve</td>
</tr>
</tbody>
</table>
## 5 Rational numbers

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 fmpq.h</td>
<td>rational numbers</td>
<td>271</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Types, macros and constants</td>
<td>271</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Memory management</td>
<td>272</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Canonicalisation</td>
<td>272</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Basic assignment</td>
<td>272</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Comparison</td>
<td>273</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Conversion</td>
<td>273</td>
</tr>
<tr>
<td>5.1.7</td>
<td>Input and output</td>
<td>275</td>
</tr>
<tr>
<td>5.1.8</td>
<td>Random number generation</td>
<td>275</td>
</tr>
<tr>
<td>5.1.9</td>
<td>Arithmetic</td>
<td>276</td>
</tr>
<tr>
<td>5.1.10</td>
<td>Modular reduction and rational reconstruction</td>
<td>278</td>
</tr>
<tr>
<td>5.1.11</td>
<td>Rational enumeration</td>
<td>278</td>
</tr>
<tr>
<td>5.1.12</td>
<td>Continued fractions</td>
<td>279</td>
</tr>
<tr>
<td>5.1.13</td>
<td>Special functions</td>
<td>280</td>
</tr>
<tr>
<td>5.1.14</td>
<td>Dedekind sums</td>
<td>280</td>
</tr>
<tr>
<td>5.2 fmpq_vec.h</td>
<td>vectors over rational numbers</td>
<td>281</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Memory management</td>
<td>281</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Randomisation</td>
<td>281</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Sorting</td>
<td>281</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Conversions</td>
<td>281</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Dot product</td>
<td>281</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Input and output</td>
<td>282</td>
</tr>
<tr>
<td>5.3 fmpq_mat.h</td>
<td>matrices over the rational numbers</td>
<td>282</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Types, macros and constants</td>
<td>282</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Memory management</td>
<td>282</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Entry access</td>
<td>283</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Basic assignment</td>
<td>283</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Addition, scalar multiplication</td>
<td>284</td>
</tr>
<tr>
<td>5.3.6</td>
<td>Input and output</td>
<td>284</td>
</tr>
<tr>
<td>5.3.7</td>
<td>Random matrix generation</td>
<td>284</td>
</tr>
<tr>
<td>5.3.8</td>
<td>Window</td>
<td>284</td>
</tr>
<tr>
<td>5.3.9</td>
<td>Concatenate</td>
<td>285</td>
</tr>
<tr>
<td>5.3.10</td>
<td>Special matrices</td>
<td>285</td>
</tr>
<tr>
<td>5.3.11</td>
<td>Basic comparison and properties</td>
<td>285</td>
</tr>
<tr>
<td>5.3.12</td>
<td>Integer matrix conversion</td>
<td>285</td>
</tr>
<tr>
<td>5.3.13</td>
<td>Modular reduction and rational reconstruction</td>
<td>286</td>
</tr>
<tr>
<td>5.3.14</td>
<td>Matrix multiplication</td>
<td>286</td>
</tr>
<tr>
<td>5.3.15</td>
<td>Kronecker product</td>
<td>287</td>
</tr>
<tr>
<td>5.3.16</td>
<td>Trace</td>
<td>287</td>
</tr>
<tr>
<td>5.3.17</td>
<td>Determinant</td>
<td>287</td>
</tr>
<tr>
<td>5.3.18</td>
<td>Nonsingular solving</td>
<td>288</td>
</tr>
<tr>
<td>5.3.19</td>
<td>Inverse</td>
<td>289</td>
</tr>
<tr>
<td>5.3.20</td>
<td>Echelon form</td>
<td>289</td>
</tr>
<tr>
<td>5.3.21</td>
<td>Gram-Schmidt Orthogonalisation</td>
<td>289</td>
</tr>
<tr>
<td>5.3.22</td>
<td>Transforms</td>
<td>289</td>
</tr>
<tr>
<td>5.3.23</td>
<td>Characteristic polynomial</td>
<td>289</td>
</tr>
<tr>
<td>5.3.24</td>
<td>Minimal polynomial</td>
<td>290</td>
</tr>
<tr>
<td>5.4 fmpq_poly.h</td>
<td>univariate polynomials over the rational numbers</td>
<td>290</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Types, macros and constants</td>
<td>290</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Memory management</td>
<td>291</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Polynomial parameters</td>
<td>292</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Accessing the numerator and denominator</td>
<td>292</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Random testing</td>
<td>292</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Assignment, swap, negation</td>
<td>293</td>
</tr>
<tr>
<td>5.4.7</td>
<td>Getting and setting coefficients</td>
<td>294</td>
</tr>
<tr>
<td>5.4.8</td>
<td>Comparison</td>
<td>295</td>
</tr>
<tr>
<td>5.4.9</td>
<td>Addition and subtraction</td>
<td>295</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>6.1 nmod.h</td>
<td>intemgers mod n (word-size n)</td>
<td></td>
</tr>
<tr>
<td>6.1.1 Modular reduction and arithmetic</td>
<td>333</td>
<td></td>
</tr>
<tr>
<td>6.1.2 Discrete Logarithms via Pohlig-Hellman</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>6.2 nmod_vec.h</td>
<td>vectors over integers mod n (word-size n)</td>
<td></td>
</tr>
<tr>
<td>6.2.1 Memory management</td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>6.2.2 Random functions</td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>6.2.3 Basic manipulation and comparison</td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>6.2.4 Printing</td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>6.2.5 Arithmetic operations</td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>6.2.6 Dot products</td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>6.3 nmod_mat.h</td>
<td>matrices over integers mod n (word-size n)</td>
<td></td>
</tr>
<tr>
<td>6.3.1 Types, macros and constants</td>
<td>337</td>
<td></td>
</tr>
<tr>
<td>6.3.2 Memory management</td>
<td>338</td>
<td></td>
</tr>
<tr>
<td>6.3.3 Basic properties and manipulation</td>
<td>338</td>
<td></td>
</tr>
<tr>
<td>6.3.4 Window</td>
<td>339</td>
<td></td>
</tr>
<tr>
<td>6.3.5 Concatenate</td>
<td>339</td>
<td></td>
</tr>
<tr>
<td>6.3.6 Printing</td>
<td>339</td>
<td></td>
</tr>
<tr>
<td>6.3.7 Random matrix generation</td>
<td>339</td>
<td></td>
</tr>
<tr>
<td>6.3.8 Comparison</td>
<td>340</td>
<td></td>
</tr>
<tr>
<td>6.3.9 Transposition and permutations</td>
<td>340</td>
<td></td>
</tr>
<tr>
<td>6.3.10 Addition and subtraction</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>6.3.11 Matrix-scalar arithmetic</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>6.3.12 Matrix multiplication</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>6.3.13 Matrix Exponentiation</td>
<td>342</td>
<td></td>
</tr>
<tr>
<td>6.3.14 Trace</td>
<td>342</td>
<td></td>
</tr>
<tr>
<td>6.3.15 Determinant and rank</td>
<td>343</td>
<td></td>
</tr>
<tr>
<td>6.3.16 Inverse</td>
<td>343</td>
<td></td>
</tr>
<tr>
<td>6.3.17 Triangular solving</td>
<td>343</td>
<td></td>
</tr>
<tr>
<td>6.3.18 Nonsingular square solving</td>
<td>344</td>
<td></td>
</tr>
<tr>
<td>6.3.19 LU decomposition</td>
<td>344</td>
<td></td>
</tr>
<tr>
<td>6.3.20 Reduced row echelon form</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>6.3.21 Nullspace</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>6.3.22 Transforms</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>6.3.23 Characteristic polynomial</td>
<td>346</td>
<td></td>
</tr>
<tr>
<td>6.3.24 Minimal polynomial</td>
<td>346</td>
<td></td>
</tr>
<tr>
<td>6.3.25 Strong echelon form and Howell form</td>
<td>346</td>
<td></td>
</tr>
<tr>
<td>6.4 nmod_poly.h</td>
<td>univariate polynomials over integers mod n (word-size n)</td>
<td></td>
</tr>
<tr>
<td>6.4.1 Simple example</td>
<td>346</td>
<td></td>
</tr>
</tbody>
</table>
6.4.2 Types, macros and constants ............................................. 347
6.4.3 Helper functions .......................................................... 347
6.4.4 Memory management ....................................................... 347
6.4.5 Polynomial properties ...................................................... 348
6.4.6 Assignment and basic manipulation ............................... 348
6.4.7 Randomization ............................................................... 349
6.4.8 Getting and setting coefficients ........................................ 350
6.4.9 Input and output ........................................................... 350
6.4.10 Comparison ................................................................. 351
6.4.11 Shifting ...................................................................... 351
6.4.12 Addition and subtraction ............................................... 352
6.4.13 Scalar multiplication and division ................................... 352
6.4.14 Bit packing and unpacking ............................................. 353
6.4.15 KS2/KS4 Reduction ....................................................... 353
6.4.16 Multiplication .............................................................. 354
6.4.17 Powering .................................................................... 356
6.4.18 Division ...................................................................... 358
6.4.19 Divisibility testing ....................................................... 361
6.4.20 Derivative and integral .................................................. 361
6.4.21 Evaluation ................................................................. 362
6.4.22 Multipoint evaluation .................................................... 362
6.4.23 Interpolation ............................................................... 363
6.4.24 Composition ............................................................... 364
6.4.25 Taylor shift ............................................................... 364
6.4.26 Modular composition ..................................................... 365
6.4.27 Greatest common divisor .............................................. 368
6.4.28 Discriminant .............................................................. 371
6.4.29 Power series composition ............................................. 372
6.4.30 Power series reversion .................................................. 372
6.4.31 Square roots ............................................................... 372
6.4.32 Power sums ............................................................... 373
6.4.33 Transcendental functions ............................................. 373
6.4.34 Special polynomials ..................................................... 375
6.4.35 Products ................................................................. 376
6.4.36 Subproduct trees ........................................................ 376
6.4.37 Inflation and deflation .................................................. 376
6.4.38 Chinese Remaindering .................................................. 377
6.4.39 Berlekamp-Massey Algorithm ....................................... 378

6.5 nmod_poly_mat.h – matrices of univariate polynomials over integers mod n (word-size n) ...................................................... 379
6.5.1 Types, macros and constants ............................................. 379
6.5.2 Memory management ....................................................... 379
6.5.3 Truncate, shift ............................................................... 379
6.5.4 Basic properties ........................................................... 380
6.5.5 Basic assignment and manipulation ................................ 380
6.5.6 Input and output ........................................................... 380
6.5.7 Random matrix generation ............................................. 380
6.5.8 Special matrices ........................................................... 381
6.5.9 Basic comparison and properties ..................................... 381
6.5.10 Norms ................................................................. 381
6.5.11 Evaluation ............................................................... 382
6.5.12 Arithmetic ............................................................... 382
6.5.13 Row reduction ............................................................ 383
6.5.14 Trace ...................................................................... 384
6.5.15 Determinant and rank .................................................. 384
6.5.16 Inverse ................................................................. 384
6.5.17 Nullspace ............................................................... 384
6.5.18 Solving ................................................................. 385
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.11.13</td>
<td>Trace</td>
</tr>
<tr>
<td>6.11.14</td>
<td>Gaussian elimination</td>
</tr>
<tr>
<td>6.11.15</td>
<td>Strong echelon form and Howell form</td>
</tr>
<tr>
<td>6.11.16</td>
<td>Inverse</td>
</tr>
<tr>
<td>6.11.17</td>
<td>LU decomposition</td>
</tr>
<tr>
<td>6.11.18</td>
<td>Triangular solving</td>
</tr>
<tr>
<td>6.11.19</td>
<td>Solving</td>
</tr>
<tr>
<td>6.11.20</td>
<td>Transforms</td>
</tr>
<tr>
<td>6.11.21</td>
<td>Characteristic polynomial</td>
</tr>
<tr>
<td>6.11.22</td>
<td>Minimal polynomial</td>
</tr>
<tr>
<td>6.12</td>
<td>\texttt{fmpz_mod_poly.h} – polynomials over integers mod n</td>
</tr>
<tr>
<td>6.12.1</td>
<td>Simple example</td>
</tr>
<tr>
<td>6.12.2</td>
<td>Types, macros and constants</td>
</tr>
<tr>
<td>6.12.3</td>
<td>Memory management</td>
</tr>
<tr>
<td>6.12.4</td>
<td>Randomisation</td>
</tr>
<tr>
<td>6.12.5</td>
<td>Attributes</td>
</tr>
<tr>
<td>6.12.6</td>
<td>Assignment and basic manipulation</td>
</tr>
<tr>
<td>6.12.7</td>
<td>Conversion</td>
</tr>
<tr>
<td>6.12.8</td>
<td>Comparison</td>
</tr>
<tr>
<td>6.12.9</td>
<td>Getting and setting coefficients</td>
</tr>
<tr>
<td>6.12.10</td>
<td>Shifting</td>
</tr>
<tr>
<td>6.12.11</td>
<td>Addition and subtraction</td>
</tr>
<tr>
<td>6.12.12</td>
<td>Scalar multiplication and division</td>
</tr>
<tr>
<td>6.12.13</td>
<td>Multiplication</td>
</tr>
<tr>
<td>6.12.14</td>
<td>Products</td>
</tr>
<tr>
<td>6.12.15</td>
<td>Division</td>
</tr>
<tr>
<td>6.12.16</td>
<td>Divisibility testing</td>
</tr>
<tr>
<td>6.12.17</td>
<td>Power series inversion</td>
</tr>
<tr>
<td>6.12.18</td>
<td>Power series division</td>
</tr>
<tr>
<td>6.12.19</td>
<td>Greatest common divisor</td>
</tr>
<tr>
<td>6.12.20</td>
<td>Mipoly</td>
</tr>
<tr>
<td>6.12.21</td>
<td>Resultant</td>
</tr>
<tr>
<td>6.12.22</td>
<td>Discriminant</td>
</tr>
<tr>
<td>6.12.23</td>
<td>Derivative</td>
</tr>
<tr>
<td>6.12.24</td>
<td>Evaluation</td>
</tr>
<tr>
<td>6.12.25</td>
<td>Multipoint evaluation</td>
</tr>
<tr>
<td>6.12.26</td>
<td>Composition</td>
</tr>
<tr>
<td>6.12.27</td>
<td>Square roots</td>
</tr>
<tr>
<td>6.12.28</td>
<td>Modular composition</td>
</tr>
<tr>
<td>6.12.29</td>
<td>Subproduct trees</td>
</tr>
<tr>
<td>6.12.30</td>
<td>Radix conversion</td>
</tr>
<tr>
<td>6.12.31</td>
<td>Input and output</td>
</tr>
<tr>
<td>6.12.32</td>
<td>Inflation and deflation</td>
</tr>
<tr>
<td>6.12.33</td>
<td>Berlekamp-Massey Algorithm</td>
</tr>
<tr>
<td>6.13</td>
<td>\texttt{fmpz_mod_poly_factor.h} – factorisation of polynomials over integers mod n</td>
</tr>
<tr>
<td>6.13.1</td>
<td>Types, macros and constants</td>
</tr>
<tr>
<td>6.13.2</td>
<td>Factorisation</td>
</tr>
<tr>
<td>6.13.3</td>
<td>Root Finding</td>
</tr>
<tr>
<td>6.14</td>
<td>\texttt{fmpz_mpoly.h} – polynomials over the integers mod n</td>
</tr>
<tr>
<td>6.14.1</td>
<td>Types, macros and constants</td>
</tr>
<tr>
<td>6.14.2</td>
<td>Context object</td>
</tr>
<tr>
<td>6.14.3</td>
<td>Memory management</td>
</tr>
<tr>
<td>6.14.4</td>
<td>Input/Output</td>
</tr>
<tr>
<td>6.14.5</td>
<td>Basic manipulation</td>
</tr>
<tr>
<td>6.14.6</td>
<td>Constants</td>
</tr>
<tr>
<td>6.14.7</td>
<td>Degrees</td>
</tr>
<tr>
<td>6.14.8</td>
<td>Coefficients</td>
</tr>
<tr>
<td>6.14.9</td>
<td>Comparison</td>
</tr>
</tbody>
</table>
6.14.10 Container operations ........................................... 444
6.14.11 Random generation ........................................... 446
6.14.12 Addition/Subtraction ....................................... 447
6.14.13 Scalar operations ........................................... 447
6.14.14 Differentiation ............................................... 448
6.14.15 Evaluation .................................................. 448
6.14.16 Multiplication ............................................... 449
6.14.17 Powering .................................................... 449
6.14.18 Division ..................................................... 449
6.14.19 Greatest Common Divisor ................................. 450
6.14.20 Square Root ................................................ 450
6.14.21 Univariate Functions ........................................ 451
6.14.22 Internal Functions ........................................... 452
6.14.23 Univariate Functions ........................................ 451
6.14.24 Differentiation ............................................... 448
6.14.25 Scalar operations ........................................... 447
6.14.26 Multiplication ............................................... 449
6.14.27 Powering .................................................... 449
6.14.28 Division ..................................................... 449
6.15 **fmpz_mod_mpoly_factor.h** – factorisation of multivariate polynomials over the integers mod n ........................................... 452
6.15.1 Types, macros and constants .................................. 452
6.15.2 Memory management ......................................... 452
6.15.3 Basic manipulation ........................................... 453
6.15.4 Factorisation .................................................. 453
6.15.5 Value comparisons ........................................... 457
6.15.6 Random generation ......................................... 468
6.15.7 Special matrices ............................................. 468
6.15.8 Value comparisons ........................................... 468
6.15.9 Arithmetic .................................................... 469
7 Groups and other structures ........................................ 455
7.1 **perm.h** – permutations ......................................... 455
7.1.1 Memory management ......................................... 455
7.1.2 Assignment ................................................... 455
7.1.3 Composition .................................................. 455
7.1.4 Parity .......................................................... 455
7.1.5 Randomisation ............................................... 456
7.2 **qfb.h** – binary quadratic forms ................................ 456
7.2.1 Introduction ................................................. 456
7.2.2 Memory management ......................................... 456
7.2.3 Hash table ..................................................... 456
7.2.4 Basic manipulation ......................................... 457
7.2.5 Comparison ................................................... 457
7.2.6 Input/output .................................................. 457
7.2.7 Computing with forms ...................................... 457
7.3 **dirichlet.h** – Dirichlet characters ................................. 459
7.3.1 Dirichlet characters .......................................... 459
7.3.2 Multiplicative group modulo \( q \) ...................... 459
7.3.3 Character type ............................................... 460
7.3.4 Character properties ........................................ 461
7.3.5 Character evaluation ....................................... 462
7.3.6 Character operations ....................................... 462
7.4 **dlog.h** – discrete logarithms mod ulong primes ................ 463
7.4.1 Types, macros and constants .................................. 463
7.4.2 Single evaluation ............................................ 463
7.4.3 Precomputations ............................................. 463
7.4.4 Vector evaluations .......................................... 464
7.4.5 Internal discrete logarithm strategies ..................... 464
7.5 **bool_mat.h** – matrices over booleans ......................... 467
7.5.1 Types, macros and constants .................................. 467
7.5.2 Memory management ......................................... 467
7.5.3 Conversions .................................................. 467
7.5.4 Input and output ............................................ 468
7.5.5 Value comparisons ........................................... 468
7.5.6 Random generation ......................................... 468
7.5.7 Special matrices ............................................. 468
7.5.8 Transpose .................................................... 469
7.5.9 Arithmetic .................................................... 469
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Number fields and algebraic numbers</td>
<td></td>
</tr>
<tr>
<td>8.1 nf.h – number fields</td>
<td>471</td>
</tr>
<tr>
<td>8.2 nf_elem.h – number field elements</td>
<td>471</td>
</tr>
<tr>
<td>8.3 fmpz.h – Gaussian integers</td>
<td>476</td>
</tr>
<tr>
<td>8.4 qqbar.h – algebraic numbers represented by minimal polynomials</td>
<td>478</td>
</tr>
<tr>
<td>9 Real and complex numbers</td>
<td></td>
</tr>
<tr>
<td>9.1 Feature overview</td>
<td>493</td>
</tr>
<tr>
<td>9.2 Using ball arithmetic</td>
<td></td>
</tr>
<tr>
<td>9.2.1 Ball semantics</td>
<td>494</td>
</tr>
<tr>
<td>9.2.2 Binary and decimal</td>
<td>494</td>
</tr>
<tr>
<td>9.2.3 Quality of enclosures</td>
<td>495</td>
</tr>
<tr>
<td>9.2.4 Predicates</td>
<td>496</td>
</tr>
<tr>
<td>9.2.5 A worked example: the sine function</td>
<td>497</td>
</tr>
<tr>
<td>9.2.6 More on precision and accuracy</td>
<td>499</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>9.2.7</td>
<td>Polynomial time guarantee</td>
</tr>
<tr>
<td>9.3</td>
<td>Technical conventions and potential issues</td>
</tr>
<tr>
<td>9.6.1</td>
<td>Types, macros and constants</td>
</tr>
<tr>
<td>9.6.2</td>
<td>Memory management</td>
</tr>
<tr>
<td>9.6.3</td>
<td>Special values</td>
</tr>
<tr>
<td>9.6.4</td>
<td>Assignment, rounding and conversions</td>
</tr>
<tr>
<td>9.6.5</td>
<td>Comparisons and bounds</td>
</tr>
<tr>
<td>9.6.6</td>
<td>Magnitude functions</td>
</tr>
<tr>
<td>9.6.7</td>
<td>Shallow assignment</td>
</tr>
<tr>
<td>9.6.8</td>
<td>Random number generation</td>
</tr>
<tr>
<td>9.6.9</td>
<td>Input and output</td>
</tr>
<tr>
<td>9.6.10</td>
<td>Addition and multiplication</td>
</tr>
<tr>
<td>9.6.11</td>
<td>Summation</td>
</tr>
<tr>
<td>9.6.12</td>
<td>Dot products</td>
</tr>
<tr>
<td>9.6.13</td>
<td>Division</td>
</tr>
<tr>
<td>9.6.14</td>
<td>Square roots</td>
</tr>
<tr>
<td>9.6.15</td>
<td>Complex arithmetic</td>
</tr>
<tr>
<td>9.6.16</td>
<td>Low-level methods</td>
</tr>
<tr>
<td>9.7</td>
<td>Types, macros and constants</td>
</tr>
<tr>
<td>9.7.2</td>
<td>Memory management</td>
</tr>
<tr>
<td>9.7.3</td>
<td>Basic manipulation</td>
</tr>
</tbody>
</table>

---

**Note:** The page numbers indicate the page where a particular section or function starts in the document.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9.7.4 Arithmetic</td>
<td>536</td>
</tr>
<tr>
<td></td>
<td>9.7.5 Approximate arithmetic</td>
<td>536</td>
</tr>
<tr>
<td></td>
<td>9.8 arb.h – real numbers</td>
<td>536</td>
</tr>
<tr>
<td></td>
<td>9.8.1 Types, macros and constants</td>
<td>537</td>
</tr>
<tr>
<td></td>
<td>9.8.2 Memory management</td>
<td>537</td>
</tr>
<tr>
<td></td>
<td>9.8.3 Assignment and rounding</td>
<td>538</td>
</tr>
<tr>
<td></td>
<td>9.8.4 Assignment of special values</td>
<td>539</td>
</tr>
<tr>
<td></td>
<td>9.8.5 Input and output</td>
<td>539</td>
</tr>
<tr>
<td></td>
<td>9.8.6 Random number generation</td>
<td>540</td>
</tr>
<tr>
<td></td>
<td>9.8.7 Radius and interval operations</td>
<td>541</td>
</tr>
<tr>
<td></td>
<td>9.8.8 Comparisons</td>
<td>544</td>
</tr>
<tr>
<td></td>
<td>9.8.9 Dot product</td>
<td>548</td>
</tr>
<tr>
<td></td>
<td>9.8.10 Powers and roots</td>
<td>548</td>
</tr>
<tr>
<td></td>
<td>9.8.11 Exponentials and logarithms</td>
<td>550</td>
</tr>
<tr>
<td></td>
<td>9.8.12 Trigonometric functions</td>
<td>550</td>
</tr>
<tr>
<td></td>
<td>9.8.13 Inverse trigonometric functions</td>
<td>550</td>
</tr>
<tr>
<td></td>
<td>9.8.14 Hyperbolic functions</td>
<td>550</td>
</tr>
<tr>
<td></td>
<td>9.8.15 Inverse hyperbolic functions</td>
<td>551</td>
</tr>
<tr>
<td></td>
<td>9.8.16 Lambert W function</td>
<td>552</td>
</tr>
<tr>
<td></td>
<td>9.8.17 Bernoulli numbers and polynomials</td>
<td>553</td>
</tr>
<tr>
<td></td>
<td>9.8.18 Gamma function and factorials</td>
<td>553</td>
</tr>
<tr>
<td></td>
<td>9.8.19 Zeta function</td>
<td>554</td>
</tr>
<tr>
<td></td>
<td>9.8.20 Polynomials of Bernoulli numbers</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>9.8.21 Beta function</td>
<td>556</td>
</tr>
<tr>
<td></td>
<td>9.8.22 Polylogarithms</td>
<td>556</td>
</tr>
<tr>
<td></td>
<td>9.8.23 Other special functions</td>
<td>556</td>
</tr>
<tr>
<td></td>
<td>9.8.24 Internals for computing elementary functions</td>
<td>557</td>
</tr>
<tr>
<td></td>
<td>9.8.25 Vector functions</td>
<td>560</td>
</tr>
<tr>
<td></td>
<td>9.9 acb.h – complex numbers</td>
<td>561</td>
</tr>
<tr>
<td></td>
<td>9.9.1 Types, macros and constants</td>
<td>562</td>
</tr>
<tr>
<td></td>
<td>9.9.2 Memory management</td>
<td>562</td>
</tr>
<tr>
<td></td>
<td>9.9.3 Basic manipulation</td>
<td>562</td>
</tr>
<tr>
<td></td>
<td>9.9.4 Input and output</td>
<td>563</td>
</tr>
<tr>
<td></td>
<td>9.9.5 Random number generation</td>
<td>564</td>
</tr>
<tr>
<td></td>
<td>9.9.6 Precision and comparisons</td>
<td>564</td>
</tr>
<tr>
<td></td>
<td>9.9.7 Complex parts</td>
<td>566</td>
</tr>
<tr>
<td></td>
<td>9.9.8 Arithmetic</td>
<td>566</td>
</tr>
<tr>
<td></td>
<td>9.9.9 Dot product</td>
<td>568</td>
</tr>
<tr>
<td></td>
<td>9.9.10 Mathematical constants</td>
<td>569</td>
</tr>
<tr>
<td></td>
<td>9.9.11 Powers and roots</td>
<td>569</td>
</tr>
<tr>
<td></td>
<td>9.9.12 Exponentials and logarithms</td>
<td>570</td>
</tr>
<tr>
<td></td>
<td>9.9.13 Trigonometric functions</td>
<td>570</td>
</tr>
<tr>
<td></td>
<td>9.9.14 Inverse trigonometric functions</td>
<td>570</td>
</tr>
<tr>
<td></td>
<td>9.9.15 Hyperbolic functions</td>
<td>571</td>
</tr>
<tr>
<td></td>
<td>9.9.16 Inverse hyperbolic functions</td>
<td>571</td>
</tr>
<tr>
<td></td>
<td>9.9.17 Lambert W function</td>
<td>572</td>
</tr>
<tr>
<td></td>
<td>9.9.18 Rising factorials</td>
<td>572</td>
</tr>
<tr>
<td></td>
<td>9.9.19 Gamma function</td>
<td>573</td>
</tr>
<tr>
<td></td>
<td>9.9.20 Zeta function</td>
<td>574</td>
</tr>
<tr>
<td></td>
<td>9.9.21 Polynomials of Bernoulli numbers</td>
<td>574</td>
</tr>
<tr>
<td></td>
<td>9.9.22 Arithmetic-geometric mean</td>
<td>575</td>
</tr>
<tr>
<td></td>
<td>9.9.23 Other special functions</td>
<td>575</td>
</tr>
<tr>
<td></td>
<td>9.9.24 Piecewise real functions</td>
<td>575</td>
</tr>
<tr>
<td></td>
<td>9.9.25 Vector functions</td>
<td>576</td>
</tr>
<tr>
<td></td>
<td>9.10 arb_poly.h – polynomials over the real numbers</td>
<td>578</td>
</tr>
<tr>
<td></td>
<td>9.10.1 Types, macros and constants</td>
<td>578</td>
</tr>
<tr>
<td></td>
<td>9.10.2 Memory management</td>
<td>578</td>
</tr>
<tr>
<td></td>
<td>9.10.3 Basic manipulation</td>
<td>579</td>
</tr>
<tr>
<td>Section</td>
<td>Topics</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>9.10.4 Conversions</td>
<td></td>
<td>580</td>
</tr>
<tr>
<td>9.10.5 Input and output</td>
<td></td>
<td>580</td>
</tr>
<tr>
<td>9.10.6 Random generation</td>
<td></td>
<td>580</td>
</tr>
<tr>
<td>9.10.7 Comparisons</td>
<td></td>
<td>580</td>
</tr>
<tr>
<td>9.10.8 Bounds</td>
<td></td>
<td>581</td>
</tr>
<tr>
<td>9.10.9 Arithmetic</td>
<td></td>
<td>581</td>
</tr>
<tr>
<td>9.10.10 Composition</td>
<td></td>
<td>583</td>
</tr>
<tr>
<td>9.10.11 Evaluation</td>
<td></td>
<td>584</td>
</tr>
<tr>
<td>9.10.12 Product trees</td>
<td></td>
<td>585</td>
</tr>
<tr>
<td>9.10.13 Multipoint evaluation</td>
<td></td>
<td>585</td>
</tr>
<tr>
<td>9.10.14 Interpolation</td>
<td></td>
<td>586</td>
</tr>
<tr>
<td>9.10.15 Differentiation</td>
<td></td>
<td>586</td>
</tr>
<tr>
<td>9.10.16 Transforms</td>
<td></td>
<td>587</td>
</tr>
<tr>
<td>9.10.17 Powers and elementary functions</td>
<td></td>
<td>588</td>
</tr>
<tr>
<td>9.10.18 Lambert W function</td>
<td></td>
<td>591</td>
</tr>
<tr>
<td>9.10.19 Gamma function and factorials</td>
<td></td>
<td>591</td>
</tr>
<tr>
<td>9.10.20 Zeta function</td>
<td></td>
<td>592</td>
</tr>
<tr>
<td>9.10.21 Root-finding</td>
<td></td>
<td>593</td>
</tr>
<tr>
<td>9.10.22 Other special polynomials</td>
<td></td>
<td>593</td>
</tr>
<tr>
<td>9.11 acb_poly.h – polynomials over the complex numbers</td>
<td></td>
<td>594</td>
</tr>
<tr>
<td>9.11.1 Types, macros and constants</td>
<td></td>
<td>594</td>
</tr>
<tr>
<td>9.11.2 Memory management</td>
<td></td>
<td>594</td>
</tr>
<tr>
<td>9.11.3 Basic properties and manipulation</td>
<td></td>
<td>594</td>
</tr>
<tr>
<td>9.11.4 Input and output</td>
<td></td>
<td>596</td>
</tr>
<tr>
<td>9.11.5 Random generation</td>
<td></td>
<td>596</td>
</tr>
<tr>
<td>9.11.6 Comparisons</td>
<td></td>
<td>596</td>
</tr>
<tr>
<td>9.11.7 Conversions</td>
<td></td>
<td>596</td>
</tr>
<tr>
<td>9.11.8 Bounds</td>
<td></td>
<td>597</td>
</tr>
<tr>
<td>9.11.9 Arithmetic</td>
<td></td>
<td>597</td>
</tr>
<tr>
<td>9.11.10 Composition</td>
<td></td>
<td>599</td>
</tr>
<tr>
<td>9.11.11 Evaluation</td>
<td></td>
<td>600</td>
</tr>
<tr>
<td>9.11.12 Product trees</td>
<td></td>
<td>600</td>
</tr>
<tr>
<td>9.11.13 Multipoint evaluation</td>
<td></td>
<td>601</td>
</tr>
<tr>
<td>9.11.14 Interpolation</td>
<td></td>
<td>601</td>
</tr>
<tr>
<td>9.11.15 Differentiation</td>
<td></td>
<td>602</td>
</tr>
<tr>
<td>9.11.16 Transforms</td>
<td></td>
<td>602</td>
</tr>
<tr>
<td>9.11.17 Elementary functions</td>
<td></td>
<td>603</td>
</tr>
<tr>
<td>9.11.18 Lambert W function</td>
<td></td>
<td>606</td>
</tr>
<tr>
<td>9.11.19 Gamma function</td>
<td></td>
<td>606</td>
</tr>
<tr>
<td>9.11.20 Power sums</td>
<td></td>
<td>607</td>
</tr>
<tr>
<td>9.11.21 Zeta function</td>
<td></td>
<td>607</td>
</tr>
<tr>
<td>9.11.22 Other special functions</td>
<td></td>
<td>608</td>
</tr>
<tr>
<td>9.11.23 Root-finding</td>
<td></td>
<td>609</td>
</tr>
<tr>
<td>9.12 arb_fmpz_poly.h – extra methods for integer polynomials</td>
<td></td>
<td>610</td>
</tr>
<tr>
<td>9.12.1 Evaluation</td>
<td></td>
<td>610</td>
</tr>
<tr>
<td>9.12.2 Utility methods</td>
<td></td>
<td>611</td>
</tr>
<tr>
<td>9.12.3 Polynomial roots</td>
<td></td>
<td>611</td>
</tr>
<tr>
<td>9.12.4 Special polynomials</td>
<td></td>
<td>612</td>
</tr>
<tr>
<td>9.13 acb_dft.h – Discrete Fourier transform</td>
<td></td>
<td>612</td>
</tr>
<tr>
<td>9.13.1 Main DFT functions</td>
<td></td>
<td>613</td>
</tr>
<tr>
<td>9.13.2 DFT on products</td>
<td></td>
<td>613</td>
</tr>
<tr>
<td>9.13.3 Convolution</td>
<td></td>
<td>614</td>
</tr>
<tr>
<td>9.13.4 FFT algorithms</td>
<td></td>
<td>614</td>
</tr>
<tr>
<td>9.14 arb_mat.h – matrices over the real numbers</td>
<td></td>
<td>616</td>
</tr>
<tr>
<td>9.14.1 Types, macros and constants</td>
<td></td>
<td>617</td>
</tr>
<tr>
<td>9.14.2 Memory management</td>
<td></td>
<td>617</td>
</tr>
<tr>
<td>9.14.3 Conversions</td>
<td></td>
<td>617</td>
</tr>
<tr>
<td>9.14.4 Random generation</td>
<td></td>
<td>618</td>
</tr>
</tbody>
</table>
9.14.5 Input and output .................................................. 618
9.14.6 Comparisons ....................................................... 618
9.14.7 Special matrices .................................................. 619
9.14.8 Transpose .......................................................... 620
9.14.9 Norms ............................................................... 620
9.14.10 Arithmetic .......................................................... 620
9.14.11 Scalar arithmetic .................................................. 621
9.14.12 Vector arithmetic ................................................. 621
9.14.13 Gaussian elimination and solving ............................... 622
9.14.14 Cholesky decomposition and solving ......................... 624
9.14.15 Characteristic polynomial and companion matrix ............. 625
9.14.16 Special functions ............................................... 626
9.14.17 Sparsity structure ............................................... 626
9.14.18 Component and error operations ............................... 627
9.14.19 Eigenvalues and eigenvectors .................................. 627
9.14.20 LLL reduction ...................................................... 627

9.15 **acb_mat.h** – matrices over the complex numbers ................. 627
9.15.1 Types, macros and constants ..................................... 628
9.15.2 Memory management .............................................. 628
9.15.3 Conversions ......................................................... 628
9.15.4 Random generation ............................................... 629
9.15.5 Input and output .................................................. 629
9.15.6 Comparisons ....................................................... 629
9.15.7 Special matrices .................................................. 630
9.15.8 Transpose ......................................................... 630
9.15.9 Norms ............................................................... 631
9.15.10 Arithmetic .......................................................... 631
9.15.11 Scalar arithmetic .................................................. 632
9.15.12 Vector arithmetic ................................................. 632
9.15.13 Gaussian elimination and solving ............................... 633
9.15.14 Characteristic polynomial and companion matrix ............. 635
9.15.15 Special functions ............................................... 635
9.15.16 Component and error operations ............................... 636
9.15.17 Eigenvalues and eigenvectors .................................. 636

9.16 **acb_hypgeom.h** – hypergeometric functions of complex variables .... 638
9.16.1 Rising factorials ................................................. 639
9.16.2 Gamma function .................................................... 639
9.16.3 Convergent series ............................................... 640
9.16.4 Asymptotic series ............................................... 642
9.16.5 Generalized hypergeometric function ............................ 642
9.16.6 Confluent hypergeometric functions ............................. 642
9.16.7 Error functions and Fresnel integrals ............................ 643
9.16.8 Bessel functions .................................................... 644
9.16.9 Modified Bessel functions ....................................... 645
9.16.10 Airy functions ..................................................... 646
9.16.11 Coulomb wave functions ........................................ 647
9.16.12 Incomplete gamma and beta functions .......................... 648
9.16.13 Exponential and trigonometric integrals ....................... 649
9.16.14 Gauss hypergeometric function ................................ 651
9.16.15 Orthogonal polynomials and functions ......................... 652
9.16.16 Dilogarithm ........................................................ 654

9.17 **arb_hypgeom.h** – hypergeometric functions of real variables ........ 655
9.17.1 Rising factorials .................................................... 655
9.17.2 Gamma function .................................................... 656
9.17.3 Binomial coefficients .............................................. 656
9.17.4 Generalized hypergeometric function ............................. 656
9.17.5 Confluent hypergeometric functions ............................. 657
9.17.6 Gauss hypergeometric function ................................ 657
9.17.7 Error functions and Fresnel integrals ........................................... 658
9.17.8 Incomplete gamma and beta functions .......................................... 658
9.17.9 Exponential and trigonometric integrals ....................................... 660
9.17.10 Bessel functions ........................................................................... 661
9.17.11 Airy functions .............................................................................. 662
9.17.12 Coulomb wave functions ............................................................... 662
9.17.13 Orthogonal polynomials and functions ......................................... 663
9.17.14 Dilogarithm .................................................................................. 664
9.17.15 Hypergeometric sums .................................................................. 664
9.18 **acb_elliptic.h** – elliptic integrals and functions of complex variables .... 664
  9.18.1 Complete elliptic integrals ............................................................. 665
  9.18.2 Legendre incomplete elliptic integrals ............................................ 665
  9.18.3 Carlson symmetric elliptic integrals ............................................... 666
  9.18.4 Weierstrass elliptic functions ........................................................ 668
9.19 **acb_modular.h** – modular forms of complex variables ...................... 669
  9.19.1 The modular group ....................................................................... 669
  9.19.2 Modular transformations ............................................................... 670
  9.19.3 Addition sequences ...................................................................... 671
  9.19.4 Jacobi theta functions .................................................................. 671
  9.19.5 Dedekind eta function .................................................................. 674
  9.19.6 Modular forms ............................................................................... 674
  9.19.7 Elliptic integrals and functions ..................................................... 675
  9.19.8 Class polynomials ....................................................................... 675
9.20 **acb_theta.h** – Riemann theta functions .......................................... 676
  9.20.1 Main user functions ...................................................................... 676
  9.20.2 Example of usage .......................................................................... 677
  9.20.3 The Siegel modular group .............................................................. 677
  9.20.4 The Siegel half space .................................................................... 679
  9.20.5 Theta characteristics .................................................................... 680
  9.20.6 Ellipsoids: types and macros .......................................................... 681
  9.20.7 Ellipsoids: memory management and computations ....................... 682
  9.20.8 Naive algorithms: error bounds ...................................................... 682
  9.20.9 Naive algorithms: main functions ................................................ 683
  9.20.10 Naive algorithms for derivatives ................................................ 684
  9.20.11 Quasi-linear algorithms: presentation ......................................... 686
  9.20.12 Quasi-linear algorithms: distances .............................................. 687
  9.20.13 Quasi-linear algorithms: AGM steps .......................................... 688
  9.20.14 Quasi-linear algorithms: main functions .................................... 688
  9.20.15 Quasi-linear algorithms: derivatives .......................................... 690
  9.20.16 The transformation formula ........................................................ 692
  9.20.17 Dimension 2 specifics ................................................................. 693
  9.20.18 Tests ......................................................................................... 696
  9.20.19 Profiling ..................................................................................... 702
9.21 **acb_dirichlet.h** – Dirichlet L-functions, Riemann zeta and related functions . 703
  9.21.1 Roots of unity ............................................................................... 703
  9.21.2 Truncated L-series and power sums .............................................. 704
  9.21.3 Riemann zeta function ................................................................. 704
  9.21.4 Riemann-Siegel formula ............................................................... 705
  9.21.5 Hurwitz zeta function .................................................................. 705
  9.21.6 Hurwitz zeta function precomputation ......................................... 706
  9.21.7 Lerch transcendent ......................................................................... 706
  9.21.8 Stieltjes constants ....................................................................... 707
  9.21.9 Dirichlet character evaluation ...................................................... 707
  9.21.10 Dirichlet character Gauss, Jacobi and theta sums ......................... 707
  9.21.11 Discrete Fourier transforms ........................................................ 709
  9.21.12 Dirichlet L-functions ................................................................. 710
  9.21.13 Hardy Z-functions .................................................................... 712
  9.21.14 Gram points ............................................................................... 712
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.22</td>
<td>bernoulli.h – support for Bernoulli numbers</td>
<td>715</td>
</tr>
<tr>
<td>9.22.1</td>
<td>Generation of Bernoulli numbers</td>
<td>715</td>
</tr>
<tr>
<td>9.22.2</td>
<td>Caching</td>
<td>716</td>
</tr>
<tr>
<td>9.22.3</td>
<td>Bounding</td>
<td>716</td>
</tr>
<tr>
<td>9.22.4</td>
<td>Isolated Bernoulli numbers</td>
<td>716</td>
</tr>
<tr>
<td>9.23</td>
<td>hypgeom.h – support for hypergeometric series</td>
<td>717</td>
</tr>
<tr>
<td>9.23.1</td>
<td>Strategy for error bounding</td>
<td>717</td>
</tr>
<tr>
<td>9.23.2</td>
<td>Types, macros and constants</td>
<td>718</td>
</tr>
<tr>
<td>9.23.3</td>
<td>Memory management</td>
<td>718</td>
</tr>
<tr>
<td>9.23.4</td>
<td>Error bounding</td>
<td>718</td>
</tr>
<tr>
<td>9.23.5</td>
<td>Summation</td>
<td>719</td>
</tr>
<tr>
<td>9.24</td>
<td>partitions.h – computation of the partition function</td>
<td>719</td>
</tr>
<tr>
<td>9.25</td>
<td>arb_calc.h – calculus with real-valued functions</td>
<td>720</td>
</tr>
<tr>
<td>9.25.1</td>
<td>Types, macros and constants</td>
<td>720</td>
</tr>
<tr>
<td>9.25.2</td>
<td>Debugging</td>
<td>721</td>
</tr>
<tr>
<td>9.25.3</td>
<td>Subdivision-based root finding</td>
<td>721</td>
</tr>
<tr>
<td>9.25.4</td>
<td>Newton-based root finding</td>
<td>722</td>
</tr>
<tr>
<td>9.26</td>
<td>acb_calc.h – calculus with complex-valued functions</td>
<td>723</td>
</tr>
<tr>
<td>9.26.1</td>
<td>Types, macros and constants</td>
<td>723</td>
</tr>
<tr>
<td>9.26.2</td>
<td>Integration</td>
<td>724</td>
</tr>
<tr>
<td>9.26.3</td>
<td>Local integration algorithms</td>
<td>726</td>
</tr>
<tr>
<td>9.26.4</td>
<td>Integration (old)</td>
<td>727</td>
</tr>
<tr>
<td>9.27</td>
<td>arb_fpwrap.h – floating-point wrappers of Arb mathematical functions</td>
<td>728</td>
</tr>
<tr>
<td>9.27.1</td>
<td>Option and return flags</td>
<td>728</td>
</tr>
<tr>
<td>9.27.2</td>
<td>Types</td>
<td>729</td>
</tr>
<tr>
<td>9.27.3</td>
<td>Functions</td>
<td>729</td>
</tr>
<tr>
<td>9.27.4</td>
<td>Calling from C</td>
<td>736</td>
</tr>
<tr>
<td>9.27.5</td>
<td>Interfacing from Python</td>
<td>737</td>
</tr>
<tr>
<td>9.27.6</td>
<td>Interfacing from Julia</td>
<td>738</td>
</tr>
<tr>
<td>9.28</td>
<td>fmpz_extras.h – extra methods for FLINT integers</td>
<td>738</td>
</tr>
<tr>
<td>9.28.1</td>
<td>Memory-related methods</td>
<td>739</td>
</tr>
<tr>
<td>9.28.2</td>
<td>Convenience methods</td>
<td>739</td>
</tr>
<tr>
<td>9.28.3</td>
<td>Inlined arithmetic</td>
<td>739</td>
</tr>
<tr>
<td>9.28.4</td>
<td>Low-level conversions</td>
<td>740</td>
</tr>
<tr>
<td>9.29</td>
<td>General formulas and bounds</td>
<td>740</td>
</tr>
<tr>
<td>9.29.1</td>
<td>Error propagation</td>
<td>740</td>
</tr>
<tr>
<td>9.29.2</td>
<td>Sums and series</td>
<td>741</td>
</tr>
<tr>
<td>9.29.3</td>
<td>Complex analytic functions</td>
<td>741</td>
</tr>
<tr>
<td>9.29.4</td>
<td>Euler-Maclaurin formula</td>
<td>742</td>
</tr>
<tr>
<td>9.30</td>
<td>Algorithms for mathematical constants</td>
<td>742</td>
</tr>
<tr>
<td>9.30.1</td>
<td>Pi</td>
<td>742</td>
</tr>
<tr>
<td>9.30.2</td>
<td>Logarithms of integers</td>
<td>743</td>
</tr>
<tr>
<td>9.30.3</td>
<td>Euler’s constant</td>
<td>743</td>
</tr>
<tr>
<td>9.30.4</td>
<td>Catalan’s constant</td>
<td>743</td>
</tr>
<tr>
<td>9.30.5</td>
<td>Apery’s constant</td>
<td>743</td>
</tr>
<tr>
<td>9.30.6</td>
<td>Khinchin’s constant</td>
<td>744</td>
</tr>
<tr>
<td>9.30.7</td>
<td>Glaisher’s constant</td>
<td>744</td>
</tr>
<tr>
<td>9.30.8</td>
<td>Reciprocal Fibonacci constant</td>
<td>744</td>
</tr>
<tr>
<td>9.31</td>
<td>Algorithms for the gamma function</td>
<td>744</td>
</tr>
<tr>
<td>9.31.1</td>
<td>The Stirling series</td>
<td>744</td>
</tr>
<tr>
<td>9.31.2</td>
<td>Rational arguments</td>
<td>745</td>
</tr>
<tr>
<td>9.32</td>
<td>Algorithms for the Hurwitz zeta function</td>
<td>745</td>
</tr>
<tr>
<td>9.32.1</td>
<td>Euler-Maclaurin summation</td>
<td>745</td>
</tr>
<tr>
<td>9.32.2</td>
<td>Parameter Taylor series</td>
<td>745</td>
</tr>
<tr>
<td>9.33</td>
<td>Algorithms for polylogarithms</td>
<td>746</td>
</tr>
<tr>
<td>9.33.1</td>
<td>Computation for small z</td>
<td>746</td>
</tr>
</tbody>
</table>
10 Exact real and complex numbers

10.1 Introduction .................................................. 753
10.1.1 Exact numbers in Calcium ............................... 753
10.1.2 FAQ .......................................................... 754

10.2 Calcium example programs .................................. 756
10.2.1 elementary.c .............................................. 756
10.2.2 binet.c .................................................... 757
10.2.3 machin.c .................................................. 758
10.2.4 swinnerton_dyer_poly.c ................................. 758
10.2.5 huge_expr.c ............................................... 759
10.2.6 hilbert_matrix.c ......................................... 760
10.2.7 dft.c ..................................................... 760

10.3 calcium.h – global definitions ............................. 763
10.3.1 Version .................................................... 763
10.3.2 Triple-valued logic ....................................... 763
10.3.3 Flint, Arb and Antic extras ............................. 763
10.3.4 Input and output ......................................... 763

10.4 ca.h – exact real and complex numbers ................. 764
10.4.1 Introduction: numbers ................................... 764
10.4.2 Introduction: special values ............................ 764
10.4.3 Number objects ........................................... 765
10.4.4 Context objects .......................................... 765
10.4.5 Memory management for numbers ..................... 766
10.4.6 Symbolic expressions ................................... 766
10.4.7 Printing ................................................... 766
10.4.8 Special values ............................................ 768
10.4.9 Assignment and conversion ............................. 768
10.4.10 Conversion of algebraic numbers ...................... 769
10.4.11 Random generation ..................................... 769
10.4.12 Representation properties .............................. 770
10.4.13 Value predicates ........................................ 771
10.4.14 Comparisons ............................................ 772
10.4.15 Field structure operations .............................. 772
10.4.16 Arithmetic ............................................... 772
10.4.17 Powers and roots ........................................ 775
10.4.18 Complex parts ........................................... 775
10.4.19 Exponentials and logarithms ........................... 777
10.4.20 Trigonometric functions ............................... 777
10.4.21 Special functions ........................................ 779
10.4.22 Numerical evaluation ................................... 779
10.4.23 Rewriting and simplification .......................... 780
10.4.24 Factorization ............................................ 780
10.4.25 Context options ......................................... 781
10.4.26 Internal representation ................................. 783

10.5 ca_vec.h – vectors of real and complex numbers ...... 784
10.5.1 Types, macros and constants .................................................. 784
10.5.2 Memory management ............................................................. 784
10.5.3 Length ............................................................................. 784
10.5.4 Assignment ........................................................................ 785
10.5.5 Special vectors ..................................................................... 785
10.5.6 Input and output ................................................................. 785
10.5.7 List operations .................................................................... 785
10.5.8 Arithmetic .......................................................................... 785
10.5.9 Comparisons and properties .................................................. 786
10.5.10 Internal representation ......................................................... 786

10.6 `ca_poly.h` – dense univariate polynomials over the real and complex numbers ........................................ 787
10.6.1 Types, macros and constants .................................................. 787
10.6.2 Memory management ............................................................. 787
10.6.3 Assignment and simple values ............................................... 788
10.6.4 Random generation ............................................................... 788
10.6.5 Input and output ................................................................. 788
10.6.6 Degree and leading coefficient ................................................ 789
10.6.7 Comparisons ..................................................................... 789
10.6.8 Arithmetic .......................................................................... 789
10.6.9 Evaluation and composition .................................................... 790
10.6.10 Derivative and integral ......................................................... 791
10.6.11 Power series division ............................................................ 791
10.6.12 Elementary functions .......................................................... 791
10.6.13 Greatest common divisor ..................................................... 791
10.6.14 Roots and factorization ......................................................... 792
10.6.15 Vectors of polynomials ........................................................ 792

10.7 `ca_mat.h` – matrices over the real and complex numbers ........................................................................... 793
10.7.1 Types, macros and constants .................................................. 793
10.7.2 Memory management ............................................................. 793
10.7.3 Assignment and conversions ............................................... 794
10.7.4 Random generation ............................................................... 794
10.7.5 Input and output ................................................................. 794
10.7.6 Special matrices .................................................................... 794
10.7.7 Comparisons and properties .................................................. 794
10.7.8 Conjugate and transpose ......................................................... 795
10.7.9 Arithmetic .......................................................................... 795
10.7.10 Powers ............................................................................. 795
10.7.11 Polynomial evaluation ........................................................ 796
10.7.12 Gaussian elimination and LU decomposition ......................... 797
10.7.13 Solving and inverse ............................................................... 798
10.7.14 Rank and echelon form ........................................................ 798
10.7.15 Determinant and trace ........................................................ 799
10.7.16 Characteristic polynomial ..................................................... 800
10.7.17 Eigenvalues and eigenvectors ................................................. 800
10.7.18 Jordan canonical form ........................................................ 800
10.7.19 Matrix functions ............................................................... 801

10.8 `ca_ext.h` – real and complex extension numbers ..................................................................................... 801
10.8.1 Type and macros ................................................................. 802
10.8.2 Memory management ............................................................. 803
10.8.3 Structure ............................................................................ 803
10.8.4 Input and output ................................................................. 803
10.8.5 Numerical evaluation ........................................................... 804
10.8.6 Cache ................................................................................. 804

10.9 `ca_field.h` – extension fields ........................................................................................................... 805
10.9.1 Type and macros ................................................................. 805
10.9.2 Memory management ............................................................. 806
10.9.3 Input and output ................................................................. 807
10.9.4 Ideal .................................................................................. 807
11 Finite fields

11.1 fq.h – finite fields

11.1.1 Types, macros and constants
11.1.2 Context Management
11.1.3 Memory management
11.1.4 Basic arithmetic
11.1.5 Roots
11.1.6 Output
11.1.7 Randomisation
11.1.8 Assignments and conversions
11.1.9 Comparison
11.1.10 Special functions
11.1.11 Bit packing

11.2 fq_default_default.h – unified finite fields

11.2.1 Types, macros and constants
11.2.2 Context Management
11.2.3 Memory management
11.2.4 Predicates
11.2.5 Basic arithmetic
11.2.6 Roots
11.2.7 Output
11.2.8 Randomisation
11.2.9 Assignments and conversions
11.2.10 Comparison
11.2.11 Special functions

11.3 fq_vec.h – vectors over finite fields

11.3.1 Memory management
11.3.2 Randomisation
11.3.3 Input and output
11.3.4 Assignment and basic manipulation
11.3.5 Comparison
11.6.15 Euclidean division ........................................ 871
11.6.16 Greatest common divisor ................................. 873
11.6.17 Divisibility testing ........................................ 874
11.6.18 Derivative .................................................. 874
11.6.19 Square root ................................................ 875
11.6.20 Evaluation .................................................. 875
11.6.21 Composition ............................................... 875
11.6.22 Output ....................................................... 878
11.6.23 Inflation and deflation .................................... 879
11.7 \texttt{fq\_default\_poly.h} – univariate polynomials over finite fields ........................................ 879
  11.7.1 Types, macros and constants .............................. 879
  11.7.2 Memory management ...................................... 879
  11.7.3 Polynomial parameters ................................... 880
  11.7.4 Randomisation ............................................. 880
  11.7.5 Assignment and basic manipulation ....................... 880
  11.7.6 Getting and setting coefficients ......................... 881
  11.7.7 Comparison ............................................... 881
  11.7.8 Addition and subtraction ................................ 881
  11.7.9 Scalar multiplication and division ....................... 882
  11.7.10 Multiplication ........................................... 882
  11.7.11 Squaring ................................................. 883
  11.7.12 Powering ................................................ 883
  11.7.13 Shifting ................................................ 883
  11.7.14 Norms .................................................... 884
  11.7.15 Euclidean division ...................................... 884
  11.7.16 Greatest common divisor ............................... 884
  11.7.17 Division testing ......................................... 885
  11.7.18 Derivative ............................................... 885
  11.7.19 Square root ............................................. 885
  11.7.20 Evaluation .............................................. 885
  11.7.21 Composition ............................................. 885
  11.7.22 Output ................................................... 886
  11.7.23 Inflation and deflation ................................. 886
11.8 \texttt{fq\_poly\_factor.h} – factorisation of univariate polynomials over finite fields .................. 887
  11.8.1 Types, macros and constants ............................. 887
  11.8.2 Memory Management ...................................... 887
  11.8.3 Basic Operations ......................................... 887
  11.8.4 Irreducibility Testing ................................... 888
  11.8.5 Factorisation ............................................. 888
  11.8.6 Root Finding ............................................. 889
11.9 \texttt{fq\_default\_poly\_factor.h} – factorisation of univariate polynomials over finite fields ............ 890
  11.9.1 Types, macros and constants ............................. 890
  11.9.2 Memory Management ...................................... 890
  11.9.3 Basic Operations ......................................... 890
  11.9.4 Irreducibility Testing ................................... 891
  11.9.5 Factorisation ............................................. 891
  11.9.6 Root Finding ............................................. 892
11.10 \texttt{fq\_embed.h} – Computing isomorphisms and embeddings of finite fields ..................... 892
11.11 \texttt{fq\_nmod.h} – finite fields (word-size characteristic) ............................................ 893
  11.11.1 Types, macros and constants ............................ 893
  11.11.2 Context Management ..................................... 894
  11.11.3 Memory management ..................................... 895
  11.11.4 Basic arithmetic ......................................... 895
  11.11.5 Roots ..................................................... 896
  11.11.6 Output .................................................... 897
  11.11.7 Randomisation ............................................ 897
  11.11.8 Assignments and conversions ............................ 897
  11.11.9 Comparison .............................................. 898
11.11.10 Special functions .......................................................... 899
11.11.11 Bit packing ................................................................. 899

11.12 `fq_nmod_vec.h` – vectors over finite fields (word-size characteristic) .......... 900
11.12.1 Memory management ..................................................... 900
11.12.2 Randomisation ............................................................... 900
11.12.3 Input and output ............................................................ 900
11.12.4 Assignment and basic manipulation .................................. 900
11.12.5 Comparison ................................................................. 901
11.12.6 Addition and subtraction ................................................ 901
11.12.7 Scalar multiplication and division .................................... 901
11.12.8 Dot products ............................................................... 901

11.13 `fq_nmod_mat.h` – matrices over finite fields (word-size characteristic) ....... 901
11.13.1 Types, macros and constants .......................................... 901
11.13.2 Memory management ..................................................... 902
11.13.3 Basic properties and manipulation .................................. 902
11.13.4 Conversions ............................................................... 903
11.13.5 Concatenate ............................................................... 903
11.13.6 Printing ................................................................. 903
11.13.7 Window ................................................................. 904
11.13.8 Random matrix generation ............................................ 904
11.13.9 Comparison ............................................................... 904
11.13.10 Addition and subtraction .............................................. 905
11.13.11 Matrix multiplication .................................................. 905
11.13.12 Inverse ............................................................... 906
11.13.13 LU decomposition ..................................................... 906
11.13.14 Reduced row echelon form ....................................... 906
11.13.15 Triangular solving .................................................. 907
11.13.16 Solving ............................................................... 908
11.13.17 Transforms ........................................................... 908
11.13.18 Characteristic polynomial .......................................... 908
11.13.19 Minimal polynomial ............................................... 908

11.14 `fq_nmod_poly.h` – univariate polynomials over finite fields (word-size characteristic) .... 909
11.14.1 Types, macros and constants .......................................... 909
11.14.2 Memory management ..................................................... 909
11.14.3 Polynomial parameters ................................................. 910
11.14.4 Randomisation ........................................................... 910
11.14.5 Assignment and basic manipulation .................................. 911
11.14.6 Getting and setting coefficients ..................................... 911
11.14.7 Comparison ............................................................... 912
11.14.8 Addition and subtraction .............................................. 912
11.14.9 Scalar multiplication and division .................................... 913
11.14.10 Multiplication ........................................................... 913
11.14.11 Squaring ............................................................... 916
11.14.12 Powering ............................................................... 917
11.14.13 Shifting ............................................................... 919
11.14.14 Norms ................................................................. 920
11.14.15 Euclidean division ................................................... 920
11.14.16 Greatest common divisor ......................................... 922
11.14.17 Divisibility testing .................................................. 923
11.14.18 Derivative ............................................................. 924
11.14.19 Square root ........................................................... 924
11.14.20 Evaluation ............................................................. 924
11.14.21 Composition ........................................................... 925
11.14.22 Output ................................................................. 927
11.14.23 Inflation and deflation ............................................... 928

11.15 `fq_nmod_poly_factor.h` – factorisation of univariate polynomials over finite fields (word-size characteristic) ................................................................. 929
11.15.1 Types, macros and constants .......................................... 929
<table>
<thead>
<tr>
<th>11.15.2 Memory Management</th>
<th>929</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.15.3 Basic Operations</td>
<td>929</td>
</tr>
<tr>
<td>11.15.4 Irreducibility Testing</td>
<td>930</td>
</tr>
<tr>
<td>11.15.5 Factorisation</td>
<td>930</td>
</tr>
<tr>
<td>11.15.6 Root Finding</td>
<td>932</td>
</tr>
<tr>
<td>11.16 <code>fq_nmod_embed.h</code> – Computing isomorphisms and embeddings of finite fields</td>
<td>932</td>
</tr>
<tr>
<td>11.17 <code>fq_nmod_mpoly.h</code> – multivariate polynomials over finite fields of word-sized characteristic</td>
<td>933</td>
</tr>
<tr>
<td>11.17.1 Types, macros and constants</td>
<td>933</td>
</tr>
<tr>
<td>11.17.2 Context object</td>
<td>934</td>
</tr>
<tr>
<td>11.17.3 Memory management</td>
<td>934</td>
</tr>
<tr>
<td>11.17.4 Input/Output</td>
<td>934</td>
</tr>
<tr>
<td>11.17.5 Basic manipulation</td>
<td>935</td>
</tr>
<tr>
<td>11.17.6 Constants</td>
<td>935</td>
</tr>
<tr>
<td>11.17.7 Degrees</td>
<td>936</td>
</tr>
<tr>
<td>11.17.8 Coefficients</td>
<td>936</td>
</tr>
<tr>
<td>11.17.9 Comparison</td>
<td>937</td>
</tr>
<tr>
<td>11.17.10 Container operations</td>
<td>937</td>
</tr>
<tr>
<td>11.17.11 Random generation</td>
<td>939</td>
</tr>
<tr>
<td>11.17.12 Addition/Subtraction</td>
<td>939</td>
</tr>
<tr>
<td>11.17.13 Scalar operations</td>
<td>939</td>
</tr>
<tr>
<td>11.17.14 Differentiation</td>
<td>940</td>
</tr>
<tr>
<td>11.17.15 Evaluation</td>
<td>940</td>
</tr>
<tr>
<td>11.17.16 Multiplication</td>
<td>940</td>
</tr>
<tr>
<td>11.17.17 Powering</td>
<td>941</td>
</tr>
<tr>
<td>11.17.18 Division</td>
<td>941</td>
</tr>
<tr>
<td>11.17.19 Greatest Common Divisor</td>
<td>941</td>
</tr>
<tr>
<td>11.17.20 Square Root</td>
<td>942</td>
</tr>
<tr>
<td>11.17.21 Univariate Functions</td>
<td>942</td>
</tr>
<tr>
<td>11.18 <code>fq_nmod_mpoly_factor.h</code> – factorisation of multivariate polynomials over finite fields of word-sized characteristic</td>
<td>943</td>
</tr>
<tr>
<td>11.18.1 Types, macros and constants</td>
<td>943</td>
</tr>
<tr>
<td>11.18.2 Memory management</td>
<td>943</td>
</tr>
<tr>
<td>11.18.3 Basic manipulation</td>
<td>943</td>
</tr>
<tr>
<td>11.18.4 Factorisation</td>
<td>944</td>
</tr>
<tr>
<td>11.19 <code>fq_zech.h</code> – finite fields (Zech logarithm representation)</td>
<td>944</td>
</tr>
<tr>
<td>11.19.1 Types, macros and constants</td>
<td>944</td>
</tr>
<tr>
<td>11.19.2 Context Management</td>
<td>945</td>
</tr>
<tr>
<td>11.19.3 Memory management</td>
<td>946</td>
</tr>
<tr>
<td>11.19.4 Basic arithmetic</td>
<td>947</td>
</tr>
<tr>
<td>11.19.5 Roots</td>
<td>948</td>
</tr>
<tr>
<td>11.19.6 Output</td>
<td>948</td>
</tr>
<tr>
<td>11.19.7 Randomisation</td>
<td>948</td>
</tr>
<tr>
<td>11.19.8 Assignments and conversions</td>
<td>949</td>
</tr>
<tr>
<td>11.19.9 Comparison</td>
<td>950</td>
</tr>
<tr>
<td>11.19.10 Special functions</td>
<td>950</td>
</tr>
<tr>
<td>11.19.11 Bit packing</td>
<td>951</td>
</tr>
<tr>
<td>11.20 <code>fq_zech_vec.h</code> – vectors over finite fields (Zech logarithm representation)</td>
<td>951</td>
</tr>
<tr>
<td>11.20.1 Memory management</td>
<td>951</td>
</tr>
<tr>
<td>11.20.2 Randomisation</td>
<td>951</td>
</tr>
<tr>
<td>11.20.3 Input and output</td>
<td>951</td>
</tr>
<tr>
<td>11.20.4 Assignment and basic manipulation</td>
<td>951</td>
</tr>
<tr>
<td>11.20.5 Comparison</td>
<td>952</td>
</tr>
<tr>
<td>11.20.6 Addition and subtraction</td>
<td>952</td>
</tr>
<tr>
<td>11.20.7 Scalar multiplication and division</td>
<td>952</td>
</tr>
<tr>
<td>11.20.8 Dot products</td>
<td>952</td>
</tr>
<tr>
<td>11.21 <code>fq_zech_mat.h</code> – matrices over finite fields (Zech logarithm representation)</td>
<td>952</td>
</tr>
<tr>
<td>11.21.1 Types, macros and constants</td>
<td>952</td>
</tr>
</tbody>
</table>
11.21.2 Memory management ........................................... 953
11.21.3 Basic properties and manipulation ............................... 953
11.21.4 Conversions .................................................. 953
11.21.5 Concatenate .................................................. 954
11.21.6 Printing ...................................................... 954
11.21.7 Window ...................................................... 954
11.21.8 Random matrix generation ...................................... 954
11.21.9 Comparison .................................................. 955
11.21.10 Addition and subtraction ...................................... 955
11.21.11 Matrix multiplication ......................................... 956
11.21.12 LU decomposition ......................................... 956
11.21.13 Reduced row echelon form .................................... 957
11.21.14 Triangular solving ........................................... 957
11.21.15 Solving ..................................................... 958
11.21.16 Transforms .................................................. 958
11.21.17 Characteristic polynomial ..................................... 959
11.21.18 Minimal polynomial .......................................... 959

11.22 fq_zech_poly.h – univariate polynomials over finite fields (Zech logarithm representation) 959
11.22.1 Types, macros and constants .................................... 959
11.22.2 Memory management ........................................... 959
11.22.3 Polynomial parameters ......................................... 960
11.22.4 Randomisation ................................................ 961
11.22.5 Assignment and basic manipulation ............................ 961
11.22.6 Getting and setting coefficients ................................ 962
11.22.7 Comparison .................................................. 962
11.22.8 Addition and subtraction ...................................... 963
11.22.9 Scalar multiplication and division ............................. 963
11.22.10 Multiplication ............................................... 964
11.22.11 Squaring ................................................... 966
11.22.12 Powering ................................................... 967
11.22.13 Shifting .................................................... 969
11.22.14 Norms ..................................................... 970
11.22.15 Euclidean division ........................................... 970
11.22.16 Greatest common divisor ..................................... 972
11.22.17 Divisibility testing .......................................... 973
11.22.18 Derivative .................................................. 974
11.22.19 Square root ................................................ 974
11.22.20 Evaluation ................................................... 974
11.22.21 Composition ............................................... 975
11.22.22 Output ...................................................... 977
11.22.23 Inflation and deflation ....................................... 978

11.23 fq_zech_poly_factor.h – factorisation of univariate polynomials over finite fields
   (Zech logarithm representation) ..................................... 979
11.23.1 Types, macros and constants .................................... 979
11.23.2 Memory Management ........................................... 979
11.23.3 Basic Operations ............................................... 979
11.23.4 Irreducibility Testing .......................................... 980
11.23.5 Factorisation .................................................. 980
11.23.6 Root Finding .................................................. 982

11.24 fq_zech_embed.h – Computing isomorphisms and embeddings of finite fields ................. 982

12 p-adic numbers .......................................................... 985
12.1 padic.h – p-adic numbers ........................................... 985
12.1.1 Introduction .................................................. 985
12.1.2 Data structures ............................................... 985
12.1.3 Context ....................................................... 986
12.1.4 Memory management .......................................... 986
12.1.5 Randomisation ................................................ 987
# 12.2 padic_poly.h — polynomials over p-adic numbers

- Module documentation ........................................................ 992
- Memory management ......................................................... 992
- Polynomial parameters ...................................................... 993
- Randomisation ................................................................ 993
- Assignment and basic manipulation ...................................... 994
- Getting and setting coefficients .......................................... 994
- Comparison ........................................................................ 995
- Addition and subtraction ..................................................... 995
- Scalar multiplication .......................................................... 995
- Multiplication .................................................................... 996
- Series inversion ................................................................ 996
- Derivative .......................................................................... 997
- Shifting ............................................................................. 997
- Evaluation .......................................................................... 997
- Composition ....................................................................... 997
- Input and output ................................................................. 998
- Testing ............................................................................... 999

# 12.3 padic_mat.h — matrices over p-adic numbers

- Module documentation ........................................................ 999
- Macros .............................................................................. 999
- Memory management ......................................................... 1000
- Basic assignment .............................................................. 1000
- Conversions ..................................................................... 1001
- Entries ............................................................................. 1001
- Comparison ...................................................................... 1001
- Input and output ............................................................... 1001
- Random matrix generation ................................................ 1002
- Transpose ....................................................................... 1002
- Addition and subtraction .................................................... 1002
- Scalar operations .............................................................. 1002
- Multiplication .................................................................. 1003

# 12.4 padic.h — unramified extensions over p-adic numbers

- Data structures .................................................................. 1003
- Context ............................................................................ 1003
- Memory management ........................................................ 1004
- Properties ....................................................................... 1005
- Randomisation ................................................................ 1005
- Assignments and conversions ............................................. 1005
- Comparison ..................................................................... 1005
- Basic arithmetic ............................................................... 1006
- Square root .................................................................... 1006
- Special functions .............................................................. 1007
- Output ............................................................................. 1007

# 13 Floating-point support code

**13.1 double_extras.h** — support functions for double arithmetic

- Random functions ............................................................. 1011
- Arithmetic ....................................................................... 1011
- Special functions .............................................................. 1011
Welcome to FLINT’s documentation! FLINT is a C library for doing number theory.

- Website: https://flintlib.org
- Source code on GitHub: https://github.com/flintlib/flint
- Issue tracker: https://github.com/flintlib/flint/issues
- Mailing list: https://groups.google.com/group/flint-devel

FLINT is free software distributed under the GNU Lesser General Public License (LGPL), version 2.1 or later.
1.1 Introduction

1.1.1 What is Flint?

FLINT is a C library of functions for doing basic arithmetic in support of computational number theory and other areas of computer algebra. It is highly optimised and can be compiled on numerous platforms.

FLINT provides highly optimised implementations of basic rings, such as the integers, rationals, \( p \)-adics, finite fields, etc., and linear algebra and univariate and multivariate polynomials over most of these rings.

FLINT also has some multithreading capabilities. To this end, the library is threadsafe, with few exceptions noted in the appropriate place, and a number of key functions have multithreaded implementations.

1.1.2 Maintainers and Authors

FLINT is currently maintained by Fredrik Johansson of INRIA Bordeaux.

FLINT was originally designed by William Hart and David Harvey. Since then FLINT was rewritten as FLINT 2 by William Hart, Fredrik Johansson and Sebastian Pancratz. Many other substantial contributions have been made by other authors, e.g. Tom Bachmann, Mike Hansen, Daniel Schultz and Andy Novocin. There have been a great number of other contributors, listed on the main Flint website and the contributors section of this documentation.

1.1.3 Requirements

FLINT and following should compile on any machine with GCC and a standard GNU toolchain, though GCC 4.8 and following are recommended.

Flint is specially optimised for x86 (32 and 64 bit) machines. There is also limited optimisation for ARM machines.

As of version 3.0, FLINT requires GMP 6.2.1 or later, and MPFR 4.1.0 or later. Note that earlier, MPIR, a fork of GMP, was supported. However, as of FLINT 3.0, this support has been dropped.

It is also required that the platform provide a \texttt{uint64_t} type if a native 64 bit type is not available. Full C99 compliance is not required.
1.1.4 Structure of Flint

FLINT is supplied as a set of modules, \texttt{fmpz}, \texttt{fmpz\_poly}, etc., each of which can be linked to a C program making use of their functionality.

All of the functions in FLINT have a corresponding test function provided in an appropriately named test file. For example, the function \texttt{fmpz\_poly\_add} located in \texttt{src/fmpz\_poly/add.c} has test code in the file \texttt{src/fmpz\_poly/test/t-add.c}.

Some modules have a \texttt{profile} directory in which profile programs can be found.

Documentation exists in the \texttt{doc/source} directory in a series of \texttt{.rst} files.

1.1.5 License

FLINT is distributed under the LGPL License, version 2.1+. There is a copy of the license included in the repository and distribution tarballs.

1.2 Building, testing and installing

1.2.1 Quick start

Building FLINT requires:

- GMP, at least version 6.2.1 (https://gmplib.org/)
- MPFR, at least version 4.1.0 (https://mpfr.org/)
- Either of the following build systems:
  - GNU Make together with GNU Autotools (Recommended)
  - CMake (Recommended only for Windows users)

On a typical Linux or Unix-like system where Autotools is available (see below for instructions using CMake), FLINT can be built and installed as follows:

```
./bootstrap.sh
./configure --disable-static
make -j N
make install
```

where \( N \) is the number of jobs number allowed to run parallel. Typically, the fastest way to build is to let \( N \) be the number of threads your CPU plus one, which can be obtained in Bash through `$(expr $(nproc) + 1)`.

We also recommend that you check that the library works as it should through `make check`, or `make -j N check` for a parallel check, before installing.

For a complete list of build settings, type

```
./configure --help
```

An example of a custom configuration command would be

```
./configure --enable-assert
--enable-avx2
--disable-static
--with-gmp-include=/home/user1/builds/includes/
```

(continues on next page)
1.2.2 Library and install paths

If you intend to install the FLINT library and header files, you can specify where they should be placed by passing `--prefix=path` to `configure`, where `path` is the directory under which the `lib` and `include` directories exist into which you wish to place the FLINT files when it is installed.

If GMP and MPFR are not installed in the default search path of your compiler (e.g. `/usr/include` and `/usr/lib`), you must specify where they are by passing their location to `configure` `--with-gmp=ABSOLUTE_PATH` for GMP and `--with-mpfr=ABSOLUTE_PATH` for MPFR. Note that the FLINT build system can handle GMP and MPFR as installed at some location and as source builds (built from source but not installed). Though, to run the FLINT tests, GMP and MPFR needs to be properly installed.

1.2.3 Testing FLINT

The full FLINT test suite can be run using

```make
make check
```

or in parallel on a multicore system using

```make
make -j check
```

Here, `make -j N check` is typically the fastest way to build when `N` equals to the number of threads your system’s CPU has plus one, that is, `make -j $(expr $(nproc) + 1) check` typically is the fastest way to check FLINT.

Number of test iterations

The number of test iterations can be changed with the `FLINT_TEST_MULTIPLIER` environment variable. For example, the following will only run 10% of the default iterations:

```bash
export FLINT_TEST_MULTIPLIER=0.1
make check
```

Conversely, `FLINT_TEST_MULTIPLIER=10` will stress test FLINT by performing 10x the default number of iterations.

Testing single modules

If you wish to simply check a single module of FLINT you can pass the option `MOD=modname` to `make check`. You can also pass a list of module names:

```bash
make check MOD=ulong_extras
make -j N check MOD="fft fmpz_mat"
```
Testing single functions

Testing a single function is also possible, although one cannot utilize make all the way through for this. For example, if you would like to test the function \texttt{fmpz\_add} and \texttt{fmpz\_sub} in the module \texttt{fmpz}, you run:

\begin{verbatim}
# Build all tests
make tests
# Run the test executable for `fmpz` with `fmpz\_add` and `fmpz\_sub` as inputs
./build/fmpz/test/main fmpz\_add fmpz\_sub
\end{verbatim}

Test coverage

To obtain coverage statistics for the FLINT test suite, assuming that gcov and lcov are installed, configure FLINT with \texttt{--enable-coverage}. Then run:

\begin{verbatim}
make \(-j\) N check
make coverage_html
\end{verbatim}

This will place a coverage report in \texttt{build/coverage}.

1.2.4 Static or dynamic library only

FLINT builds static and shared libraries by default, except on platforms where this is not supported. If you do not require either a shared or static library then you may pass \texttt{--disable-static} or \texttt{--disable-shared} to configure. This can substantially speed up the build.

1.2.5 AVX2 instructions

On x86-64 machines with AVX2 support, compiling FLINT with the \texttt{--enable-avx2} option can improve performance substantially, notably by enabling the small-prime FFT. Currently this option is not enabled by default.

1.2.6 TLS, reentrancy and single mode

FLINT uses thread local storage by default (\texttt{--enable-tls}). However, if reentrancy is required on systems that do not support this, one can pass \texttt{--disable-tls} and mutexes will be used instead (requires POSIX). As most modern systems support thread local storage, it is not recommended to build FLINT without TLS.

There are two modes in which FLINT may installed: the default “single” mode, which is faster, but makes use of thread local storage for its memory manager and to handle threading, and a slower but less complicated “reentrant” mode. The later is useful when debugging a program where tracing allocations is important.

If you wish to select the single mode, pass the \texttt{--disable-reentrant} option to configure, though note that this is the default. The reentrant mode is selected by passing the option \texttt{--enable-reentrant} to configure.
1.2.7 ABI and architecture support

On some systems, e.g. Sparc and some Macs, more than one ABI is available. FLINT chooses the ABI based on the CPU type available, however its default choice can be overridden by passing either ABI=64 or ABI=32 to configure.

To build on MinGW64 it is necessary to pass ABI=64 to configure, as FLINT is otherwise unable to distinguish it from MinGW32.

In some cases, it is necessary to override the CPU/OS defaults. This can be done by specifying the build system triplet to configure via --build=arch-vendor-os.

It is also possible to override the default CC, AR and CFLAGS used by FLINT by passing CC=full_path_to_compiler, etc., to FLINT’s configure.

1.2.8 CMake build for Windows users

For Windows users, we also provide a way to install FLINT using CMake. Note, however, that FLINT’s CMake script only exists to provide Windows users a way to install FLINT. For UNIX-type systems, please use Autotools along with GNU Make instead, as described at the top of this page.

If you wish to install FLINT with CMake on Windows, simply type:

```
mkdir build && cd build
cmake .. -DBUILD_SHARED_LIBS=ON
cmake --build . --target install
```

1.2.9 Uninstalling FLINT

To uninstall FLINT with GNU make, type:

```
make uninstall
```

Now to use FLINT, simply include the appropriate header files for the FLINT modules you wish to use in your C program. Then compile your program, linking against the FLINT library, GMP, MPFR and pthreads with the options -lflint -lmpfr -lgmp -lpthread.

To clean up the local build files, use:

```
make clean
make distclean
```

1.2.10 Assertion checking

FLINT has an assert system. If you want a debug build you can pass --enable-assert to configure. However, this will slow FLINT considerably, so asserts should not be enabled (--disable-assert, the default) for deployment.
1.2.11 Linking and running code

Here is an example program to get started using FLINT:

```c
#include "flint/flint.h"
#include "flint/arb.h"

int main()
{
    arb_t x;
    arb_init(x);
    arb_const_pi(x, 50 * 3.33);
    arb_printn(x, 50, 0); flint_printf("\n\n");
    flint_printf("Computed with FLINT-%s\n", flint_version);
    arb_clear(x);
}
```

Compile it with:

```
gcc test.c -lflint
```

You may also have to pass the flags `-lmpfr` and `-lgmp` to the compiler. If the FLINT header and library files are not in a standard location such as `/usr/local`, you may also have to provide flags such as:

```
-I/path/to/flint -L/path/to/flint
```

Finally, to run the program, make sure that the linker can find `libflint`. If it is installed in a nonstandard location, you can for example add this path to the `LD_LIBRARY_PATH` environment variable.

The output of the example program should be something like the following:

```
[3.1415926535897932384626433832795028841971693993751 +/- 4.43e-50]
Computed with flint-3.0.0
```

1.3 Bug reporting

1.3.1 Reporting bugs

The maintainers wishes to be made aware of any and all bugs. Please open an issue at the GitHub repository (https://github.com/flintlib/flint) or send an email with your bug report to the FLINT devel list https://groups.google.com/group/flint-devel.

If possible please include details of your system, how Flint was compiled/installed, the versions of GMP and MPFR as well as precise details of how to replicate the bug.

Note that FLINT needs to be linked against version 6.2.1 or later of GMP, version 4.1.0 or later of MPFR. Version 4.8 or later of GCC is recommended for parallel builds.
1.4 Contributing to FLINT

1.4.1 Code conventions

Four steps are needed to add a new function:

- Add the function `module_foo()` in a new file `src/module/foo.c`.
- Add a corresponding test program in a new file `src/module/test/t-foo.c`.
- Add the function prototype to `src/module.h`.
- Document the function in `doc/source/module.rst`.

The build system takes care of everything else automatically.

Test code (see below) can be omitted if `module_foo()` is a trivial helper function, but it should at least be tested indirectly via another function in that case. Auxiliary functions needed to implement `module_foo()` but which have no use elsewhere should be declared as `static` in `src/module/foo.c`. If `module_foo()` is very short, it can be declared inline directly in `module.h` with the `MODULE_INLINE` macro.

Use the following checklist regarding code style:

- Try to keep names and function arguments consistent with existing code.
- Follow the conventions regarding types, aliasing rules, etc. described in `Technical conventions and potential issues` and in `code_conventions.txt`.
- Use basic FLINT constants, types and functions: `FLINT_BITS`, `flint_malloc/flint_free`, `flint_abort`, `flint_printf`, etc.
- Complex macros should be avoided.
- Indentation is four spaces.
- Curly braces normally go on a new line.
- Binary operators are surrounded by spaces (but parentheses and brackets are not).
- Logically distinct chunks of code (variable declarations, initialization, precomputations, the main loop, cleanup, etc.) should be separated by a single blank line.
- Lines are up to 79 characters long, but this rule can be broken if it helps readability.
- Add correct copyright notices at the top of each file.

1.4.2 Test code

The easiest way to write a test program for a new function is to adapt the test code for an existing, similar function.

Most of the test code in FLINT uses the strategy of computing the same result in two or more different ways (for example, using functional equations, interchanging the order of parameter, or varying the precision and other algorithm parameters) and verifying that the results are consistent. It is also a good idea to test that aliasing works. Input data is usually generated randomly, but in some cases including precomputed reference values also makes sense.

Faster test code is better. A single test program should not take more than 1 seconds to run with the default number of iterations, and preferably no more than 0.1 seconds. Most functions can be tested effectively in a few milliseconds. Think of what the corner cases are and try to generate random input biased toward such cases. The `randtest()` functions attempt to generate corner cases automatically, but some thought may be needed to use them optimally. Try to ensure that the test code fails if you deliberately break the tested function in any way. It is also a good idea to run the test code once with `FLINT_TEST_MULTIPLIER=10.0` or higher. If a function’s input space is too large to probe effectively for
corner cases with random input, that can be a hint that the function should be split into smaller logical parts that can be tested separately.

The test code must complete without errors when run with valgrind. The most common mistake leading to memory corruption or memory leaks is to miss or duplicate an init() or clear() call. Check that the init() and clear() calls exactly match the variable declarations in each code block, including the test code itself.

Profiling code is not needed in most cases, but it is often a good idea to run some benchmarks at least during the initial development of a new feature. The TIMEIT_START/TIMEIT_STOP and SHOW_MEMORY_USAGE macros in FLINT are useful for quick measurements.

1.5 Contributors

1.5.1 Contributors

FLINT has only been possible due to an extraordinary number of high quality contributions from a vast array of people.

A complete list of contributors is available on the FLINT website at: https://flintlib.org/authors.html

If you believe your name is missing from this list, please contact us immediately on the flint-devel list. The list is updated at the time of each new release of FLINT.

1.6 Examples

1.6.1 Example programs

FLINT comes with example programs to demonstrate current and future FLINT features. To build the example programs run:

```
make examples
```

The example programs are built in the build/examples directory.

For Arb and Calcium there are separate example pages Arb example programs and Calcium example programs. Below are some general examples.

- **partitions** Demonstrates the partition counting code, e.g. build/examples/partitions 1000000000 will compute the number of partitions of $10^9$.

- **delta_qexp** Computes the $n$-th term of the delta function, e.g. build/examples/delta_qexp 1000000 will compute the one million-th term of the $q$-expansion of delta.

- **crt** Demonstrates the integer Chinese Remainder code, e.g. build/examples/crt 10382788 will build up the given integer from its value mod various primes.

- **multi_crt** Demonstrates the fast tree version of the integer Chinese Remainder code, e.g. build/examples/multi_crt 100493287498239 13 will build up the given integer from its value mod the given number of primes.

- **stirling_matrix** Generates Stirling number matrices of the first and second kind and computes their product, which should come out as the identity matrix. The matrices are printed to standard output. For example build/examples/stirling_matrix 10 does this with 10 by 10 matrices.

- **fmpz_poly_factor_zassenhaus** Demonstrates the factorisation of a small polynomial. A larger polynomial is also provided on disk and a small (obvious) change to the example program will read this file instead of using the hard coded polynomial.
• **padic** Gives examples of the usage of many functions in the padic module.
• **fmpz_poly_q** Gives a very simple example of the fmpz_poly_q module.
• **fmpq_poly** Gives a very simple example of the fmpq_poly module.

1.7 Memory management

1.7.1 Memory allocation functions

The file *flint.h* defines functions *flint_malloc*, *flint_realloc*, *flint_calloc* and *flint_free*. They have the same interface as the standard library functions, but may perform additional error checking.

By default the memory allocation functions wrap the system’s *malloc*, *realloc*, *calloc* and *free*. The user can override this behaviour by calling *__flint_set_memory_functions* passing the *malloc*, *realloc*, *calloc* and *free* function pointers as parameters (see *flint.h* for the exact prototype). The current memory functions can be returned in a similar manner by calling *__flint_get_memory_functions* passing the address of pointers in which the function pointers can be stored.

Memory allocated with *flint_malloc* must be freed with *flint_free* and not with *free*.

1.7.2 Global caches and cleanup

FLINT may cache some data (such as allocated integers and tables of prime numbers) to speed up various computations. If FLINT is built in threadsafe mode, most caches are thread-local (some are always global and shared among the threads).

Data cached by the current thread can be freed by calling the *flint_cleanup()* function. The user can register additional cleanup functions to be invoked by *flint_cleanup()* by passing a pointer to a function with signature `void cleanup_function(void)` to *flint_register_cleanup_function()*.

The user should call *flint_cleanup_master()* exactly once right before exiting a program. This cleans up all caches in all threads and should result in a clean output with tools like *valgrind* if there are no memory leaks.

1.7.3 Temporary allocation

FLINT allows for temporary allocation of memory using *alloca* to allocate on the stack if the allocation is small enough.

The following program demonstrates how to use this facility to allocate two different arrays.

```c
#include <gmp.h>
#include "flint.h"

void myfun(void)
{
    /* other variable declarations */
    mp_ptr a, b;
    TMP_INIT;

    /* arbitrary code */

    TMP_START; /* we are about to do some allocation */
```

(continues on next page)
It is very important to note that temporary allocations should not be made in recursive functions or in loop bodies, as many small allocations on the stack can exhaust the stack causing a stack overflow.

1.8 Portability

1.8.1 Portable FLINT types

For platform independence, FLINT provides two types ulong and slong to replace unsigned long and long respectively. These are guaranteed to be the same size as GMP’s mp_limb_t and mp_limb_signed_t types, respectively.

A full list of types provided by FLINT is available in code_conventions.txt in the top-level source tree.

As FLINT supports Windows 64 on which the FLINT ulong and slong types are 64 bits, whilst unsigned long and long are only 32 bits, it is necessary to have a special format specifier which is 64 bits on Windows 64 instead of the usual "%lu" and "%ld".

For this purpose FLINT provides its own I/O functions, flint_printf, flint_fprintf, flint_sprintf, flint_scanf, flint_fscanf and flint_sscanf, which work exactly as the usual system versions, but which take the "%wu" and "%wd" format specifiers, which support FLINT ulong and slong types respectively.

Also, instead of using constants 123UL and 123L, FLINT provides the macros UWORD(123) and WORD(123) respectively for constants of type ulong and slong respectively.

The maximum and minimum values that can be represented by these types are given by UWORD_MAX and WORD_MAX respectively.

1.9 Threading

1.9.1 Multithreaded FLINT

FLINT provides a number of multithreaded functions, which use multiple threads by default if FLINT was built with at least pthreads. (This functionality works best when thread local storage is also available on the system.)

By default, FLINT will just use one thread. To control the maximum number of threads FLINT uses, one can call the function flint_set_num_threads(n), where n is the maximum number of threads to use.

One can also query the current thread limit by calling flint_get_num_threads().

Each version of FLINT brings new functions that are threaded by default.
Many core algorithms such as the FFT (for large integer and polynomial operations, including some factoring algorithms), integer factoring and multivariate polynomial algorithms are threaded in FLINT.

### 1.9.2 Writing threaded functions in FLINT

Flint uses a custom thread pool for threading. This involves creating a worker function, requesting threads from the thread pool, starting the threads, waiting for them to finish, then giving the threads back to the pool. Simple examples of this include `fmpz_mod_mat_mul_classical_threaded` and `fmpz_poly_taylor_shift_multi_mod`.

The user should not have to run specialised versions of functions to get threading. This means that user facing functions should generally not have `_threaded` appended to their name. Either there is a single function that does the job, and it happens to be threaded, or there is a best-of-breed function that calls the appropriate threaded function when this is the best strategy.

There are some instances where it may be desirable (e.g. for testing purposes, or because naming proves difficult) where one wants a `_threaded` in the name. But these cases should be rare.

In some cases, one does not want functions to request threads from the pool themselves, but to accept threads from another function which has already obtained them. Such functions will accept an array of thread pool handles and a number of threads. The naming convention for such functions is to append `_threaded_pool` to the end of their name. However, the usual distinctions between underscore and non-underscore functions should still apply.

Functions should request `flint_get_num_threads()` threads from the thread pool. The function should not exceed this number of threads in total. In general a thread that is woken should start zero additional workers. However, if this is not the desired behaviour, an option exists to the function for waking worker threads to alter how many threads it can start. In some cases it is also necessary to temporarily restrict the number of worker threads a given function can start. This is accomplished by calling `flint_set_num_workers()` and then once the function is called, calling `flint_reset_num_workers()`. Any threaded function which calls `flint_get_num_threads()` to determine how many threads to request from the thread pool will be appropriately restricted by such calls.

Note that if `flint_get_num_threads()` returns n then the number of workers that can be started is n - 1 (in addition to the thread the function is already running in). For this reason our documentation often distinguishes number of workers and number of threads. Please refer to the thread pool interface and Flint threading interface documentation to see the exact specification.

### 1.9.3 Functional parallel programming helpers

The following convenience function are defined in `thread_support.h`. They are currently experimental, and the interfaces might change in a future version.

```c
slong flint_get_num_available_threads()
```

Returns the number of threads that are not currently in use.

typedef void (*do_func_t)(slong i, void *args)

```c
void flint_parallel_do(do_func_t f, void *args, slong n, int thread_limit, int flags)
```

Evaluate `f(i, args)` for `0 \leq i < n - 1` in parallel using up to `thread_limits` threads (including the master thread). If `thread_limit` is nonpositive, the number of threads defaults to `flint_get_num_threads()`.

The following flags are supported:

- `FLINT_PARALLEL_UNIFORM` - assumes that the cost of function calls is roughly constant, so that scheduling uniformly into blocks is efficient.
- `FLINT_PARALLEL_STRIDED` - assumes that the cost increases or decreases monotonically with `i`, so that strided scheduling is efficient.
- `FLINT_PARALLEL_DYNAMIC` (not implemented) - use dynamic scheduling.
FLINT_PARALLEL_VERBOSE - print information.

typedef void (*bsplit_merge_func_t)(void*, void*, void*, void*)
typedef void (*bsplit_basecase_func_t)(void*, slong, slong, void*)
typedef void (*bsplit_init_func_t)(void*, void*)
typedef void (*bsplit_clear_func_t)(void*, void*)

void flint_parallel_binary_splitting(void *res, bsplit_basecase_func_t basecase,
                                      bsplit_merge_func_t merge, size_t sizeof_res,
                                      bsplit_init_func_t init, bsplit_clear_func_t clear, void
                                      *args, slong a, slong b, slong basecase_cutoff, int
                                      thread_limit, int flags)

Sets res to \( f(a) \circ f(a+1) \circ \cdots \circ f(b-1) \) computed using parallel binary splitting, using up to thread limits threads (including the master thread). If thread_limit is nonpositive, the number of threads defaults to flint_get_num_threads().

The function basecase(res, a, b, args) gets called when \( b - a \) does not exceed basecase_cutoff, which must be at least 1.

The function merge(res, x, y, args) implements the associative operation \( (x \circ y) \), writing the result to res. If called with FLINT_PARALLEL_B_SPLIT_LEFT_INPLACE in flags, the same space will be used for res and x.

A result is assumed to fit in a structure of size sizeof_res. The functions init(res, args) and clear(res, args) initialize and clear intermediate result objects.
2.1 flint.h – global definitions

2.1.1 Macros

The file `flint.h` contains various useful macros.

The macro constant `FLINT_BITS` is set at compile time to be the number of bits per limb on the machine. FLINT requires it to be either 32 or 64 bits. Other architectures are not currently supported.

The macro constant `FLINT_D_BITS` is set at compile time to be the number of bits per double on the machine or one less than the number of bits per limb, whichever is smaller. This will have the value 53 or 31 on currently supported architectures. Numerous internal functions using precomputed inverses only support operands up to `FLINT_D_BITS` bits, hence the macro.

The macro `FLINT_ABS(x)` returns the absolute value of `x` for primitive signed numerical types. It might fail for least negative values such as `INT_MIN` and `WORD_MIN`.

The macro `FLINT_MIN(x, y)` returns the minimum of `x` and `y` for primitive signed or unsigned numerical types. This macro is only safe to use when `x` and `y` are of the same type, to avoid problems with integer promotion.

Similar to the previous macro, `FLINT_MAX(x, y)` returns the maximum of `x` and `y`.

The macro `FLINT_SWAP(T, x, y)` swaps `x` and `y`, where `x` and `y` are of type `T`. For instance, with `x` and `y` of type `fmpz_poly_t`, one can write `FLINT_SWAP(fmpz_poly_struct, *x, *y)` to swap the content of `x` with the content of `y`.

`FLINT_SGN(x)`

Returns the sign of `x` where `x` is interpreted as a `slong`, that is, returns −1 if `x < 0`, 0 if `x = 0` and 1 if `x > 0`.

`mp_limb_t FLINT_BIT_COUNT(mp_limb_t x)`

Returns the number of binary bits required to represent an `ulong x`. If `x` is zero, returns 0.

Derived from this there are the two macros `FLINT_FLOG2(x)` and `FLINT_CLOG2(x)` which, for any `x ≥ 1`, compute ⌊log₂ `x`⌋ and ⌈log₂ `x`⌉.

To determine the current FLINT version a number of macros are available. For example, if the current FLINT version is 2.4.0 then `__FLINT_VERSION` will have the value 2, `__FLINT_MINOR` will have the value 4 and `__FLINT_PATCHLEVEL` will have the value 0.

The `__FLINT_RELEASE` macro gives a single number representing the FLINT version. For example, it will have the value 20400 for version 2.4.0.

The `FLINT_VERSION` macro is a static text string giving the version number, e.g. “2.4” or “2.4.1”. Note that if the final digit is a zero it is suppressed.
2.1.2 Integer types

The char, short and int types are assumed to be two’s complement types with exactly 8, 16 and 32 bits. This is not technically guaranteed by the C standard, but it is true on mainstream platforms.

Since the C types long and unsigned long do not have a standardized size in practice, FLINT defines slong and ulong types which are guaranteed to be 32 bits on a 32-bit system and 64 bits on a 64-bit system. They are also guaranteed to have the same size as GMP’s mp_limb_t. GMP builds with a different limb size configuration are not supported at all. For convenience, the macro FLINT_BITS specifies the word length (32 or 64) of the system.

type slong

The slong type is used for precisions, bit counts, loop indices, array sizes, and the like, even when those values are known to be nonnegative. It is also used for small integer-valued coefficients. In method names, an slong parameter is denoted by si, for example arb_add_si()

The constants WORD_MIN and WORD_MAX give the range of this type. This type can be printed with flint_printf using the format string %wd.

type ulong

The ulong type is used for integer-valued coefficients that are known to be unsigned, and for values that require the full 32-bit or 64-bit range. In method names, a ulong parameter is denoted by ui, for example arb_add_ui().

The constant UWORD_MAX gives the range of this type. This type can be printed with flint_printf using the format string %wu.

The following GMP-defined types are used in methods that manipulate the internal representation of numbers (using limb arrays).

type mp_limb_t

A single limb.

type mp_ptr

Pointer to a writable array of limbs.

type mp_srcptr

Pointer to a read-only array of limbs.

type mp_size_t

A limb count (always nonnegative).

type flint_bitcnt_t

A bit offset within an array of limbs (always nonnegative).

2.1.3 Allocation Functions

void *flint_malloc(size_t size)

Allocate size bytes of memory.

void *flint_realloc(void *ptr, size_t size)

Realocate an area of memory previously allocated by flint_malloc(), flint_realloc(), or flint_calloc().

void *flintcalloc(size_t num, size_t size)

Allocate num objects of size bytes each, and zero the allocated memory.

void flint_free(void *ptr)

Free a section of memory allocated by flint_malloc(), flint_realloc(), or flint_calloc().
2.1.4 Random Numbers

type **flint_rand_s**
A structure holding the state of a flint pseudo random number generator.

type **flint_rand_t**
An array of length 1 of **flint_rand_s**.

**flint_rand_s** *flint_rand_alloc**(void)
Allocates a **flint_rand_t** object to be used like a heap-allocated **flint_rand_t** in external libraries. The random state is not initialised.

void **flint_rand_free**( **flint_rand_s** *state)
Frees a random state object as allocated using **flint_rand_alloc()**.

void **flint_randinit**( **flint_rand_t** state)
Initialize a **flint_rand_t**.

void **flint_randclear**( **flint_rand_t** state)
Free all memory allocated by **flint_rand_init()**.

2.1.5 Thread functions

void **flint_set_num_threads**(int num_threads)
Set up a thread pool of **num_threads** - 1 worker threads (in addition to the master thread) and set the maximum number of worker threads the master thread can start to **num_threads** - 1.

This function may only be called globally from the master thread. It can also be called at a global level to change the size of the thread pool, but an exception is raised if the thread pool is in use (threads have been woken but not given back). The function cannot be called from inside worker threads.

int **flint_get_num_threads**(void)
When called at the global level, this function returns one more than the number of worker threads in the Flint thread pool, i.e. it returns the number of workers in the thread pool plus one for the master thread.

In general, this function returns one more than the number of additional worker threads that can be started by the current thread.

Use **thread_pool_wake()** to set this number for a given worker thread.

See also: **flint_get_num_available_threads()**.

int **flint_set_num_workers**(int num_workers)
Restricts the number of worker threads that can be started by the current thread to **num_workers**. This function can be called from any thread.

Assumes that the Flint thread pool is already set up.

The function returns the old number of worker threads that can be started.

The function can only be used to reduce the number of workers that can be started from a thread. It cannot be used to increase the number. If a higher number is passed, the function has no effect.

The number of workers must be restored to the original value by a call to **flint_reset_num_workers()** before the thread is returned to the thread pool.

The main use of this function and **flint_reset_num_workers()** is to cheaply and temporarily restrict the number of workers that can be started, e.g. by a function that one wishes to call from a thread, and cheaply restore the number of workers to its original value before exiting the current thread.
void flint_reset_num_workers(int num_workers)

    After a call to flint_set_num_workers() this function must be called to set the number of workers
    that may be started by the current thread back to its original value.

2.1.6 Input/Output

int flint_printf(const char *format, ...)
int flint_fprintf(FILE *fs, const char *format, ...)
int flint_vprintf(const char *format, va_list vlist)
int flint_vfprintf(FILE *fs, const char *format, va_list vlist)

    These functions are extensions of the C standard library functions printf, fprintf, vprintf, and
    vfprintf.

    The first extension is the addition of the length modifier w, used for printing the types ulong,
    slong and mp_limb_t. As these types are either defined as signed and unsigned long int or long
    long int, this comes in handy. Just like long int and long long int, the conversion format
    specifier are allowed to be d, i, o, x, X and u.

    The second and final extension is printing of FLINT types. Currently supported types are the base
    types ulong, slong, fmpz_t, fmpq_t, mag_t, arf_t, arb_t and acb_t as well as the context
    structures for modulo arithmetic nmod_t and fmpz_mod_ctx_t. We also support the GMP types
    mpz_t and mpq_t.

    We currently support printing vectors of pointers to the following base types: slong, ulong, fmpz,
    fmpq, mag_struct, arf_struct, arb_struct and acb_struct.

    We also support printing matrices of the following types: nmod_mat_t, fmpz_mat_t, fmpq_mat_t,
    arb_mat_t and acb_mat_t.

    Finally, we currently support printing polynomial of the following types: nmod_poly_t,
    fmpz_poly_t, fmpq_poly_t, arb_poly_t and acb_poly_t.

ulong bulong;
slong bslong;
fmpz_t bfmpz;
fmpq_t bfmpq;
mag_t bmag;
arf_t barf;
arb_t barb;
acb_t bacb;
nmod_t bnmod;
fmpz_mod_ctx_t bfmpz_mod_ctx;
mpz_t bmpz;
mpq_t bmpq;

    /* Initialize and set variables */

flint_printf(
    "ulong: %{ulong}\n"
    "slong: %{slong}\n"
    "fmpz: %{fmpz}\n"
    "fmpq: %{fmpq}\n"
    "mag: %{mag}\n"
    "arf: %{arf}\n"
    "arb: %{arb}\n"
    "acb: %{acb}\n"
    "nmod: %{nmod}\n"
    "fmpz_mod_ctx: %{fmpz_mod_ctx}\n"
slong * vslong; slong vslong_len;
mp_ptr vnmod; slong vnmod_len; /* The base type for nmod is ulong */
fmpz * vfmpz; slong vfmpz_len;
/* fmpz_mod vectors are given by the type `fmpz *` */
fmpq * vfmpq; slong vfmpq_len;
mag_ptr vmag; slong vmag_len;
arf_ptr varf; slong varf_len;
arb_ptr varb; slong varb_len;
acb_ptr vacb; slong vacb_len;
/* Initialize and set variables */
flint_printf(
    "slong vector: %{slong*}\n"
    "nmod vector: %{ulong*}\n"
    "fmpz vector: %{fmpz*}\n"
    "fmpq vector: %{fmpq*}\n"
    "mag vector: %{mag*}\n"
    "arf vector: %{arf*}\n"
    "arb vector: %{arb*}\n"
    "acb vector: %{acb*}\n"
    vslong, vslong_len, /* They require a vector length specifier */
    vnmod, vnmod_len,
    vfmpz, vfmpz_len,
    vfmpq, vfmpq_len,
    vmag, vmag_len,
    varf, varf_len,
    varb, varb_len,
    vacb, vacb_len);

nmod_mat_t mnmod;
fmmpz_mat_t mfmpz;
fmpz_mod_mat_t mfmpz_mod;
fmpq_mat_t mfmpq;
arb_mat_t marb;
acb_mat_t macb;
/* Initialize and set variables */
flint_printf(


Note: Printing of FLINT types does not currently support any flags.

Note: Any use of %n flags will be invalid, but will not generate any error.

Note: Invalid formats using variable minimum field width and/or precision such as "%* p" may be wrongly parsed, and may result in a different result compared to the C standard library functions.

int flint_sprintf(char *s, const char *str, ...)

This function is an extension of the C standard library functions sprintf. It is currently advised not to use this function as it is currently not coherent with flint_printf().

int flint_scanf(const char *str, ...)
int flint_fscanf(FILE *f, const char *str, ...)
int flint_sscanf(const char *s, const char *str, ...)

These are equivalent to the standard library functions scanf, fscanf, and sscanf with an additional length modifier "w" for reading an mp_limb_t type.
2.1.7 Exceptions

void flint_abort(void)
   FLINT version of the C standard function abort.

void flint_set_abort(void (*func)(void))
   Sets the flint_abort() function to call func instead of abort.

enum flint_err_t
   An error code with one of the following values
   
   FLINT_ERROR
      Describes a generic error.
   
   FLINT_OVERFLOW
      Describes an overflow.
   
   FLINT_IMPINV
      Describes an impossible inversion.
   
   FLINT_DOMERR
      Describes a domain error.
   
   FLINT_DIVZERO
      Describes a division by zero.
   
   FLINT_EXPOF
      Describes an exponent overflow.
   
   FLINT_INEXACT
      Describes an inexact operation.
   
   FLINT_TEST_FAIL
      Describes a test fail.

void flint_throw(flint_err_t exc, const char *msg, ...)
   Throws an error of type exc with message msg and aborts via flint_abort(). The printing
   back-end function is flint_fprintf(), and so it allows for printing of FLINT types as well.

2.2 profiler.h – performance profiling

2.2.1 Timer based on the cycle counter

void timeit_start(timeit_t t)
void timeit_stop(timeit_t t)
   Gives wall and user time - useful for parallel programming.
   
   Example usage:
   
   timeit_t t0;
   // ...
   timeit_start(t0);
   // do stuff, take some time
   timeit_stop(t0);
### 2.2.2 Framework for repeatedly sampling a single target

void **prof_repeat**(double *min, double *max, profile_target_t target, void *arg)

Allows one to automatically time a given function. Here is a sample usage:

Suppose one has a function one wishes to profile:

```c
void myfunc(ulong a, ulong b);
```

One creates a struct for passing arguments to our function:

```c
typedef struct {
    ulong a, b;
} myfunc_t;
```

A sample function:

```c
void sample_myfunc(void * arg, ulong count) {
    myfunc_t * params = (myfunc_t *) arg;
    ulong a = params->a;
    ulong b = params->b;
    for (ulong i = 0; i < count; i++) {
```
Then we do the profile:

```c
double min, max;
myfunc_t params;
params.a = 3;
params.b = 4;
prof_repeat(&min, &max, sample_myfunc, &params);
flint_printf("Min time is %lf, max time is %lf
", min, max);
```

If either of the first two parameters to `prof_repeat` is `NULL`, that value is not stored.

One may set the minimum time in microseconds for a timing run by adjusting `DURATION_THRESHOLD` and one may set a target duration in microseconds by adjusting `DURATION_TARGET` in `profiler.h`.

### 2.2.3 Memory usage

```c
void get_memory_usage(meminfo_t meminfo)
```

Obtains information about the memory usage of the current process. The `meminfo` object contains the slots `size` (virtual memory size), `peak` (peak virtual memory size), `rss` (resident set size), `hwm` (peak resident set size). The values are stored in kilobytes (1024 bytes). This function currently only works on Linux.

### 2.2.4 Simple profiling macros

- **TIMEIT_REPEAT** (timer, reps)
- **TIMEIT_END_REPEAT** (timer, reps)
  Repeatedly runs the code between the `TIMEIT_REPEAT` and the `TIMEIT_END_REPEAT` markers, automatically increasing the number of repetitions until the elapsed time exceeds the timer resolution. The macro takes as input a predefined `timeit_t` object and an integer variable to hold the number of repetitions.

- **TIMEIT_START**
- **TIMEIT_STOP**
  Repeatedly runs the code between the `TIMEIT_START` and the `TIMEIT_STOP` markers, automatically increasing the number of repetitions until the elapsed time exceeds the timer resolution, and then prints the average elapsed cpu and wall time for a single repetition.

- **TIMEIT_ONCE_START**
- **TIMEIT_ONCE_STOP**
  Runs the code between the `TIMEIT_ONCE_START` and the `TIMEIT_ONCE_STOP` markers exactly once and then prints the elapsed cpu and wall time. This does not give a precise measurement if the elapsed time is short compared to the timer resolution.
SHOW_MEMORY_USAGE
Retrieves memory usage information via get_memory_usage and prints the results.

2.3 thread_pool.h – thread pool

2.3.1 Thread pool
type thread_pool_t
This is a thread pool.
type thread_pool_handle
This is a handle to a thread in a thread pool.

void thread_pool_init(thread_pool_t T, slong size)
Initialise T and create size sleeping threads that are available to work. If size ≤ 0 no threads are created and future calls to thread_pool_request() will return 0 (unless thread_pool_set_size() has been called).

slong thread_pool_get_size(thread_pool_t T)
Return the number of threads in T.

int thread_pool_set_size(thread_pool_t T, slong new_size)
If all threads in T are in the available state, resize T and return 1. Otherwise, return 0.

slong thread_pool_request(thread_pool_t T, thread_pool_handle *out, slong requested)
Put at most requested threads in the unavailable state and return their handles. The handles are written to out and the number of handles written is returned. These threads must be released by a call to thread_pool_give_back.

void thread_pool_wake(thread_pool_t T, thread_pool_handle i, int max_workers, void (*f)(void*), void *a)
Wake up a sleeping thread i and have it work on f(a). The thread being woken will be allowed to start max_workers additional worker threads. Usually this value should be set to 0.

void thread_pool_wait(thread_pool_t T, thread_pool_handle i)
Wait for thread i to finish working and go back to sleep.

void thread_pool_give_back(thread_pool_t T, thread_pool_handle i)
Put thread i back in the available state. This thread should be sleeping when this function is called.

void thread_pool_clear(thread_pool_t T)
Release any resources used by T. All threads should be given back before this function is called.

2.4 mpoly.h – support functions for multivariate polynomials

An array of type ulong * or fmpz ** is used to communicate exponent vectors. These exponent vectors must have length equal to the number of variables in the polynomial ring. The element of this exponent vector at index 0 corresponds to the most significant variable in the monomial ordering. For example, if the polynomial is \( 7 \cdot x^2 \cdot y + 8 \cdot y \cdot z + 9 \) and the variables are ordered so that \( x > y > z \), the degree function will return \( \{2, 1, 1\} \). Similarly, the exponent vector of the 0-index term of this polynomial is \( \{2, 1, 0\} \), while the 2-index term has exponent vector \( \{0, 0, 0\} \) and coefficient 9.
2.4.1 Orderings

**type** ordering_t

An enumeration of supported term orderings. Currently one of ORD_LEX, ORD_DEGLEX or ORD_DEGREVLEX.

**type** mpoly_ctx_struct

**type** mpoly_ctx_t

An mpoly_ctx_struct is a structure holding information about the number of variables and the term ordering of a multivariate polynomial.

**void** mpoly_ctx_init(mpoly_ctx_t ctx, slong nvars, const ordering_t ord)

Initialize a context for specified number of variables and ordering.

**void** mpoly_ctx_clear(mpoly_ctx_t mctx)

Clean up any space used by a context object.

ordering_t mpoly_ordering_randtest(flint_rand_t state)

Return a random ordering. The possibilities are ORD_LEX, ORD_DEGLEX and ORD_DEGREVLEX.

**void** mpoly_ctx_init_rand(mpoly_ctx_t mctx, flint_rand_t state, slong max_nvars)

Initialize a context with a random choice for the ordering.

int mpoly_ordering_isdeg(const mpoly_ctx_t ctx)

Return 1 if the ordering of the given context is a degree ordering (deglex or degrevlex).

int mpoly_ordering_isrev(const mpoly_ctx_t cth)

Return 1 if the ordering of the given context is a reverse ordering (currently only degrevlex).

**void** mpoly_ordering_print(ordering_t ord)

Print a string (either “lex”, “deglex” or “degrevlex”) to standard output, corresponding to the given ordering.

2.4.2 Monomial arithmetic

These functions in this section are only provided as inline functions as they are somewhat trivial. This is in order to minimize the FLINT binary.

**void** mpoly_monomial_add(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N)

Set (exp_ptr, N) to the sum of the monomials (exp2, N) and (exp3, N), assuming bits <= FLINT_BITS.

**void** mpoly_monomial_add_mp(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N)

Set (exp_ptr, N) to the sum of the monomials (exp2, N) and (exp3, N).

**void** mpoly_monomial_sub(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N)

Set (exp_ptr, N) to the difference of the monomials (exp2, N) and (exp3, N), assuming bits <= FLINT_BITS

**void** mpoly_monomial_sub_mp(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N)

Set (exp_ptr, N) to the difference of the monomials (exp2, N) and (exp3, N).

**int** mpoly_monomial_overflows(ulong *exp2, slong N, ulong mask)

Return true if any of the fields of the given monomial (exp2, N) has overflowed (or is negative). The mask is a word with the high bit of each field set to 1. In other words, the function returns 1 if any word of exp2 has any of the nonzero bits in mask set. Assumes that bits <= FLINT_BITS.

**int** mpoly_monomial_overflows_mp(ulong *exp_ptr, slong N, flint_bitcnt_t bits)

Return true if any of the fields of the given monomial (exp_ptr, N) has overflowed. Assumes that bits >= FLINT_BITS.
int mpolynomial_overflows1(ulong exp, ulong mask)
    As per mpolynomial_overflows with N = 1.

void mpolynomial_set(ulong *exp2, const ulong *exp3, slong N)
    Set the monomial (exp2, N) to (exp3, N).

void mpolynomial_swap(ulong *exp2, ulong *exp3, slong N)
    Swap the words in (exp2, N) and (exp3, N).

void mpolynomial_mul_ui(ulong *exp2, const ulong *exp3, slong N, ulong c)
    Set the words of (exp2, N) to the words of (exp3, N) multiplied by c.

2.4.3 Monomial comparison

These functions in this section are only provided as inline functions as they are somewhat trivial. This is in order to minimize the FLINT binary.

int mpolynomial_is_zero(const ulong *exp, slong N)
    Return 1 if (exp, N) is zero.

int mpolynomial_equal(const ulong *exp2, const ulong *exp3, slong N)
    Return 1 if the monomials (exp2, N) and (exp3, N) are equal.

void mpolynomial_get_cmpmask(ulong *cmpmask, slong N, ulong bits, const mpolynomial_ctx_t mctx)
    Get the mask (cmpmask, N) for comparisons. bits should be set to the number of bits in the exponents to be compared. Any function that compares monomials should use this comparison mask.

int mpolynomial_lt(const ulong *exp2, const ulong *exp3, slong N, const ulong *cmpmask)
    Return 1 if (exp2, N) is less than (exp3, N).

int mpolynomial_gt(const ulong *exp2, const ulong *exp3, slong N, const ulong *cmpmask)
    Return 1 if (exp2, N) is greater than (exp3, N).

int mpolynomial_cmp(const ulong *exp2, const ulong *exp3, slong N, const ulong *cmpmask)
    Return 1 if (exp2, N) is greater than, 0 if it is equal to and -1 if it is less than (exp3, N).

2.4.4 Monomial divisibility

These functions in this section are only provided as inline functions as they are somewhat trivial. This is in order to minimize the FLINT binary.

int mpolynomial_divides(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N, ulong mask)
    Return 1 if the monomial (exp2, N) divides (exp3, N) if so set (exp_ptr, N) to the quotient monomial. The mask is a word with the high bit of each bit field set to 1. Assumes that bits <= FLINT_BITS.

int mpolynomial_divides_mp(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N, flint_bitcnt_t bits)
    Return 1 if the monomial (exp2, N) divides (exp3, N). If so set (exp_ptr, N) to the quotient monomial. Assumes that bits >= FLINT_BITS.

int mpolynomial_divides1(ulong *exp_ptr, const ulong exp2, const ulong exp3, ulong mask)
    As per mpolynomial_divides with N = 1.

int mpolynomial_divides_tight(slong e1, slong e2, slong *prods, slong num)
    Return 1 if the monomial e2 divides the monomial e1, where the monomials are stored using factorial representation. The array (prods, num) should consist of 1, b1, b1 x b2, ..., where the bi are the bases of the factorial number representation.
2.4.5 Basic manipulation

\texttt{flint\_bitcnt\_t mpoly\_exp\_bits\_required\_ui} (const \texttt{ulong *}user\_exp, const \texttt{mpoly\_ctx\_t} mctx)

Returns the number of bits required to store \texttt{user\_exp} in packed format. The returned number of bits includes space for a zeroed signed bit.

\texttt{flint\_bitcnt\_t mpoly\_exp\_bits\_required\_ffmpz} (const \texttt{fmpz *}user\_exp, const \texttt{mpoly\_ctx\_t} mctx)

Returns the number of bits required to store \texttt{user\_exp} in packed format. The returned number of bits includes space for a zeroed signed bit.

\texttt{flint\_bitcnt\_t mpoly\_exp\_bits\_required\_pfmpz} (const \texttt{fmpz *}const *user\_exp, const \texttt{mpoly\_ctx\_t} mctx)

Returns the number of bits required to store \texttt{user\_exp} in packed format. The returned number of bits includes space for a zeroed signed bit.

\texttt{void mpoly\_max\_fields\_ui\_sp} (\texttt{ulong *}max\_fields, const \texttt{ulong *}poly\_exps, \texttt{slong} len, \texttt{ulong} bits, const \texttt{mpoly\_ctx\_t} mctx)

Compute the field-wise maximum of packed exponents from \texttt{poly\_exps} of length \texttt{len} and unpack the result into \texttt{max\_fields}. The maximums are assumed to fit a ulong.

\texttt{void mpoly\_max\_fields\_fmpz} (\texttt{fmpz *}max\_fields, const \texttt{ulong *}poly\_exps, \texttt{slong} len, \texttt{ ulong} bits, const \texttt{mpoly\_ctx\_t} mctx)

Compute the field-wise maximum of packed exponents from \texttt{poly\_exps} of length \texttt{len} and unpack the result into \texttt{max\_fields}.

\texttt{void mpoly\_max\_degrees\_tight} (\texttt{slong *}max\_exp, \texttt{ulong *}exps, \texttt{slong} len, \texttt{ulong} *prods, \texttt{slong} num)

Return an array of \texttt{num} integers corresponding to the maximum degrees of the exponents in the array of exponent vectors (\texttt{exps}, \texttt{len}), assuming that the exponent are packed in a factorial representation. The array (\texttt{prods}, \texttt{num}) should consist of 1, \(b_1\), \(b_1 \times b_2\),..., where the \(b_i\) are the bases of the factorial number representation. The results are stored in the array \texttt{max\_exp}, with the entry corresponding to the most significant base of the factorial representation first in the array.

\texttt{int mpoly\_monomial\_exists} (\texttt{slong *}index, const \texttt{ulong *}poly\_exps, const \texttt{ulong} *exp, \texttt{slong} len, \texttt{ulong} *cmpmask)

Returns true if the given exponent vector \texttt{exp} exists in the array of exponent vectors (\texttt{poly\_exps}, \texttt{len}), otherwise, returns false. If the exponent vector is found, its index into the array of exponent vectors is returned. Otherwise, \texttt{index} is set to the index where this exponent could be inserted to preserve the ordering. The index can be in the range \([0, \texttt{len}]\).

\texttt{void mpoly\_search\_monomials} (\texttt{slong **e\_ind}, \texttt{ulong} *e, \texttt{slong} *e\_score, \texttt{slong} *t1, \texttt{slong} *t2, \texttt{slong} *t3, \texttt{slong} lower, \texttt{slong} upper, const \texttt{ulong} *a, \texttt{slong} a\_len, const \texttt{ulong} *b, \texttt{slong} b\_len, \texttt{slong} N, const \texttt{ulong} *cmpmask)

Given packed exponent vectors \texttt{a} and \texttt{b}, compute a packed exponent \texttt{e} such that the number of monomials in the cross product \(a \times b\) that are less than or equal to \texttt{e} is between \texttt{lower} and \texttt{upper}. This number is stored in \texttt{e\_store}. If no such monomial exists, one is chosen so that the number of monomials is as close as possible. This function assumes that 1 is the smallest monomial and needs three arrays \(t_1, t_2,\) and \(t_3\) of the size as \(a\) for workspace. The parameter \texttt{e\_ind} is set to one of \(t_1, t_2,\) and \(t_3\) and gives the locations of the monomials in \(a \times b\).

2.4.6 Setting and getting monomials

\texttt{int mpoly\_term\_exp\_fits\_ui} (\texttt{ulong *}exps, \texttt{ulong} bits, \texttt{slong} n, const \texttt{mpoly\_ctx\_t} mctx)

Return whether every entry of the exponent vector of index \(n\) in \texttt{exps} fits into a \texttt{ulong}.

\texttt{int mpoly\_term\_exp\_fits\_si} (\texttt{ulong *}exps, \texttt{ulong} bits, \texttt{slong} n, const \texttt{mpoly\_ctx\_t} mctx)

Return whether every entry of the exponent vector of index \(n\) in \texttt{exps} fits into a \texttt{slong}.
void mpoly_get_monomial_ui(ulong *exps, const ulong *poly_exps, ulong bits, const mpoly_ctx_t mctx)
    Convert the packed exponent poly_exps of bit count bits to a monomial from the user’s perspective. The exponents are assumed to fit a ulong.

void mpoly_get_monomial_ffmpz(fmpz *exps, const ulong *poly_exps, flint_bitcnt_t bits, const mpoly_ctx_t mctx)
    Convert the packed exponent poly_exps of bit count bits to a monomial from the user’s perspective.

void mpoly_get_monomial_pfmpz(fmpz **exps, const ulong *poly_exps, flint_bitcnt_t bits, const mpoly_ctx_t mctx)
    Convert the packed exponent poly_exps of bit count bits to a monomial from the user’s perspective.

void mpoly_set_monomial_ui(ulong *exp1, const ulong *exp2, ulong bits, const mpoly_ctx_t mctx)
    Convert the user monomial exp2 to packed format using bits.

void mpoly_set_monomial_ffmpz(ulong *exp1, const fmpz *exp2, flint_bitcnt_t bits, const mpoly_ctx_t mctx)
    Convert the user monomial exp2 to packed format using bits.

void mpoly_set_monomial_pfmpz(ulong *exp1, fmpz *const *exp2, flint_bitcnt_t bits, const mpoly_ctx_t mctx)
    Convert the user monomial exp2 to packed format using bits.

2.4.7 Packing and unpacking monomials

void mpoly_pack_vec_ui(ulong *exp1, const ulong *exp2, ulong bits, slong nfields, slong len)
    Packs a vector exp2 into {exp1} using a bit count of bits. No checking is done to ensure that the vector actually fits into bits bits. The number of fields in each vector is nfields and the total number of vectors to unpack is len.

void mpoly_pack_vec_fmpz(ulong *exp1, const fmpz *exp2, flint_bitcnt_t bits, slong nfields, slong len)
    Packs a vector exp2 into {exp1} using a bit count of bits. No checking is done to ensure that the vector actually fits into bits bits. The number of fields in each vector is nfields and the total number of vectors to unpack is len.

void mpoly_unpack_vec_ui(ulong *exp1, const ulong *exp2, ulong bits, slong nfields, slong len)
    Unpacks vector exp2 of bit count bits into exp1. The number of fields in each vector is nfields and the total number of vectors to unpack is len.

void mpoly_unpack_vec_fmpz(fmpz *exp1, const ulong *exp2, flint_bitcnt_t bits, slong nfields, slong len)
    Unpacks vector exp2 of bit count bits into exp1. The number of fields in each vector is nfields and the total number of vectors to unpack is len.

int mpoly_repack_monomials(ulong *exps1, ulong bits1, const ulong *exps2, ulong bits2, slong len, const mpoly_ctx_t mctx)
    Convert an array of length len of exponents exps2 packed using bits bits2 into an array exps1 using bits bits1. No checking is done to ensure that the result fits into bits bits1.

void mpoly_pack_monomials_tight(ulong *exp1, const ulong *exp2, slong len, const ulong *mults, slong num, slong len)
    Given an array of possibly packed exponent vectors exp2 of length len, where each field of each exponent vector is packed into the given number of bits, return the corresponding array of monomial vectors packed using a factorial numbering scheme. The “bases” for the factorial numbering scheme are given as an array of integers mults, the first entry of which corresponds to the field of least
significance in each input exponent vector. Obviously the maximum exponent to be packed must be less than the corresponding base in \textit{mults}.

The number of multipliers is given by \textit{num}. The code only considers least significant \textit{num} fields of each exponent vectors and ignores the rest. The number of ignored fields should be passed in \textit{extras}.

\begin{verbatim}
void mpoly_unpack_monomials_tight(ulong *e1, ulong *e2, slong len, slong *mults, slong num, slong bits)
\end{verbatim}

Given an array of exponent vectors \textit{e2} of length \textit{len} packed using a factorial numbering scheme, unpack the monomials into an array \textit{e1} of exponent vectors in standard packed format, where each field has the given number of bits. The “bases” for the factorial numbering scheme are given as an array of integers \textit{mults}, the first entry of which corresponds to the field of least significance in each exponent vector.

The number of multipliers is given by \textit{num}. The code only considers least significant \textit{num} fields of each exponent vectors and ignores the rest. The number of ignored fields should be passed in \textit{extras}.

### 2.4.8 Chunking

\begin{verbatim}
void mpoly_main_variable_terms1(slong *i1, slong *n1, const ulong *exp1, slong l1, slong len1, slong k, slong num, slong bits)
\end{verbatim}

Given an array of exponent vectors \textit{(exp1, len1)}, each exponent vector taking one word of space, with each exponent being packed into the given number of bits, compute \textit{l1} starting offsets \textit{i1} and lengths \textit{n1} (which may be zero) to break the exponents into chunks. Each chunk consists of exponents have the same degree in the main variable. The index of the main variable is given by \textit{k}. The variables are indexed from the variable of least significance, starting from 0. The value \textit{l1} should be the degree in the main variable, plus one.

### 2.4.9 Chained heap functions

\begin{verbatim}
int _mpoly_heap_insert(mpoly_heap_s *heap, ulong *exp, void *x, slong *next_loc, slong *heap_len, slong N, const ulong *cmpmask)
\end{verbatim}

Given a heap, insert a new node \textit{x} corresponding to the given exponent into the heap. Heap elements are ordered by the exponent \textit{(exp, N)}, with the largest element at the head of the heap. A pointer to the current heap length must be passed in via \textit{heap_len}. This will be updated by the function. Note that the index 0 position in the heap is not used, so the length is always one greater than the number of elements.

\begin{verbatim}
void _mpoly_heap_insert1(mpoly_heap1_s *heap, ulong exp, void *x, slong *next_loc, slong *heap_len, ulong maskhi)
\end{verbatim}

As per \_mpoly_heap_insert except that \textit{N} = 1, and \textit{maskhi} = \textit{cmpmask}[0].

\begin{verbatim}
void * _mpoly_heap_pop(mpoly_heap_s *heap, slong *heap_len, slong N, const ulong *cmpmask)
\end{verbatim}

Pop the head of the heap. It is cast to a \textit{void *}. A pointer to the current heap length must be passed in via \textit{heap_len}. This will be updated by the function. Note that the index 0 position in the heap is not used, so the length is always one greater than the number of elements. The \textit{maskhi} and \textit{masklo} values are zero except for degrevlex ordering, where they are as per the monomial comparison operations above.

\begin{verbatim}
void * _mpoly_heap_pop1(mpoly_heap1_s *heap, slong *heap_len, ulong maskhi)
\end{verbatim}

As per \_mpoly_heap_pop1 except that \textit{N} = 1, and \textit{maskhi} = \textit{cmpmask}[0].
2.5 machine_vectors.h – SIMD-accelerated operations on fixed-length vectors

This module currently requires building FLINT with support for AVX2 or NEON instructions. Some functions may require that vectors are aligned in memory.

2.5.1 Types

- type **vec1n**
- type **vec2n**
- type **vec4n**
- type **vec8n**
  Vector with 1, 2, 4, or 8 **ulong** entries.
- type **vec1d**
- type **vec2d**
- type **vec4d**
- type **vec8d**
  Vector with 1, 2, 4, or 8 **double** entries.

2.5.2 Printing

- void **vec4d_print(vec4d a)**
- void **vec4n_print(vec4n a)**

2.5.3 Access and conversions

- **vec1d vec1d_load(const double *a)**
- **vec4d vec4d_load(const double *a)**
- **vec8d vec8d_load(const double *a)**
- **vec1d vec1d_load_aligned(const double *a)**
- **vec4d vec4d_load_aligned(const double *a)**
- **vec8d vec8d_load_aligned(const double *a)**
- **vec1d vec1d_load_unaligned(const double *a)**
- **vec4d vec4d_load_unaligned(const double *a)**
- **vec8d vec8d_load_unaligned(const double *a)**
- **vec4n vec4n_load_unaligned(const ulong *a)**
- **vec8n vec8n_load_unaligned(const ulong *a)**
- void **vec1d_store(double *z, vec1d a)**
- void **vec4d_store(double *z, vec4d a)**
- void **vec8d_store(double *z, vec8d a)**
- void **vec1d_store_aligned(double *z, vec1d a)**
- void **vec4d_store_aligned(double *z, vec4d a)**
void vec8d_store_aligned(double *z, vec8d a)

void vec1d_store_unaligned(double *z, vec1d a)

void vec4d_store_unaligned(double *z, vec4d a)

void vec4n_store_unaligned(ulong *z, vec4n a)

void vec8d_store_unaligned(double *z, vec8d a)

double vec4d_get_index(vec4d a, const int i)

double vec8d_get_index(vec8d a, int i)

    Extract the entry at index \( i \).

vec1d vec1d_set_d(double a)

vec4d vec4d_set_d(double a)

vec4n vec4n_set_n(ulong a)

vec8d vec8d_set_d(double a)

vec8n vec8n_set_n(ulong a)

    Set all entries to the same value.

vec4d vec4d_set_d4(double a0, double a1, double a2, double a3)

vec4n vec4n_set_n4(ulong a0, ulong a1, ulong a2, ulong a3)

vec8d vec8d_set_d8(double a0, double a1, double a2, double a3, double a4, double a5, double a6, double a7)

    Create vector from distinct entries.

vec4n vec4d_convert_limited_vec4n(vec4d a)

vec8d vec8n_convert_limited_vec8d(vec8n a)

### 2.5.4 Permutations

vec4d vec4d_unpacklo(vec4d a, vec4d b)

vec4d vec4d_unpackhi(vec4d a, vec4d b)

vec4d vec4d_permute_0_2_1_3(vec4d a)

vec4d vec4d_permute_3_1_2_0(vec4d a)

vec4d vec4d_permute_3_2_1_0(vec4d a)

vec4d vec4d_permute2_0_2(vec4d a, vec4d b)

vec4d vec4d_permute2_1_3(vec4d a, vec4d b)

vec4d vec4d_unpack_lo_permute_0_2_1_3(vec4d u, vec4d v)

vec4d vec4d_unpack_hi_permute_0_2_1_3(vec4d u, vec4d v)

vec4d vec4d_unpackhi_permute_3_1_2_0(vec4d u, vec4d v)

vec4d vec4d_unpacklo_permute_3_1_2_0(vec4d u, vec4d v)

VECD_TRANSPOSE(z0, z1, z2, z3, a0, a1, a2, a3)

    Sets the rows \( z \) to the transpose of the 4x4 matrix given by rows \( a \).
2.5.5 Comparisons

int vec1d_same(double a, double b)
int vec4d_same(vec4d a, vec4d b)
int vec8d_same(vec8d a, vec8d b)

Check whether the vectors are equal.

vec4d vec4d_cmp_ge(vec4d a, vec4d b)
vec4d vec4d_cmp_gt(vec4d a, vec4d b)

Entrywise comparisons.

2.5.6 Arithmetic and basic operations

vec1d vec1d_round(vec1d a)
vec4d vec4d_round(vec4d a)
vec8d vec8d_round(vec8d a)
vec1d vec1d_zero()
vec4d vec4d_zero()
vec8d vec8d_zero()
vec1d vec1d_one()
vec4d vec4d_one()
vec8d vec8d_one()
vec1d vec1d_add(vec1d a, vec1d b)
vec1d vec1d_sub(vec1d a, vec1d b)
vec4d vec4d_add(vec4d a, vec4d b)
vec4d vec4d_sub(vec4d a, vec4d b)
vec4n vec4n_add(vec4n a, vec4n b)
vec4n vec4n_sub(vec4n a, vec4n b)
vec8d vec8d_add(vec8d a, vec8d b)
vec8d vec8d_sub(vec8d a, vec8d b)
vec1d vec1d_addsub(vec1d a, vec1d b)
vec4d vec4d_addsub(vec4d a, vec4d b)
vec1d vec1d_neg(vec1d a)
vec4d vec4d_neg(vec4d a)
vec8d vec8d_neg(vec8d a)
vec1d vec1d_abs(vec1d a)
vec4d vec4d_abs(vec4d a)
vec1d vec1d_max(vec1d a, vec1d b)
vec1d vec1d_min(vec1d a, vec1d b)
vec4d vec4d_max(vec4d a, vec4d b)
vec4d vec4d_min(vec4d a, vec4d b)
vec8d vec8d_max(vec8d a, vec8d b)
vec8d vec8d_min(vec8d a, vec8d b)
vec1d vec1d_mul(vec1d a, vec1d b)
vec4d vec4d_mul(vec4d a, vec4d b)
vec8d vec8d_mul(vec8d a, vec8d b)
vec1d vec1d_half(vec1d a)
vec4d vec4d_half(vec4d a)
vec1d vec1d_div(vec1d a, vec1d b)
vec4d vec4d_div(vec4d a, vec4d b)
vec8d vec8d_div(vec8d a, vec8d b)
vec1d vec1d_fmadd(vec1d a, vec1d b, vec1d c)
vec4d vec4d_fmadd(vec4d a, vec4d b, vec4d c)
vec8d vec8d_fmadd(vec8d a, vec8d b, vec8d c)
vec1d vec1d_fmsub(vec1d a, vec1d b, vec1d c)
vec4d vec4d_fmsub(vec4d a, vec4d b, vec4d c)
vec8d vec8d_fmsub(vec8d a, vec8d b, vec8d c)
vec1d vec1d_fnmadd(vec1d a, vec1d b, vec1d c)
vec4d vec4d_fnmadd(vec4d a, vec4d b, vec4d c)
vec8d vec8d_fnmadd(vec8d a, vec8d b, vec8d c)
vec1d vec1d_fnmsub(vec1d a, vec1d b, vec1d c)
vec4d vec4d_fnmsub(vec4d a, vec4d b, vec4d c)
vec8d vec8d_fnmsub(vec8d a, vec8d b, vec8d c)
vec1d vec1d_blendv(vec1d a, vec1d b, vec1d c)
vec4d vec4d_blendv(vec4d a, vec4d b, vec4d c)
vec8d vec8d_blendv(vec8d a, vec8d b, vec8d c)
vec4n vec4n_bit_shift_right(vec4n a, ulong b)
vec8n vec8n_bit_shift_right(vec8n a, ulong b)
vec4n vec4n_bit_and(vec4n a, vec4n b)
vec8n vec8n_bit_and(vec8n a, vec8n b)

2.5.7 Modular arithmetic

These functions are used internally by the small-prime FFT. Some double variants assume an odd modulus $n < 2^{50}$. Other assumptions are not yet documented.

int vec1d_same_mod(vec1d a, vec1d b, vec1d n, vec1d ninv)
int vec4d_same_mod(vec4d a, vec4d b, vec4d n, vec4d ninv)

Return whether $a$ and $b$ are the same mod $n$.

vec1d vec1d_reduce_pm1no_to_0n(vec1d a, vec1d n)
vec1d vec4d_reduce_pm1no_to_0n(vec4d a, vec4d n)
vec8d vec8d_reduce_pm1no_to_0n(vec8d a, vec8d n)

Return $a$ mod $n$ reduced to $[0, n)$ assuming $a \in (-n, n)$.

vec1d vec1d_reduce_to_pmin(vec1d a, vec1d n, vec1d ninv)
vec4d vec4d_reduce_to_pmin(vec4d a, vec4d n, vec4d ninv)
vec8d vec8d_reduce_to_pmin(vec8d a, vec8d n, vec8d ninv)

Return $a$ mod $n$ reduced to $[-n, n]$. 
vec8d vec8d_reduce_to_pm1no(vec8d a, vec8d n, vec8d ninv)
    Return \(a \mod n\) reduced to \((-n, n)\).

vec1d vec1d_reduce_0n_to_pmhn(vec1d a, vec1d n)
vec4d vec4d_reduce_0n_to_pmhn(vec4d a, vec4d n)
    Return \(a \mod n\) reduced to \([-n/2, n/2]\) given \(a \in [0, n]\).

vec1d vec1d_reduce_pmin_to_pmhn(vec1d a, vec1d n)
vec4d vec4d_reduce_pmin_to_pmhn(vec4d a, vec4d n)
vec8d vec8d_reduce_pmin_to_pmhn(vec8d a, vec8d n)
    Return \(a \mod n\) reduced to \([-n/2, n/2]\) given \(a \in [-n, n]\).

vec1d vec1d_reduce_2n_to_n(vec1d a, vec1d n)
vec4d vec4d_reduce_2n_to_n(vec4d a, vec4d n)
vec8d vec8d_reduce_2n_to_n(vec8d a, vec8d n)
    Return \(a \mod n\) reduced to \([0, n)\) given \(a \in [0, 2n)\).

vec1d vec1d_reduce_to_0n(vec1d a, vec1d n, vec1d ninv)
vec4d vec4d_reduce_to_0n(vec4d a, vec4d n, vec4d ninv)
vec8d vec8d_reduce_to_0n(vec8d a, vec8d n, vec8d ninv)
    Return \(a \mod n\) reduced to \([0, n)\).

vec1d vec1d_mulmod(vec1d a, vec1d b, vec1d n, vec1d ninv)
vec4d vec4d_mulmod(vec4d a, vec4d b, vec4d n, vec4d ninv)
vec8d vec8d_mulmod(vec8d a, vec8d b, vec8d n, vec8d ninv)
    Return \(ab \mod n\) in \([-n, n]\) with assumptions.

vec1d vec1d_nmulmod(vec1d a, vec1d b, vec1d n, vec1d ninv)
vec4d vec4d_nmulmod(vec4d a, vec4d b, vec4d n, vec4d ninv)
vec8d vec8d_nmulmod(vec8d a, vec8d b, vec8d n, vec8d ninv)
    Return \(ab \mod n\) in \([-n, n]\) with assumptions.

vec4n vec4n_addmod(vec4n a, vec4n b, vec4n n)
vec8n vec8n_addmod(vec8n a, vec8n b, vec8n n)
    Return \(a + b \mod n\) in \([0, n]\)

vec4n vec4n_addmod_limited(vec4n a, vec4n b, vec4n n)
vec8n vec8n_addmod_limited(vec8n a, vec8n b, vec8n n)
    Return \(a + b \mod n\) in \([0, n]\), assuming that \(n < 2^{63}\).
3.1 gr.h – generic structures and their elements

3.1.1 Introduction

Parents and elements

To work with an element \( x \in R \) of a particular mathematical structure \( R \), we use a context object to represent \( R \) (the “parent” of \( x \)). Elements are passed around as pointers. Note:

- Parents are not stored as part of the elements; the user must track the context objects for all variables.
- Operations are strictly type-stable: elements only change parent when performing an explicit conversion.

The structure \( R \) will typically be a ring, but the framework supports general objects (including groups, monoids, and sets without any particular structure whatsoever). We use these terms in a strict mathematical sense: a “ring” must exactly satisfy the ring axioms. It can have inexact representations, but this inexactness must be handled rigorously.

To give an idea of how the interface works, this example program computes \( 3^{100} \) in the ring of integers and prints the value:

```c
#include "gr.h"

int main()
{
    int status;
    gr_ctx_t ZZ; /* a parent (context object) */
    gr_ptr x; /* an element */
    gr_ctx_init_fmpz(ZZ); /* ZZ = ring of integers with fmpz_t elements */
    GR_TMP_INIT(x, ZZ) /* allocate element on the stack */
    status = gr_set_ui(x, 3, ZZ); /* x = 3 */
    status |= gr_pow_ui(x, x, 100, ZZ); /* x = x ^ 100 */
    status |= gr_println(x, ZZ);
    GR_TMP_CLEAR(x, ZZ)
    gr_ctx_clear(ZZ);
    return status;
}
```
Parent and element types

type gr_ptr
    Pointer to a ring element or array of contiguous ring elements. This is an alias for void * so that it can be used with any C type.

type gr_srcptr
    Pointer to a read-only ring element or read-only array of contiguous ring elements. This is an alias for const void * so that it can be used with any C type.

type gr_ctx_struct

type gr_ctx_t
    A context object representing a mathematical structure $R$. It contains the following data:
    • The size (number of bytes) of each element.
    • A pointer to a method table.
    • Optionally a pointer to data defining parameters of the ring (e.g. modulus of a residue ring; element ring and dimensions of a matrix ring; precision of an inexact ring).

A $\text{gr_ctx_t}$ is defined as an array of length one of type $\text{gr_ctx_struct}$, permitting a $\text{gr_ctx_t}$ to be passed by reference. Context objects are not normally passed as const in order to allow storing mutable caches, additional debugging information, etc.

type gr_ctx_ptr
    Pointer to a context object.

There is no type to represent a single generic element as a struct since we do not know the size of a generic element at compile time. Memory for single elements can either be allocated on the stack with the special macros provided below, or as usual with malloc. Methods can also be used with particular C types like fmpz_t when the user knows the type. Users may wish to define their own union types when only some particular types will appear in an application.

Error handling

To compute over a structure $R$, it is useful to conceptually extend to a larger set $R' = R \cup \{\text{undefined, unknown}\}$.

• Adding an undefined (error) value allows us to extend partial functions to total functions.
• An unknown value is useful in cases where a result may exist in principle but cannot be computed.

An alternative to having an undefined value is to choose some arbitrary default value in $R$, say undefined = 0 in a ring. This is often done in proof assistants, but in a regular programming environment, we typically want some way to detect domain errors.

Representing $R'$ as a type-level extension of $R$ is tricky in C since we would either have to wrap elements in a larger structure or reserve bit patterns in each type for special values. In any case, it is useful to assume in low-level code that elements really represent elements of the intended structure so that there are fewer special cases to handle. We also need some form of error handling for conversions to standard C types. For these reasons, we handle special values (undefined, unknown) using return codes.

Functions can return a combination of the following status flags:

GR_SUCCESS
    The operation finished as expected, i.e. the result is a correct element of the target type.

GR_DOMAIN
    The result does not have a value in the domain of the target ring or type, i.e. the result is mathematically undefined. This occurs, for example, on division by zero or when attempting to compute the square root of a non-square. It also occurs when attempting to convert a too large value to a bounded type (example: get_ui() with input $n \geq 2^{64}$).
GR_UNABLE

The operation could not be performed because of limitations of the implementation or the data representation, i.e. the result is unknown. Typical reasons:

- The result would be too large to fit in memory
- The inputs are inexact and an exact comparison is needed
- The computation would take too long
- An algorithm is not yet implemented for this case

If this flag is set, there is also potentially a domain error (but this is unknown).

GR_TEST_FAIL

Test failure. This is only used in test code.

When the status code is any other value than GR_SUCCESS, any output variables may be set to meaningless values.

C functions that return a status code are marked with the WARN_UNUSED_RESULT attribute. This allows compilers to emit warnings when the status code is ignored.

Flags can be OR’ed and checked only at the top level of a computation to avoid complex control flow:

```c
status = GR_SUCCESS;
gr |= gr_add(res, a, b, ctx);
gr |= gr_pow_ui(res, res, 2, ctx);
...
```

If we do not care about recovering from undefined/unknown results, the following macro is useful:

GR_MUST_SUCCEED(expr)

Evaluates expr and asserts that the return value is GR_SUCCESS. On failure, calls flint_abort().

For uniformity, most operations return a status code, even operations that are not typically expected to fail. Exceptions include the following:

- Pure “container” operations like init, clear and swap do not return a status code.
- Pure predicate functions (see below) return T_TRUE / T_FALSE / T_UNKNOWN instead of computing a separate boolean value and error code.

**Predicates**

We use the following type (borrowed from Calcium) instead of a C int to represent boolean results, allowing the possibility that the value is not computable:

```c
define truth_t

Represent one of the following truth values:

T_TRUE
T_FALSE
T_UNKNOWN

Warning: the constants T_TRUE and T_FALSE do not correspond to 1 and 0. It is erroneous to write, for example !t if t is a truth_t. One should instead write t != T_TRUE, t == T_FALSE, etc. depending on whether the unknown case should be included or excluded.
3.1.2 Context operations

```c
slong gr_ctx_sizeof_elem(gr_ctx_t ctx)
  Return sizeof(type) where type is the underlying C type for elements of ctx.

int gr_ctx_clear(gr_ctx_t ctx)
  Clears the context object ctx, freeing any memory allocated by this object.

  Some context objects may require that no elements are cleared after calling this method, and may
  leak memory if not all elements have been cleared when calling this method.

  If ctx is derived from a base ring, the base ring context may also be required to stay alive until
  after this method is called.
```

```c
int gr_ctx_write(gr_stream_t out, gr_ctx_t ctx)
int gr_ctx_print(gr_ctx_t ctx)
int gr_ctx_println(gr_ctx_t ctx)
int gr_ctx_get_str(char **s, gr_ctx_t ctx)
  Writes a description of the structure ctx to the stream out, prints it to stdout, or sets s to a pointer
  to a heap-allocated string of the description (the user must free the string with flint_free). The
  println version prints a trailing newline.
```

```c
int gr_ctx_set_gen_name(gr_ctx_t ctx, const char *s)
int gr_ctx_set_gen_names(gr_ctx_t ctx, const char **s)
  Set the name of the generator (univariate polynomial ring, finite field, etc.) or generators (multi-
  variate). The name is used when printing and may be used to choose coercions.
```

3.1.3 Element operations

Memory management

```c
void gr_init(gr_ptr res, gr_ctx_t ctx)
  Initializes res to a valid variable and sets it to the zero element of the ring ctx.

void gr_clear(gr_ptr res, gr_ctx_t ctx)
  Clears res, freeing any memory allocated by this object.

void gr_swap(gr_ptr x, gr_ptr y, gr_ctx_t ctx)
  Swaps x and y efficiently.

void gr_set_shallow(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
  Sets res to a shallow copy of x, copying the struct data.

gr_ptr gr_heap_init(gr_ctx_t ctx)
  Return a pointer to a single new heap-allocated element of ctx set to 0.

void gr_heap_clear(gr_ptr x, gr_ctx_t ctx)
  Free the single heap-allocated element x of ctx which should have been created with
  gr_heap_init().

gr_ptr gr_heap_init_vec(slong len, gr_ctx_t ctx)
  Return a pointer to a new heap-allocated vector of len initialized elements.

void gr_heap_clear_vec(gr_ptr x, slong len, gr_ctx_t ctx)
  Clear the len elements in the heap-allocated vector len and free the vector itself.
```

The following macros support allocating temporary variables efficiently. Data will be allocated on the
stack using alloca unless the size is excessive (risking stack overflow), in which case the implementation
transparently switches to malloc/free instead. The usage pattern is as follows:
Init and clear macros must match exactly, as variables may be allocated contiguously in a block.

Warning: never use these macros directly inside a loop. This is likely to overflow the stack, as memory will not be reclaimed until the function exits. Instead, allocate the needed space before entering any loops, move the loop body to a separate function, or allocate the memory on the heap if needed.

```
{  
  gr_ptr x, y;  
  GR_TMP_INIT2(x1, x2, ctx);  
  /* do computations with x1, x2 */  
  GR_TMP_CLEAR2(x1, x2, ctx);  
}
```

GR_TMP_INIT_VEC(vec, len, ctx)

GR_TMP_CLEAR_VEC(vec, len, ctx)

Allocates and frees a vector of len contiguous elements, all initialized to the value 0, assigning the first element to the pointer vec.

GR_TMP_INIT(x1, ctx)

GR_TMP_INIT2(x1, x2, ctx)

GR_TMP_INIT3(x1, x2, x3, ctx)

GR_TMP_INIT4(x1, x2, x3, x4, ctx)

GR_TMP_INIT5(x1, x2, x3, x4, x5, ctx)

Allocates one or several temporary elements, all initialized to the value 0, assigning the elements to the pointers x1, x2, etc.

GR_TMP_CLEAR(x1, ctx)

GR_TMP_CLEAR2(x1, x2, ctx)

GR_TMP_CLEAR3(x1, x2, x3, ctx)

GR_TMP_CLEAR4(x1, x2, x3, x4, ctx)

GR_TMP_CLEAR5(x1, x2, x3, x4, x5, ctx)

Corresponding macros to clear temporary variables.

**Random elements**

int gr_randtest(gr_ptr res, flint_rand_t state, gr_ctx_t ctx)

Sets res to a random element of the domain ctx. The distribution is determined by the implementation. Typically the distribution is non-uniform in order to find corner cases more easily in test code.

int gr_randtest_not_zero(gr_ptr res, flint_rand_t state, gr_ctx_t ctx)

Sets res to a random nonzero element of the domain ctx. This operation will fail and return GR_DOMAIN in the zero ring.

int gr_randtest_small(gr_ptr res, flint_rand_t state, gr_ctx_t ctx)

Sets res to a “small” element of the domain ctx. This is suitable for randomized testing where a “large” argument could result in excessive computation time.
Input, output and string conversion

\begin{verbatim}
int gr_write(gr_stream_t out, gr_srcptr x, gr_ctx_t ctx)
int gr_print(gr_srcptr x, gr_ctx_t ctx)
int gr_println(gr_srcptr x, gr_ctx_t ctx)
int gr_get_str(char **s, gr_srcptr x, gr_ctx_t ctx)
\end{verbatim}

Writes a description of the element \(x\) to the stream \(out\), or prints it to \(stdout\), or sets \(s\) to a pointer to a heap-allocated string of the description (the user must free the string with \texttt{flint_free}). The \texttt{println} version prints a trailing newline.

\begin{verbatim}
int gr_set_str(gr_ptr res, const char *x, gr_ctx_t ctx)
int gr_write_n(gr_stream_t out, gr_srcptr x, slong n, gr_ctx_t ctx)
int gr_get_str_n(char **s, gr_srcptr x, slong n, gr_ctx_t ctx)
\end{verbatim}

String conversion where real and complex numbers may be rounded to \(n\) digits.

Assignment and conversions

\begin{verbatim}
int gr_set(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_set_other(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t ctx)
int gr_set_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_set_si(gr_ptr res, slong x, gr_ctx_t ctx)
int gr_set_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
int gr_set_fmpq(gr_ptr res, const fmpq_t x, gr_ctx_t ctx)
int gr_set_d(gr_ptr res, double x, gr_ctx_t ctx)
int gr_get_si(slong *res, gr_srcptr x, gr_ctx_t ctx)
int gr_get_ui(ulong *res, gr_srcptr x, gr_ctx_t ctx)
int gr_get_fmpz(fmpz_t res, gr_srcptr x, gr_ctx_t ctx)
int gr_get_fmpq(fmpq_t res, gr_srcptr x, gr_ctx_t ctx)
int gr_get_d(double *res, gr_srcptr x, gr_ctx_t ctx)
int gr_set_fmpz_2exp_fmpz(gr_ptr res, const fmpz_t a, const fmpz_t b, gr_ctx_t ctx)
int gr_get_fmpz_2exp_fmpz(fmpz_t res1, fmpz_t res2, gr_srcptr x, gr_ctx_t ctx)
int gr_set_fmpz_10exp_fmpz(gr_ptr res, const fmpz_t a, const fmpz_t b, gr_ctx_t ctx)
int gr_get_fmpz_10exp_fmpz(fmpz_t res1, fmpz_t res2, gr_srcptr x, gr_ctx_t ctx)
int gr_set_fexpr(fexpr_t res, gr_srcptr x, gr_ctx_t ctx)
int gr_get_fexpr_serialize(fexpr_t res, gr_srcptr x, gr_ctx_t ctx)
\end{verbatim}

Sets \(res\) to the value \(x\). If no reasonable conversion to the domain \(ctx\) is possible, returns \texttt{GR_DOMAIN}.

Sets \(res\) to a copy of the element \(x\).

Sets \(res\) to the element \(x\) of the structure \(x\_ctx\) which may be different from \(ctx\). This returns the \texttt{GR_DOMAIN} flag if \(x\) is not an element of \(ctx\) or cannot be converted unambiguously to \(ctx\). The \texttt{GR_UNABLE} flag is returned if the conversion is not implemented.

Sets \(res\) to \(a \cdot 2^b\).

Sets \(res\) to \(a \cdot 10^b\).

Sets \(res\) to a symbolic expression representing \(x\). The \texttt{serialize} version may generate a representation of the internal representation which is not intended to be human-readable.
int gr_set_fexpr (gr_ptr res, fexpr_vec_t inputs, gr_vec_t outputs, const fexpr_t x, gr_ctx_t ctx)

Sets res to the evaluation of the expression x in the given ring or structure. The user must provide vectors inputs and outputs which may be empty initially and which may be used as scratch space during evaluation. Non-empty vectors may be given to map symbols to predefined values.

**Special values**

int gr_zero (gr_ptr res, gr_ctx_t ctx)
int gr_one (gr_ptr res, gr_ctx_t ctx)
int gr_neg_one (gr_ptr res, gr_ctx_t ctx)

Sets res to the ring element 0, 1 or -1.

int gr_gen (gr_ptr res, gr_ctx_t ctx)

Sets res to a generator of this domain. The meaning of “generator” depends on the domain.

int gr_gens (gr_vec_t res, gr_ctx_t ctx)
int gr_gens_recursive (gr_vec_t res, gr_ctx_t ctx)

Sets res to a vector containing the generators of this domain where this makes sense, for example in a multivariate polynomial ring. The recursive version also includes any generators of the base ring, and of any recursive base rings.

**Basic properties**

truth_t gr_is_zero (gr_srcptr x, gr_ctx_t ctx)
truth_t gr_is_one (gr_srcptr x, gr_ctx_t ctx)
truth_t gr_is_neg_one (gr_srcptr x, gr_ctx_t ctx)

Returns whether x is equal to the ring element 0, 1 or -1, respectively.

truth_t gr_equal (gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)

Returns whether the elements x and y are equal.

truth_t gr_is_integer (gr_srcptr x, gr_ctx_t ctx)

Returns whether x represents an integer.

truth_t gr_is_rational (gr_srcptr x, gr_ctx_t ctx)

Returns whether x represents a rational number.

**Arithmetic**

User-defined rings should supply neg, add, sub and mul methods; the variants with other operand types have generic fallbacks that may be overridden for performance. The fmpq versions may return GR_DOMAIN if the denominator is not invertible. The other versions accept operands belonging to a different domain, attempting to perform a coercion into the target domain.

int gr_neg (gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

Sets res to -x.

int gr_add (gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_add_ui (gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_add_si (gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_add_fmpz (gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_add_fmpq (gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)
int gr_add_other (gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
Iterated arithmetic operations are best performed using vector functions. See in particular 
\texttt{\_gr\_vec\_dot()} and \texttt{\_gr\_vec\_dot\_rev()}.
Division

The default implementations of the following methods check for divisors 0, 1, -1 and otherwise return `GR_UNABLE`. Particular rings should override the methods when an inversion or division algorithm is available.

```c
truth_t gr_is_invertible(gr_srcptr x, gr_ctx_t ctx)
```
Returns whether \(x\) has a multiplicative inverse in the present ring, i.e. whether \(x\) is a unit.

```c
int gr_inv(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```
Sets res to the multiplicative inverse of \(x\) in the present ring, if such an element exists. Returns the flag `GR_DOMAIN` if \(x\) is not invertible, or `GR_UNABLE` if the implementation is unable to perform the computation.

```c
int gr_div(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
```
Sets res to the quotient \(x/y\). In a field, this returns `GR_DOMAIN` if \(y\) is zero; in an integral domain, it returns `GR_DOMAIN` if \(y\) is zero or if the quotient does not exist. In a non-integral domain, we consider a quotient to exist only if it is unique, and otherwise return `GR_DOMAIN`; see `gr_div_nonunique()` for a different behavior.

Returns the flag `GR_UNABLE` if the implementation is unable to perform the computation.

```c
int gr_div_nonunique(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
```
Sets res to an arbitrary solution \(q\) of the equation \(x = qy\). Returns the flag `GR_DOMAIN` if no such solution exists. Returns the flag `GR_UNABLE` if the implementation is unable to perform the computation. This method allows dividing \(x/y\) in some cases where `gr_div()` fails:

- 0/0 has solutions (for example, 0) in any ring.
- It allows solving division problems in nonintegral domains. For example, it allows assigning a value to 6/2 in \(R = \mathbb{Z}/10\mathbb{Z}\) even though \(2^{-1}\) does not exist in \(R\). In this case, both 3 and 8 are possible solutions, and which one is chosen is unpredictable.

```c
truth_t gr_divides(gr_srcptr d, gr_srcptr x, gr_ctx_t ctx)
```
Returns whether \(d \mid x\); that is, whether there is an element \(q\) such that \(x = dq\). Note that this corresponds to divisibility in the sense of `gr_div_nonunique()`, which is weaker than that of `gr_div()`. For example, 0 \(\mid 0\) is true even in rings where 0/0 is undefined.
In a Euclidean ring, these functions perform some version of Euclidean division with remainder, where the choice of quotient is implementation-defined. For example, it is standard to use the round-to-floor quotient in $\mathbb{Z}$ and a round-to-nearest quotient in $\mathbb{Z}[i]$. In non-Euclidean rings, these functions may implement some generalization of Euclidean division with remainder.

**Powering**

- `int gr_pow(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`
- `int gr_pow_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)`
- `int gr_pow_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)`
- `int gr_pow_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)`
- `int gr_pow_fmpq(gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)`
- `int gr_pow_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)`
- `int gr_other_pow(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)`

Sets `res` to the power $x^y$, the interpretation of which depends on the ring when $y \not\in \mathbb{Z}$. Returns the flag `GR_DOMAIN` if this power cannot be assigned a meaningful value in the present ring, or `GR_UNABLE` if the implementation is unable to perform the computation.

For subrings of $\mathbb{C}$, it is implied that the principal power $x^y = \exp(y \log(x))$ is computed for $x \neq 0$.

Default implementations of the powering methods support raising elements to integer powers using a generic implementation of exponentiation by squaring. Particular rings should override these methods with faster versions or to support more general notions of exponentiation when possible.

**Square roots**

The default implementations of the following methods check for the elements 0 and 1 and otherwise return `GR_UNABLE`. Particular rings should override the methods when a square root algorithm is available.

In subrings of $\mathbb{C}$, it is implied that the principal square root is computed; in other cases (e.g. in finite fields), the choice of root is implementation-dependent.

- `truth_t gr_is_square(gr_srcptr x, gr_ctx_t ctx)`
  Returns whether $x$ is a perfect square in the present ring.
- `int gr_sqrt(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`
- `int gr_rsqrt(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`

Sets `res` to a square root of $x$ (respectively reciprocal square root) in the present ring, if such an element exists. Returns the flag `GR_DOMAIN` if $x$ is not a perfect square (also for zero, when computing the reciprocal square root), or `GR_UNABLE` if the implementation is unable to perform the computation.

**Greatest common divisors**

- `int gr_gcd(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`
  Sets `res` to a greatest common divisor (GCD) of $x$ and $y$. Since the GCD is unique only up to multiplication by a unit, an implementation-defined representative is chosen.

- `int gr_lcm(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`
  Sets `res` to a least common multiple (LCM) of $x$ and $y$. Since the LCM is unique only up to multiplication by a unit, an implementation-defined representative is chosen.
Factorization

```c
int gr_factor(gr_ptr c, gr_vec_t factors, gr_vec_t exponents, gr_srcptr x, int flags, gr_ctx_t ctx)
```

Given \( x \in R \), computes a factorization

\[
x = c f_1^{e_1} \cdots f_n^{e_n}
\]

where \( f_k \) will be irreducible or prime (depending on \( R \)).

The prefactor \( c \) stores a unit, sign, or coefficient, e.g. the sign \(-1, 0 \) or \(+1 \) in \( \mathbb{Z} \), or a sign multiplied by the coefficient content in \( \mathbb{Z}[x] \). Note that this function outputs \( c \) as an element of the same ring as the input: for example, in \( \mathbb{Z}[x] \), \( c \) will be a constant polynomial rather than an element of the coefficient ring. The exponents \( e_k \) are output as a vector of \( \text{fmpz} \) elements.

The factors \( f_k \) are guaranteed to be distinct, but they are not guaranteed to be sorted in any particular order.

Fractions

```c
int gr_numerator(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_denominator(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

Return a numerator \( p \) and denominator \( q \) such that \( x = p/q \). For typical fraction fields, the denominator will be minimal and canonical. However, some rings may return an arbitrary denominator as long as the numerator matches. The default implementations simply return \( p = x \) and \( q = 1 \).

Integer and complex parts

```c
int gr_floor(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_ceil(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_trunc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_nint(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

In the real and complex numbers, sets \( res \) to the integer closest to \( x \), respectively rounding towards minus infinity, plus infinity, zero, or the nearest integer (with tie-to-even).

```c
int gr_abs(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

Sets \( res \) to the absolute value of \( x \), which may be defined both in complex rings and in any ordered ring.

```c
int gr_i(gr_ptr res, gr_ctx_t ctx)
```

Sets \( res \) to the imaginary unit.

```c
int gr_conj(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_re(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_im(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_sgn(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_csgn(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_arg(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

These methods may return the flag \( \text{GR\_DOMAIN} \) (or \( \text{GR\_UNABLE} \)) when the ring is not a subring of the real or complex numbers.
Infinities and extended values

int gr_pos_inf(gr_ptr res, gr_ctx_t ctx)
int gr_neg_inf(gr_ptr res, gr_ctx_t ctx)
int gr_uinf(gr_ptr res, gr_ctx_t ctx)
int gr_undefined(gr_ptr res, gr_ctx_t ctx)
int gr_unknown(gr_ptr res, gr_ctx_t ctx)

Ordering methods

int gr_cmp(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_cmp_other(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
Sets res to -1, 0 or 1 according to whether x is less than, equal or greater than the absolute value of y. This may return GR DOMAIN if the ring is not an ordered ring.

int gr_cmpabs(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_cmpabs_other(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
Sets res to -1, 0 or 1 according to whether the absolute value of x is less than, equal or greater than the absolute value of y. This may return GR_DOMAIN if the ring is not an ordered ring.

Enclosure and interval methods

int gr_set_interval_mid_rad(gr_ptr res, gr_srcptr m, gr_srcptr r, gr_ctx_t ctx)
In ball representations of the real numbers, sets res to the interval $m \pm r$.

In vector spaces over the real numbers represented using balls, intervals are handled independently for the generators; for example, in the complex numbers, $a + bi \pm (0.1 + 0.2i)$ is equivalent to $(a \pm 0.1) + (b \pm 0.2)i$.

Finite field methods

int gr_ctx_fq_prime(fmpz_t p, gr_ctx_t ctx)
int gr_ctx_fq_degree(slong *deg, gr_ctx_t ctx)
int gr_ctx_fq_order(fmpz_t q, gr_ctx_t ctx)
int gr_fq_frobenius(gr_ptr res, gr_srcptr x, slong e, gr_ctx_t ctx)
int gr_fq_multiplicative_order(fmpz_t res, gr_srcptr x, gr_ctx_t ctx)
int gr_fq_norm(fmpz_t res, gr_srcptr x, gr_ctx_t ctx)
int gr_fq_trace(fmpz_t res, gr_srcptr x, gr_ctx_t ctx)
truth_t gr_fq_is_primitive(gr_srcptr x, gr_ctx_t ctx)
int gr_fq_pth_root(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
3.2 gr.h (continued) – implementing rings

Defining a ring requires putting appropriate data into a `gr_ctx_t` parent object, most importantly the method table and the size of elements.

3.2.1 Example

This is an extract from the `fmpz` wrapper in `gr/fmpz.c`:

```c
/* Some methods */
...
int _gr_fmpz_add(fmpz_t res, const fmpz_t x, const fmpz_t y, const gr_ctx_t ctx)
{
    fmpz_add(res, x, y);
    return GR_SUCCESS;
}
...

/* The method table */
int _fmpz_methods_initialized = 0;

gr_static_method_table _fmpz_methods;

gr_method_tab_input _fmpz_methods_input[] =
{
    {GR_METHOD_CTX_IS_RING, (gr_funcptr) gr_generic_ctx_predicate_true},
    ...
    {GR_METHOD_INIT, (gr_funcptr) _gr_fmpz_init},
    {GR_METHOD_CLEAR, (gr_funcptr) _gr_fmpz_clear},
    ...
    {GR_METHOD_ADD_FMPZ, (gr_funcptr) _gr_fmpz_add},
    ...
    {0, (gr_funcptr) NULL},
};

/* Context object initializer */

void gr_ctx_init_fmpz(gr_ctx_t ctx)
{
    ctx->which_ring = GR_CTX_FMPZ;
    ctx->sizeof_elem = sizeof(fmpz);
    ctx->size_limit = WORD_MAX;

    ctx->methods = _fmpz_methods;

    if (!_fmpz_methods_initialized)
    {
        gr_method_tab_init(_fmpz_methods, _fmpz_methods_input);
        _fmpz_methods_initialized = 1;
    }
}
```

Note that the method table only has to be constructed once, allowing new context objects for the same domain to be initialized cheaply.
3.2.2 Method table

type gr_funcptr
    Typedef for a pointer to a function with signature int func(void), used to represent method table entries.

type gr_method
    Enumeration type for indexing method tables. Enum values named GR_METHOD_INIT, GR_METHOD_ADD_UI, etc. correspond to methods gr_init, gr_add_ui, etc. The number of methods is given by GR_METHOD_TAB_SIZE, which can be used to declare static method tables.

type gr_static_method_table
    Typedef for an array of length GR_METHOD_TAB_SIZE with gr_funcptr entries.

type gr_method_tab_input
    Typedef representing a (index, function pointer) pair.

void gr_method_tab_init(gr_funcptr *methods, gr_method_tab_input *tab)
    Initializes the method table methods. This first inserts default and generic methods in all slots, and then overwrites with the specialized methods listed in tab.

3.2.3 Placeholder and trivial methods

int gr_not_implemented(void)
    This function does nothing and returns GR_UNABLE. It is used as a generic fallback method when no implementation is available.

int gr_not_in_domain(void)
    This function does nothing and returns GR_DOMAIN. It can be used for an operation that never makes sense in the present domain, e.g. for the constant π in the rational numbers.

truth_t gr_generic_ctx_predicate(gr_ctx_t ctx)
    Does nothing and returns T_UNKNOWN, used as a generic fallback for predicate methods.

truth_t gr_generic_ctx_predicate_true(gr_ctx_t ctx)
    A predicate that does nothing and returns T_TRUE.

truth_t gr_generic_ctx_predicate_false(gr_ctx_t ctx)
    A predicate that does nothing and returns T_FALSE.

3.2.4 Required methods

A context object must at minimum define the following methods for a ring:

- init
- clear
- swap
- randtest
- write
- zero
- one
- equal
- set
- set_si
• set_ui
• set_fmpz
• neg
• add
• sub
• mul

Other methods have generic defaults which may be overridden for performance or completeness.

Implementing context predicates (ctx_is_integral_domain, ctx_is_field, etc.) is strongly recommended so that the most appropriate algorithms can be used in generic implementations.

Rings with cheap operations on single elements should also provide non-generic versions of performance-critical vector operations to minimize overhead. The most important vector operations include:

• vec_init
• vec_clear
• vec_swap
• vec_zero
• vec_neg
• vec_add
• vec_sub
• vec_mul_scalar_ui/si
• vec_addmul_scalar_ui/si
• vec_dot
• vec_dot_rev

Dot products, for example, are the main building block for classical polynomial multiplication and matrix multiplication. The methods

• poly_mullow
• matrix_mul

should be overridden for rings where faster-than-classical polynomial and matrix multiplication is possible. Other higher-complexity generic algorithms will try to reduce to polynomial and matrix multiplication automatically, but may in turn need to be overridden to select accurate cutoffs between different algorithms.

3.2.5 Testing rings

void gr_test_ring(gr_ctx_t R, slong iters, int test_flags)

Test correctness of the ring R. This calls test functions for various methods, each being repeated up to iters times.
3.3 gr.h (continued) – builtin domains and types

3.3.1 Coercions

```c
int gr_ctx_cmp_coercion(gr_ctx_t ctx1, gr_ctx_t ctx2)
```

Returns 1 if coercing elements into `ctx1` is more meaningful, and returns -1 otherwise.

3.3.2 Domain properties

```c
truth_t gr_ctx_is_finite(gr_ctx_t ctx)
truth_t gr_ctx_is_multiplicative_group(gr_ctx_t ctx)
truth_t gr_ctx_is_ring(gr_ctx_t ctx)
truth_t gr_ctx_is_commutative_ring(gr_ctx_t ctx)
truth_t gr_ctx_is_integral_domain(gr_ctx_t ctx)
truth_t gr_ctx_is_unique_factorization_domain(gr_ctx_t ctx)
truth_t gr_ctx_is_field(gr_ctx_t ctx)
truth_t gr_ctx_is_algebraically_closed(gr_ctx_t ctx)
truth_t gr_ctx_is_finite_characteristic(gr_ctx_t ctx)
truth_t gr_ctx_is_ordered_ring(gr_ctx_t ctx)
truth_t gr_ctx_is_zero_ring(gr_ctx_t ctx)
```

Returns whether the structure satisfies the respective mathematical property. The result can be `T_UNKNOWN`.

```c
truth_t gr_ctx_is_exact(gr_ctx_t ctx)
```

Returns whether the representation of elements is always exact.

```c
truth_t gr_ctx_is_canonical(gr_ctx_t ctx)
```

Returns whether the representation of elements is always canonical.

```c
truth_t gr_ctx_has_real_prec(gr_ctx_t ctx)
```

Returns whether `ctx` or a base field thereof represents real or complex numbers using finite-precision approximations. This returns `T_TRUE` both for floating-point approximate fields and for rigorous fields based on ball or interval arithmetic.

```c
int gr_ctx_set_real_prec(gr_ctx_t ctx, slong prec)
int gr_ctx_get_real_prec(slong *prec, gr_ctx_t ctx)
```

Sets or retrieves the floating-point precision in bits.

3.3.3 Groups

```c
void gr_ctx_init_perm(gr_ctx_t ctx, ulong n)
```

Initializes `ctx` to the symmetric group \( S_n \) representing permutations of \([0, 1, \ldots, n-1]\). Elements are currently represented as pointers (the representation may change in the future).

```c
void gr_ctx_init_psl2z(gr_ctx_t ctx)
```

Initializes `ctx` to the modular group \( \text{PSL}(2, \mathbb{Z}) \) with elements of type `psl2z_t`.

```c
int gr_ctx_init_dirichlet_group(gr_ctx_t ctx, ulong q)
```

Initializes `ctx` to the Dirichlet group \( G_q \) with elements of type `dirichlet_char_t`. Fails and returns `GR_DOMAIN` if \( q \) is zero. Fails and returns `GR_UNABLE` if \( q \) has a prime factor larger than \( 10^{16} \), which is currently unsupported by the implementation.
3.3.4 Base rings and fields

void gr_ctx_init_random(gr_ctx_t ctx, flint_rand_t state)
    Initializes ctx to a random ring. This will currently only generate base rings and composite rings over certain simple base rings.

void gr_ctx_init_fmpz(gr_ctx_t ctx)
    Initializes ctx to the ring of integers $\mathbb{Z}$ with elements of type fmpz.

void gr_ctx_init_fmpq(gr_ctx_t ctx)
    Initializes ctx to the field of rational numbers $\mathbb{Q}$ with elements of type fmpq.

void gr_ctx_init_fmpzi(gr_ctx_t ctx)
    Initializes ctx to the ring of Gaussian integers $\mathbb{Z}[i]$ with elements of type fmpzi_t.

void gr_ctx_init_nmod8(gr_ctx_t ctx, unsigned char n)
    void gr_ctx_init_nmod32(gr_ctx_t ctx, unsigned int n)
    Initializes ctx to the ring $\mathbb{Z}/n\mathbb{Z}$ of integers modulo $n$ where elements have type uint8 or uint32. The modulus must be nonzero.

void gr_ctx_init_nmod(gr_ctx_t ctx, ulong n)
    Initializes ctx to the ring $\mathbb{Z}/n\mathbb{Z}$ of integers modulo $n$ where elements have type ulong. We require $n \neq 0$.

void gr_ctx_init_fmpz_mod(gr_ctx_t ctx, const fmpz_t n)
    Initializes ctx to the ring $\mathbb{Z}/n\mathbb{Z}$ of integers modulo $n$ where elements have type fmpz. The modulus must be positive.

void gr_ctx_fmpz_mod_set_primality(gr_ctx_t ctx, truth_t is_prime)
    For a ring initialized with gr_ctx_init_fmpz_mod(), indicate whether the modulus is prime. This can speed up some computations.

void gr_ctx_init_fq(gr_ctx_t ctx, const fmpz_t p, slong d, const char *var)
void gr_ctx_init_fq_nmod(gr_ctx_t ctx, ulong p, slong d, const char *var)
void gr_ctx_init_fq_zech(gr_ctx_t ctx, ulong p, slong d, const char *var)
    Initializes ctx to the finite field $\mathbb{F}_q$ where $q = p^d$. It is assumed (not checked) that $p$ is prime. The variable name var can be NULL to use a default. The corresponding element types are fq_t, fq_nmod_t, fq_zech_t. The fq_zech context requires $q < 2^{64}$ (and in practice a much smaller value than this).

void gr_ctx_init_nf(gr_ctx_t ctx, const fmpq_poly_t poly)
void gr_ctx_init_nf_fmpz_poly(gr_ctx_t ctx, const fmpz_poly_t poly)
    Initializes ctx to the number field with defining polynomial poly which must be irreducible (this is not checked). The elements have type nf_elem_t.

void gr_ctx_init_real_qqbar(gr_ctx_t ctx)
void gr_ctx_init_complex_qqbar(gr_ctx_t ctx)
    Initializes ctx to the field of real or complex algebraic numbers with elements of type qqbar_t.

void gr_ctx_init_real_arb(gr_ctx_t ctx, slong prec)
void gr_ctx_init_complex_acb(gr_ctx_t ctx, slong prec)
    Initializes ctx to the field of real or complex numbers represented by elements of type arb_t and acb_t.

void gr_ctx_arb_set_prec(gr_ctx_t ctx, slong prec)
slong gr_ctx_arb_get_prec(gr_ctx_t ctx)
    Sets or retrieves the bit precision of ctx which must be an Arb context (this is currently not checked).
void gr_ctx_init_real_ca(gr_ctx_t ctx)
void gr_ctx_init_complex_ca(gr_ctx_t ctx)
void gr_ctx_init_real_algebraic_ca(gr_ctx_t ctx)
void gr_ctx_init_complex_algebraic_ca(gr_ctx_t ctx)

Initializes ctx to the field of real, complex, real algebraic or complex algebraic numbers represented by elements of type ca_t.

void gr_ctx_ca_set_option(gr_ctx_t ctx, slong option, slong value)
slong gr_ctx_ca_get_option(gr_ctx_t ctx, slong option)

Sets or retrieves options of a Calcium context object.

3.3.5 Extended number sets

void gr_ctx_init_complex_extended_ca(gr_ctx_t ctx)

Like gr_ctx_init_complex_ca() but allows special values (infinities, undefined).

3.3.6 Floating-point arithmetic

Although domains of floating-point numbers approximate real and complex fields, they are not rings or fields. Floating-point arithmetic can be used in many places where a ring or field is normally assumed, but predicates like “is field” return false.

- Equality compares equality of floating-point numbers, with the special rule that NaN is not equal to itself.
- In general, the following implementations do not currently guarantee correct rounding except for atomic arithmetic operations (add, sub, mul, div, sqrt) on real floating-point numbers.

void gr_ctx_init_real_float_arf(gr_ctx_t ctx, slong prec)

Initializes ctx to the floating-point arithmetic with elements of type arf_t and a default precision of prec bits.

void gr_ctx_init_complex_float_acf(gr_ctx_t ctx, slong prec)

Initializes ctx to the complex floating-point arithmetic with elements of type acf_t and a default precision of prec bits.

3.3.7 Vectors

void gr_ctx_init_vector_gr_vec(gr_ctx_t ctx, gr_ctx_t base_type)

Initializes ctx to the domain of all vectors (of any length) over the given base_type. Elements have type gr_vec_struct.

void gr_ctx_init_vector_space_gr_vec(gr_ctx_t ctx, gr_ctx_t base_type, slong n)

Initializes ctx to the space of all vectors of length n over the given base_type. Elements have type gr_vec_struct.
3.3.8 Matrices

void gr_ctx_init_matrix_domain(gr_ctx_t ctx, gr_ctx_t base_ring)
    Initializes ctx to the domain of all matrices (of any shape) over the given base_ring. Elements have type gr_mat_struct.

void gr_ctx_init_matrix_space(gr_ctx_t ctx, gr_ctx_t base_ring, slong n, slong m)
    Initializes ctx to the space of matrices over base_ring with n rows and m columns. Elements have type gr_mat_struct.

void gr_ctx_init_matrix_ring(gr_ctx_t ctx, gr_ctx_t base_ring, slong n)
    Initializes ctx to the ring of matrices over base_ring with n rows columns. Elements have type gr_mat_struct.

3.3.9 Polynomial rings

void gr_ctx_init_fmpz_poly(gr_ctx_t ctx)
    Initializes ctx to a ring of integer polynomials of type fmpz_poly_struct.

void gr_ctx_init_fmpq_poly(gr_ctx_t ctx)
    Initializes ctx to a ring of rational polynomials of type fmpq_poly_struct.

void gr_ctx_init_gr_poly(gr_ctx_t ctx, gr_ctx_t base_ring)
    Initializes ctx to a ring of densely represented univariate polynomials over the given base_ring. Elements have type gr_poly_struct.

void gr_ctx_init_fmpz_mpoly(gr_ctx_t ctx, slong nvars, const ordering_t ord)
    Initializes ctx to a ring of sparsely represented multivariate polynomials in nvars variables over the integers, with monomial ordering ord. Elements have type fmpz_mpoly_struct.

void gr_ctx_init_gr_mpoly(gr_ctx_t ctx, gr_ctx_t base_ring, slong nvars, const ordering_t ord)
    Initializes ctx to a ring of sparsely represented multivariate polynomials in nvars variables over the given base_ring, with monomial ordering ord. Elements have type gr_mpoly_struct.

3.3.10 Fraction fields

void gr_ctx_init_fmpz_mpoly_q(gr_ctx_t ctx, slong nvars, const ordering_t ord)
    Initializes ctx to a ring of sparsely represented multivariate fractions in nvars variables over the integers (equivalently, rationals), with monomial ordering ord. Elements have type fmpz_mpoly_q_struct.

3.3.11 Symbolic expressions

void gr_ctx_init_fexpr(gr_ctx_t ctx)
    Initializes ctx to handle symbolic expressions. Elements have type fexpr_struct.
3.4 \texttt{gr\_generic.h} – basic algorithms and fallback implementations for generic elements

```c
void \texttt{gr\_generic\_init}(void)
void \texttt{gr\_generic\_clear}(void)
void \texttt{gr\_generic\_swap}(void)
void \texttt{gr\_generic\_randtest}(void)
void \texttt{gr\_generic\_write}(void)
void \texttt{gr\_generic\_zero}(void)
void \texttt{gr\_generic\_one}(void)
void \texttt{gr\_generic\_equal}(void)
void \texttt{gr\_generic\_set}(void)
void \texttt{gr\_generic\_set\_si}(void)
void \texttt{gr\_generic\_set\_ui}(void)
void \texttt{gr\_generic\_set\_fmpz}(void)
void \texttt{gr\_generic\_neg}(void)
void \texttt{gr\_generic\_add}(void)
void \texttt{gr\_generic\_sub}(void)
void \texttt{gr\_generic\_mul}(void)

int \texttt{gr\_generic\_ctx\_clear(gr\_ctx\_t ctx)}

void \texttt{gr\_generic\_set\_shallow(gr\_ptr res, gr\_srcptr x, const gr\_ctx\_t ctx)}

int \texttt{gr\_generic\_write\_n(gr\_stream\_t out, gr\_srcptr x, slong n, gr\_ctx\_t ctx)}

int \texttt{gr\_generic\_randtest\_not\_zero(gr\_ptr x, flint\_rand\_t state, gr\_ctx\_t ctx)}

int \texttt{gr\_generic\_randtest\_small(gr\_ptr x, flint\_rand\_t state, gr\_ctx\_t ctx)}

truth\_t \texttt{gr\_generic\_is\_zero(gr\_srcptr x, gr\_ctx\_t ctx)}

truth\_t \texttt{gr\_generic\_is\_one(gr\_srcptr x, gr\_ctx\_t ctx)}

truth\_t \texttt{gr\_generic\_is\_neg\_one(gr\_srcptr x, gr\_ctx\_t ctx)}

int \texttt{gr\_generic\_neg\_one(gr\_ptr res, gr\_ctx\_t ctx)}

int \texttt{gr\_generic\_set\_other(gr\_ptr res, gr\_srcptr x, gr\_ctx\_t xctx, gr\_ctx\_t ctx)}

int \texttt{gr\_generic\_set\_fmpq(gr\_ptr res, const fmpq\_t y, gr\_ctx\_t ctx)}

3.4.1 Generic string parsing

GR\_PARSE\_BALANCE\_ADDITIONS

GR\_PARSE\_RING\_EXponents

int \texttt{gr\_generic\_set\_str\_expr(gr\_ptr res, const char *s, int flags, gr\_ctx\_t ctx)}

int \texttt{gr\_generic\_set\_str(gr\_ptr res, const char *s, gr\_ctx\_t ctx)}

int \texttt{gr\_generic\_set\_str\_balance\_additions(gr\_ptr res, const char *s, gr\_ctx\_t ctx)}

int \texttt{gr\_generic\_set\_str\_ring\_exponents(gr\_ptr res, const char *s, gr\_ctx\_t ctx)}

Parses expression string. Generators returned by \texttt{gr\_gens\_recursive()} are handled automatically. We have the following flags:
3.4.2 Generic arithmetic

- **GR_PARSE_RING_EXPONENTS** - by default, only (nonnegative) integer literals are allowed for exponents. If this flag is set, exponents are parsed as arbitrary subexpressions within the same ring.

- **GR_PARSE_BALANCE_ADDITIONS** - attempt to improve performance for huge sums by reordering additions (useful for polynomials)

### 3.4.2 Generic arithmetic

```c
int gr_generic_add_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_add_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_add_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_add_fmpq(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_add_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_other_add(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)
int gr_generic_sub_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_sub_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_sub_fmpq(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_sub_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_other_sub(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)
int gr_generic_mul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_mul_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_mul_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_mul_fmpq(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_mul_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_other_mul(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)
int gr_generic_addmul(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_generic_addmul_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_addmul_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_addmul_fmpq(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_addmul_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_submul(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_generic_submul_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_submul_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_submul_fmpq(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_submul_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_mul_two(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_generic_sqr(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_generic_mul_2exp_si(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_mul_2exp_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_set_fmpz_2exp_fmpz(gr_ptr res, const fmpz_t x, const fmpz_t y, gr_ctx_t ctx)
```
int \texttt{gr\_generic\_get\_fmpz\_2exp\_fmpz}(fmpz\_t res1, fmpz\_t res2, gr\_ptr x, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_inv}(gr\_ptr res, gr\_sreptr x, gr\_ctx\_t ctx)

\texttt{truth\_t gr\_generic\_is\_invertible}(gr\_sreptr x, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_div\_fmpz}(gr\_ptr res, gr\_sreptr x, const fmpz\_t y, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_div\_ui}(gr\_ptr res, gr\_sreptr x, ulong y, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_div\_si}(gr\_ptr res, gr\_sreptr x, slong y, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_div\_fmpz\_ui}(gr\_ptr res, gr\_sreptr x, const fmpz\_t y, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_div\_other}(gr\_ptr res, gr\_sreptr x, gr\_sreptr y, gr\_ctx\_t y\_ctx, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_other\_div}(gr\_ptr res, gr\_sreptr x, gr\_ctx\_t x\_ctx, gr\_sreptr y, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_div\_exact}(gr\_ptr res, gr\_sreptr x, gr\_sreptr y, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_fmpz\_sliding}(gr\_ptr f, gr\_sreptr g, const fmpz\_t pow, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_ui\_sliding}(gr\_ptr f, gr\_sreptr g, ulong pow, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_fmpz\_binexp}(gr\_ptr res, gr\_sreptr x, const fmpz\_t exp, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_ui\_binexp}(gr\_ptr res, gr\_sreptr x, ulong e, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_fmpz}(gr\_ptr res, gr\_sreptr x, const fmpz\_t e, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_si}(gr\_ptr res, gr\_sreptr x, slong e, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_ui}(gr\_ptr res, gr\_sreptr x, ulong e, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_fmpz\_mpoly\_ui}(gr\_ptr res, gr\_sreptr x, const fmpz\_t pow, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_ui\_mpoly\_ui}(gr\_ptr res, gr\_sreptr x, const fmpz\_t exp, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_ui\_binexp}(gr\_ptr res, gr\_sreptr x, const fmpz\_t exp, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_pow\_other}(gr\_ptr res, gr\_sreptr x, gr\_sreptr y, gr\_ctx\_t y\_ctx, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_other\_pow}(gr\_ptr res, gr\_sreptr x, gr\_ctx\_t x\_ctx, gr\_sreptr y, gr\_ctx\_t ctx)

int \texttt{gr\_mpz\_poly\_evaluate\_horner}(gr\_ptr res, const fmpz\_t f, slong len, gr\_sreptr x, gr\_ctx\_t ctx)

int \texttt{gr\_mpz\_poly\_evaluate\_horner\_binary}(gr\_ptr res, const fmpz\_poly\_t f, gr\_sreptr x, gr\_ctx\_t ctx)

int \texttt{gr\_mpz\_poly\_evaluate\_rectangular}(gr\_ptr res, const fmpz\_t f, gr\_sreptr x, gr\_ctx\_t ctx)

int \texttt{gr\_mpz\_poly\_evaluate\_rectangular\_binary}(gr\_ptr res, const fmpz\_poly\_t f, gr\_sreptr x, gr\_ctx\_t ctx)

Sets res to the value of the integer polynomial \( f \) evaluated at the argument \( x \).

\begin{verbatim}
int gr\_mpz\_mpoly\_evaluate\_rectangular\_binary(gr\_ptr res, const fmpz\_poly\_mpoly\_t f, gr\_sreptr x, const fmpz\_poly\_ctx\_t mctx, gr\_ctx\_t ctx)
\end{verbatim}

Sets res to value of the multivariate polynomial \( f \) (with corresponding context object \( mctx \)) evaluated at the vector of arguments in \( x \).

\texttt{truth\_t gr\_generic\_is\_square}(gr\_sreptr x, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_sqrt}(gr\_ptr res, gr\_sreptr x, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_rsqrt}(gr\_ptr res, gr\_sreptr x, gr\_ctx\_t ctx)

Currently these methods check for the special values 0 and 1.

int \texttt{gr\_generic\_numer\_denominator}(gr\_ptr res, gr\_sreptr x, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_cmp}(int *res, gr\_sreptr x, gr\_sreptr y, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_cmp\_ui\_binexp}(int *res, gr\_sreptr x, gr\_sreptr y, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_cmp\_other}(int *res, gr\_sreptr x, gr\_sreptr y, gr\_ctx\_t y\_ctx, gr\_ctx\_t ctx)

int \texttt{gr\_generic\_cmp\_other\_ui\_binexp}(int *res, gr\_sreptr x, gr\_sreptr y, gr\_ctx\_t y\_ctx, gr\_ctx\_t ctx)
3.4.3 Generic special functions

To do: move to gr_special

```c
int gr_generic_bernoulli_ui(gr_ptr res, ulong n, gr_ctx_t ctx)
int gr_generic_bernoulli_fmpz(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_generic_bernoulli_vec(gr_ptr res, slong len, gr_ctx_t ctx)
int gr_generic_eulernum_ui(gr_ptr res, ulong n, gr_ctx_t ctx)
int gr_generic_eulernum_fmpz(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_generic_eulernum_vec(gr_ptr res, slong len, gr_ctx_t ctx)
int gr_generic_stirling_s1u_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_generic_stirling_s1_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_generic_stirling_s2_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_generic_stirling_s1u_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
int gr_generic_stirling_s1_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
int gr_generic_stirling_s2_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
```

3.4.4 Generic vector methods

To do: move to gr_vec

```c
void gr_generic_vec_init(gr_ptr vec, slong len, gr_ctx_t ctx)
void gr_generic_vec_clear(gr_ptr vec, slong len, gr_ctx_t ctx)
void gr_generic_vec_swap(gr_ptr vec1, gr_ptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_vec_zero(gr_ptr vec1, gr_ptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_vec_set(gr_ptr res, gr_srcptr src, slong len, gr_ctx_t ctx)
int gr_generic_vec_neg(gr_ptr res, gr_srcptr src, slong len, gr_ctx_t ctx)
int gr_generic_vec_normalise(slong *res, gr_srcptr vec, slong len, gr_ctx_t ctx)
slong gr_generic_vec_normalise_weak(gr_srcptr vec, slong len, gr_ctx_t ctx)
int gr_generic_vec_mul_scalar_2exp_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int gr_generic_vec_scalar_addmul(gr_ptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_vec_scalar_submul(gr_ptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_vec_scalar_addmul_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int gr_generic_vec_scalar_submul_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
truth_t gr_generic_vec_equal(gr_srcptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_vec_is_zero(gr_srcptr vec, slong len, gr_ctx_t ctx)
int gr_generic_vec_dot(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)
```
int gr_generic_vec_dot_rev(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_dot_ui(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const ulong *vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_dot_si(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const slong *vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_dot_fmpz(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const fmpz_t *vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_set_powers(gr_ptr res, gr_srcptr x, slong len, gr_ctx_t ctx)

int gr_generic_vec_reciprocals(gr_ptr res, slong len, gr_ctx_t ctx)

int gr_generic_vec_add(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)

int gr_generic_vec_sub(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)

int gr_generic_vec_mul(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)

int gr_generic_vec_div(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)

int gr_generic_vec_divexact(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)

int gr_generic_vec_pow(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)

int gr_generic_vec_pow_scalar_ui(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, const ulong *vec2, gr_ctx_t ctx)

int gr_generic_vec_pow_scalar_si(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, const slong *vec2, gr_ctx_t ctx)

int gr_generic_vec_pow_fmpz(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, const fmpz_t *vec2, gr_ctx_t ctx)

int gr_generic_vec_div_scalar_ui(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)

int gr_generic_vec_div_scalar_si(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)

int gr_generic_vec_div_scalar_fmpz(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, const fmpz_t *vec2, gr_ctx_t ctx)

int gr_generic_vec_dot_fmpz(gr_ptr res, gr_srcptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)

int gr_generic_vec_dot-ui(gr_ptr res, slong *vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_dot-rev(gr_ptr res, gr_ptr vec1, slong len, gr_ctx_t ctx)

Chapter 3. Generic rings
3.4. gr_generic.h – basic algorithms and fallback implementations for generic elements

int gr_generic_vec_pow_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int gr_generic_vec_add_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int gr_generic_vec_sub_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int gr_generic_vec_mul_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int gr_generic_vec_div_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int gr_generic_vec_divexact_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int gr_generic_vec_divvec_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)

int gr_generic_vec_pow_other(gr_ptr vec1, gr_srcptr vec2, slong len, const int c, gr_ctx_t ctx)
int gr_generic_vec_add_other(gr_ptr vec1, gr_srcptr vec2, slong len, const int c, gr_ctx_t ctx)
int gr_generic_vec_sub_other(gr_ptr vec1, gr_srcptr vec2, slong len, const int c, gr_ctx_t ctx)
int gr_generic_vec_mul_other(gr_ptr vec1, gr_srcptr vec2, slong len, const int c, gr_ctx_t ctx)
int gr_generic_vec_div_other(gr_ptr vec1, gr_srcptr vec2, slong len, const int c, gr_ctx_t ctx)
int gr_generic_vec_divexact_other(gr_ptr vec1, gr_srcptr vec2, slong len, const int c, gr_ctx_t ctx)
int gr_generic_vec_div_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t cctx, gr_ctx_t ctx)
int gr_generic_vec_divexact_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t cctx, gr_ctx_t ctx)
int gr_generic_vec_pow_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t cctx, gr_ctx_t ctx)
int gr_generic_scalar_other_add_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_sub_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_mul_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_div_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_divexact_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_pow_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len, gr_ctx_t ctx)
3.5 gr_special.h – special arithmetic and transcendental functions

3.5.1 Mathematical constants

```c
int gr_pi(gr_ptr res, gr_ctx_t ctx)
int gr_euler(gr_ptr res, gr_ctx_t ctx)
int gr_catalan(gr_ptr res, gr_ctx_t ctx)
int gr_khinchin(gr_ptr res, gr_ctx_t ctx)
int gr_glaisher(gr_ptr res, gr_ctx_t ctx)
```

Standard real constants: \( \pi \), Euler’s constant \( \gamma \), Catalan’s constant, Khinchin’s constant, Glaisher’s constant.

3.5.2 Elementary functions

```c
int gr_exp(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_expm1(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_exp2(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_exp10(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_exp_pi_i(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log1p(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log2(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log10(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log_pi_i(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sin(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cos(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sin_cos(gr_ptr res1, gr_ptr res2, gr_srcptr x, gr_ctx_t ctx)
int gr_tan(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cot(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sec(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_csc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sin_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cos_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sin_cos_pi(gr_ptr res1, gr_ptr res2, gr_srcptr x, gr_ctx_t ctx)
int gr_tan_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cot_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sec_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_csc_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sinc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sin_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cos_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sin_cos_pi(gr_ptr res1, gr_ptr res2, gr_srcptr x, gr_ctx_t ctx)
int gr_tan_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cot_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sec_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_csc_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sinc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```
int gr_csch(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_asin(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acos(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_atan(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_atan2(gr_ptr res, gr_srcptr y, gr_srcptr x, gr_ctx_t ctx)
int gr_acot(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_asec(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acsc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_asec_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acsc_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_atan_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acosh(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_atanh(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acoth(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_asech(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acsch(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_bin(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_bin_ui(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_bin_uiui(gr_ptr res, ulong y, gr_ctx_t ctx)
int gr_bin_vec(gr_ptr res, gr_srcptr x, ulong len, gr_ctx_t ctx)
int gr_bin_ui_vec(gr_ptr res, ulong y, ulong len, gr_ctx_t ctx)

3.5.3 Factorials and gamma functions

int gr_fac(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_fac_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_fac_fmpz(gr_ptr res, fmpz_t x, gr_ctx_t ctx)
int gr_fac_vec(gr_ptr res, slong len, gr_ctx_t ctx)

Factorial $x!$. The vec version writes the first $len$ consecutive values $1, 2, 6, \ldots, (len - 1)!$ to the preallocated vector $res$.

int gr_rfac(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_rfac_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_rfac_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
int gr_rfac_vec(gr_ptr res, slong len, gr_ctx_t ctx)

Reciprocal factorial. The vec version writes the first $len$ consecutive values $1, 1/2, 1/6, \ldots, 1/(len - 1)!$ to the preallocated vector $res$.

int gr_binui(gr_ptr res, gr_srcptr y, gr_ctx_t ctx)
int gr_binuiui(gr_ptr res, ulong y, gr_ctx_t ctx)
int gr_binvec(gr_ptr res, gr_srcptr x, slong len, gr_ctx_t ctx)
int gr_binui_vec(gr_ptr res, ulong y, slong len, gr_ctx_t ctx)

Binomial coefficient $\binom{x}{y}$. The vec versions write the first $len$ consecutive values $\binom{x}{0}, \binom{x}{1}, \ldots, \binom{x}{len-1}$ to the preallocated vector $res$. For constructing a two-dimensional array of binomial coefficients (Pascal’s triangle), it is more efficient to call gr_mat_pascal() than to call these functions repeatedly.
### 3.5.4 Combinatorial numbers

The `vec` version of functions for number sequences \( c_n \) write the `len` consecutive values \( c_0, c_1, \ldots, c_{len-1} \) to the preallocated vector `res`.

```c
int gr_bernoulli_ui(gr_ptr res, ulong n, gr_ctx_t ctx)
int gr_bernoulli_fmpz(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_bernoulli_vec(gr_ptr res, slong len, gr_ctx_t ctx)
```

Bernoulli numbers \( B_n \).

```c
int gr_eulernum_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_eulernum_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
int gr_eulernum_vec(gr_ptr res, slong len, gr_ctx_t ctx)
```

Euler numbers \( E_n \).

```c
int gr_fib_ui(gr_ptr res, ulong n, gr_ctx_t ctx)
int gr_fib_fmpz(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_fib_vec(gr_ptr res, slong len, gr_ctx_t ctx)
```

Fibonacci numbers \( F_n \).

```c
int gr_stirling_s1u_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_stirling_s1_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_stirling_s2_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_stirling_s1u_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
int gr_stirling_s1_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
```

Rising and falling factorials \( x(x+1) \cdots (x+y-1) \) and \( x(x-1) \cdots (x-y+1) \), or their generalizations to non-integer \( y \) via the gamma function.

```c
int gr_gamma(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_gamma_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
int gr_gamma_fmpq(gr_ptr res, const fmpq_t x, gr_ctx_t ctx)
```

Gamma function \( \Gamma(x) \), its reciprocal \( 1/\Gamma(x) \), the log-gamma function \( \log \Gamma(x) \), and the digamma function \( \psi(x) \).

```c
int gr_barnes_g(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log_barnes_g(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

Barnes \( G \)-function.

```c
int gr_beta(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
```

Beta function \( B(x,y) \).

```c
int gr_doublefac_ui(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

Double factorial \( x!! \).

```c
int gr_harmonic_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
```

Harmonic number \( H_x \).
int gr_stirling_s2_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
Stirling numbers $S(x, y)$: unsigned of the first kind, signed of the first kind, and second kind. The vec versions write the len consecutive values $S(x,0), S(x,1), \ldots, S(x,len - 1)$ to the preallocated vector res. For constructing a two-dimensional array of Stirling numbers, it is more efficient to call gr_mat_stirling() than to call these functions repeatedly.

int gr_bellnum_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_bellnum_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
int gr_bellnum_ui(gr_ptr res, slong len, gr_ctx_t ctx)
Bell numbers $B_n$.

int gr_partitions_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_partitions_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
int gr_partitions_vec(gr_ptr res, slong len, gr_ctx_t ctx)
Partition numbers $p(n)$.

3.5.5 Error function and exponential integrals

int gr_erf(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_erfc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_erfxc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_erfi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_erfinv(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_erfci(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_fresnel_s(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_fresnel_c(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_fresnel_sgr(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_fresnel_cgr(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_gamma_upper(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_beta_lower(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
int gr_exp_integral(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_exp_integral_ei(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sin_integral(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cos_integral(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sinh_integral(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cosh_integral(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_dilog(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

3.5.6 Orthogonal polynomials

int gr_chebyshev_t_fmpz(gr_ptr res, const fmpz_t n, gr_srcptr x, gr_ctx_t ctx)
int gr_chebyshev_t(gr_ptr res, gr_srcptr n, gr_srcptr x, gr_ctx_t ctx)
int gr_chebyshev_u_fmpz(gr_ptr res, const fmpz_t n, gr_srcptr x, gr_ctx_t ctx)
int gr_chebyshev_u(gr_ptr res, gr_srcptr n, gr_srcptr x, gr_ctx_t ctx)
int gr_jacobi_p(gr_ptr res, gr_srcptr n, gr_srcptr a, gr_srcptr b, gr_srcptr z, gr_ctx_t ctx)
int gr_gegenbauer_c(gr_ptr res, gr_srcptr n, gr_srcptr m, gr_srcptr z, gr_ctx_t ctx)

64 Chapter 3. Generic rings
3.5. Bessel, Airy and Coulomb functions

int gr_bessel_j(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_y(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_i(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_k(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_jy(gr_ptr res1, gr_ptr res2, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_i_scaled(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_k_scaled(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_airy(gr_ptr res1, gr_ptr res2, gr_ptr res3, gr_ptr res4, gr_srcptr x, gr_ctx_t ctx)
int gr_airy_ai(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_airy_bi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_airy_ai_prime(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_airy_bi_prime(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_airy_ai_zero(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_airy_bi_zero(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_airy_ai_prime_zero(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_coulomb(gr_ptr res1, gr_ptr res2, gr_ptr res3, gr_ptr res4, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
int gr_coulomb_f(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
int gr_coulomb_g(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
int gr_coulomb_hpos(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
int gr_coulomb_hneg(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)

3.5.8 Hypergeometric functions

int gr_hypgeom_0f1(gr_ptr res, gr_srcptr a, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_hypgeom_1f1(gr_ptr res, gr_srcptr a, gr_srcptr b, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_hypgeom_2f1(gr_ptr res, gr_srcptr a, gr_srcptr b, gr_srcptr c, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_hypgeom_pfq(gr_ptr res, const gr_vec_t a, const gr_vec_t b, gr_srcptr z, int flags, gr_ctx_t ctx)

3.5. gr_special.h – special arithmetic and transcendental functions
3.5.9 Riemann zeta, polylogarithms and Dirichlet L-functions

```c
int gr_zeta(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_zeta_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_hurwitz_zeta(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_polygamma(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_polylog(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_riemann_xi(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
int gr_dirichlet_chi_fmpz(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi, const fmpz_t n, gr_ctx_t ctx)
int gr_dirichlet_chi_vec(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, gr_ctx_t ctx)
int gr_dirichlet_l(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi, gr_srcptr s, gr_ctx_t ctx)
int gr_dirichlet_l_all(gr_vec_t res, const dirichlet_group_t G, gr_srcptr s, gr_ctx_t ctx)
int gr_dirichlet_hardy_theta(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi, gr_srcptr t, gr_ctx_t ctx)
int gr_dirichlet_hardy_z(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi, gr_srcptr t, gr_ctx_t ctx)
```

3.5.10 Elliptic integrals

```c
int gr_agm1(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_agm(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_elliptic_k(gr_ptr res, gr_srcptr m, gr_ctx_t ctx)
int gr_elliptic_e(gr_ptr res, gr_srcptr m, gr_ctx_t ctx)
int gr_elliptic_pi(gr_ptr res, gr_srcptr n, gr_srcptr m, gr_ctx_t ctx)
int gr_elliptic_f(gr_ptr res, gr_srcptr phi, gr_srcptr m, int pi, gr_ctx_t ctx)
int gr_elliptic_e_inc(gr_ptr res, gr_srcptr phi, gr_srcptr m, int pi, gr_ctx_t ctx)
int gr_elliptic_pi_inc(gr_ptr res, gr_srcptr n, gr_srcptr phi, gr_srcptr m, int pi, gr_ctx_t ctx)
int gr_carlson_rc(gr_ptr res, gr_srcptr x, gr_srcptr y, int flags, gr_ctx_t ctx)
int gr_carlson_rf(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_carlson_rd(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_carlson_rg(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_carlson_rj(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_srcptr w, int flags, gr_ctx_t ctx)
```
3.5.11 Elliptic, modular and theta functions

```c
int gr_jacobi_theta(gr_ptr res1, gr_ptr res2, gr_ptr res3, gr_ptr res4, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_jacobi_theta_1(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_jacobi_theta_2(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_jacobi_theta_3(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_jacobi_theta_4(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_dedekind_eta(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)
int gr_dedekind_eta_q(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)
int gr_modular_j(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)
int gr_modular_lambda(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)
int gr_modular_delta(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)
int gr_hilbert_class_poly(gr_ptr res, slong D, gr_srcptr x, gr_ctx_t ctx)
int gr_eisenstein_e(gr_ptr res, ulong n, gr_srcptr tau, gr_ctx_t ctx)
int gr_eisenstein_g(gr_ptr res, ulong n, gr_srcptr tau, gr_ctx_t ctx)
int gr_eisenstein_g_vec(gr_ptr res, gr_srcptr tau, slong len, gr_ctx_t ctx)
int gr_elliptic_invariants(gr_ptr res1, gr_ptr res2, gr_srcptr tau, gr_ctx_t ctx)
int gr_elliptic_roots(gr_ptr res1, gr_ptr res2, gr_ptr res3, gr_srcptr tau, gr_ctx_t ctx)
int gr_weierstrass_p(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_weierstrass_p_prime(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_weierstrass_p_inv(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_weierstrass_zeta(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_weierstrass_sigma(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
```
3.6 gr_vec.h – vectors over generic rings

3.6.1 Types and basic operations

type gr_vec_struct

type gr_vec_t

void gr_vec_init(gr_vec_t vec, slong len, gr_ctx_t ctx)

Initializes vec to a vector of length len with elements in the ring ctx. The length must be nonnegative. All entries are set to zero.

void gr_vec_clear(gr_vec_t vec, gr_ctx_t ctx)

Clears the vector vec.

GR_VEC_ENTRY(vec, i, sz)

Macro to access the i-th element in the vector vec, indexed from zero, assuming that entries have size sz. The index must be in bounds.

gr_ptr gr_vec_entry_ptr(gr_vec_t vec, slong i, gr_ctx_t ctx)

Returns a pointer to the i-th element in the vector vec, indexed from zero. The index must be in bounds.

slong gr_vec_length(const gr_vec_t vec, gr_ctx_t ctx)

Returns the length of the vector vec.

void gr_vec_fit_length(gr_vec_t vec, slong len, gr_ctx_t ctx)

Allocates space for at least len elements in the vector vec. This does not change the size of the vector.

void gr_vec_set_length(gr_vec_t vec, slong len, gr_ctx_t ctx)

Resizes the vector to length len, which must be nonnegative. The vector will be extended with zeros.

int gr_vec_set(gr_vec_t res, const gr_vec_t src, gr_ctx_t ctx)

Sets res to a copy of the vector src.

int gr_vec_append(gr_vec_t vec, gr_srcptr x, gr_ctx_t ctx)

Appends the element x to the end of vector vec.

int _gr_vec_write(gr_stream_t out, gr_srcptr vec, slong len, gr_ctx_t ctx)

int gr_vec_write(gr_stream_t out, const gr_vec_t vec, gr_ctx_t ctx)

int gr_vec_print(const gr_vec_t vec, gr_ctx_t ctx)

GR_ENTRY(vec, i, size)

Macro to access the i-th entry of a gr_ptr or gr_srcptr vector vec, where each element is size bytes.

void _gr_vec_init(gr_ptr vec, slong len, gr_ctx_t ctx)

Initialize len elements of vec to the value 0. The pointer vec must already refer to allocated memory.

void _gr_vec_clear(gr_ptr vec, slong len, gr_ctx_t ctx)

Clears len elements of vec. This frees memory allocated by individual elements, but does not free the memory allocated by vec itself.

void _gr_vec_swap(gr_ptr vec1, gr_ptr vec2, slong len, gr_ctx_t ctx)

Swap the entries of vec1 and vec2.

int _gr_vec_randtest(gr_ptr res, flint_rand_t state, slong len, gr_ctx_t ctx)
3.6. gr_vec.h – vectors over generic rings

3.6.2 Arithmetic

int _gr_vec_set(gr_ptr res, gr_srcptr src, slong len, gr_ctx_t ctx)

truth_t _gr_vec_equal(gr_srcptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)

int _gr_vec_zero(gr_ptr vec, slong len, gr_ctx_t ctx)

truth_t _gr_vec_is_zero(gr_srcptr vec, slong len, gr_ctx_t ctx)

int _gr_vec_normalise(slong *res, gr_srcptr vec, slong len, gr_ctx_t ctx)

slong _gr_vec_normalise_weak(gr_srcptr vec, slong len, gr_ctx_t ctx)

Binary operations applied elementwise.

int _gr_vec_add_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)

int _gr_vec_sub_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)

int _gr_vec_mul_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)

int _gr_vec_div_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)

int _gr_vec_add_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, slong len, gr_ctx_t ctx)

int _gr_vec_sub_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, slong len, gr_ctx_t ctx)

int _gr_vec_mul_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, slong len, gr_ctx_t ctx)

int _gr_vec_div_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, slong len, gr_ctx_t ctx)

int _gr_vec_pow_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, slong len, gr_ctx_t ctx)

int _gr_other_add_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len, gr_ctx_t ctx)

int _gr_other_add_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx3, slong len, gr_ctx_t ctx)
Flint Documentation, Release 3.1.0-dev

70 Chapter 3. Generic rings

int _gr_other_sub_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len, gr_ctx_t ctx)
int _gr_other_mul_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len, gr_ctx_t ctx)
int _gr_other_div_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len, gr_ctx_t ctx)
int _gr_other_divexact_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len, gr_ctx_t ctx)
int _gr_other_pow_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len, gr_ctx_t ctx)

Binary operations applied elementwise, allowing a different type for one of the vectors.

int _gr_vec_add_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_sub_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_mul_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_div_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_pow_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_divexact_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)

Flint Documentation, Release 3.1.0-dev
int _gr_vec_mul_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_div_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_divexact_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_pow_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_add_scalar_fmpq(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpq_t c, gr_ctx_t ctx)
int _gr_vec_sub_scalar_fmpq(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpq_t c, gr_ctx_t ctx)
int _gr_vec_mul_scalar_fmpq(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpq_t c, gr_ctx_t ctx)
int _gr_vec_div_scalar_fmpq(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpq_t c, gr_ctx_t ctx)
int _gr_vec_divexact_scalar_fmpq(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpq_t c, gr_ctx_t ctx)
int _gr_vec_pow_scalar_fmpq(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpq_t c, gr_ctx_t ctx)

3.6.3 Sums and products

int _gr_vec_sum(gr_ptr res, gr_srcptr vec, slong len, gr_ctx_t ctx)
int _gr_vec_product(gr_ptr res, gr_srcptr vec, slong len, gr_ctx_t ctx)

3.6.4 Dot products

int _gr_vec_dot(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int _gr_vec_dot_si(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const slong *vec2, slong len, gr_ctx_t ctx)
int _gr_vec_dot_ui(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const ulong *vec2, slong len, gr_ctx_t ctx)
int _gr_vec_dot_fmpz(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const fmpz *vec2, slong len, gr_ctx_t ctx)

Sets res to $c \pm \sum_{i=0}^{n-1} a_i b_i$.

int _gr_vec_dot_rev(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)

Sets res to $c \pm \sum_{i=0}^{n-1} a_i b_{n-1-i}$.
3.6.5 Other functions

int _gr_vec_step (gr_ptr vec, gr_srcptr start, gr_srcptr step, slong len, gr_ctx_t ctx)

int _gr_vec_reciprocals (gr_ptr res, slong len, gr_ctx_t ctx)

Sets res to the vector of reciprocals of the positive integers 1, 2, ... up to len inclusive.

int _gr_vec_set_powers (gr_ptr res, gr_srcptr x, slong len, gr_ctx_t ctx)
3.7 gr_mat.h – dense matrices over generic rings

A \texttt{gr_mat_t} represents a matrix implemented as a dense array of entries in a generic ring \(R\).

- In this module, the context object \texttt{ctx} always represents the coefficient ring \(R\) unless otherwise stated. Creating a context object representing a matrix space only becomes necessary when one wants to manipulate matrices using generic ring methods like \texttt{gr_add} instead of the designated matrix methods like \texttt{gr_mat_add}.

- Matrix functions generally assume that input as well as output operands have compatible shapes. Some functions return \texttt{GR_DOMAIN} for matrices with the wrong shape, but this is not always consistent.

- Some operations (like rank, LU factorization) generally only make sense when the base ring is an integral domain. Typically the algorithms designed for integral domains also work over non-integral domains as long as all inversions of nonzero elements succeed. If an inversion fails, the algorithm will return the \texttt{GR_DOMAIN} or \texttt{GR_UNABLE} flag. This might not yet be entirely consistent.

3.7.1 Type compatibility

The \texttt{gr_mat} type has the same data layout as most Flint, Arb and Calcium matrix types. Methods in this module can therefore be mixed freely with methods in the corresponding Flint, Arb and Calcium modules when the underlying coefficient type is the same.

It is not directly compatible with the \texttt{nmod_mat} type, which stores modulus data as part of the matrix object.

3.7.2 Types, macros and constants

\texttt{type gr_mat_struct}

\texttt{type gr_mat_t}

Contains a pointer to an array of coefficients \texttt{(entries)}, the number of rows \texttt{(r)}, the number of columns \texttt{(c)}, and an array to pointers marking the start of each row \texttt{(rows)}.

A \texttt{gr_mat_t} is defined as an array of length one of type \texttt{gr_mat_struct}, permitting a \texttt{gr_mat_t} to be passed by reference.

\texttt{GR_MAT_ENTRY(mat, i, j, sz)}

Macro to access the entry at row \texttt{i} and column \texttt{j} of the matrix \texttt{mat} whose entries have size \texttt{sz} bytes.

\texttt{gr_ptr gr_mat_entry_ptr(gr_mat_t mat, slong i, slong j, gr_ctx_t ctx)}

Function returning a pointer to the entry at row \texttt{i} and column \texttt{j} of the matrix \texttt{mat}. The indices must be in bounds.

\texttt{gr_mat_nrows(mat, ctx)}

Macro accessing the number of rows of \texttt{mat}.

\texttt{gr_mat_ncols(mat, ctx)}

Macro accessing the number of columns of \texttt{mat}.
3.7.3 Memory management

void \texttt{gr\_mat\_init}(\texttt{gr\_mat\_t mat, slong rows, slong cols, gr\_ctx\_t ctx})

Initializes \texttt{mat} to a matrix with the given number of rows and columns.

int \texttt{gr\_mat\_init\_set}(\texttt{gr\_mat\_t res, const gr\_mat\_t mat, gr\_ctx\_t ctx})

Initializes \texttt{res} to a copy of the matrix \texttt{mat}.

void \texttt{gr\_mat\_clear}(\texttt{gr\_mat\_t mat, gr\_ctx\_t ctx})

Clears the matrix.

void \texttt{gr\_mat\_swap}(\texttt{gr\_mat\_t mat1, gr\_mat\_t mat2, gr\_ctx\_t ctx})

Swaps \texttt{mat1} and \texttt{mat2} efficiently.

int \texttt{gr\_mat\_swap\_entrywise}(\texttt{gr\_mat\_t mat1, const gr\_mat\_t mat2, gr\_ctx\_t ctx})

Performs a deep swap of \texttt{mat1} and \texttt{mat2}, swapping the individual entries rather than the top-level structures.

3.7.4 Window matrices

void \texttt{gr\_mat\_window\_init}(\texttt{gr\_mat\_t window, const gr\_mat\_t mat, slong r1, slong c1, slong r2, slong c2, gr\_ctx\_t ctx})

Initializes \texttt{window} to a window matrix into the submatrix of \texttt{mat} starting at the corner at row \texttt{r1} and column \texttt{c1} (inclusive) and ending at row \texttt{r2} and column \texttt{c2} (exclusive). The indices must be within bounds.

void \texttt{gr\_mat\_window\_clear}(\texttt{gr\_mat\_t window, gr\_ctx\_t ctx})

Frees the window matrix.

3.7.5 Input and output

int \texttt{gr\_mat\_write}(\texttt{gr\_stream\_t out, const gr\_mat\_t mat, gr\_ctx\_t ctx})

Write \texttt{mat} to the stream \texttt{out}.

int \texttt{gr\_mat\_print}(\texttt{const gr\_mat\_t mat, gr\_ctx\_t ctx})

Prints \texttt{mat} to standard output.

3.7.6 Comparisons

\texttt{truth\_t gr\_mat\_equal}(\texttt{const gr\_mat\_t mat1, const gr\_mat\_t mat2, gr\_ctx\_t ctx})

Returns whether \texttt{mat1} and \texttt{mat2} are equal.

3.7.7 Assignment and special values

\texttt{truth\_t gr\_mat\_is\_zero}(\texttt{const gr\_mat\_t mat, gr\_ctx\_t ctx})

\texttt{truth\_t gr\_mat\_is\_one}(\texttt{const gr\_mat\_t mat, gr\_ctx\_t ctx})

\texttt{truth\_t gr\_mat\_is\_neg\_one}(\texttt{const gr\_mat\_t mat, gr\_ctx\_t ctx})

Returns whether \texttt{mat} respectively is the zero matrix or the scalar matrix with 1 or -1 on the main diagonal.

\texttt{truth\_t gr\_mat\_is\_scalar}(\texttt{const gr\_mat\_t mat, gr\_ctx\_t ctx})

Returns whether \texttt{mat} is a scalar matrix, being a diagonal matrix with identical elements on the main diagonal.
int gr_mat_zero(gr_mat_t res, gr_ctx_t ctx)
    Sets res to the zero matrix.

int gr_mat_one(gr_mat_t res, gr_ctx_t ctx)
    Sets res to the scalar matrix with 1 on the main diagonal and zero elsewhere.

int gr_mat_set(gr_mat_t res, const gr_mat_t mat, gr_ctx_t ctx)
    Sets res to the value of mat.

int gr_mat_set_fmpz(gr_mat_t res, const fmpz_t c, gr_ctx_t ctx)
    Set res to the scalar matrix with c on the main diagonal and zero elsewhere.

int gr_mat_set_fmpq(gr_mat_t res, const fmpq_t c, gr_ctx_t ctx)

3.7.8 Basic row, column and entry operations

int gr_mat_concat_horizontal(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)
    Concatenates the horizontal matrices mat1 and mat2.

int gr_mat_concat_vertical(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)
    Concatenates the vertical matrices mat1 and mat2.

int gr_mat_transpose(gr_mat_t B, const gr_mat_t A, gr_ctx_t ctx)
    Sets B to the transpose of A.

int gr_mat_swap_rows(gr_mat_t mat, slong *perm, slong r, slong s, gr_ctx_t ctx)
    Swaps rows r and s of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

int gr_mat_swap_cols(gr_mat_t mat, slong *perm, slong r, slong s, gr_ctx_t ctx)
    Swaps columns r and s of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.

int gr_mat_invert_rows(gr_mat_t mat, slong *perm, gr_ctx_t ctx)
    Swaps rows i and r - i of mat for 0 <= i < r/2, where r is the number of rows of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

int gr_mat_invert_cols(gr_mat_t mat, slong *perm, gr_ctx_t ctx)
    Swaps columns i and c - i of mat for 0 <= i < c/2, where c is the number of columns of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.

truth_t gr_mat_is_empty(const gr_mat_t mat, gr_ctx_t ctx)
    Returns whether mat is an empty matrix, having either zero rows or zero column. This predicate is always decidable (even if the underlying ring is not computable), returning T_TRUE or T_FALSE.

truth_t gr_mat_is_square(const gr_mat_t mat, gr_ctx_t ctx)
    Returns whether mat is a square matrix, having the same number of rows as columns (not the same thing as being a perfect square!). This predicate is always decidable (even if the underlying ring is not computable), returning T_TRUE or T_FALSE.
3.7.9 Arithmetic

```c
int gr_mat_neg(gr_mat_t res, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_add(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)
int gr_mat_sub(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)
int gr_mat_mul_classical(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)
int gr_mat_mul_strassen(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)
int gr_mat_mul_generic(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)
int gr_mat_mul(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)
```

Matrix multiplication. The default function can be overloaded by specific rings; otherwise, it falls back to `gr_mat_mul_generic()` which currently only performs classical multiplication.

```c
int gr_mat_sqr(gr_mat_t res, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_add_scalar(gr_mat_t res, const gr_mat_t mat, gr_srcptr c, gr_ctx_t ctx)
int gr_mat_sub_scalar(gr_mat_t res, const gr_mat_t mat, gr_srcptr c, gr_ctx_t ctx)
int gr_mat_mul_scalar(gr_mat_t res, const gr_mat_t mat, gr_srcptr c, gr_ctx_t ctx)
int gr_mat_addmul_scalar(gr_mat_t res, const gr_mat_t mat, gr_srcptr c, gr_ctx_t ctx)
int gr_mat_submul_scalar(gr_mat_t res, const gr_mat_t mat, gr_srcptr c, gr_ctx_t ctx)
int gr_mat_div_scalar(gr_mat_t res, const gr_mat_t mat, gr_srcptr c, gr_ctx_t ctx)
```

Sets res to the matrix obtained by evaluating the scalar polynomial poly with matrix argument mat.

```c
int _gr_mat_gr_poly_evaluate(gr_mat_t res, gr_srcptr poly, slong len, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_gr_poly_evaluate(gr_mat_t res, const gr_poly_t poly, const gr_mat_t mat, gr_ctx_t ctx)
```

3.7.10 Diagonal and triangular matrices

```c
truth_t gr_mat_is_upper_triangular(const gr_mat_t mat, gr_ctx_t ctx)
truth_t gr_mat_is_lower_triangular(const gr_mat_t mat, gr_ctx_t ctx)
truth_t gr_mat_is_diagonal(const gr_mat_t mat, gr_ctx_t ctx)
```

Returns whether mat is upper (respectively lower) triangular, having zeros everywhere below (respectively above) the main diagonal. The matrix need not be square.

```c
int gr_mat_mul_diag(gr_mat_t res, const gr_mat_t A, const gr_vec_t D, gr_ctx_t ctx)
int gr_mat_diag_mul(gr_mat_t res, const gr_vec_t D, const gr_mat_t A, gr_ctx_t ctx)
```

Set res to the product $AD$ or $DA$ respectively, where $D$ is a diagonal matrix represented as a vector of entries.
3.7.11 Gaussian elimination

int gr_mat_find_nonzero_pivot_large_abs(slong *pivot_row, gr_mat_t mat, slong start_row, slong end_row, slong column, gr_ctx_t ctx)

int gr_mat_find_nonzero_pivot_generic(slong *pivot_row, gr_mat_t mat, slong start_row, slong end_row, slong column, gr_ctx_t ctx)

int gr_mat_find_nonzero_pivot(slong *pivot_row, gr_mat_t mat, slong start_row, slong end_row, slong column, gr_ctx_t ctx)

Attempts to find a nonzero element in column number column of the matrix mat in a row between start_row (inclusive) and end_row (exclusive). On success, sets pivot_row to the row index and returns GR_SUCCESS. If no nonzero pivot element exists, returns GR_DOMAIN. If no nonzero pivot element exists and zero-testing fails for some element, returns the flag GR_UNABLE.

This function may be destructive: any elements that are nontrivially zero but can be certified zero may be overwritten by exact zeros.

int gr_mat_lu_classical(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check, gr_ctx_t ctx)

int gr_mat_lu_recursive(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check, gr_ctx_t ctx)

int gr_mat_lu(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check, gr_ctx_t ctx)

Computes a generalized LU decomposition \( A = PLU \) of a given matrix \( A \), writing the rank of \( A \) to rank.

If \( A \) is a nonsingular square matrix, \( LU \) will be set to a unit diagonal lower triangular matrix \( L \) and an upper triangular matrix \( U \) (the diagonal of \( L \) will not be stored explicitly).

If \( A \) is an arbitrary matrix of rank \( r \), \( U \) will be in row echelon form having \( r \) nonzero rows, and \( L \) will be lower triangular but truncated to \( r \) columns, having implicit ones on the \( r \) first entries of the main diagonal. All other entries will be zero.

If a nonzero value for rank_check is passed, the function will abandon the output matrix in an undefined state and set the rank to 0 if \( A \) is detected to be rank-deficient. This currently only works as expected for square matrices.

The algorithm can fail if it fails to certify that a pivot element is zero or nonzero, in which case the correct rank cannot be determined. It can also fail if a pivot element is not invertible. In these cases the GR_UNABLE and/or GR_DOMAIN flags will be returned. On failure, the data in the output variables rank, P and LU will be meaningless.

The classical version uses iterative Gaussian elimination. The recursive version uses a block recursive algorithm to take advantage of fast matrix multiplication.

int gr_mat_fflu(slong *rank, slong *P, gr_mat_t LU, gr_ptr den, const gr_mat_t A, int rank_check, gr_ctx_t ctx)

Similar to gr_mat_lu(), but computes a fraction-free LU decomposition using the Bareiss algorithm. The denominator is written to den.

3.7.12 Solving

int gr_mat_nonsingular_solve_tril_classical(gr_mat_t X, const gr_mat_t L, const gr_mat_t B, int unit, gr_ctx_t ctx)

int gr_mat_nonsingular_solve_tril_recursive(gr_mat_t X, const gr_mat_t L, const gr_mat_t B, int unit, gr_ctx_t ctx)

int gr_mat_nonsingular_solve_tril(gr_mat_t X, const gr_mat_t L, const gr_mat_t B, int unit, gr_ctx_t ctx)

int gr_mat_nonsingular_solve_triu_classical(gr_mat_t X, const gr_mat_t L, const gr_mat_t B, int unit, gr_ctx_t ctx)
Solves the lower triangular system $LX = B$ or the upper triangular system $UX = B$, respectively. Division by the diagonal entries must be possible; if not a division fails, `GR_DOMAIN` is returned even if the system is solvable. If `unit` is set, the main diagonal of $L$ or $U$ is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The classical versions perform the computations iteratively while the recursive versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

Solves $AX = B$. If $A$ is not invertible, returns `GR_DOMAIN` even if the system has a solution.

Solves $AX = B$ given a precomputed FFLU or LU factorization of $A$.

Solves $AX = B$ over the fraction field of the present ring (assumed to be an integral domain), returning $X$ with an implied denominator `den`. If $A$ is not invertible over the fraction field, returns `GR_DOMAIN` even if the system has a solution.

Solves $AX = B$ where $A$ is not necessarily square and not necessarily invertible. Assuming that the ring is a field, a return value of `GR_DOMAIN` indicates that the system has no solution. If there are multiple solutions, an arbitrary solution is returned.

### 3.7.13 Determinant and trace

Sets `res` to the determinant of the square matrix `mat`. Various algorithms are available:

- The `berkowitz` version uses the division-free Berkowitz algorithm performing $O(n^4)$ operations. Since no zero tests are required, it is guaranteed to succeed if the ring arithmetic succeeds.
• The cofactor version performs cofactor expansion. This is currently only supported for matrices up to size 4, and for larger matrices returns the GR_UNABLE flag.

• The lu and ff lu versions use rational LU decomposition and fraction-free LU decomposition (Bareiss algorithm) respectively, requiring $O(n^3)$ operations. These algorithms can fail if zero certification or inversion fails, in which case the GR_UNABLE flag is returned.

• The generic, generic_field and generic_integral_domain versions choose an appropriate algorithm for a generic ring depending on the availability of division.

• The default method can be overloaded.

If the matrix is not square, GR_DOMAIN is returned.

int gr_mat_trace(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)

Sets res to the trace (sum of entries on the main diagonal) of the square matrix mat. If the matrix is not square, GR_DOMAIN is returned.

3.7.14 Rank

int gr_mat_rank_fflu(slong *rank, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_rank_lu(slong *rank, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_rank(slong *rank, const gr_mat_t mat, gr_ctx_t ctx)

Sets res to the rank of mat. The default method returns GR_DOMAIN if the element ring is not an integral domain, in which case the usual rank is not well-defined. The ff lu and lu variants currently do not check the element domain, and simply return this flag if they encounter an impossible inverse in the execution of the respective algorithms.

3.7.15 Row echelon form

int gr_mat_rref_lu(slong *rank, gr_mat_t R, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_rref_fflu(slong *rank, gr_mat_t R, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_rref(slong *rank, gr_mat_t R, const gr_mat_t A, gr_ctx_t ctx)

Sets R to the reduced row echelon form of A, also setting rank to its rank.

int gr_mat_rref_den_fflu(slong *rank, gr_mat_t R, gr_ptr den, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_rref_den(slong *rank, gr_mat_t R, gr_ptr den, const gr_mat_t A, gr_ctx_t ctx)

Like rref, but computes the reduced row echelon multiplied by a common (not necessarily minimal) denominator which is written to den. This can be used to compute the rref over an integral domain which is not a field.

3.7.16 Nullspace

int gr_mat_nullspace(gr_mat_t X, const gr_mat_t A, gr_ctx_t ctx)

Sets X to a basis for the (right) nullspace of A. On success, the output matrix will be resized to the correct number of columns.

The basis is not guaranteed to be presented in a canonical or minimal form.

If the ring is not a field, this is implied to compute a nullspace basis over the fraction field. The result may be meaningless if the ring is not an integral domain.
3.7.17 Inverse and adjugate

int gr_mat_inv(gr_mat_t res, const gr_mat_t mat, gr_ctx_t ctx)

Sets res to the inverse of mat, computed by solving $AA^{-1} = I$.

Returns GR_DOMAIN if it can be determined that mat is not invertible over the present ring (warning: this may not work over non-integral domains). If invertibility cannot be proved, returns GR_UNABLE.

To compute the inverse over the fraction field, one may use gr_mat_nonsingular_solve_den() or gr_mat_adjugate().

int gr_mat_adjugate_charpoly(gr_mat_t adj, gr_ptr det, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_adjugate_cofactor(gr_mat_t adj, gr_ptr det, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_adjugate(gr_mat_t adj, gr_ptr det, const gr_mat_t mat, gr_ctx_t ctx)

Sets adj to the adjugate matrix of mat, simultaneously setting det to the determinant of mat. We have $\text{adj}(AA) = A \text{adj}(A)\det(A)$ and $A^{-1} = \text{adj}(A)/\det(A)$ when A is invertible.

The cofactor version uses cofactor expansion, requiring the evaluation of $n^2$ determinants. The charpoly version computes and then evaluates the characteristic polynomial, requiring $O(n^{1/2})$ matrix multiplications plus $O(n^3)$ or $O(n^4)$ operations for the characteristic polynomial itself depending on the algorithm used.

3.7.18 Characteristic polynomial

int _gr_mat_charpoly(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_charpoly(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)

Computes the characteristic polynomial using a default algorithm choice. The underscore method assumes that res is a preallocated array of $n + 1$ coefficients.

int _gr_mat_charpoly_berkowitz(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_charpoly_berkowitz(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)

Sets res to the characteristic polynomial of the square matrix mat, computed using the division-free Berkowitz algorithm. The number of operations is $O(n^4)$ where $n$ is the size of the matrix.

int _gr_mat_charpoly_danilevsky_inplace(gr_ptr res, gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_charpoly_danilevsky(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
int _gr_mat_charpoly_danilevsky(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)
int gr_mat_charpoly_danilevsky(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)

Sets res to the characteristic polynomial of the square matrix mat, computed using the Danilevsky algorithm, Hessenberg reduction using Gaussian elimination, and Hessenberg reduction using Householder reflections. The number of operations of each method is $O(n^3)$ where $n$ is the size of the matrix. The inplace version overwrites the input matrix.

These methods require divisions and can therefore fail when the ring is not a field. They also require zero tests. The householder version also requires square roots. The flags GR_UNABLE or GR_DOMAIN are returned when an impossible division or square root is encountered or when a comparison cannot be performed.
int gr_mat_charpoly_faddeev_bsgs(gr_poly_t res, gr_mat_t adj, const gr_mat_t mat, gr_ctx_t ctx)

Sets res to the characteristic polynomial of the square matrix mat, computed using the Faddeev-LeVerrier algorithm. If the optional output argument adj is not NULL, it is set to the adjugate matrix, which is computed free of charge.

The bsgs version uses a baby-step giant-step strategy, also known as the Preparata-Sarwate algorithm. This reduces the complexity from $O(n^4)$ to $O(n^{3.5})$ operations at the cost of requiring $n^{0.5}$ temporary matrices to be stored.

This method requires divisions by small integers and can therefore fail (returning the GR_UNABLE or GR_DOMAIN flags) in finite characteristic or when the underlying ring does not implement a division algorithm.

int _gr_mat_charpoly_from_hessenberg(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)

int gr_mat_charpoly_from_hessenberg(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)

Sets res to the characteristic polynomial of the square matrix mat, which is assumed to be in Hessenberg form (this is currently not checked).

### 3.7.19 Minimal polynomial

int gr_mat_minpoly_field(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)

Compute the minimal polynomial of the matrix mat. The algorithm assumes that the coefficient ring is a field.

### 3.7.20 Similarity transformations

int gr_mat_apply_row_similarity(gr_mat_t M, slong r, gr_ptr d, gr_ctx_t ctx)

Applies an elementary similarity transform to the $n \times n$ matrix $M$ in-place.

If $P$ is the $n \times n$ identity matrix the zero entries of whose row $r$ (0-indexed) have been replaced by $d$, this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

### 3.7.21 Eigenvalues

int gr_mat_eigenvalues(gr_vec_t lambda, gr_vec_t mult, const gr_mat_t mat, int flags, gr_ctx_t ctx)

int gr_mat_eigenvalues_other(gr_vec_t lambda, gr_vec_t mult, const gr_mat_t mat, gr_ctx_t mat_ctx, int flags, gr_ctx_t ctx)

Finds all eigenvalues of the given matrix in the ring defined by ctx, storing the eigenvalues without duplication in lambda (a vector with elements of type ctx) and the corresponding multiplicities in mult (a vector with elements of type fmpz).

The interface is essentially the same as that of gr_poly_roots(); see its documentation for details.

int gr_mat_diagonalization_precomp(gr_vec_t D, gr_mat_t L, gr_mat_t R, const gr_mat_t A, const gr_vec_t eigenvalues, const gr_vec_t mult, gr_ctx_t ctx)

int gr_mat_diagonalization_generic(gr_vec_t D, gr_mat_t L, gr_mat_t R, const gr_mat_t A, int flags, gr_ctx_t ctx)

int gr_mat_diagonalization(gr_vec_t D, gr_mat_t L, gr_mat_t R, const gr_mat_t A, int flags, gr_ctx_t ctx)

3.7. gr_mat.h – dense matrices over generic rings
Computes a diagonalization \( LAR = D \) given a square matrix \( A \), where \( D \) is a diagonal matrix (returned as a vector) of the eigenvalues repeated according to their multiplicities, \( L \) is a matrix of left eigenvectors, and \( R \) is a matrix of right eigenvectors, normalized such that \( L = R^{-1} \). This implies that \( A = RDL = RDR^{-1} \). Either \( L \) or \( R \) (or both) can be set to NULL to omit computing the respective matrix.

If the matrix has entries in a field then a return flag of GR_DOMAIN indicates that the matrix is non-diagonalizable over this field.

The precomp version requires as input a precomputed set of eigenvalues with corresponding multiplicities, which can be computed with \( gr\text{-}mat\_eigenvalues() \).

### 3.7.22 Jordan decomposition

```c
int gr_mat_set_jordan_blocks(gr_mat_t mat, const gr_vec_t lambda, slong num_blocks, slong *block_lambda, slong *block_size, gr_ctx_t ctx)
int gr_mat_jordan_blocks(gr_vec_t lambda, slong *num_blocks, slong *block_lambda, slong *block_size, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_jordan_transformation(gr_mat_t mat, const gr_vec_t lambda, slong num_blocks, slong *block_lambda, slong *block_size, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_jordan_form(gr_mat_t J, gr_mat_t P, const gr_mat_t A, gr_ctx_t ctx)
```

### 3.7.23 Matrix functions

```c
int gr_mat_exp_jordan(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_exp(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_log_jordan(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_log(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
```

### 3.7.24 Hessenberg form

```c
truth_t gr_mat_is_hessenberg(const gr_mat_t mat, gr_ctx_t ctx)

Sets \( res \) to an upper Hessenberg form of \( mat \). The gauss version uses Gaussian elimination. The householder version uses Householder reflections.

These methods require divisions and zero testing and can therefore fail (returning GR_UNABLE or GR_DOMAIN) when the ring is not a field. The householder version additionally requires complex conjugation and the ability to compute square roots.
3.7.25 Random matrices

```c
int gr_mat_randtest(gr_mat_t res, flint_rand_t state, gr_ctx_t ctx)
Sets res to a random matrix. The distribution is nonuniform.
```

```c
int gr_mat_randops(gr_mat_t mat, flint_rand_t state, slong count, gr_ctx_t ctx)
Randomises mat in-place by performing elementary row or column operations. More precisely, at most count random additions or subtractions of distinct rows and columns will be performed.
```

```c
int gr_mat_randpermdiag(int *parity, gr_mat_t mat, flint_rand_t state, gr_ptr diag, slong n, gr_ctx_t ctx)
Sets mat to a random permutation of the diagonal matrix with n leading entries given by the vector diag. Returns GR_DOMAIN if the main diagonal of mat does not have room for at least n entries. The parity (0 or 1) of the permutation is written to parity.
```

```c
int gr_mat_randrank(gr_mat_t mat, flint_rand_t state, slong rank, gr_ctx_t ctx)
Sets mat to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank. The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling gr_mat_randops() .
This operation only makes sense over integral domains (currently not checked).
```

3.7.26 Special matrices

For the following functions, the user supplies an output matrix with the intended number of rows and columns.

```c
int gr_mat_ones(gr_mat_t res, gr_ctx_t ctx)
Sets all entries in res to one.
```

```c
int gr_mat_pascal(gr_mat_t res, int triangular, gr_ctx_t ctx)
Sets res to a Pascal matrix, whose entries are binomial coefficients. If triangular is 0, constructs a full symmetric matrix with the rows of Pascal’s triangle as successive antidiagonals. If triangular is 1, constructs the upper triangular matrix with the rows of Pascal’s triangle as columns, and if triangular is -1, constructs the lower triangular matrix with the rows of Pascal’s triangle as rows.
```

```c
int gr_mat_stirling(gr_mat_t res, int kind, gr_ctx_t ctx)
Sets res to a Stirling matrix, whose entries are Stirling numbers. If kind is 0, the entries are set to the unsigned Stirling numbers of the first kind. If kind is 1, the entries are set to the signed Stirling numbers of the first kind. If kind is 2, the entries are set to the Stirling numbers of the second kind.
```

```c
int gr_mat_hilbert(gr_mat_t res, gr_ctx_t ctx)
Sets res to the Hilbert matrix, which has entries 1/(i + j + 1) for i, j ≥ 0.
```

```c
int gr_mat_hadamard(gr_mat_t res, gr_ctx_t ctx)
If possible, sets res to a Hadamard matrix of the provided size and returns GR_SUCCESS. Returns GR_DOMAIN if no Hadamard matrix of the given size exists, and GR_UNABLE if the implementation does not know how to construct a Hadamard matrix of the given size.
A Hadamard matrix of size n can only exist if n is 0, 1, 2, or a multiple of 4. It is not known whether a Hadamard matrix exists for every size that is a multiple of 4. This function uses the Paley construction, which succeeds for all n of the form n = 2^e or n = 2^e(q + 1) where q is an odd prime power. Orders n for which Hadamard matrices are known to exist but for which this construction fails are 92, 116, 156, ... (OEIS A046116).
```
3.7.27 Helper functions for reduction

```c
int gr_mat_reduce_row(slong *column, gr_mat_t A, slong *P, slong *L, slong m, gr_ctx_t ctx)
```

Reduce row $n$ of the matrix $A$, assuming the prior rows are in Gauss form. However those rows may not be in order. The entry $i$ of the array $P$ is the row of $A$ which has a pivot in the $i$-th column. If no such row exists, the entry of $P$ will be $-1$. The function sets column to the column in which the $n$-th row has a pivot after reduction. This will always be chosen to be the first available column for a pivot from the left. This information is also updated in $P$. Entry $i$ of the array $L$ contains the number of possibly nonzero columns of $A$ row $i$. This speeds up reduction in the case that $A$ is chambered on the right. Otherwise the entries of $L$ can all be set to the number of columns of $A$. We require the entries of $L$ to be monotonic increasing.
3.8 gr_poly.h – dense univariate polynomials over generic rings

A `gr_poly_t` represents a univariate polynomial \( f \in R[X] \) implemented as a dense array of coefficients in a generic ring \( R \).

In this module, the context object `ctx` always represents the coefficient ring \( R \) unless otherwise stated. Creating a context object representing the polynomial ring \( R[X] \) only becomes necessary when one wants to manipulate polynomials using generic ring methods like \texttt{gr_add} instead of the designated polynomial methods like \texttt{gr_poly_add}.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (often requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

3.8.1 Type compatibility

The `gr_poly` type has the same data layout as the following polynomial types: \texttt{fmpz_poly}, \texttt{fq_poly}, \texttt{fq_nmod_poly}, \texttt{fq_zech_poly}, \texttt{arb_poly}, \texttt{acb_poly}, \texttt{ca_poly}. Methods in this module can therefore be mixed freely with methods in the corresponding FLINT modules when the underlying coefficient type is the same. It is not directly compatible with the following types: \texttt{fmpq_poly} (coefficients are stored with a common denominator), \texttt{nmod_poly} (modulus data is stored as part of the polynomial object).

3.8.2 Weak normalization

A `gr_poly_t` is always normalised by removing leading zeros. For rings without decidable equality (e.g., rings with inexact representation), only coefficients that are provably zero will be removed, and there can thus be spurious leading zeros in the internal representation. Methods that depend on knowing the exact degree of a polynomial will act appropriately, typically by returning \texttt{GR_UNABLE} when it is unknown whether the leading stored coefficient is nonzero.

3.8.3 Types, macros and constants

```c
typedef struct gr_poly_struct {
    gr_ptr coeffs;
    slong length;
    gr_ctx_t ctx;
} gr_poly_t;
```

A `gr_poly_t` is defined as an array of length one of type `gr_poly_struct`, permitting a `gr_poly_t` to be passed by reference.

3.8.4 Memory management

```c
void gr_poly_init(gr_poly_t poly, gr_ctx_t ctx);
void gr_poly_init2(gr_poly_t poly, slong len, gr_ctx_t ctx);
void gr_poly_clear(gr_poly_t poly, gr_ctx_t ctx);
```

```c
gr_ptr gr_poly_entry_ptr(gr_poly_t poly, slong i, gr_ctx_t ctx);
gr_srcptr gr_poly_entry_srcptr(const gr_poly_t poly, slong i, gr_ctx_t ctx);
```

```c
slong gr_poly_length(const gr_poly_t poly, gr_ctx_t ctx);
```
void \texttt{gr\_poly\_swap} (\texttt{gr\_poly\_t} poly1, \texttt{gr\_poly\_t} poly2, \texttt{gr\_ctx\_t} ctx)

void \texttt{gr\_poly\_fit\_length} (\texttt{gr\_poly\_t} poly, \texttt{slong} len, \texttt{gr\_ctx\_t} ctx)

void \texttt{\_gr\_poly\_set\_length} (\texttt{gr\_poly\_t} poly, \texttt{slong} len, \texttt{gr\_ctx\_t} ctx)

### 3.8.5 Basic manipulation

void \texttt{\_gr\_poly\_normalise} (\texttt{gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set} (\texttt{gr\_poly\_t} res, \texttt{const gr\_poly\_t} src, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_get\_fmpz\_poly} (\texttt{gr\_poly\_t} res, \texttt{const fmpz\_poly\_t} src, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_fmpq\_poly} (\texttt{gr\_poly\_t} res, \texttt{const fmpq\_poly\_t} src, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_gr\_poly\_other} (\texttt{gr\_poly\_t} res, \texttt{const gr\_poly\_t} x, \texttt{gr\_ctx\_t} x\_ctx, \texttt{gr\_ctx\_t} ctx)

int \texttt{\_gr\_poly\_reverse} (\texttt{gr\_ptr} res, \texttt{gr\_srcptr} poly, \texttt{slong} len, \texttt{slong} n, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_reverse} (\texttt{gr\_poly\_t} res, \texttt{const gr\_poly\_t} poly, \texttt{slong} n, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_truncate} (\texttt{gr\_poly\_t} res, \texttt{const gr\_poly\_t} poly, \texttt{slong} newlen, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_zero} (\texttt{gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_one} (\texttt{gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_neg\_one} (\texttt{gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_gen} (\texttt{gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_write} (\texttt{gr\_stream\_t} out, \texttt{const gr\_poly\_t} poly, \texttt{const char *} x, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_print} (\texttt{const gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_rand\_test} (\texttt{gr\_poly\_t} poly, \texttt{flint\_rand\_t} state, \texttt{slong} len, \texttt{gr\_ctx\_t} ctx)

\texttt{truth\_t} \texttt{\_gr\_poly\_equal} (\texttt{gr\_srcptr} poly1, \texttt{slong} len1, \texttt{gr\_srcptr} poly2, \texttt{slong} len2, \texttt{gr\_ctx\_t} ctx)

\texttt{truth\_t} \texttt{gr\_poly\_equal} (\texttt{const gr\_poly\_t} poly1, \texttt{const gr\_poly\_t} poly2, \texttt{gr\_ctx\_t} ctx)

\texttt{truth\_t} \texttt{gr\_poly\_is\_zero} (\texttt{const gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

\texttt{truth\_t} \texttt{gr\_poly\_is\_one} (\texttt{const gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

\texttt{truth\_t} \texttt{gr\_poly\_is\_gen} (\texttt{const gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

\texttt{truth\_t} \texttt{gr\_poly\_is\_scalar} (\texttt{const gr\_poly\_t} poly, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_scalar} (\texttt{gr\_poly\_t} poly, \texttt{gr\_srcptr} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_si} (\texttt{gr\_poly\_t} poly, \texttt{slong} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_ui} (\texttt{gr\_poly\_t} poly, \texttt{ulong} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_fmpz} (\texttt{gr\_poly\_t} poly, \texttt{const fmpz\_t} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_fmpq} (\texttt{gr\_poly\_t} poly, \texttt{const fmpq\_t} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_coeff\_scalar} (\texttt{gr\_poly\_t} poly, \texttt{slong} n, \texttt{gr\_srcptr} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_coeff\_si} (\texttt{gr\_poly\_t} poly, \texttt{slong} n, \texttt{slong} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_coeff\_ui} (\texttt{gr\_poly\_t} poly, \texttt{slong} n, \texttt{ulong} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_coeff\_fmpz} (\texttt{gr\_poly\_t} poly, \texttt{slong} n, \texttt{const fmpz\_t} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_set\_coeff\_fmpq} (\texttt{gr\_poly\_t} poly, \texttt{slong} n, \texttt{const fmpq\_t} c, \texttt{gr\_ctx\_t} ctx)

int \texttt{gr\_poly\_get\_coeff\_scalar} (\texttt{gr\_ptr} res, \texttt{const gr\_poly\_t} poly, \texttt{slong} n, \texttt{gr\_ctx\_t} ctx)
3.8.6 Arithmetic

int gr_poly_neg(gr_poly_t res, const gr_poly_t src, gr_ctx_t ctx)
int _gr_poly_add(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, gr_ctx_t ctx)
int gr_poly_add(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t ctx)
int _gr_poly_sub(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, gr_ctx_t ctx)
int gr_poly_sub(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t ctx)
int _gr_poly_mul(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, gr_ctx_t ctx)
int gr_poly_mul(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t ctx)
int _gr_poly_mullow_generic(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,
  slong len, gr_ctx_t ctx)
int _gr_poly_mullow(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, slong len,
  gr_ctx_t ctx)
int gr_poly_mullow(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, slong len,
  gr_ctx_t ctx)
int gr_poly_mul_scalar(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)

3.8.7 Powering

int _gr_poly_pow_series_ui_binexp(gr_ptr res, gr_srcptr f, slong flen, ulong exp, slong len,
  gr_ctx_t ctx)
int gr_poly_pow_series_ui_binexp(gr_poly_t res, const gr_poly_t poly, ulong exp, slong len,
  gr_ctx_t ctx)
int _gr_poly_pow_series_ui(gr_ptr res, gr_srcptr f, slong flen, ulong exp, slong len, gr_ctx_t ctx)
int gr_poly_pow_series_ui(gr_poly_t res, const gr_poly_t poly, ulong exp, slong len, gr_ctx_t ctx)
int _gr_poly_pow_ui_binexp(gr_ptr res, gr_srcptr f, slong flen, ulong exp, gr_ctx_t t ctx)
int gr_poly_pow_ui_binexp(gr_poly_t res, const gr_poly_t poly, ulong exp, gr_ctx_t t ctx)
int _gr_poly_pow_ui(gr_ptr res, gr_srcptr f, slong flen, ulong exp, gr_ctx_t t ctx)
int gr_poly_pow_ui(gr_poly_t res, const gr_poly_t poly, ulong exp, gr_ctx_t t ctx)
int gr_poly_pow_fmpz(gr_poly_t res, const gr_poly_t poly, const fmpz_t exp, gr_ctx_t t ctx)
int _gr_poly_pow_series_fmpq_recurrence(gr_ptr h, gr_srcptr f, slong flen, const fmpq_t exp, slong len, int precomp, gr_ctx_t t ctx)
int gr_poly_pow_series_fmpq_recurrence(gr_poly_t res, const gr_poly_t poly, const fmpq_t exp,
  slong len, gr_ctx_t t ctx)

3.8.8 Shifting

int _gr_poly_shift_left(gr_ptr res, gr_srcptr poly, slong len, slong n, gr_ctx_t t ctx)
int gr_poly_shift_left(gr_poly_t res, const gr_poly_t poly, slong n, gr_ctx_t t ctx)
int _gr_poly_shift_right(gr_ptr res, gr_srcptr poly, slong len, slong n, gr_ctx_t t ctx)
int gr_poly_shift_right(gr_poly_t res, const gr_poly_t poly, slong n, gr_ctx_t t ctx)
3.8.9 Scalar division

int gr_poly_div_scalar(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)

3.8.10 Division with remainder

int _gr_poly_divrem_divconquer_preinv1(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_srcptr invB, slong cutoff, gr_ctx_t ctx)
int _gr_poly_divrem_divconquer_noinv(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, slong cutoff, gr_ctx_t ctx)
int _gr_poly_divrem_divconquer(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, slong cutoff, gr_ctx_t ctx)
int gr_poly_divrem_divconquer(gr_poly_t Q, gr_poly_t R, const gr_poly_t A, const gr_poly_t B, slong cutoff, gr_ctx_t ctx)
int _gr_poly_divrem_basecase_preinv1(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_srcptr invB, gr_ctx_t ctx)
int _gr_poly_divrem_basecase_noinv(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_ctx_t ctx)
int _gr_poly_divrem_basecase(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_ctx_t ctx)
int gr_poly_divrem_basecase(gr_poly_t Q, gr_poly_t R, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
int _gr_poly_divrem_newton(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_ctx_t ctx)
int gr_poly_divrem_newton(gr_poly_t Q, gr_poly_t R, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
int _gr_poly_divrem(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_ctx_t ctx)
int gr_poly_divrem(gr_poly_t Q, gr_poly_t R, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)

These functions implement Euclidean division with remainder: given polynomials $A, B \in K[x]$ where $K$ is a field, with $B \neq 0$, there is a unique quotient $Q$ and remainder $R$ such that $A = BQ + R$ and either $R = 0$ or $\deg(R) < \deg(B)$. If $B$ is provably zero, GR_DOMAIN is returned.

When $K$ is a commutative ring and $\text{lc}(B)$ is a unit in $K$, the situation is the same as over fields. In particular, Euclidean division with remainder always makes sense over commutative rings when $B$ is monic. If $\text{lc}(B)$ is not a unit, the division still makes sense if the coefficient quotient $\text{lc}(r) / \text{lc}(B)$ exists for each partial remainder $r$. Indeed, the basecase and divconquer algorithms return GR_DOMAIN precisely when encountering a leading quotient $\text{lc}(r) / \text{lc}(B) \notin K$. However, the newton algorithm as currently implemented returns GR_DOMAIN when $\text{lc}(B)^{-1} \notin K$.

The underscore methods make the following assumptions:

- $Q$ has room for $\text{lenA} - \text{lenB} + 1$ coefficients.
- $R$ has room for $\text{lenB} - 1$ coefficients.
- $\text{lenA} \geq \text{lenB} \geq 1$.
- $Q$ is not aliased with either $A$ or $B$.
- $R$ is not aliased with $B$.
- $R$ may be aliased with $A$, in which case all $\text{lenA}$ entries may be used as scratch space. Note that in this case, only the low $\text{lenB} - 1$ coefficients of $R$ actually represent valid coefficients on output: the higher scratch coefficients will not necessarily be zeroed.
- The divisor $B$ is normalized to have nonzero leading coefficient. (The non-underscore methods check for leading coefficients that are not provably nonzero and return GR_UNABLE.)
For divide-and-conquer (including Newton-like) algorithms, `cutoff` has the following meaning: we use the basecase algorithm for lengths $n < \text{cutoff}$ and the divide-and-conquer algorithm for $n \geq \text{cutoff}$. Using cutoff = $n$ thus results in exactly one divide-and-conquer step with a basecase length of $\lceil n/2 \rceil$. One should avoid calling the Newton methods with $n < \text{cutoff}$ as this may result in much worse performance if those methods do not have a specific escape check for that case.

The `newton` versions use Newton iteration, switching to a basecase algorithm when the length is smaller than the specified `cutoff`. Division uses the Karp-Markstein algorithm.

### 3.8.11 Power series division

For divide-and-conquer (including Newton-like) algorithms, `cutoff` has the following meaning: we use the basecase algorithm for lengths $n < \text{cutoff}$ and the divide-and-conquer algorithm for $n \geq \text{cutoff}$. Using cutoff = $n$ thus results in exactly one divide-and-conquer step with a basecase length of $\lceil n/2 \rceil$. One should avoid calling the Newton methods with $n < \text{cutoff}$ as this may result in much worse performance if those methods do not have a specific escape check for that case.

The `newton` versions use Newton iteration, switching to a basecase algorithm when the length is smaller than the specified `cutoff`. Division uses the Karp-Markstein algorithm.
3.8.12 Exact division

These functions compute a quotient $Q = A/B$ which is known to be exact (without remainder) in $R[x]$ (or in $R[[x]]/x^n$ in the case of series division). Given a nonexact division, they are allowed to set $Q$ to an arbitrary polynomial and return GR_SUCCESS instead of returning an error flag.

$R$ is assumed to be an integral domain (this is not checked).

For exact division, we have the choice of starting the division from the most significant terms (classical division) or the least significant (power series division). Which direction is more efficient depends in part on whether the leading or trailing coefficient of $B$ is cheaper to use for divisions. In a generic setting, this is hard to predict.

The bidirectional algorithms combine two half-divisions from both ends. This halves the number of operations in the basecase regime, though an extra coefficient inversion may be needed.

The noinv versions perform repeated divexact operations in the scalar domain without attempting to invert the leading (or trailing) coefficient, while other versions check invertibility first. There are no divexact_preinv1 versions because those are identical to the div_preinv1 counterparts.
int gr_poly_divexact_basecase(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
int _gr_poly_divexact_series_basecase_noinv(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen, slong len, gr_ctx_t ctx)
int _gr_poly_divexact_series_basecase(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen, slong len, gr_ctx_t ctx)
int gr_poly_divexact_series_basecase(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, slong len, gr_ctx_t ctx)

3.8.13 Square roots

int _gr_poly_sqrt_series_newton(gr_ptr res, gr_srcptr f, slong flen, slong len, slong cutoff, gr_ctx_t ctx)
int gr_poly_sqrt_series_newton(gr_poly_t res, const gr_poly_t f, slong len, slong cutoff, gr_ctx_t ctx)
int gr_poly_sqrt_series_basecase(gr_ptr res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_sqrt_series_miller(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_sqrt_series_miller(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int gr_poly_sqrt_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_rsqrt_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_rsqrt_series_newton(gr_ptr res, gr_srcptr f, slong flen, slong len, slong cutoff, gr_ctx_t ctx)
int gr_poly_rsqrt_series_newton(gr_poly_t res, const gr_poly_t f, slong len, slong cutoff, gr_ctx_t ctx)
int gr_poly_rsqrt_series_basecase(gr_ptr res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_rsqrt_series_miller(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_rsqrt_series_miller(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int gr_poly_rsqrt_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_rsqrt_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)

3.8.14 Evaluation

int _gr_poly_evaluate_rectangular(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr x, gr_ctx_t ctx)
int gr_poly_evaluate_rectangular(gr_ptr res, const gr_poly_t poly, gr_srcptr x, gr_ctx_t ctx)
int _gr_poly_evaluate_modular(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr x, gr_ctx_t ctx)
int gr_poly_evaluate_modular(gr_ptr res, const gr_poly_t poly, gr_srcptr x, gr_ctx_t ctx)
int _gr_poly_evaluate_horner(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr x, gr_ctx_t ctx)
int gr_poly_evaluate_horner(gr_ptr res, const gr_poly_t poly, gr_srcptr x, gr_ctx_t ctx)
int _gr_poly_evaluate_other_horner(gr_ptr res, gr_srcptr f, slong len, const gr_srcptr x, gr_ctx_t ctx)
int gr_poly_evaluate_other_horner(gr_ptr res, const gr_poly_t poly, gr_srcptr x, gr_ctx_t ctx)
Set res to poly evaluated at x.

int _gr_poly_evaluate_other_horner(gr_ptr res, gr_srcptr f, slong len, const gr_srcptr x, gr_ctx_t ctx)
int gr_poly_evaluate_other_horner(gr_ptr res, const gr_poly_t f, gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t t_ctx)
int _gr_poly_evaluate_other_rectangular(gr_ptr res, const gr_poly_t f, const gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t t_ctx)
int gr_poly_evaluate_other_rectangular(gr_ptr res, const gr_poly_t f, const gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t t_ctx)
int _gr_poly_evaluate_otherืนีภร (gr_ptr res, const gr_poly_t f, const gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t t_ctx)
int gr_poly_evaluate_other盛典, const gr_poly_t f, const gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t t_ctx)

Set res to poly evaluated at x, where the coefficients of f belong to ctx while both x and res belong to x_ctx.

3.8.15 Multipoint evaluation and interpolation

gr_ptr * _gr_poly_tree_alloc(slong len, gr_ctx_t ctx)
void _gr_poly_tree_free(gr_ptr *tree, slong len, gr_ctx_t ctx)
int _gr_poly_tree_build(gr_ptr *tree, gr_srcptr roots, slong len, gr_ctx_t ctx)
int _gr_poly_evaluate_vec_fast_precomp(gr_ptr vs, gr_srcptr poly, slong plen, gr_ptr *tree, slong len, gr_ctx_t ctx)
int _gr_poly_evaluate_vec_fast(gr_ptr ys, gr_srcptr poly, slong plen, gr_srcptr xs, slong n, gr_ctx_t ctx)
int gr_poly_evaluate_vec_fast(gr_vec_t ys, const gr_poly_t poly, const gr_vec_t xs, gr_ctx_t ctx)
int _gr_poly_evaluate_vec_iter(gr_ptr ys, gr_srcptr poly, slong plen, gr_srcptr xs, slong n, gr_ctx_t ctx)
int gr_poly_evaluate_vec_iter(gr_vec_t ys, const gr_poly_t poly, const gr_vec_t xs, gr_ctx_t ctx)

3.8.16 Composition

int _gr_poly_taylor_shift_horner(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr c, gr_ctx_t t_ctx)
int _gr_poly_taylor_shift_horner(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t t_ctx)
int _gr_poly_taylor_shift_divconquer(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr c, gr_ctx_t t_ctx)
int gr_poly_taylor_shift_divconquer(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t t_ctx)
int _gr_poly_taylor_shift_convolution(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr c, gr_ctx_t t_ctx)
int gr_poly_taylor_shift_convolution(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t t_ctx)
int _gr_poly_taylor_shift(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr c, gr_ctx_t t_ctx)
int gr_poly_taylor_shift(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t t_ctx)

Sets res to the Taylor shift \( f(x + c) \), where \( f \) is given by poly, computed respectively using an optimized form of Horner’s rule, divide-and-conquer, a single convolution, and an automatic choice between the three algorithms. The underscore methods support aliasing.
3.8.17 Power series composition and reversion

Sets res to the composition \( f(g(x)) \) where \( f \) is given by \( \text{poly1} \) and \( g \) is given by \( \text{poly2} \), respectively using Horner’s rule, divide-and-conquer, and an automatic choice between the two algorithms. The default algorithm also handles special-form input \( g = ax^n + c \) efficiently by performing a Taylor shift followed by a rescaling. The underscore methods do not support aliasing of the output with either input polynomial.

Sets res to the power series composition \( h(x) = f(g(x)) \) truncated to order \( O(x^n) \) where \( f \) is given by \( \text{poly1} \) and \( g \) is given by \( \text{poly2} \), respectively using Horner’s rule, the Brent-Kung baby step-giant step algorithm [BrentKung1978], divide-and-conquer, and an automatic choice between the algorithms.

The default algorithm also handles short input and special-form input \( g = ax^n \) efficiently.

We require that the constant term in \( g(x) \) is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial, and do not zero-pad the result.
int gr_poly_revert_series(gr_poly_t res, const gr_poly_t f, slong n, gr_ctx_t ctx)
Sets res to the power series reversion $f^{-1}(x)$ which satisfies $f^{-1}(f(x)) = f(f^{-1}(x)) = x \mod x^n$. For the series reversion to exist, we require that the constant term in $f$ is zero and that the linear coefficient is invertible. The flag GR_DOMAIN is returned otherwise.

The lagrange and lagrange_fast algorithms require the ability to divide by $2, 3, \ldots, n-1$ and will return the GR_UNABLE flag in too small characteristic.

The underscore methods do not support aliasing of the output with the input.

The Newton method is described in [BrentKung1978]; the lagrange algorithm implements the Lagrange inversion formula, while the lagrange_fast algorithm implements the baby-step giant-step algorithm described in [Joh2015b].

### 3.8.18 Derivative and integral

int _gr_poly_derivative(gr_ptr res, gr_srcptr poly, slong len, gr_ctx_t ctx)
int gr_poly_derivative(gr_poly_t res, const gr_poly_t poly, gr_ctx_t ctx)
int _gr_poly_nth_derivative(gr_ptr res, gr_srcptr poly, ulong n, slong len, gr_ctx_t ctx)
int gr_poly_nth_derivative(gr_poly_t res, const gr_poly_t poly, ulong n, gr_ctx_t ctx)
int _gr_poly_integral(gr_ptr res, gr_srcptr poly, slong len, gr_ctx_t ctx)
int gr_poly_integral(gr_poly_t res, const gr_poly_t poly, gr_ctx_t ctx)

### 3.8.19 Monic polynomials

int _gr_poly_make_monic(gr_ptr res, gr_srcptr poly, slong len, gr_ctx_t ctx)
int gr_poly_make_monic(gr_poly_t res, const gr_poly_t src, gr_ctx_t ctx)

truth_t _gr_poly_is_monic(gr_srcptr poly, slong len, gr_ctx_t ctx)
truth_t gr_poly_is_monic(const gr_poly_t res, gr_ctx_t ctx)

### 3.8.20 GCD

int _gr_poly_hgcd(gr_ptr r, slong *sgn, gr_ptr *M, slong *lenM, gr_ptr A, slong *lenA, gr_ptr B, slong *lenB, gr_srcptr a, slong lena, gr_srcptr b, slong lenb, slong cutoff, gr_ctx_t ctx)
Computes the HGCD of $a$ and $b$, that is, a matrix $M$, a sign $\sigma$ and two polynomials $A$ and $B$ such that

$$(A, B)^t = \sigma M^{-1}(a, b)^t.$$ 
Assumes that $\text{len}(a) > \text{len}(b) > 0$.
Assumes that $A$ and $B$ have space of size at least $\text{len}(a)$ and $\text{len}(b)$, respectively. On exit, *lenA and *lenB will contain the correct lengths of $A$ and $B$.
If $r$ is not NULL, writes to that variable the corresponding value for computing resultants using the HGCD algorithm.

int _gr_poly_gcd_hgcd(gr_ptr G, slong *lenG, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, slong inner_cutoff, slong cutoff, gr_ctx_t ctx)
int gr_poly_gcd_hgcd(gr_poly_t G, const gr_poly_t A, const gr_poly_t B, slong inner_cutoff, slong cutoff, gr_ctx_t ctx)
3.8.21 Resultant

Sets res to the resultant of poly1 and poly2. The underscore methods assume that len1 ≥ len2 ≥ 1 and that the leading coefficients are nonzero.

The euclidean algorithm is the ordinary Euclidean algorithm. The hgcd version uses the quasilinear half-GCD algorithm. It requires two extra tuning parameters inner_cutoff (recursion threshold passed forward to the HGCD algorithm) and cutoff. Both algorithms can fail when run over non-fields; they will return GR_DOMAIN when encountering an impossible inverse.

The small version uses division-free straight-line programs optimized for short polynomials. It returns GR_UNABLE if the polynomials are too large. Currently this function handles the cases where len1 ≤ 2 or len2 ≤ 3.

The sylvester version constructs the Sylvester matrix and computes its determinant. This is useful over inexact rings and as a fallback for rings without division.
The default version attempts to choose an appropriate algorithm automatically.
Currently no algorithm has been implemented that is appropriate for integral domains.

### 3.8.22 Squarefree factorization

TODO: currently only fields of characteristic 0 are supported.

```c
int gr_poly_factor_squarefree(gr_ptr c, gr_vec_t fac, gr_vec_t exp, const gr_poly_t poly, gr_ctx_t ctx)
```

Computes a squarefree factorization of `poly`.

The constant `c` is set to an element of the scalar ring. The factors in `fac` are set to polynomials; the user must thus initialize it to a vector of polynomials of the same type as `poly` (and not to the parent `ctx`). The exponent vector `exp` must be initialized to the `fmpz` type.

```c
int gr_poly_squarefree_part(gr_poly_t res, const gr_poly_t poly, gr_ctx_t ctx)
```

Sets `res` to the squarefree part of `poly`.

### 3.8.23 Roots

```c
int gr_poly_roots(gr_vec_t roots, gr_vec_t mult, const gr_poly_t poly, int flags, gr_ctx_t ctx)
```

```c
int gr_poly_roots_other(gr_vec_t roots, gr_vec_t mult, const gr_poly_t poly, gr_ctx_t poly_ctx, int flags, gr_ctx_t ctx)
```

Finds all roots of the given polynomial in the ring defined by `ctx`, storing the roots without duplication in `roots` (a vector with elements of type `ctx`) and the corresponding multiplicities in `mult` (a vector with elements of type `fmpz`).

If the target ring is not an algebraically closed field, then the sum of multiplicities can be smaller than the degree of the polynomial. For example, with `fmpz` coefficients, we only find integer roots. The other version of this function takes as input a polynomial with entries in a different ring `poly_ctx`. For example, we can compute `qqbar` or `arb` roots for a polynomial with `fmpz` coefficients.

Whether the roots are sorted in any particular order is ring-dependent.

We consider roots of the zero polynomial to be ill-defined and return `GR_DOMAIN` in that case.

### 3.8.24 Power series special functions

```c
int gr_poly_asin_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
```

```c
int gr_poly_acos_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
```

```c
int gr_poly_asinh_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
```

```c
int gr_poly_acosh_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
```

```c
int gr_poly_atan_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
```

```c
int gr_poly_atanh_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
```

```c
int gr_poly_log_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
```
int _gr_poly_log1p_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_log1p_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_exp_series_basecase(gr_ptr f, gr_srcptr h, slong hlen, slong n, gr_ctx_t ctx)
int gr_poly_exp_series_basecase(gr_poly_t f, const gr_poly_t h, slong n, gr_ctx_t ctx)
int _gr_poly_exp_series_basecase_mul(gr_poly_t f, gr_srcptr h, slong hlen, slong n, gr_ctx_t ctx)
int gr_poly_exp_series_basecase_mul(gr_poly_t f, const gr_poly_t h, slong n, gr_ctx_t ctx)
int _gr_poly_exp_series_newton(gr_ptr f, gr_ptr g, gr_srcptr h, slong hlen, slong n, slong cutoff, gr_ctx_t ctx)
int gr_poly_exp_series_newton(gr_poly_t f, const gr_poly_t h, slong n, gr_ctx_t ctx)
int _gr_poly_exp_series_generic(gr_ptr f, gr_srcptr h, slong hlen, slong n, gr_ctx_t ctx)
int gr_poly_exp_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int _gr_poly_exp_series(gr_poly_t f, const gr_poly_t h, slong n, gr_ctx_t ctx)
int _gr_poly_sin_cos_series_basecase(gr_ptr s, gr_ptr c, gr_srcptr h, slong hlen, slong n, int times_pi, gr_ctx_t ctx)
int gr_poly_sin_cos_series_basecase(gr_poly_t s, gr_poly_t c, const gr_poly_t h, slong n, int times_pi, gr_ctx_t ctx)
int _gr_poly_sin_cos_series_tangent(gr_ptr s, gr_ptr c, gr_srcptr h, slong hlen, slong n, int times_pi, gr_ctx_t ctx)
int gr_poly_sin_cos_series_tangent(gr_poly_t s, gr_poly_t c, const gr_poly_t h, slong n, int times_pi, gr_ctx_t ctx)

The basecase version uses a simple recurrence for the coefficients, requiring \(O(nm)\) operations where \(m\) is the length of \(h\).

The tangent version uses the tangent half-angle formulas to compute the sine and cosine via \_acb_poly_tan_series(). This requires \(O(M(n))\) operations. When \(h = h_0 + h_1\) where the constant term \(h_0\) is nonzero, the evaluation is done as \(\sin(h_0 + h_1) = \cos(h_0)\sin(h_1) + \sin(h_0)\cos(h_1)\), \(\cos(h_0 + h_1) = \cos(h_0)\cos(h_1) - \sin(h_0)\sin(h_1)\).

The basecase and tangent versions take a flag \texttt{times\_pi} specifying that the input is to be multiplied by \(\pi\).
3.9 \texttt{gr_mpoly.h} – sparse multivariate polynomials over generic rings

A \texttt{gr_mpoly_t} represents a multivariate polynomial $f \in R[X_1, \ldots, X_n]$ implemented as an array of coefficients in a generic ring $R$ together with an array of packed exponents.

3.9.1 Weak normalization

A \texttt{gr_mpoly_t} is always normalised by removing zero coefficients. For rings without decidable equality (e.g. rings with inexact representation), only coefficients that are provably zero will be removed, and there can thus be spurious zeros in the internal representation. Methods that depend on knowing the exact structure of a polynomial will act appropriately, typically by returning \texttt{GR_UNABLE} when it is unknown whether any stored coefficients are nonzero.

3.9.2 Types, macros and constants

type \texttt{gr_mpoly_struct}

type \texttt{gr_mpoly_t}

A \texttt{gr_mpoly_t} is defined as an array of length one of type \texttt{gr_mpoly_struct}, permitting a \texttt{gr_mpoly_t} to be passed by reference.

3.9.3 Memory management

\texttt{void gr_mpoly_init}(\texttt{gr_mpoly_t} \texttt{A}, \texttt{const mpolyt_t mctx}, \texttt{gr_ctx_t cctx})

Initializes and sets \texttt{A} to the zero polynomial.

\texttt{void gr_mpoly_init3}(\texttt{gr_mpoly_t} \texttt{A}, \texttt{slong alloc}, \texttt{flint_bitcnt_t bits}, \texttt{const mpolyt_t mctx}, \texttt{gr_ctx_t cctx})

Initializes \texttt{A} with space allocated for the given number of coefficients and exponents with the given number of bits.

\texttt{void gr_mpoly_clear}(\texttt{gr_mpoly_t} \texttt{A}, \texttt{const mpolyt_t mctx}, \texttt{gr_ctx_t cctx})

Clears \texttt{A}, freeing all allocated data.

3.9.4 Basic manipulation

\texttt{void gr_mpoly_swap}(\texttt{gr_mpoly_t} \texttt{A}, \texttt{gr_mpoly_t B}, \texttt{const mpolyt_t mctx}, \texttt{gr_ctx_t cctx})

Swaps \texttt{A} and \texttt{B} efficiently.

\texttt{int gr_mpoly_set}(\texttt{gr_mpoly_t} \texttt{A}, \texttt{const gr_mpoly_t B}, \texttt{const mpolyt_t mctx}, \texttt{gr_ctx_t cctx})

Sets \texttt{A} to \texttt{B}.

\texttt{int gr_mpoly_zero}(\texttt{gr_mpoly_t} \texttt{A}, \texttt{const mpolyt_t mctx}, \texttt{gr_ctx_t cctx})

Sets \texttt{A} to the zero polynomial.

\texttt{truth_t gr_mpoly_is_zero}(\texttt{const gr_mpoly_t A}, \texttt{const mpolyt_t mctx}, \texttt{gr_ctx_t cctx})

Returns whether \texttt{A} is the zero polynomial.

\texttt{int gr_mpoly_gen}(\texttt{gr_mpoly_t} \texttt{A}, \texttt{slong var}, \texttt{const mpolyt_t mctx}, \texttt{gr_ctx_t cctx})

Sets \texttt{A} to the generator with index \texttt{var} (indexed from zero).

\texttt{truth_t gr_mpoly_is_gen}(\texttt{const gr_mpoly_t A}, \texttt{slong var}, \texttt{const mpolyt_t mctx}, \texttt{gr_ctx_t cctx})

Returns whether \texttt{A} is the generator with index \texttt{var} (indexed from zero).
3.9.5 Comparisons

\texttt{truth\_t gr\_mpoly\_equal(\text{const gr\_mpoly\_t A, const gr\_mpoly\_t B, const mpoly\_ctx\_t mctx, gr\_ctx\_t cctx})}

Returns whether A and B are equal.

3.9.6 Random generation

\texttt{int gr\_mpoly\_randtest\_bits(\text{gr\_mpoly\_t A, flint\_rand\_t state, slong length, flint\_bitcnt\_t exp\_bits, const mpoly\_ctx\_t mctx, gr\_ctx\_t cctx})}

Sets A to a random polynomial with up to length terms and up to exp\_bits bits in the exponents.

3.9.7 Input and output

\texttt{int gr\_mpoly\_write\_pretty(gr\_stream\_t out, const gr\_mpoly\_t A, const char **x, const mpoly\_ctx\_t mctx, gr\_ctx\_t cctx)\n\texttt{int gr\_mpoly\_print\_pretty(const gr\_mpoly\_t A, const char **x, const mpoly\_ctx\_t mctx, gr\_ctx\_t cctx)}}

Prints A using the strings in x for the variables. If x is \textit{NULL}, defaults are used.

3.9.8 Coefficient and exponent access

\texttt{int gr\_mpoly\_get\_coeff\_scalar\_fmpz(gr\_ptr c, const gr\_mpoly\_t A, const fmpz *exp, const mpoly\_ctx\_t mctx, gr\_ctx\_t cctx)}

\texttt{int gr\_mpoly\_get\_coeff\_scalar\_ui(gr\_ptr c, const gr\_mpoly\_t A, const ulong *exp, const mpoly\_ctx\_t mctx, gr\_ctx\_t cctx)}

Sets c to the coefficient in A with exponents exp.

\texttt{int gr\_mpoly\_set\_coeff\_scalar\_fmpz(gr\_mpoly\_t A, gr\_srcptr c, const fmpz *exp, const mpoly\_ctx\_t mctx, gr\_ctx\_t cctx)}

\texttt{int gr\_mpoly\_set\_coeff\_ui\_fmpz(gr\_mpoly\_t A, const fmpq\_t c, const fmpz *exp, const mpoly\_ctx\_t mctx, gr\_ctx\_t cctx)}

Sets the coefficient with exponents exp in A to the scalar c which must be an element of or coercible to the coefficient ring.
### 3.9.9 Arithmetic

```c
int gr_mpoly_neg(gr_mpoly_t A, const gr_mpoly_t B, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```
Sets A to the negation of B.

```c
int gr_mpoly_add(gr_mpoly_t A, const gr_mpoly_t B, const gr_mpoly_t C, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```
Sets A to the difference of B and C.

```c
int gr_mpoly_sub(gr_mpoly_t A, const gr_mpoly_t B, const gr_mpoly_t C, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```
Sets A to the difference of B and C.

```c
int gr_mpoly_mul(gr_mpoly_t A, const gr_mpoly_t B, const gr_mpoly_t C, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```
Sets A to the product of B and C. The monomial version assumes that C is a monomial.

```c
int gr_mpoly_mul_johnson(gr_mpoly_t A, const gr_mpoly_t B, const gr_mpoly_t C, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```

```c
int gr_mpoly_mul_monomial(gr_mpoly_t A, const gr_mpoly_t B, const gr_mpoly_t C, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```
Sets A to the product of B and C. The monomial version assumes that C is a monomial.

```c
int gr_mpoly_mul_scalar(gr_mpoly_t A, const gr_mpoly_t B, gr_srcptr c, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```
Sets A to B multiplied by the scalar c which must be an element of or coercible to the coefficient ring.

### 3.9.10 Container operations

Mostly intended for internal use.

```c
void _gr_mpoly_fit_length(gr_ptr *coeffs, slong *coeffs_alloc, ulong **exps, slong *exps_alloc, slong N, slong length, gr_ctx_t cctx)
```
Ensures that A has space for len coefficients and exponents.

```c
void gr_mpoly_fit_length(gr_mpoly_t A, slong len, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```

```c
void gr_mpoly_fit_bits(gr_mpoly_t A, flint_bitcnt_t bits, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```

```c
void gr_mpoly_fit_length_fit_bits(gr_mpoly_t A, slong len, flint_bitcnt_t bits, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```

```c
void gr_mpoly_fit_length_reset_bits(gr_mpoly_t A, slong len, flint_bitcnt_t bits, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```

```c
void _gr_mpoly_set_length(gr_mpoly_t A, slong newlen, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```

```c
void _gr_mpoly_push_exp_ui(gr_mpoly_t A, const ulong *exp, const mpoly_ctx_t mctx, gr_ctx_t cctx)
```
int gr_mpoly_push_term_scalar_ui(gr_mpoly_t A, gr_srcptr c, const ulong *exp, const mpoly_ctx_t mx, gr_ctx_t ctx)

void _gr_mpoly_push_exp_fmpz(gr_mpoly_t A, const fmpz *exp, const mpoly_ctx_t mx, gr_ctx_t ctx)

int gr_mpoly_push_term_scalar_fmpz(gr_mpoly_t A, gr_srcptr c, const fmpz *exp, const mpoly_ctx_t mx, gr_ctx_t ctx)

void gr_mpoly_sort_terms(gr_mpoly_t A, const mpoly_ctx_t mx, gr_ctx_t ctx)

int gr_mpoly_combine_like_terms(gr_mpoly_t A, const mpoly_ctx_t mx, gr_ctx_t ctx)

truth_t gr_mpoly_is_canonical(const gr_mpoly_t A, const mpoly_ctx_t mx, gr_ctx_t ctx)

void gr_mpoly_assert_canonical(const gr_mpoly_t A, const mpoly_ctx_t mx, gr_ctx_t ctx)
4.1 ulong_extras.h – arithmetic and number-theoretic functions for single-word integers

This module implements functions for single limb unsigned integers, including arithmetic with a precomputed inverse and modular arithmetic.

The module includes functions for square roots, factorisation and primality testing. Almost all the functions in this module are highly developed and extremely well optimised.

The basic type is the mp_limb_t as defined by GMP. Functions which take a precomputed inverse either have the suffix preinv and take an mp_limb_t precomputed inverse as computed by n_preinvert_limb or have the suffix _precomp and accept a double precomputed inverse as computed by n_precompute_inverse.

Sometimes three functions with similar names are provided for the same task, e.g. n_mod_precomp, n_mod2_precomp and n_mod2_preinv. If the part of the name that designates the functionality ends in 2 then the function has few if any limitations on its inputs. Otherwise the function may have limitations such as being limited to 52 or 53 bits. In practice we found that the preinv functions are generally faster anyway, so most times it pays to just use the n_blah2_preinv variants.

Some functions with the n_ll_ or n_lll_ prefix accept parameters of two or three limbs respectively.

4.1.1 Simple example

The following example computes $a b \pmod{n}$ using a precomputed inverse, where $a = 12345678$, $b = 87654321$ and $n = 111111111$.

```c
#include <stdio.h>
#include "ulong_extras.h"

int main() {
    mp_limb_t r, a, b, n, ninv;

    a = UWORD(12345678);
    b = UWORD(87654321);
    n = UWORD(111111111);
    ninv = n_preinvert_limb(n);

    r = n_mulmod2_preinv(a, b, n, ninv);

    flint_printf("%wu*%wu mod %wu is %wu\n", a, b, n, r);
}
```

The output is:

103
4.1.2 Random functions

ulong n_randlimb(flint_rand_t state)

Returns a uniformly pseudo random limb.

The algorithm generates two random half limbs \( s_j, j = 0, 1 \), by iterating respectively \( v_{i+1} = (v_i a + b) \mod p_j \) for some initial seed \( v_0 \), randomly chosen values \( a \) and \( b \) and \( p_0 = 4294967311 = \text{nextprime}(2^{32}) \) on a 64-bit machine and \( p_0 = \text{nextprime}(2^{16}) \) on a 32-bit machine and \( p_{i+1} = \text{nextprime}(p_i) \).

ulong n_randbits(flint_rand_t state, unsigned int bits)

Returns a uniformly pseudo random number with the given number of bits. The most significant bit is always set, unless zero is passed, in which case zero is returned.

ulong n_randtest_bits(flint_rand_t state, int bits)

Returns a uniformly pseudo random number with the given number of bits. The most significant bit is always set, unless zero is passed, in which case zero is returned. The probability of a value with a sparse binary representation being returned is increased. This function is intended for use in test code.

ulong n_randint(flint_rand_t state, ulong limit)

Returns a uniformly pseudo random number up to but not including the given limit. If zero is passed as a parameter, an entire random limb is returned.

ulong n_urandint(flint_rand_t state, ulong limit)

Returns a uniformly pseudo random number up to but not including the given limit. If zero is passed as a parameter, an entire random limb is returned. This function provides somewhat better randomness as compared to n_randint(), especially for larger values of limit.

ulong n_randtest(flint_rand_t state)

Returns a pseudo random number with a random number of bits, from 0 to FLINT_BITS. The probability of the special values 0, 1, COEFF_MAX and WORD_MAX is increased as is the probability of a value with sparse binary representation. This random function is mainly used for testing purposes. This function is intended for use in test code.

ulong n_randtest_not_zero(flint_rand_t state)

As for n_randtest(), but does not return 0. This function is intended for use in test code.

ulong n_randprime(flint_rand_t state, ulong bits, int proved)

Returns a random prime number (proved = 1) or probable prime (proved = 0) with bits bits, where bits must be at least 2 and at most FLINT_BITS.

ulong n_randtest_prime(flint_rand_t state, int proved)

Returns a random prime number (proved = 1) or probable prime (proved = 0) with size randomly chosen between 2 and FLINT_BITS bits. This function is intended for use in test code.
4.1.3 Basic arithmetic

ulong n_pow(ulong n, ulong exp)
Returns $n^{\text{exp}}$. No checking is done for overflow. The exponent may be zero. We define $0^0 = 1$.
The algorithm simply uses a for loop. Repeated squaring is unlikely to speed up this algorithm.

ulong n_flog(ulong n, ulong b)
Returns $\lfloor \log_b n \rfloor$.
Assumes that $n \geq 1$ and $b \geq 2$.

ulong n_clog(ulong n, ulong b)
Returns $\lceil \log_b n \rceil$.
Assumes that $n \geq 1$ and $b \geq 2$.

ulong n_clog_2exp(ulong n, ulong b)
Returns $\lceil \log_2 2^n \rceil$.
Assumes that $b \geq 2$.

4.1.4 Miscellaneous

ulong n_revbin(ulong n, ulong b)
Returns the binary reverse of $n$, assuming it is $b$ bits in length, e.g. $n_{\text{revbin}}(10110, 6)$ will return 110100.

int n_sizeinbase(ulong n, int base)
Returns the exact number of digits needed to represent $n$ as a string in base base assumed to be between 2 and 36. Returns 1 when $n = 0$.

4.1.5 Basic arithmetic with precomputed inverses

ulong n_preinvert_limb_prenorm(ulong n)
Computes an approximate inverse invxl of the limb xl, with an implicit leading `1`. More formally it computes:

$$\text{invxl} = (B^2 - B \times l - 1)/x = (B^2 - 1)/x - B$$

Note that $x$ must be normalised, i.e. with msb set. This inverse makes use of the following theorem of Torbjorn Granlund and Peter Montgomery--[Lemma--8.1][GraMon1994]:

Let $d$ be normalised, $d < B$, i.e. it fits in a word, and suppose that $md < B^2 \leq (m+1)d$. Let $0 \leq n \leq Bd - 1$. Write $n = n_2B + n_1B/2 + n_0$ with $n_1 = 0$ or 1 and $n_0 < B/2$. Suppose $q_1B + q_0 = n_2B + (n_2 + n_1)(m - B) + n_1(d - B/2) + n_0$ and $0 \leq q_0 < B$. Then $0 \leq q_1 < B$ and $0 \leq n - q_1d < 2d$.

In the theorem, $m$ is the inverse of $d$. If we let $m = \text{invxl} + B$ and $d = x$ we have $md = B^2 - 1 < B^2$ and $(m + 1)x = B^2 + d - 1 \geq B^2$.

The theorem is often applied as follows: note that $n_0$ and $n_1(d - B/2)$ are both less than $B/2$. Also note that $n_1(m - B) < B$. Thus the sum of all these terms contributes at most 1 to $q_1$. We are left with $n_2B + n_2(m - B)$. But note that $(m - B)$ is precisely our precomputed inverse invxl. If we write $q_1B + q_0 = n_2B + n_2(m - B)$, then from the theorem, we have $0 \leq n - q_1d < 3d$, i.e. the quotient is out by at most 2 and is always either correct or too small.
ulong n_preinvert_limb(ulong n)

Returns a precomputed inverse of \( n \), as defined in [GraMol2010]. This precomputed inverse can be used with all of the functions that take a precomputed inverse whose names are suffixed by _preinv.

We require \( n > 0 \).

double n_precompute_inverse(ulong n)

Returns a precomputed inverse of \( n \) with double precision value \( 1/n \). This precomputed inverse can be used with all of the functions that take a precomputed inverse whose names are suffixed by _precomp.

We require \( n > 0 \).

ulong n_mod_precomp(ulong a, ulong n, double ninv)

Returns \( a \mod n \) given a precomputed inverse of \( n \) computed by n_precompute_inverse(). We require \( n < 2^{\text{FLINT\_D\_BITS}} \) and \( a < 2^{(\text{FLINT\_BITS}-1)} \) and \( 0 \leq a < n^2 \).

We assume the processor is in the standard round to nearest mode. Thus \( ninv \) is correct to 53 binary bits, the least significant bit of which we shall call a place, and can be at most half a place out. When \( a \) is multiplied by \( ninv \), the binary representation of \( a \) is exact and the mantissa is less than 2, thus we see that \( a \times ninv \) can be at most one out in the mantissa. We now truncate \( a \times ninv \) to the nearest integer, which is always a round down. Either we already have an integer, or we need to make a change down of at least 1 in the last place. In the latter case we either get precisely the exact quotient or below it as when we rounded the product to the nearest place we changed by at most half a place. In the case that truncating to an integer takes us below the exact quotient, we have rounded down by less than 1 plus half a place. But as the product is less than \( n \) and \( n \) is less than \( 2^{53} \), half a place is less than 1, thus we are out by less than 2 from the exact quotient, i.e. the quotient we have computed is the quotient we are after or one too small. That leaves only the case where we had to round up to the nearest place which happened to be an integer, so that truncating to an integer didn’t change anything. But this implies that the exact quotient \( a/n \) is less than \( 2^{-54} \) from an integer. We deal with this rare case by subtracting 1 from the quotient. Then the quotient we have computed is either exactly what we are after, or one too small.

ulong n_mod2_precomp(ulong a, ulong n, double ninv)

Returns \( a \mod n \) given a precomputed inverse of \( n \) computed by n_precompute_inverse(). There are no restrictions on \( a \) or on \( n \).

As for n_mod_precomp() for \( n < 2^{53} \) and \( a < n^2 \) the computed quotient is either what we are after or one too large or small. We deal with these cases. Otherwise we can be sure that the top 52 bits of the quotient are computed correctly. We take the remainder and adjust the quotient by multiplying the remainder by \( ninv \) to compute another approximate quotient as per mod_precomp(). Now the remainder may be either negative or positive, so the quotient we compute may be one out in either direction.

ulong n_divrem2_preinv(ulong *q, ulong a, ulong n, ulong ninv)

Returns \( a \mod n \) and sets \( q \) to the quotient of \( a \) by \( n \), given a precomputed inverse of \( n \) computed by n_preinvert_limb(). There are no restrictions on \( a \) and the only restriction on \( n \) is that it be nonzero.

This uses the algorithm of Granlund and Möller [GraMol2010]. First \( n \) is normalised and \( a \) is shifted into two limbs to compensate. Then their algorithm is applied verbatim and the remainder shifted back.

ulong n_div2_preinv(ulong a, ulong n, ulong ninv)

Returns the Euclidean quotient of \( a \) by \( n \) given a precomputed inverse of \( n \) computed by n_preinvert_limb(). There are no restrictions on \( a \) and the only restriction on \( n \) is that it be nonzero.

This uses the algorithm of Granlund and Möller [GraMol2010]. First \( n \) is normalised and \( a \) is shifted into two limbs to compensate. Then their algorithm is applied verbatim.
ulong n_mod2_preinv(ulong a, ulong n, ulong ninv)

Returns \( a \mod n \) given a precomputed inverse of \( n \) computed by \texttt{n_preinvert_limb()} . There are no restrictions on \( a \) and the only restriction on \( n \) is that it be nonzero.

This uses the algorithm of Granlund and Möller [GraMol2010]. First \( n \) is normalised and \( a \) is shifted into two limbs to compensate. Then their algorithm is applied verbatim and the result shifted back.

ulong n_divrem2_precomp(ulong *q, ulong a, ulong n, double npre)

Returns \( a \mod n \) given a precomputed inverse of \( n \) computed by \texttt{n_precompute_inverse()} and sets \( q \) to the quotient. There are no restrictions on \( a \) or on \( n \).

This is as for \texttt{n_mod2_precomp()} with some additional care taken to retain the quotient information. There are also special cases to deal with the case where \( a \) is already reduced modulo \( n \) and where \( n \) is 64 bits and \( a \) is not reduced modulo \( n \).

ulong n_ll_mod_preinv(ulong a_hi, ulong a_lo, ulong n, ulong ninv)

Returns \( a \mod n \) given a precomputed inverse of \( n \) computed by \texttt{n_preinvert_limb()} . There are no restrictions on \( a \), which will be two limbs \((a_{hi}, a_{lo})\), or on \( n \).

The old version of this function merely reduced the top limb \( a_{hi} \) modulo \( n \) so that \texttt{udiv_grmnd_preinv()} could be used.

The new version reduces the top limb modulo \( n \) as per \texttt{n_mod2_preinv()} and then the algorithm of Granlund and Möller [GraMol2010] is used again to reduce modulo \( n \).

ulong n_lll_mod_preinv(ulong a_hi, ulong a_mi, ulong a_lo, ulong n, ulong ninv)

Returns \( a \mod n \), where \( a \) has three limbs \((a_{hi}, a_{mi}, a_{lo})\), given a precomputed inverse of \( n \) computed by \texttt{n_preinvert_limb()} . It is assumed that \( a_{hi} \) is reduced modulo \( n \). There are no restrictions on \( n \).

This function uses the algorithm of Granlund and Möller [GraMol2010] to first reduce the top two limbs modulo \( n \), then does the same on the bottom two limbs.

ulong n_mulmod_precomp(ulong a, ulong b, ulong n, double ninv)

Returns \( ab \mod n \) given a precomputed inverse of \( n \) computed by \texttt{n_precompute_inverse()} . We require \( n < 2^{\text{FLINT\_D\_BITS}} \) and \( 0 \leq a, b < n \).

We assume the processor is in the standard round to nearest mode. Thus \texttt{ninv} is correct to 53 binary bits, the least significant bit of which we shall call a place, and can be at most half a place out. The product of \( a \) and \( b \) is computed with error at most half a place. When \( a \cdot b \) is multiplied by \texttt{ninv} we find that the exact quotient and computed quotient differ by less than two places. As the quotient is less than \texttt{n} this means that the exact quotient is at most 1 away from the computed quotient. We truncate this quotient to an integer which reduces the value by less than 1. We end up with a value which can be no more than two above the quotient we are after and no less than two below. However an argument similar to that for \texttt{n_mod_precomp()} shows that the truncated computed quotient cannot be two smaller than the truncated exact quotient. In other words the computed integer quotient is at most two above and one below the quotient we are after.

ulong n_mulmod2_preinv(ulong a, ulong b, ulong n, ulong ninv)

Returns \( ab \mod n \) given a precomputed inverse of \( n \) computed by \texttt{n_preinvert_limb()} . There are no restrictions on \( a, b \) or on \( n \). This is implemented by multiplying using \texttt{umul_ppmm()} and then reducing using \texttt{n_ll_mod_preinv()} .

ulong n_mulmod2(ulong a, ulong b, ulong n)

Returns \( ab \mod n \). There are no restrictions on \( a, b \) or on \( n \). This is implemented by multiplying using \texttt{umul_ppmm()} and then reducing using \texttt{n_ll_mod_preinv()} after computing a precomputed inverse.

ulong n_mulmod_preinv(ulong a, ulong b, ulong n, ulong ninv, ulong norm)

Returns \( ab \pmod n \) given a precomputed inverse of \( n \) computed by \texttt{n_preinvert_limb()}, assuming \( a \) and \( b \) are reduced modulo \( n \) and \( n \) is normalised, i.e. with most significant bit set. There are no other restrictions on \( a, b \) or \( n \).

4.1. \texttt{ulong\_extras.h} – arithmetic and number-theoretic functions for single-word integers 107
The value \( \text{norm} \) is provided for convenience. As \( n \) is required to be normalised, it may be that \( a \) and \( b \) have been shifted to the left by \( \text{norm} \) bits before calling the function. Their product then has an extra factor of \( 2^{\text{norm}} \). Specifying a nonzero \( \text{norm} \) will shift the product right by this many bits before reducing it.

The algorithm used is that of Granlund and Möller [GraMol2010].

### 4.1.6 Greatest common divisor

\[
\text{ulong n_gcd(ulong } x, \text{ ulong } y) \\
\text{Returns the greatest common divisor } g \text{ of } x \text{ and } y. \text{ No assumptions are made about the values } x \text{ and } y. \\
\text{This function wraps GMP’s mpn_gcd_1.}
\]

\[
\text{ulong n_gcdinv(ulong *a, ulong x, ulong y) } \text{ Returns the greatest common divisor } g \text{ of } x \text{ and } y \text{ and computes } a \text{ such that } 0 \leq a < y \text{ and } ax \equiv \text{gcd}(x,y) \mod y, \text{ when this is defined. We require } x < y. \\
\text{When } y = 1 \text{ the greatest common divisor is set to } 1 \text{ and } a \text{ is set to } 0. \\
\text{This is merely an adaption of the extended Euclidean algorithm computing just one cofactor and reducing it modulo } y.
\]

\[
\text{ulong n_xgcd(ulong *a, ulong *b, ulong x, ulong y) } \text{ Returns the greatest common divisor } g \text{ of } x \text{ and } y \text{ and unsigned values } a \text{ and } b \text{ such that } ax - by = g. \text{ We require } x \geq y. \\
\text{We claim that computing the extended greatest common divisor via the Euclidean algorithm always results in cofactor } |a| < x/2, |b| < x/2, \text{ with perhaps some small degenerate exceptions.} \\
\text{We proceed by induction.} \\
\text{Suppose we are at some step of the algorithm, with } x_n = qy_n + r \text{ with } r \geq 1, \text{ and suppose } 1 = sy_n - tr \text{ with } s < y_n/2, t < y_n/2 \text{ by hypothesis.} \\
\text{Write } 1 = sy_n - t(x_n - qy_n) = (s + tq)y_n - tx_n. \\
\text{It suffices to show that } (s + tq) < x_n/2 \text{ as } t < y_n/2 < x_n/2, \text{ which will complete the induction step.} \\
\text{But at the previous step in the backsubstitution we would have had } 1 = sr - cd \text{ with } s < r/2 \text{ and } c < r/2. \\
\text{Then } s + tq < r/2 + y_n/2q = (r + qy_n)/2 = x_n/2. \\
\text{See the documentation of } \text{n_gcd()} \text{ for a description of the branching in the algorithm, which is faster than using division.}
\]

### 4.1.7 Jacobi and Kronecker symbols

\[
\text{int n_jacobi(mp_limb_signed_t x, ulong y) } \\
\text{Computes the Jacobi symbol } \left( \frac{x}{y} \right) \text{ for any } x \text{ and odd } y.
\]

\[
\text{int n_jacobi_unsigned(ulong x, ulong y) } \\
\text{Computes the Jacobi symbol, allowing } x \text{ to go up to a full limb.}
\]
4.1.8 Modular Arithmetic

ulong n_addmod(ulong a, ulong b, ulong n)
Returns \((a + b) \mod n\).

ulong n_submod(ulong a, ulong b, ulong n)
Returns \((a - b) \mod n\).

ulong n_invmmod(ulong x, ulong y)
Returns the inverse of \(x\) modulo \(y\), if it exists. Otherwise an exception is thrown.
This is merely an adaption of the extended Euclidean algorithm with appropriate normalisation.

ulong n_powmod_precomp(ulong a, mp_limb_signed_t exp, ulong n, double npre)
Returns \(a^\exp \mod n\) given a precomputed inverse of \(n\) computed by \texttt{n_precompute_inverse()}.
We require \(n < 2^{253}\) and \(0 \leq a < n\). There are no restrictions on \(\exp\), i.e. it can be negative.
This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo \(n\) at each step.

ulong n_powmod_ui_precomp(ulong a, ulong exp, ulong n, double npre)
Returns \(a^\exp \mod n\) given a precomputed inverse of \(n\) computed by \texttt{n_precompute_inverse()}.
The exponent \(\exp\) is unsigned and so can be larger than allowed by \texttt{n_powmod_precomp()}.
This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo \(n\) at each step.

ulong n_powmod(ulong a, mp_limb_signed_t exp, ulong n)
Returns \(a^\exp \mod n\). We require \(n < 2^{FLINT\_D\_BITS}\) and \(0 \leq a < n\). There are no restrictions on \(\exp\), i.e. it can be negative.
This is implemented by precomputing an inverse and calling the \texttt{precomp} version of this function.

ulong n_powmod2_preinv(ulong a, mp_limb_signed_t exp, ulong n, ulong ninv)
Returns \((a^\exp) \mod n\) given a precomputed inverse of \(n\) computed by \texttt{n_preinvert_limb()}. We require \(0 \leq a < n\), but there are no restrictions on \(n\) or on \(\exp\), i.e. it can be negative.
This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo \(n\) at each step.

If \(\exp\) is negative but \(a\) is not invertible modulo \(n\), an exception is raised.

ulong n_powmod2(ulong a, mp_limb_signed_t exp, ulong n)
Returns \((a^\exp) \mod n\). We require \(0 \leq a < n\), but there are no restrictions on \(n\) or on \(\exp\), i.e. it can be negative.
This is implemented by precomputing an inverse limb and calling the \texttt{preinv} version of this function.

If \(\exp\) is negative but \(a\) is not invertible modulo \(n\), an exception is raised.

ulong n_powmod2_ui_preinv(ulong a, ulong exp, ulong n, ulong ninv)
Returns \((a^\exp) \mod n\) given a precomputed inverse of \(n\) computed by \texttt{n_preinvert_limb()}. We require \(0 \leq a < n\), but there are no restrictions on \(n\) or on \(\exp\). The exponent \(\exp\) must not be negative.
This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo \(n\) at each step.

ulong n_powmod2_fmpz_preinv(ulong a, const fmpz_t exp, ulong n, ulong ninv)
Returns \((a^\exp) \mod n\) given a precomputed inverse of \(n\) computed by \texttt{n_preinvert_limb()}. We require \(0 \leq a < n\). The exponent \(\exp\) must not be negative.
This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo \(n\) at each step.
ulong n_sqrtmod(ulong a, ulong p)
    If \( p \) is prime, compute a square root of \( a \) modulo \( p \) if \( a \) is a quadratic residue modulo \( p \), otherwise return 0.

    If \( p \) is not prime the result is with high probability 0, indicating that \( p \) is not prime, or \( a \) is not a square modulo \( p \). Otherwise the result is meaningless.

    Assumes that \( a \) is reduced modulo \( p \).

slong n_sqrtmod_2pow(ulong **sqrt, ulong a, slong exp)
    Computes all the square roots of \( a \) modulo \( 2^\exp \). The roots are stored in an array which is created and whose address is stored in the location pointed to by \( \text{sqrt} \). The array of roots is allocated by the function but must be cleaned up by the user by calling \text{flint_free}. The number of roots is returned by the function. If \( a \) is not a quadratic residue modulo \( 2^\exp \) then 0 is returned by the function and the location \( \text{sqrt} \) points to is set to NULL.

slong n_sqrtmod_primepow(ulong **sqrt, ulong a, ulong p, slong exp)
    Computes all the square roots of \( a \) modulo \( p^\exp \). The roots are stored in an array which is created and whose address is stored in the location pointed to by \( \text{sqrt} \). The array of roots is allocated by the function but must be cleaned up by the user by calling \text{flint_free}. The number of roots is returned by the function. If \( a \) is not a quadratic residue modulo \( p^\exp \) then 0 is returned by the function and the location \( \text{sqrt} \) points to is set to NULL.

slong n_sqrtmodn(ulong **sqrt, ulong a, n_factor_t *fac)
    Computes all the square roots of \( a \) modulo \( m \) given the factorisation of \( m \) in \( \text{fac} \). The roots are stored in an array which is created and whose address is stored in the location pointed to by \( \text{sqrt} \). The array of roots is allocated by the function but must be cleaned up by the user by calling \text{flint_free}. The number of roots is returned by the function. If \( a \) is not a quadratic residue modulo \( m \) then 0 is returned by the function and the location \( \text{sqrt} \) points to is set to NULL.

mp_limb_t n_mulmod_shoup(mp_limb_t w, mp_limb_t t, mp_limb_t w_precomp, mp_limb_t p)
    Returns \( wt \mod p \) given a precomputed scaled approximation of \( w/p \) computed by \text{n_mulmod_precomp_shoup}. The value of \( p \) should be less than \( 2^{\text{FLINT_BITS}}-1 \). \( w \) and \( t \) should be less than \( p \). Works faster than \text{n_mulmod2_preinv} if \( w \) fixed and \( t \) from array (for example, scalar multiplication of vector).

mp_limb_t n_mulmod_precomp_shoup(mp_limb_t w, mp_limb_t p)
    Returns \( w' \), scaled approximation of \( w/p \). \( w' \) is equal to the integer part of \( w \cdot 2^{\text{FLINT_BITS}}/p \).

4.1.9 Divisibility testing

int n_divides(mp_limb_t *q, mp_limb_t n, mp_limb_t p)
    Returns 1 if \( p \) divides \( n \) and sets \( q \) to the quotient, otherwise returns 0 and sets \( q \) to 0.

4.1.10 Prime number generation and counting

void n_primes_init(n_primes_t iter)
    Initialises the prime number iterator \( \text{iter} \) for use.

void n_primes_clear(n_primes_t iter)
    Frees memory allocated by the prime number iterator \( \text{iter} \).

ulong n_primes_next(n_primes_t iter)
    Returns the next prime number and advances the state of \( \text{iter} \). The first call returns 2.

Small primes are looked up from \text{flint_small_primes}. When this table is exhausted, primes are generated in blocks by calling \text{n_primes_sieve_range}.
void n_primes_jump_after(n_primes_t iter, ulong n)
    Changes the state of iter to start generating primes after n (excluding n itself).

void n_primes_extend_small(n_primes_t iter, ulong bound)
    Extends the table of small primes in iter to contain at least two primes larger than or equal to bound.

void n_primes_sieve_range(n_primes_t iter, ulong a, ulong b)
    Sets the block endpoints of iter to the smallest and largest odd numbers between a and b inclusive, and sieves to mark all odd primes in this range. The iterator state is changed to point to the first number in the sieved range.

void n_compute_primes(ulong num_primes)
    Precomputes at least num_primes primes and their double precomputed inverses and stores them in an internal cache. Assuming that FLINT has been built with support for thread-local storage, each thread has its own cache.

const ulong *n_primes_arr_readonly(ulong num_primes)
    Returns a pointer to a read-only array of the first num_primes prime numbers. The computed primes are cached for repeated calls. The pointer is valid until the user calls n_cleanup_primes() in the same thread.

const double *n_prime_inverses_arr_readonly(ulong n)
    Returns a pointer to a read-only array of inverses of the first num_primes prime numbers. The computed primes are cached for repeated calls. The pointer is valid until the user calls n_cleanup_primes() in the same thread.

void n_cleanup_primes()
    Frees the internal cache of prime numbers used by the current thread. This will invalidate any pointers returned by n_primes_arr_readonly() or n_prime_inverses_arr_readonly().

ulong n_nextprime(ulong n, int proved)
    Returns the next prime after n. Assumes the result will fit in an ulong. If proved is 0, i.e. false, the prime is not proven prime, otherwise it is.

ulong n_prime_pi(ulong n)
    Returns the value of the prime counting function \(\pi(n)\), i.e. the number of primes less than or equal to n. The invariant \(n_{\text{prime}}(n_{\text{nth prime}}(n)) = n\).
    Currently, this function simply extends the table of cached primes up to an upper limit and then performs a binary search.

void n_prime_pi_bounds(ulong *lo, ulong *hi, ulong n)
    Calculates lower and upper bounds for the value of the prime counting function \(\pi(n)\). If \(\text{lo}\) and \(\text{hi}\) point to the same location, the high value will be stored.
    This does a table lookup for small values, then switches over to some proven bounds.
    The upper approximation is \(1.25506n/\ln n\), and the lower is \(n/\ln n\). These bounds are due to Rosser and Schoenfeld [RosSch1962] and valid for \(n \geq 17\).
    We use the number of bits in \(n\) (or one less) to form an approximation to \(\ln n\), taking care to use a value too small or too large to maintain the inequality.

ulong n_nth_prime(ulong n)
    Returns the \(n\)th prime number \(p_n\), using the mathematical indexing convention \(p_1 = 2, p_2 = 3, \ldots\).
    This function simply ensures that the table of cached primes is large enough and then looks up the entry.

4.1. ulong_extras.h – arithmetic and number-theoretic functions for single-word integers
void n_nth_prime_bounds(ulong *lo, ulong *hi, ulong n)
Calculates lower and upper bounds for the \( n \)th prime number \( p_n \), \( lo <= p_n <= hi \). If \( lo \) and \( hi \) point to the same location, the high value will be stored. Note that this function will overflow for sufficiently large \( n \).

We use the following estimates, valid for \( n > 5 \):

\[
p_n > n(\ln n + \ln \ln n - 1)
\]

\[
p_n < n(\ln n + \ln \ln n)
\]

\[
p_n < n(\ln n + \ln \ln n - 0.9427) \quad (n \geq 15985)
\]

The first inequality was proved by Dusart [Dus1999], and the last is due to Massias and Robin [MasRob1996]. For a further overview, see http://primes.utm.edu/howmany.shtml.

We bound \( \ln n \) using the number of bits in \( n \) as in \( n_prime_pi_bounds() \), and estimate \( \ln \ln n \) to the nearest integer; this function is nearly constant.

### 4.1.11 Primality testing

#### int n_is_oddprime_small(ulong n)

Returns 1 if \( n \) is an odd prime smaller than FLINT_ODDPRIME_SMALL_CUTOFF. Expects \( n \) to be odd and smaller than the cutoff.

This function merely uses a lookup table with one bit allocated for each odd number up to the cutoff.

#### int n_is_oddprime_binary(ulong n)

This function performs a simple binary search through the table of cached primes for \( n \). If it exists in the array it returns 1, otherwise 0. For the algorithm to operate correctly \( n \) should be odd and at least 17.

Lower and upper bounds are computed with \( n_prime_pi_bounds() \). Once we have bounds on where to look in the table, we refine our search with a simple binary algorithm, taking the top or bottom of the current interval as necessary.

#### int n_is_prime_pocklington(ulong n, ulong iterations)

Tests if \( n \) is a prime using the Pocklington–Lehmer primality test. If 1 is returned \( n \) has been proved prime. If 0 is returned \( n \) is composite. However –1 may be returned if nothing was proved either way due to the number of iterations being too small.

The most time consuming part of the algorithm is factoring \( n - 1 \). For this reason \( n_factor_partial() \) is used, which uses a combination of trial factoring and Hart’s one line factor algorithm [Har2012] to try to quickly factor \( n - 1 \). Additionally if the cofactor is less than the square root of \( n - 1 \) the algorithm can still proceed.

One can also specify a number of iterations if less time should be taken. Simply set this to \( \text{WORD}(0) \) if this is irrelevant. In most cases a greater number of iterations will not significantly affect timings as most of the time is spent factoring.

See https://mathworld.wolfram.com/PocklingtonsTheorem.html for a description of the algorithm.

#### int n_is_prime_pseudosquare(ulong n)

Tests if \( n \) is a prime according to Theorem 2.7 [LukPatWil1996].

We first factor \( N \) using trial division up to some limit \( B \). In fact, the number of primes used in the trial factoring is at most FLINT_PSEUDOSQUARES_CUTOFF.

Next we compute \( N/B \) and find the next pseudosquare \( L_p \) above this value, using a static table as per https://oeis.org/A002189/b002189.txt.

As noted in the text, if \( p \) is prime then Step 3 will pass. This test rejects many composites, and so by this time we suspect that \( p \) is prime. If \( N \) is 3 or 7 modulo 8, we are done, and \( N \) is prime.
We now run a probable prime test, for which no known counterexamples are known, to reject any composites. We then proceed to prove $N$ prime by executing Step 4. In the case that $N$ is 1 modulo 8, if Step 4 fails, we extend the number of primes $p_i$ at Step 3 and hope to find one which passes Step 4. We take the test one past the largest $p$ for which we have pseudosquares $L_p$ tabulated, as this already corresponds to the next $L_p$ which is bigger than $2^{64}$ and hence larger than any prime we might be testing.

As explained in the text, Condition 4 cannot fail if $N$ is prime.

The possibility exists that the probable prime test declares a composite prime. However in that case an error is printed, as that would be of independent interest.

```c
int n_is_prime(ulong n)
Tests if n is a prime. This first sieves for small prime factors, then simply calls n_is_probabprime(). This has been checked against the tables of Feitsma and Galway [Page 143 of [CraPom2005]], regarding Fibonacci pseudoprimes. This has been checked against the tables of Feitsma and Galway. This has been checked against the tables of Feitsma and Galway http://www.cecm.sfu.ca/Pseudoprimes/index-2-to-64.html and thus constitutes a check for primality (rather than just pseudoprimality) up to $2^{64}$.

In future, this test may produce and check a certificate of primality. This is likely to be significantly slower for prime inputs.

int n_is_strong_probabprime_precomp(ulong n, double npre, ulong a, ulong d)
Tests if $n$ is a strong probable prime to the base $a$. We require that $d$ is set to the largest odd factor of $n - 1$ and $npre$ is a precomputed inverse of $n$ computed with n_precompute_inverse(). We also require that $n < 2^{34}$, $a$ to be reduced modulo $n$ and not 0 and $n$ to be odd.

If we write $n - 1 = 2^r d$ where $d$ is odd then $n$ is a strong probable prime to the base $a$, i.e. an $a$-SPRP, if either $a^d = 1 \pmod n$ or $(a^d)^{2^r} = -1 \pmod n$ for some $r$ less than $s$.

A description of strong probable primes is given here: https://mathworld.wolfram.com/StrongPseudoprime.html

int n_is_strong_probabprime2_preinv(ulong n, ulong ninv, ulong a, ulong d)
Tests if $n$ is a strong probable prime to the base $a$. We require that $d$ is set to the largest odd factor of $n - 1$ and $npre$ is a precomputed inverse of $n$ computed with n_preinvert_limb(). We require $a$ to be reduced modulo $n$ and non-zero, and $n$ to be odd.

If we write $n - 1 = 2^r d$ where $d$ is odd then $n$ is a strong probable prime to the base $a$ (an $a$-SPRP) if either $a^d = 1 \pmod n$ or $(a^d)^{2^r} = -1 \pmod n$ for some $r$ less than $s$.

A description of strong probable primes is given here: https://mathworld.wolfram.com/StrongPseudoprime.html

int n_is_probabprime_fermat(ulong n, ulong i)
Returns 1 if $n$ is a base $i$ Fermat probable prime. Requires $1 < i < n$ and that $i$ does not divide $n$.

By Fermat’s Little Theorem if $i^{n-1}$ is not congruent to 1 then $n$ is not prime.

int n_is_probabprime_fibonacci(ulong n)
Let $F_j$ be the $j$th element of the Fibonacci sequence 0, 1, 1, 2, 3, 5,…, starting at $j = 0$. Then if $n$ is prime we have $F_{n-(n/5)} = 0 \pmod n$, where $(n/5)$ is the Jacobi symbol.

For further details, see pp. 142 [CraPom2005].

We require that $n$ is not divisible by 2 or 5.

int n_is_probabprime_BPSW(ulong n)
Implements a Baillie–Pomerance–Selfridge–Wagstaff probable primality test. This is a variant of the usual BPSW test (which only uses strong base-2 probable prime and Lucas-Selfridge tests, see Baillie and Wagstaff [BaiWag1980]).

This implementation makes use of a weakening of the usual Baillie-PSW test given in [Chen2003], namely replacing the Lucas test with a Fibonacci test when $n \equiv 2, 3 \pmod 5$ (see also the comment on page 143 of [CraPom2005]), regarding Fibonacci pseudoprimes.

There are no known counterexamples to this being a primality test.

4.1. ulong_extras.h – arithmetic and number-theoretic functions for single-word integers 113
Up to $2^{64}$ the test we use has been checked against tables of pseudoprimes. Thus it is a primality test up to this limit.

```c
int n_is_probabprime_lucas(ulong n)
```

For details on Lucas pseudoprimes, see [pp. 143] [CraPom2005].

We implement a variant of the Lucas pseudoprime test similar to that described by Baillie and Wagstaff [BaiWag1980].

```c
int n_is_probabprime(ulong n)
```

Tests if $n$ is a probable prime. Up to `FLINT_ODDPRIME_SMALL_CUTOFF` this algorithm uses `n_is_oddprime_small()` which uses a lookup table.

Next it calls `n_compute_primes()` with the maximum table size and uses this table to perform a binary search for $n$ up to the table limit.

Then up to 1050535501 it uses a number of strong probable prime tests, `n_is_strong_probabprime_preinv()`, etc., for various bases. The output of the algorithm is guaranteed to be correct up to this bound due to exhaustive tables, described at [http://uucode.com/obf/dalbec/alg.html](http://uucode.com/obf/dalbec/alg.html).

Beyond that point the BPSW probabilistic primality test is used, by calling the function `n_is_probabprime_BPSW()`. There are no known counterexamples, and it has been checked against the tables of Feitsma and Galway and up to the accuracy of those tables, this is an exhaustive check up to $2^{64}$, i.e. there are no counterexamples.

### 4.1.12 Chinese remaindering

```c
ulong n_CRT(ulong r1, ulong m1, ulong r2, ulong m2)
```

Use the Chinese Remainder Theorem to return the unique value $0 \leq x < M$ congruent to $r_1$ modulo $m_1$ and $r_2$ modulo $m_2$, where $M = m_1 \times m_2$ is assumed to fit a ulong.

It is assumed that $m_1$ and $m_2$ are positive integers greater than 1 and coprime. It is assumed that $0 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$.

### 4.1.13 Square root and perfect power testing

```c
ulong n_sqrt(ulong a)
```

Computes the integer truncation of the square root of $a$.

The implementation uses a call to the IEEE floating point `sqrt` function. The integer itself is represented by the nearest double and its square root is computed to the nearest place. If $a$ is one below a square, the rounding may be up, whereas if it is one above a square, the rounding will be down. Thus the square root may be one too large in some instances which we then adjust by checking if we have the right value. We also have to be careful when the square of this too large value causes an overflow. The same assumptions hold for a single precision float provided the square root itself can be represented in a single float, i.e. for $a < 281474976710656 = 2^{46}$.

```c
ulong n_sqrtrem(ulong r, ulong a)
```

Computes the integer truncation of the square root of $a$.

The integer itself is represented by the nearest double and its square root is computed to the nearest place. If $a$ is one below a square, the rounding may be up, whereas if it is one above a square, the rounding will be down. Thus the square root may be one too large in some instances which we then adjust by checking if we have the right value. We also have to be careful when the square of this too large value causes an overflow. The same assumptions hold for a single precision float provided the square root itself can be represented in a single float, i.e. for $a < 281474976710656 = 2^{46}$.

The remainder is computed by subtracting the square of the computed square root from $a$. 

114 Chapter 4. Integers
int n_is_square(ulong x)
    Returns 1 if \( x \) is a square, otherwise 0.

This code first checks if \( x \) is a square modulo 64, 63 = 3 \times 3 \times 7 and 65 = 5 \times 13, using lookup tables, and if so it then takes a square root and checks that the square of this equals the original value.

int n_is_perfect_power235(ulong n)
    Returns 1 if \( n \) is a perfect square, cube or fifth power.

This function uses a series of modular tests to reject most non-235-powers. Each modular test returns a value from 0 to 7 whose bits respectively indicate whether the value is a square, cube or fifth power modulo the given modulus. When these are logically AND-ed together, this gives a powerful test which will reject most non-235 powers.

If a bit remains set indicating it may be a square, a standard square root test is performed. Similarly a cube root or fifth root can be taken, if indicated, to determine whether the power of that root is exactly equal to \( n \).

int n_is_perfect_power(ulong *root, ulong n)
    If \( n = r^k \), return \( k \) and set \( \text{root} \) to \( r \). Note that 0 and 1 are considered squares. No guarantees are made about \( r \) or \( k \) being the minimum possible value.

ulong n_rootrem(ulong *remainder,ulong n, ulong root)
    This function uses the Newton iteration method to calculate the nth root of a number. First approximation is calculated by an algorithm mentioned in this article: https://en.wikipedia.org/wiki/Fast_inverse_square_root. Instead of the inverse square root, the nth root is calculated.

Returns the integer part of \( n ^ {1/root} \). Remainder is set as \( n - \text{base}^\text{root} \). In case \( n < 1 \) or \( \text{root} < 1 \), 0 is returned.

ulong n_cbrt(ulong n)
    This function returns the integer truncation of the cube root of \( n \). First approximation is calculated by an algorithm mentioned in this article: https://en.wikipedia.org/wiki/Fast_inverse_square_root. Instead of the inverse square root, the cube root is calculated. This function uses different algorithms to calculate the cube root, depending upon the size of \( n \). For numbers greater than 2^{46}, it uses \( n_{cb}t_{che}byshev_{approx}() \). Otherwise, it makes use of the iteration, \( x \leftarrow x - (x \cdot x \cdot x - a) \cdot x / (2 \cdot x \cdot x \cdot x + a) \) for getting a good estimate, as mentioned in the paper by W. Kahan [Kahan1991].

ulong n_cbrt_newton_iteration(ulong n)
    This function returns the integer truncation of the cube root of \( n \). Makes use of Newton iterations to get a close value, and then adjusts the estimate so as to get the correct value.

ulong n_cbrt_binary_search(ulong n)
    This function returns the integer truncation of the cube root of \( n \). Uses binary search to get the correct value.

ulong n_cbrt_chebyshev_approx(ulong n)
    This function returns the integer truncation of the cube root of \( n \). The number is first expressed in the form \( x \cdot 2^{e_{xp}} \). This ensures \( x \) is in the range [0.5, 1]. Cube root of \( x \) is calculated using Chebyshev’s approximation polynomial for the function \( y = x^{1/3} \). The values of the coefficient are calculated from the Python module mpmath, https://mpmath.org, using the function chebyfit. \( x \) is multiplied by \( 2^{e_{xp}} \) and the cube root of 1, 2 or 4 (according to \( e_{xp}%3 \)).

ulong n_cbrtrem(ulong *remainder, ulong n)
    This function returns the integer truncation of the cube root of \( n \). Remainder is set as \( n \) minus the cube of the value returned.
4.1.14 Factorisation

void n_factor_init(n_factor_t *factors)
    Initializes factors.

int n_remove(ulong *n, ulong p)
    Removes the highest possible power of \( p \) from \( n \), replacing \( n \) with the quotient. The return value is the highest power of \( p \) that divided \( n \). Assumes \( n \) is not 0.

    For \( p = 2 \), trailing zeroes are counted. For other primes \( p \), repeatedly squared and stored in a table of powers with the current highest power of \( p \) removed at each step until no higher power can be removed. The algorithm then proceeds down the power tree again removing powers of \( p \) until none remain.

int n_remove2_precomp(ulong *n, ulong p, double ppre)
    Removes the highest possible power of \( p \) from \( n \), replacing \( n \) with the quotient. The return value is the highest power of \( p \) that divided \( n \). Assumes \( n \) is not 0. We require \( ppre \) to be set to a precomputed inverse of \( p \) computed with n_precompute_inverse().

    For \( p = 2 \), trailing zeroes are counted. For other primes \( p \), we make repeated use of n_divrem2_precomp() until division by \( p \) is no longer possible.

void n_factor_insert(n_factor_t *factors, ulong p, ulong exp)
    Inserts the given prime power factor \( p^\text{exp} \) into the n_factor_t factors. See the documentation for n_factor_trial() for a description of the n_factor_t type.

    The algorithm performs a simple search to see if \( p \) already exists as a prime factor in the structure. If so the exponent there is increased by the supplied exponent. Otherwise a new factor \( p^\text{exp} \) is added to the end of the structure.

    There is no test code for this function other than its use by the various factoring functions, which have test code.

ulong n_factor_trial_range(n_factor_t *factors, ulong n, ulong start, ulong num_primes)
    Trial factor \( n \) with the first \( \text{num} \text{primes} \) primes, but starting at the prime with index \( \text{start} \) (counting from zero).

    One requires an initialised n_factor_t structure, but factors will be added by default to an already used n_factor_t. Use the function n_factor_init() defined in ulong_extras if initialisation has not already been completed on factors.

    Once completed, num will contain the number of distinct prime factors found. The field \( p \) is an array of ulongs containing the distinct prime factors, \( \text{exp} \) an array containing the corresponding exponents.

    The return value is the unfactored cofactor after trial factoring is done.

    The function calls n_compute_primes() automatically. See the documentation for that function regarding limits.

    The algorithm stops when the current prime has a square exceeding \( n \), as no prime factor of \( n \) can exceed this unless \( n \) is prime.

    The precomputed inverses of all the primes computed by n_compute_primes() are utilised with the n_remove2_precomp() function.

ulong n_factor_trial(n_factor_t *factors, ulong n, ulong num_primes)
    This function calls n_factor_trial_range(), with the value of 0 for start. By default this adds factors to an already existing n_factor_t or to a newly initialised one.

ulong n_factor_power235(ulong *exp, ulong n)
    Returns 0 if \( n \) is not a perfect square, cube or fifth power. Otherwise it returns the root and sets \( \text{exp} \) to either 2, 3 or 5 appropriately.
This function uses a series of modular tests to reject most non-235-powers. Each modular test returns a value from 0 to 7 whose bits respectively indicate whether the value is a square, cube, or fifth power modulo the given modulus. When these are logically AND-ed together, this gives a powerful test which will reject most non-235 powers.

If a bit remains set indicating it may be a square, a standard square root test is performed. Similarly a cube root or fifth root can be taken, if indicated, to determine whether the power of that root is exactly equal to $n$.

\[ \text{ulong } n\text{\_factor\_one\_line}(\text{ulong } n, \text{ulong } \text{iters}) \]
This implements Bill Hart’s one line factoring algorithm [Har2012]. It is a variant of Fermat’s algorithm which cycles through a large number of multipliers instead of incrementing the square root. It is faster than SQUFOF for $n$ less than about $2^{40}$.

\[ \text{ulong } \text{n\_factor\_lehman}(\text{ulong } n) \]
Lehman’s factoring algorithm. Currently works up to $10^{16}$, but is not particularly efficient and so is not used in the general factor function. Always returns a factor of $n$.

\[ \text{ulong } \text{n\_factor\_SQUFOF}(\text{ulong } n, \text{ulong } \text{iters}) \]
Attempts to split $n$ using the given number of iterations of SQUFOF. Simply set \text{iters} to \text{WORD}(0) for maximum persistence.

The version of SQUFOF implemented here is as described by Gower and Wagstaff [GowWag2008]. We start by trying SQUFOF directly on $n$. If that fails we multiply it by each of the primes in flint\_primes\_small in turn. As this multiplication may result in a two limb value we allow this in our implementation of SQUFOF. As SQUFOF works with values about half the size of $n$ it only needs single limb arithmetic internally.

If SQUFOF fails to factor $n$ we return 0, however with \text{iters} large enough this should never happen.

\[ \text{void } \text{n\_factor}(\text{n\_factor\_t } *\text{factors}, \text{ulong } n, \text{int } \text{proved}) \]
Factors $n$ with no restrictions on $n$. If the prime factors are required to be checked with a primality test, one may set \text{proved} to 1, otherwise set it to 0, and they will only be probable primes. NB: at the present there is no difference because the probable prime tests have been exhaustively tested up to $2^{64}$.

However, in future, this flag may produce and separately check a primality certificate. This may be quite slow (and probably no less reliable in practice).

For details on the \text{n\_factor\_t} structure, see \text{n\_factor\_trial()}.

This function first tries trial factoring with a number of primes specified by the constant \text{FLINT\_FACTOR\_TRIAL\_PRIMES}. If the cofactor is 1 or prime the function returns with all the factors.

Otherwise, the cofactor is placed in the array \text{factor\_arr}. Whilst there are factors remaining in there which have not been split, the algorithm continues. At each step each factor is first checked to determine if it is a perfect power. If so it is replaced by the power that has been found. Next if the factor is small enough and composite, in particular, less than \text{FLINT\_FACTOR\_ONE\_LINE\_MAX} then \text{n\_factor\_one\_line()} is called with \text{FLINT\_FACTOR\_ONE\_LINE\_ITERS} to try and split the factor. If that fails or the factor is too large for \text{n\_factor\_one\_line()} then \text{n\_factor\_SQUFOF()} is called, with \text{FLINT\_FACTOR\_SQUFOF\_ITERS}. If that fails an error results and the program aborts. However this should not happen in practice.

\[ \text{ulong } \text{n\_factor\_trial\_partial}(\text{n\_factor\_t } *\text{factors}, \text{ulong } n, \text{ulong } *\text{prod}, \text{ulong } \text{num\_primes}, \text{ulong } \text{limit}) \]
Attempts trial factoring of $n$ with the first \text{num\_primes} primes, but stops when the product of prime factors so far exceeds \text{limit}.

One requires an initialised \text{n\_factor\_t} structure, but factors will be added by default to an already used \text{n\_factor\_t}. Use the function \text{n\_factor\_init()} defined in ulong\_extras if initialisation has not already been completed on \text{factors}.

4.1. ulong\_extras.h – arithmetic and number-theoretic functions for single-word integers
Once completed, num will contain the number of distinct prime factors found. The field \( p \) is an array of ulongs containing the distinct prime factors, \( \text{exp} \) an array containing the corresponding exponents.

The return value is the unfactored cofactor after trial factoring is done. The value \( \text{prod} \) will be set to the product of the factors found.

The function calls \( \text{n}_\text{compute}_\text{primes()} \) automatically. See the documentation for that function regarding limits.

The algorithm stops when the current prime has a square exceeding \( n \), as no prime factor of \( n \) can exceed this unless \( n \) is prime.

The precomputed inverses of all the primes computed by \( \text{n}_\text{compute}_\text{primes()} \) are utilised with the \( \text{n}_\text{remove2}_\text{precomp()} \) function.

\( \text{ulong} \ \text{n}_\text{factor}_\text{partial}(\text{n}_\text{factor}_\text{t} \ * \text{factors}, \ \text{ulong} \ n, \ \text{ulong} \ \text{limit}, \ \text{int} \ \text{proved}) \)

Factors \( n \), but stops when the product of prime factors so far exceeds \( \text{limit} \).

One requires an initialised \( \text{n}_\text{factor}_\text{t} \) structure, but factors will be added by default to an already used \( \text{n}_\text{factor}_\text{t} \). Use the function \( \text{n}_\text{factor}_\text{init()} \) defined in \( \text{ulong}_\text{extras} \) if initialisation has not already been completed on \( \text{factors} \).

On exit, \( \text{num} \) will contain the number of distinct prime factors found. The field \( \text{p} \) is an array of ulongs containing the distinct prime factors, \( \text{exp} \) an array containing the corresponding exponents.

The return value is the unfactored cofactor after factoring is done.

The factors are proved prime if \( \text{proved} \) is 1, otherwise they are merely probably prime.

\( \text{ulong} \ \text{n}_\text{factor}_\text{pp1}(\text{ulong} \ n, \ \text{ulong} \ \text{B1}, \ \text{ulong} \ \text{c}) \)

Factors \( n \) using Williams’ \( \text{p} + 1 \) factoring algorithm, with prime limit set to \( \text{B1} \). We require \( c \) to be set to a random value. Each trial of the algorithm with a different value of \( c \) gives another chance to factor \( n \), with roughly exponentially decreasing chance of finding a missing factor. If \( p + 1 \) (or \( p - 1 \)) is not smooth for any factor \( p \) of \( n \), the algorithm will never succeed. The value \( c \) should be less than \( n \) and greater than 2.

If the algorithm succeeds, it returns the factor, otherwise it returns 0 or 1 (the trivial factors modulo \( n \)).

\( \text{ulong} \ \text{n}_\text{factor}_\text{pp1}_\text{wrapper}(\text{ulong} \ n) \)

A simple wrapper around \( \text{n}_\text{factor}_\text{pp1} \) which works in the range 31-64 bits. Below this point, trial factoring will always succeed. This function mainly exists for \( \text{n}_\text{factor} \) and is tuned to minimise the time for \( \text{n}_\text{factor} \) on numbers that reach the \( \text{n}_\text{factor}_\text{pp1} \) stage, i.e. after trial factoring and one line factoring.

\( \text{int} \ \text{n}_\text{factor}_\text{pollard}_\text{brent}_\text{single}(\text{mp}_\text{limb}_\text{t} \ * \text{factor}, \ \text{mp}_\text{limb}_\text{t} \ n, \ \text{mp}_\text{limb}_\text{t} \ \text{ninvv}, \ \text{mp}_\text{limb}_\text{t} \ \text{ai}, \ \text{mp}_\text{limb}_\text{t} \ \text{xi}, \ \text{mp}_\text{limb}_\text{t} \ \text{normbits}, \ \text{mp}_\text{limb}_\text{t} \ \text{max}_\text{iters}) \)

Pollard Rho algorithm (with Brent modification) for integer factorization. Assumes that the \( n \) is not prime. \( \text{factor} \) is set as the factor if found. It is not assured that the factor found will be prime. Does not compute the complete factorization, just one factor. Returns 1 if factorization is successful (non trivial factor is found), else returns 0. Assumes \( n \) is normalized (shifted by normbits bits), and takes as input a precomputed inverse of \( n \) as computed by \( \text{n}_\text{preinvert}_\text{limb()} \). \( \text{ai} \) and \( \text{xi} \) should also be shifted left by \( \text{normbits} \).

\(\text{ai} \) is the constant of the polynomial used, \( \text{xi} \) is the initial value. \( \text{max}_\text{iters} \) is the number of iterations tried in process of finding the cycle.

The algorithm used is a modification of the original Pollard Rho algorithm, suggested by Richard Brent in the paper, available at https://maths-people.anu.edu.au/~brent/pd/rpb051i.pdf

\( \text{int} \ \text{n}_\text{factor}_\text{pollard}_\text{brent}(\text{mp}_\text{limb}_\text{t} \ * \text{factor}, \ \text{flint}_\text{rand}_\text{t} \ \text{state}, \ \text{mp}_\text{limb}_\text{t} \ \text{n}_\text{in}, \ \text{mp}_\text{limb}_\text{t} \ \text{max}_\text{tries}, \ \text{mp}_\text{limb}_\text{t} \ \text{max}_\text{iters}) \)
Pollard Rho algorithm, modified as suggested by Richard Brent. Makes a call to \texttt{n\_factor\_pollard\_brent\_single()}. The input parameters \(a_i\) and \(x_i\) for \texttt{n\_factor\_pollard\_brent\_single()} are selected at random.

If the algorithm fails to find a non trivial factor in one call, it tries again (this time with a different set of random values). This process is repeated a maximum of \(\max\_\_\text{tries}\) times.

Assumes \(n\) is not prime. \texttt{factor} is set as the factor found, if factorization is successful. In such a case, 1 is returned. Otherwise, 0 is returned. Factor discovered is not necessarily prime.

### 4.1.15 Arithmetic functions

**\texttt{int n\_moebius\_mu(ulong n)}**

Computes the Moebius function \(\mu(n)\), which is defined as \(\mu(n) = 0\) if \(n\) has a prime factor of multiplicity greater than 1, \(\mu(n) = -1\) if \(n\) has an odd number of distinct prime factors, and \(\mu(n) = 1\) if \(n\) has an even number of distinct prime factors. By convention, \(\mu(0) = 0\).

For even numbers, we use the identities \(\mu(4n) = 0\) and \(\mu(2n) = -\mu(n)\). Odd numbers up to a cutoff are then looked up from a precomputed table storing \(\mu(n) + 1\) in groups of two bits.

For larger \(n\), we first check if \(n\) is divisible by a small odd square and otherwise call \texttt{n\_factor()} and count the factors.

**\texttt{void n\_moebius\_mu\_vec(int *mu, ulong len)}**

Computes \(\mu(n)\) for \(n = 0, 1, \ldots, \text{len} - 1\). This is done by sieving over each prime in the range, flipping the sign of \(\mu(n)\) for every multiple of a prime \(p\) and setting \(\mu(n) = 0\) for every multiple of \(p^2\).

**\texttt{int n\_is\_squarefree(ulong n)}**

Returns 0 if \(n\) is divisible by some perfect square, and 1 otherwise. This simply amounts to testing whether \(\mu(n) \neq 0\). As special cases, 1 is considered squarefree and 0 is not considered squarefree.

**\texttt{ulong n\_euler\_phi(ulong n)}**

Computes the Euler totient function \(\phi(n)\), counting the number of positive integers less than or equal to \(n\) that are coprime to \(n\).

### 4.1.16 Factorials

**\texttt{ulong n\_factorial\_fast\_mod2\_preinv(ulong n, ulong p, ulong pinv)}**

Returns \(n! \mod p\) given a precomputed inverse of \(p\) as computed by \texttt{n\_preinvert\_limb()}. \(p\) is not required to be a prime, but no special optimisations are made for composite \(p\). Uses fast multipoint evaluation, running in about \(O(n^{1/2})\) time.

**\texttt{ulong n\_factorial\_mod2\_preinv(ulong n, ulong p, ulong pinv)}**

Returns \(n! \mod p\) given a precomputed inverse of \(p\) as computed by \texttt{n\_preinvert\_limb()}. \(p\) is not required to be a prime, but no special optimisations are made for composite \(p\).

Uses a lookup table for small \(n\), otherwise computes the product if \(n\) is not too large, and calls the fast algorithm for extremely large \(n\).
4.1.17 Primitive Roots and Discrete Logarithms

ulong n_primitive_root_prime_prefactor(ulong p, n_factor_t *factors)
Returns a primitive root for the multiplicative subgroup of \(\mathbb{Z}/p\mathbb{Z}\) where \(p\) is prime given the
factorisation (factors) of \(p - 1\).

ulong n_primitive_root_prime(ulong p)
Returns a primitive root for the multiplicative subgroup of \(\mathbb{Z}/p\mathbb{Z}\) where \(p\) is prime.

ulong n_discrete_log_bsgs(ulong b, ulong a, ulong n)
Returns the discrete logarithm of \(b\) with respect to \(a\) in the multiplicative subgroup of \(\mathbb{Z}/n\mathbb{Z}\) when
\(\mathbb{Z}/n\mathbb{Z}\) is cyclic. That is, it returns a number \(x\) such that \(a^x = b \mod n\). The multiplicative subgroup
is only cyclic when \(n\) is 2, 4, \(p^k\), or \(2p^k\) where \(p\) is an odd prime and \(k\) is a positive integer.

4.1.18 Elliptic curve method for factorization of mp_limb_t

void n_factor_ecm_double(mp_limb_t *x, mp_limb_t *z, mp_limb_t x0, mp_limb_t z0,
mp_limb_t n, n_ecm_t n_ecm_inf)
Sets the point \((x : z)\) to two times \((x_0 : z_0)\) modulo \(n\) according to the formula
\[ x = (x_0 + z_0)^2 \cdot (x_0 - z_0)^2 \mod n, \]
\[ z = 4x_0z_0 ((x_0 - z_0)^2 + 4a_24x_0z_0) \mod n. \]
This group doubling is valid only for points expressed in Montgomery projective coordinates.

void n_factor_ecm_add(mp_limb_t *x, mp_limb_t *z, mp_limb_t x1, mp_limb_t z1,
mp_limb_t x2, mp_limb_t z2, mp_limb_t x0, mp_limb_t z0, mp_limb_t n, n_ecm_t
n_ecm_inf)
Sets the point \((x : z)\) to the sum of \((x_1 : z_1)\) and \((x_2 : z_2)\) modulo \(n\), given the difference \((x_0 : z_0)\)
according to the formula
This group doubling is valid only for points expressed in Montgomery projective coordinates.

void n_factor_ecm_mul_montgomery_ladder(mp_limb_t *x, mp_limb_t *z, mp_limb_t x0,
mp_limb_t z0, mp_limb_t k, mp_limb_t n, n_ecm_t
n_ecm_inf)
Montgomery ladder algorithm for scalar multiplication of elliptic points.
Sets the point \((x : z)\) to \(k(x_0 : z_0)\) modulo \(n\).
Valid only for points expressed in Montgomery projective coordinates.

int n_factor_ecm_select_curve(mp_limb_t *f, mp_limb_t sigma, mp_limb_t n, n_ecm_t
n_ecm_inf)
Selects a random elliptic curve given a random integer sigma, according to Suyama’s parameterization.
If the factor is found while selecting the curve, 1 is returned. In case the curve found is
not suitable, 0 is returned.
Also selects the initial point \(x_0\), and the value of \((a + 2)/4\), where \(a\) is a curve parameter. Sets \(z_0\)
as 1 (shifted left by n_ecm_inf->normbits). All these are stored in the n_ecm_t struct.
The curve selected is of Montgomery form, the points selected satisfy the curve and are projective coordinates.

int n_factor_ecm_stage_I(mp_limb_t *f, const mp_limb_t *prime_array, mp_limb_t num,
mp_limb_t B1, mp_limb_t n, n_ecm_t n_ecm_inf)
Stage I implementation of the ECM algorithm.
\(f\) is set as the factor if found. \(num\) is number of prime numbers \(<\) the bound \(B1\). \prime_array\) is
an array of first \(B1\) primes. \(n\) is the number being factored.
If the factor is found, 1 is returned, otherwise 0.
int n_factor_ecm_stage_II(mp_limb_t *f, mp_limb_t B1, mp_limb_t B2, mp_limb_t P, mp_limb_t n, n_ecm_t n_ecm_inf)

Stage II implementation of the ECM algorithm.

f is set as the factor if found. B1, B2 are the two bounds. P is the primorial (approximately equal to $\sqrt{B_2}$). n is the number being factored.

If the factor is found, 1 is returned, otherwise 0.

int n_factor_ecm(mp_limb_t *f, mp_limb_t curves, mp_limb_t B1, mp_limb_t B2, flint_rand_t state, mp_limb_t n)

Outer wrapper function for the ECM algorithm. It factors n which must fit into a mp_limb_t.

The function calls stage I and II, and the precomputations (builds prime_array for stage I, GCD_table and prime_table for stage II).

f is set as the factor if found. curves is the number of random curves being tried. B1, B2 are the two bounds or stage I and stage II. n is the number being factored.

If a factor is found in stage I, 1 is returned. If a factor is found in stage II, 2 is returned. If a factor is found while selecting the curve, −1 is returned. Otherwise 0 is returned.

4.2 fmpz.h – integers

By default, an fmpz_t is implemented as an array of fmpz’s of length one to allow passing by reference as one can do with GMP’s mpz_t type. The fmpz_t type is simply a single limb, though the user does not need to be aware of this except in one specific case outlined below.

In all respects, fmpz_t’s act precisely like GMP’s mpz_t’s, with automatic memory management, however, in the first place only one limb is used to implement them. Once an fmpz_t overflows a limb then a multiprecision integer is automatically allocated and instead of storing the actual integer data the slong which implements the type becomes an index into a FLINT wide array of mpz_t’s.

These internal implementation details are not important for the user to understand, except for three important things.

Firstly, fmpz_t’s will be more efficient than mpz_t’s for single limb operations, or more precisely for signed quantities whose absolute value does not exceed FLINT_BITS - 2 bits.

Secondly, for small integers that fit into `FLINT_BITS - 2` bits much less memory will be used than for an mpz_t. When very many fmpz_t’s are used, there can be important cache benefits on account of this.

Thirdly, it is important to understand how to deal with arrays of fmpz_t’s. As for mpz_t’s, there is an underlying type, an fmpz, which can be used to create the array, e.g.

```c
fmpz myarr[100];
```

Now recall that an fmpz_t is an array of length one of fmpz’s. Thus, a pointer to an fmpz can be used in place of an fmpz_t. For example, to find the sign of the third integer in our array we would write

```c
int sign = fmpz_sgn(myarr + 2);
```

The fmpz module provides routines for memory management, basic manipulation and basic arithmetic. Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.
4.2.1 Simple example

The following example computes the square of the integer 7 and prints the result.

```c
#include "fmpz.h"

int main()
{
    fmpz_t x, y;
    fmpz_init(x);
    fmpz_init(y);
    fmpz_set_ui(x, 7);
    fmpz_mul(y, x, x);
    fmpz_print(x);
    flint_printf("^2 = ");
    fmpz_print(y); 
    flint_printf("
    ");
    fmpz_clear(x);
    fmpz_clear(y);
}
```

\[7^2 = 49\]

4.2.2 Types, macros and constants

**type fmpz**

The FLINT multi-precision integer type uses an inline representation for small integers, specifically when the absolute value is at most \(2^{62} - 1\) (on 64-bit machines) or \(2^{30} - 1\) (on 32-bit machines). It switches automatically to a GMP integer for larger values.

An fmpz is implemented as an slong. When its second most significant bit is 0 the fmpz represents an ordinary slong integer whose absolute value is at most FLINT_BITS - 2 bits.

When the second most significant bit is 1 then the value represents a pointer (the pointer is shifted right 2 bits and the second most significant bit is set to 1. This relies on the fact that malloc always allocates memory blocks on a 4 or 8 byte boundary).

**type fmpz_t**

An array of length 1 of fmpz’s. This is used to pass fmpz’s around by reference without fuss, similar to the way mpz_t works.

**COEFF_MAX**

The largest (positive) value an fmpz can be if just an slong.

**COEFF_MIN**

The smallest (negative) value an fmpz can be if just an slong.

**fmpz PTR_TO_COEFF(___mpz_struct *ptr)**

A macro to convert an mpz_t (or more generally any ___mpz_struct *) to an fmpz (shifts the pointer right by 2 and sets the second most significant bit).

**___mpz_struct *COEFF_TO_PTR(fmpz f)**

A macro to convert an fmpz which represents a pointer into an actual pointer to an ___mpz_struct (i.e. to an mpz_t).

**COEFF_IS_MPZ(f)**

A macro which returns 1 if f represents an mpz_t, otherwise 0 is returned.
__mpz_struct *__fmpz_new_mpz(void)
Initialises a new mpz_t and returns a pointer to it. This is only used internally.

void __fmpz_clear_mpz(fmpz f)
Clears the mpz_t “pointed to” by the fmpz f. This is only used internally.

void __fmpz_cleanup_mpz_content()
This function does nothing in the reentrant version of fmpz.

void __fmpz_cleanup()
This function does nothing in the reentrant version of fmpz.

__mpz_struct *__fmpz_promote(fmpz_t f)
If f doesn’t represent an mpz_t, initialise one and associate it to f.

__mpz_struct *__fmpz_promote_val(fmpz_t f)
If f doesn’t represent an mpz_t, initialise one and associate it to f, but preserve the value of f.
This function is for internal use. The resulting fmpz will be backed by an mpz_t that can be passed to GMP, but the fmpz will be in an inconsistent state with respect to the other Flint fmpz functions such as fmpz_is_zero, etc.

void __fmpz_demote(fmpz_t f)
If f represents an mpz_t clear it and make f just represent an slong.

void __fmpz_demote_val(fmpz_t f)
If f represents an mpz_t and its value will fit in an slong, preserve the value in f which we make to represent an slong, and clear the mpz_t.

int __fmpz_is_canonical(const fmpz_t f)
Returns 1 if the internal representation of f is correctly normalised and demoted; 0 otherwise.

4.2.3 Memory management

void fmpz_init(fmpz_t f)
A small fmpz_t is initialised, i.e. just a slong. The value is set to zero.

void fmpz_init2(fmpz_t f, ulong limbs)
Initialises the given fmpz_t to have space for the given number of limbs.
If limbs is zero then a small fmpz_t is allocated, i.e. just a slong. The value is also set to zero.
It is not necessary to call this function except to save time. A call to fmpz_init will do just fine.

void fmpz_clear(fmpz_t f)
Clears the given fmpz_t, releasing any memory associated with it, either back to the stack or the OS, depending on whether the reentrant or non-reentrant version of FLINT is built.

void fmpz_init_set(fmpz_t f, const fmpz_t g)

void fmpz_init_set_ui(fmpz_t f, ulong g)

void fmpz_init_set_si(fmpz_t f, slong g)
Initialises f and sets it to the value of g.
4.2.4 Random generation

For thread-safety, the randomisation methods take as one of their parameters an object of type `flint_rand_t`. Before calling any of the randomisation functions such an object first has to be initialised with a call to `flint_randinit()`. When one is finished generating random numbers, one should call `flint_randclear()` to clean up.

```c
void fmpz_randbits(fmpz_t f, flint_rand_t state, flint_bitcnt_t bits)
Generates a random signed integer whose absolute value has precisely the given number of bits.

void fmpz_randtest(fmpz_t f, flint_rand_t state, flint_bitcnt_t bits)
Generates a random signed integer whose absolute value has a number of bits which is random from 0 up to bits inclusive.

void fmpz_randtest_unsigned(fmpz_t f, flint_rand_t state, flint_bitcnt_t bits)
Generates a random unsigned integer whose value has a number of bits which is random from 0 up to bits inclusive.

void fmpz_randtest_not_zero(fmpz_t f, flint_rand_t state, flint_bitcnt_t bits)
As per fmpz_randtest, but the result will not be 0. If bits is set to 0, an exception will result.

void fmpz_randm(fmpz_t f, flint_rand_t state, const fmpz_t m)
Generates a random integer in the range 0 to m – 1 inclusive.

void fmpz_randtest_mod(fmpz_t f, flint_rand_t state, const fmpz_t m)
Generates a random integer in the range 0 to m – 1 inclusive, with an increased probability of generating values close to the endpoints.

void fmpz_randtest_mod_signed(fmpz_t f, flint_rand_t state, const fmpz_t m)
Generates a random integer in the range (–m/2, m/2], with an increased probability of generating values close to the endpoints or close to zero.

void fmpz_randprime(fmpz_t f, flint_rand_t state, flint_bitcnt_t bits, int proved)
Generates a random prime number with the given number of bits.

The generation is performed by choosing a random number and then finding the next largest prime, and therefore does not quite give a uniform distribution over the set of primes with that many bits.

Random number generation is performed using the standard Flint random number generator, which is not suitable for cryptographic use.

If proved is nonzero, then the integer returned is guaranteed to actually be prime.

4.2.5 Conversion

```c
slong fmpz_get_si(const fmpz_t f)
Returns f as a slong. The result is undefined if f does not fit into a slong.

ulong fmpz_get_ui(const fmpz_t f)
Returns f as an ulong. The result is undefined if f does not fit into an ulong or is negative.

void fmpz_get_uiui(mp_limb_t *hi, mp_limb_t *low, const fmpz_t f)
If f consists of two limbs, then *hi and *low are set to the high and low limbs, otherwise *low is set to the low limb and *hi is set to 0.

mp_limb_t fmpz_get_nmod(const fmpz_t f, nmod_t mod)
Returns f mod n.

double fmpz_get_d(const fmpz_t f)
Returns f as a double, rounding down towards zero if f cannot be represented exactly. The outcome is undefined if f is too large to fit in the normal range of a double.
```
void \texttt{fmpz\_set\_mpf} (\texttt{fmpz\_t} \texttt{f}, \texttt{const mpf\_t} \texttt{x})
\begin{itemize}
  \item Sets \texttt{f} to the \texttt{mpf\_t} \texttt{x}, rounding down towards zero if the value of \texttt{x} is fractional.
\end{itemize}

void \texttt{fmpz\_get\_mpf} (\texttt{mpf\_t} \texttt{x}, \texttt{const fmpz\_t} \texttt{f})
\begin{itemize}
  \item Sets the value of the \texttt{mpf\_t} \texttt{x} to the value of \texttt{f}.
\end{itemize}

void \texttt{fmpz\_get\_mpfr} (\texttt{mpfr\_t} \texttt{x}, \texttt{const fmpz\_t} \texttt{f}, \texttt{mpfr\_rnd\_t} \texttt{rnd})
\begin{itemize}
  \item Sets the value of \texttt{x} from \texttt{f}, rounded toward the given direction \texttt{rnd}.
\end{itemize}
\textbf{Note:} Requires that \texttt{mpfr.h} has been included before any FLINT header is included.

\begin{verbatim}
double fmpz\_get\_d\_2exp (slong *exp, \texttt{const fmpz\_t} \texttt{f})
\begin{itemize}
  \item Returns \texttt{f} as a normalized \texttt{double} along with a $2$-exponent \texttt{exp}, i.e. if \texttt{r} is the return value then \texttt{f} = $r2^{\texttt{exp}}$, to within 1 ULP.
\end{itemize}
\end{verbatim}

void \texttt{fmpz\_get\_mpz} (\texttt{mpz\_t} \texttt{x}, \texttt{const fmpz\_t} \texttt{f})
\begin{itemize}
  \item Sets the \texttt{mpz\_t} \texttt{x} to the same value as \texttt{f}.
\end{itemize}

\begin{verbatim}
int fmpz\_get\_mpn (mp\_ptr *n, \texttt{fmpz\_t} \texttt{n\_in})
\begin{itemize}
  \item Sets the \texttt{mp\_ptr} \texttt{n} to the same value as \texttt{n\_in}. Returned integer is number of limbs allocated to \texttt{n}, minimum number of limbs required to hold the value stored in \texttt{n\_in}.
\end{itemize}
\end{verbatim}

\begin{verbatim}
char *fmpz\_get\_str (char *str, int \texttt{b}, \texttt{const fmpz\_t} \texttt{f})
\begin{itemize}
  \item Returns the representation of \texttt{f} in base \texttt{b}, which can vary between 2 and 62, inclusive.
  \item If \texttt{str} is \texttt{NULL}, the result string is allocated by the function. Otherwise, it is up to the caller to ensure that the allocated block of memory is sufficiently large.
\end{itemize}
\end{verbatim}

void \texttt{fmpz\_set\_si} (\texttt{fmpz\_t} \texttt{f}, \texttt{slong} \texttt{val})
\begin{itemize}
  \item Sets \texttt{f} to the given \texttt{slong} value.
\end{itemize}

void \texttt{fmpz\_set\_ui} (\texttt{fmpz\_t} \texttt{f}, \texttt{ulong} \texttt{val})
\begin{itemize}
  \item Sets \texttt{f} to the given \texttt{ulong} value.
\end{itemize}

void \texttt{fmpz\_set\_d} (\texttt{fmpz\_t} \texttt{f}, \texttt{double} \texttt{c})
\begin{itemize}
  \item Sets \texttt{f} to the \texttt{double} \texttt{c}, rounding down towards zero if the value of \texttt{c} is fractional. The outcome is undefined if \texttt{c} is infinite, not-a-number, or subnormal.
\end{itemize}

void \texttt{fmpz\_set\_d\_2exp} (\texttt{fmpz\_t} \texttt{f}, \texttt{double} \texttt{d}, \texttt{ulong} \texttt{exp})
\begin{itemize}
  \item Sets \texttt{f} to the nearest integer to $d2^{\texttt{exp}}$.
\end{itemize}

void \texttt{fmpz\_neg\_ui} (\texttt{fmpz\_t} \texttt{f}, \texttt{ulong} \texttt{val})
\begin{itemize}
  \item Sets \texttt{f} to the given \texttt{ulong} value, and then negates \texttt{f}.
\end{itemize}

void \texttt{fmpz\_set\_uiui} (\texttt{fmpz\_t} \texttt{f}, \texttt{mp\_limb\_t} \texttt{hi}, \texttt{mp\_limb\_t} \texttt{lo})
\begin{itemize}
  \item Sets \texttt{f} to \texttt{lo}, plus \texttt{hi} shifted to the left by FLINT\_BITS.
\end{itemize}

void \texttt{fmpz\_neg\_uiui} (\texttt{fmpz\_t} \texttt{f}, \texttt{mp\_limb\_t} \texttt{hi}, \texttt{mp\_limb\_t} \texttt{lo})
\begin{itemize}
  \item Sets \texttt{f} to \texttt{lo}, plus \texttt{hi} shifted to the left by FLINT\_BITS, and then negates \texttt{f}.
\end{itemize}

void \texttt{fmpz\_set\_signed\_uiui} (\texttt{fmpz\_t} \texttt{f}, \texttt{ulong} \texttt{hi}, \texttt{ulong} \texttt{lo})
\begin{itemize}
  \item Sets \texttt{f} to \texttt{lo}, plus \texttt{hi} shifted to the left by FLINT\_BITS, interpreted as a signed two’s complement integer with 2 \* FLINT\_BITS bits.
\end{itemize}

void \texttt{fmpz\_set\_signed\_uiuiui} (\texttt{fmpz\_t} \texttt{f}, \texttt{ulong} \texttt{hi}, \texttt{ulong} \texttt{mid}, \texttt{ulong} \texttt{lo})
\begin{itemize}
  \item Sets \texttt{f} to \texttt{lo}, plus \texttt{mid} shifted to the left by FLINT\_BITS, plus \texttt{hi} shifted to the left by 2\*FLINT\_BITS bits, interpreted as a signed two’s complement integer with 3 \* FLINT\_BITS bits.
\end{itemize}

void \texttt{fmpz\_set\_ui\_array} (\texttt{fmpz\_t} \texttt{out}, \texttt{const ulong\_t} \texttt{in}, \texttt{ulong\_t} \texttt{n})
\begin{itemize}
  \item Sets \texttt{out} to the nonnegative integer $\texttt{in}[0] + \texttt{in}[1]\texttt{X} + \ldots + \texttt{in}[\texttt{n} - 1]\texttt{X}^{(\texttt{n} - 1)}$ where $\texttt{X} = 2^{\texttt{FLINT\_BITS}}$. It is assumed that $\texttt{n} > 0$.
\end{itemize}
void \texttt{fmpz\_set\_signed\_ui\_array}(\texttt{fmpz\_t} \texttt{out}, \texttt{const ulong *in}, \texttt{slong n})

Sets \texttt{out} to the integer represented in \texttt{in[0]}, ..., \texttt{in[n - 1]} as a signed two’s complement integer with \texttt{n * FLINT\_BITS} bits. It is assumed that \texttt{n} > 0. The function operates as a call to \texttt{fmpz\_set\_ui\_array()} followed by a symmetric remainder modulo \(2^n \cdot \text{FLINT\_BITS}\).

void \texttt{fmpz\_get\_ui\_array}(\texttt{ulong *out}, \texttt{slong n}, \texttt{const fmpz\_t in})

Assuming that the nonnegative integer \texttt{in} can be represented in the form \texttt{out[0] + out[1]\cdot X + ... + out[n - 1]\cdot X^{(n - 1)}} , where \(X = 2^{\text{FLINT\_BITS}}\), sets the corresponding elements of \texttt{out} so that this is true. It is assumed that \texttt{n} > 0.

void \texttt{fmpz\_get\_signed\_ui\_array}(\texttt{ulong *out}, \texttt{slong n}, \texttt{const fmpz\_t in})

Retrieves the value of \texttt{in} modulo \(2^n \cdot \text{FLINT\_BITS}\) and puts the \texttt{n} words of the result in \texttt{out[0]}, ..., \texttt{out[n-1]}. This will give a signed two’s complement representation of \texttt{in} (assuming \texttt{in} doesn’t overflow the array).

void \texttt{fmpz\_get\_signed\_ui\_ui}(\texttt{ulong *hi}, \texttt{ulong *lo}, \texttt{const fmpz\_t in})

Retrieves the value of \texttt{in} modulo \(2^{2\cdot \text{FLINT\_BITS}}\) and puts the high and low words into \texttt{*hi} and \texttt{*lo} respectively.

void \texttt{fmpz\_set\_mpz}(\texttt{fmpz\_t f}, \texttt{const mpz\_t x})

Sets \texttt{f} to the given \texttt{mpz\_t} value.

int \texttt{fmpz\_set\_str}(\texttt{fmpz\_t f}, \texttt{const char *str}, \texttt{int b})

Sets \texttt{f} to the value given in the null-terminated string \texttt{str}, in base \texttt{b}. The base \texttt{b} can vary between 2 and 62, inclusive. Returns 0 if the string contains a valid input and -1 otherwise.

void \texttt{fmpz\_set\_ui\_smod}(\texttt{fmpz\_t f}, \texttt{mp\_limb\_t x}, \texttt{mp\_limb\_t m})

Sets \texttt{f} to the signed remainder \(y \equiv x \mod m\) satisfying \(-m/2 < y \leq m/2\), given \(x\) which is assumed to satisfy \(0 \leq x < m\).

void \texttt{flint\_mpz\_init\_set\_readonly}(\texttt{mpz\_t z}, \texttt{const fmpz\_t f})

Sets the uninitialised \texttt{mpz\_t} \texttt{z} to the value of the readonly \texttt{fmpz\_t} \texttt{f}.

Note that it is assumed that \texttt{f} does not change during the lifetime of \texttt{z}.

The integer \texttt{z} has to be cleared by a call to \texttt{flint\_mpz\_clear\_readonly()}. The suggested use of the two functions is as follows:

```c
fmpz_t f;
...
{
  mpz_t z;
  flint_mpz_init_set_readonly(z, f);
  foo(..., z);
  flint_mpz_clear_readonly(z);
}
```

This provides a convenient function for user code, only requiring to work with the types \texttt{fmpz\_t} and \texttt{mpz\_t}.

In critical code, the following approach may be favourable:

```c
fmpz_t f;
...
{
  __mpz_struct *z;
  z = _fmpz_promote_val(f);
  foo(..., z);
}
```

(continues on next page)
void **flint_mpz_clear_readonly(mpz_t z)
    Clears thereadonly mpz_t z.

void **fmpz_init_set_readonly(fmpz_t f, const mpz_t z)
    Sets theuninitialised fmpz_t f to areadonly version of the integer z.
    Notethat the value of z is assumed to remain constant throughout the lifetime of f.
    The fmpz_t f has to be cleared by calling the function fmpz_clear_readonly().
    The suggested use of the two functions is as follows:

```c
mpz_t z;
...
{
    fmpz_t f;
    fmpz_init_set_readonly(f, z);
    foo(..., f);
    fmpz_clear_readonly(f);
}
```

void **fmpz_clear_readonly(fmpz_t f)
    Clears the readonly fmpz_t f.

### 4.2.6 Input and output

**int fmpz_read(fmpz_t f)**
    Readsamultiprecision integer from stdin. The format is an optional minus sign, followed by one
    or more digits. The first digit should be non-zero unless it is the only digit.
    In case of success, returnsapositive number. In case of failure, returns a non-positive number.
    This convention is adopted in light of the return values of scanf from the standard library and
    mpz_inp_str from GMP.

**int fmpz_fread(FILE *file, fmpz_t f)**
    Readsamultiprecision integer from the stream file. The format is an optional minus sign, followed
    by one or more digits. The first digit should be non-zero unless it is the only digit.
    In case of success, returnsapositive number. In case of failure, returns a non-positive number.
    This convention is adopted in light of the return values of scanf from the standard library and
    mpz_inp_str from GMP.

**size_t fmpz_inp_raw(fmpz_t x, FILE *fin)**
    Readsamultiprecision integer from the stream file. The format is raw binary format write by
    fmpz_out_raw().
    In case of success, returnapositive number, indicating number of bytes read. In case of failure 0.
    This function calls the mpz_inp_raw function in library gmp. So that it can read the raw data
    write by mpz_inp_raw directly.

**int fmpz_fprint(FILE *fs, const fmpz_t x)**
int fmpz_print(const fmpz_t x)
    Prints the value \( x \) to \( \text{fs} \) or \( \text{stdout} \), without a carriage return. The value is printed as either 0, the decimal digits of a positive integer, or a minus sign followed by the digits of a negative integer.
    Returns the number of characters written to file stream.

size_t fmpz_out_raw(FILE *fout, const fmpz_t x)
    Writes the value \( x \) to file. The value is written in raw binary format. The integer is written in portable format, with 4 bytes of size information, and that many bytes of limbs. Both the size and the limbs are written in decreasing significance order (i.e., in big-endian).
    The output can be read with fmpz_inp_raw.
    In case of success, return a positive number, indicating number of bytes written. In case of failure, return 0.
    The output of this can also be read by mpz_inp_raw from GMP, since this function calls the mpz_inp_raw function in library gmp.

4.2.7 Basic properties and manipulation

size_t fmpz_sizeinbase(const fmpz_t f, int b)
    Returns the size of the absolute value of \( f \) in base \( b \), measured in numbers of digits. The base \( b \) can be between 2 and 62, inclusive.

flint_bitcnt_t fmpz_bits(const fmpz_t f)
    Returns the number of bits required to store the absolute value of \( f \). If \( f \) is 0 then 0 is returned.

mp_size_t fmpz_size(const fmpz_t f)
    Returns the number of limbs required to store the absolute value of \( f \). If \( f \) is zero then 0 is returned.

int fmpz_sgn(const fmpz_t f)
    Returns \(-1\) if the sign of \( f \) is negative, \(+1\) if it is positive, otherwise returns 0.

flint_bitcnt_t fmpz_val2(const fmpz_t f)
    Returns the exponent of the largest power of two dividing \( f \), or equivalently the number of trailing zeros in the binary expansion of \( f \). If \( f \) is zero then 0 is returned.

void fmpz_swap(fmpz_t f, fmpz_t g)
    Efficiently swaps \( f \) and \( g \). No data is copied.

void fmpz_set(fmpz_t f, const fmpz_t g)
    Sets \( f \) to the same value as \( g \).

void fmpz_zero(fmpz_t f)
    Sets \( f \) to zero.

void fmpz_one(fmpz_t f)
    Sets \( f \) to one.

int fmpz_abs_fits_ui(const fmpz_t f)
    Returns whether the absolute value of \( f \) fits into an ulong.

int fmpz_fits_si(const fmpz_t f)
    Returns whether the value of \( f \) fits into a slong.

void fmpz_setbit(fmpz_t f, ulong i)
    Sets bit index \( i \) of \( f \).

int fmpz_tstbit(const fmpz_t f, ulong i)
    Test bit index \( i \) of \( f \) and return 0 or 1, accordingly.
mp_limb_t fmpz_abs_lbound_ui_2exp (slong *exp, const fmpz_t x, int bits)
   For nonzero \(x\), returns a mantissa \(m\) with exactly \(bits\) bits and sets \(exp\) to an exponent \(e\), such that \(|x| \geq m2^e\). The number of bits must be between 1 and FLINT_BITS inclusive. The mantissa is guaranteed to be correctly rounded.

mp_limb_t fmpz_abs_ubound_ui_2exp (slong *exp, const fmpz_t x, int bits)
   For nonzero \(x\), returns a mantissa \(m\) with exactly \(bits\) bits and sets \(exp\) to an exponent \(e\), such that \(|x| \leq m2^e\). The number of bits must be between 1 and FLINT_BITS inclusive. The mantissa is either correctly rounded or one unit too large (possibly meaning that the exponent is one too large, if the mantissa is a power of two).

4.2.8 Comparison

int fmpz_cmp (const fmpz_t f, const fmpz_t g)
int fmpz_cmp_ui (const fmpz_t f, ulong g)
int fmpz_cmp_si (const fmpz_t f, slong g)
   Returns a negative value if \(f < g\), positive value if \(g < f\), otherwise returns 0.
int fmpz_cmpabs (const fmpz_t f, const fmpz_t g)
   Returns a negative value if \(|f| < |g|\), positive value if \(|g| < |f|\), otherwise returns 0.
int fmpz_cmp2abs (const fmpz_t f, const fmpz_t g)
   Returns a negative value if \(|f| < 2|g|\), positive value if \(2|g| < |f|\), otherwise returns 0.
int fmpz_equal (const fmpz_t f, const fmpz_t g)
int fmpz_equal_ui (const fmpz_t f, ulong g)
int fmpz_equal_si (const fmpz_t f, slong g)
   Returns 1 if \(f\) is equal to \(g\), otherwise returns 0.
int fmpz_is_zero (const fmpz_t f)
   Returns 1 if \(f\) is 0, otherwise returns 0.
int fmpz_is_one (const fmpz_t f)
   Returns 1 if \(f\) is equal to one, otherwise returns 0.
int fmpz_is_pm1 (const fmpz_t f)
   Returns 1 if \(f\) is equal to one or minus one, otherwise returns 0.
int fmpz_is_even (const fmpz_t f)
   Returns whether the integer \(f\) is even.
int fmpz_is_odd (const fmpz_t f)
   Returns whether the integer \(f\) is odd.

4.2.9 Basic arithmetic

void fmpz_neg (fmpz_t fl, const fmpz_t f2)
   Sets \(f_1\) to \(-f_2\).
void fmpz_abs (fmpz_t fl, const fmpz_t f2)
   Sets \(f_1\) to the absolute value of \(f_2\).
void fmpz_add (fmpz_t fl, const fmpz_t f, const fmpz_t h)
void fmpz_add_ui (fmpz_t fl, const fmpz_t g, ulong h)
void fmpz_add_si(fmpz_t f, const fmpz_t g, slong h)
    Sets \( f \) to \( g + h \).

void fmpz_sub(fmpz_t f, const fmpz_t g, const fmpz_t h)
void fmpz_sub_ui(fmpz_t f, const fmpz_t g, ulong h)
void fmpz_sub_si(fmpz_t f, const fmpz_t g, slong h)
    Sets \( f \) to \( g - h \).

void fmpz_mul(fmpz_t f, const fmpz_t g, const fmpz_t h)
void fmpz_mul_ui(fmpz_t f, const fmpz_t g, ulong h)
void fmpz_mul_si(fmpz_t f, const fmpz_t g, slong h)
    Sets \( f \) to \( g \times h \).

void fmpz_mul2_uiui(fmpz_t f, const fmpz_t g, ulong x, ulong y)
    Sets \( f \) to \( g \times x \times y \) where \( x \) and \( y \) are of type \text{ulong}.

void fmpz_mul_2exp(fmpz_t f, const fmpz_t g, ulong e)
    Sets \( f \) to \( g \times 2^e \).
    Note: Assumes that \( e + \text{FLINT\_BITS} \) does not overflow.

void fmpz_one_2exp(fmpz_t f, ulong e)
    Sets \( f \) to \( 2^e \).

void fmpz_addmul(fmpz_t f, const fmpz_t g, const fmpz_t h)
void fmpz_addmul_ui(fmpz_t f, const fmpz_t g, ulong h)
void fmpz_addmul_si(fmpz_t f, const fmpz_t g, slong h)
    Sets \( f \) to \( f + g \times h \).

void fmpz_submul(fmpz_t f, const fmpz_t g, const fmpz_t h)
void fmpz_submul_ui(fmpz_t f, const fmpz_t g, ulong h)
void fmpz_submul_si(fmpz_t f, const fmpz_t g, slong h)
    Sets \( f \) to \( f - g \times h \).

void fmpz_fmma(fmpz_t f, const fmpz_t a, const fmpz_t b, const fmpz_t c, const fmpz_t d)
    Sets \( f \) to \( a \times b + c \times d \).

void fmpz_fmmas(fmpz_t f, const fmpz_t a, const fmpz_t b, const fmpz_t c, const fmpz_t d)
    Sets \( f \) to \( a \times b - c \times d \).

void fmpz_cdiv_qr(fmpz_t f, fmpz_t s, const fmpz_t g, const fmpz_t h)
void fmpz_fdiv_qr(fmpz_t f, fmpz_t s, const fmpz_t g, const fmpz_t h)
void fmpz_tdiv_qr(fmpz_t f, fmpz_t s, const fmpz_t g, const fmpz_t h)
void fmpz_ndiv_qr(fmpz_t f, fmpz_t s, const fmpz_t g, const fmpz_t h)
void fmpz_cdiv_q(fmpz_t f, const fmpz_t g, const fmpz_t h)
void fmpz_fdiv_q(fmpz_t f, const fmpz_t g, const fmpz_t h)
void fmpz_tdiv_q(fmpz_t f, const fmpz_t g, const fmpz_t h)
void fmpz_cdiv_q_si(fmpz_t f, const fmpz_t g, slong h)
void fmpz_fdiv_q_si(fmpz_t f, const fmpz_t g, slong h)
void fmpz_tdiv_q_si(fmpz_t f, const fmpz_t g, slong h)
void fmpz_cdiv_q_ui(fmpz_t f, const fmpz_t g, ulong h)
void fmpz_fdiv_q_ui(fmpz_t f, const fmpz_t g, ulong h)
void fmpz_tdiv_q_ui(fmpz_t f, const fmpz_t g, ulong h)
void fmpz_cdiv_q_2exp(fmpz_t f, const fmpz_t g, ulong exp)
void fmpz_fdiv_q_2exp(fmpz_t f, const fmpz_t g, ulong exp)
void fmpz_tdiv_q_2exp(fmpz_t f, const fmpz_t g, ulong exp)
void fmpz_cdiv_r_2exp(fmpz_t s, const fmpz_t g, ulong exp)
void fmpz_fdiv_r_2exp(fmpz_t s, const fmpz_t g, ulong exp)
void fmpz_tdiv_r_2exp(fmpz_t s, const fmpz_t g, ulong exp)

Sets \( f \) to the quotient of \( g \) by \( h \) and/or \( s \) to the remainder. For the 2exp functions, \( g = 2^{\text{exp}} \).

If \( h \) is 0 an exception is raised.

Rounding is made in the following way:

- \text{fdiv} rounds the quotient via floor rounding.
- \text{cdiv} rounds the quotient via ceil rounding.
- \text{tdiv} rounds the quotient via truncation, i.e. rounding towards zero.
- \text{ndiv} rounds the quotient such that the remainder has the smallest absolute value. In case of ties, it rounds the quotient towards zero.

ulong fmpz_cdiv_ui(const fmpz_t g, ulong h)
ulong fmpz_fdiv_ui(const fmpz_t g, ulong h)
ulong fmpz_tdiv_ui(const fmpz_t g, ulong h)

Returns the absolute value remainder of \( g \) divided by \( h \), following the convention of rounding as seen above. If \( h \) is zero an exception is raised.

void fmpz_divexact(fmpz_t f, const fmpz_t g, const fmpz_t h)
void fmpz_divexact_si(fmpz_t f, const fmpz_t g, slong h)
void fmpz_divexact_ui(fmpz_t f, const fmpz_t g, ulong h)

Sets \( f \) to the quotient of \( g \) and \( h \), assuming that the division is exact, i.e. \( g \) is a multiple of \( h \). If \( h \) is 0 an exception is raised.

void fmpz_divexact2_uiui(fmpz_t f, const fmpz_t g, ulong x, ulong y)

Sets \( f \) to the quotient of \( g \) and \( h = x \times y \), assuming that the division is exact, i.e. \( g \) is a multiple of \( h \). If \( x \) or \( y \) is 0 an exception is raised.

int fmpz_divisible(const fmpz_t f, const fmpz_t g)
int fmpz_divisible_si(const fmpz_t f, slong g)

Returns 1 if there is an integer \( q \) with \( f = q g \) and 0 if there is none.

int fmpz_divides(fmpz_t q, const fmpz_t g, const fmpz_t h)

Returns 1 if there is an integer \( q \) with \( f = q g \) and sets \( q \) to the quotient. Otherwise returns 0 and sets \( q \) to 0.

void fmpz_mod(fmpz_t f, const fmpz_t g, const fmpz_t h)

Sets \( f \) to the remainder of \( g \) divided by \( h \) such that the remainder is positive. Assumes that \( h \) is not zero.

4.2. fmpz.h – integers
ulong fmpz_mod_ui(fmpz_t f, const fmpz_t g, ulong h)
Sets f to the remainder of g divided by h such that the remainder is positive and also returns this value. Raises an exception if h is zero.

void fmpz_smod(fmpz_t f, const fmpz_t g, const fmpz_t h)
Sets f to the signed remainder y ≡ g mod h satisfying −|h|/2 < y ≤ |h|/2.

void fmpz_preinvn_init(fmpz_preinvn_t inv, const fmpz_t f)
Compute a precomputed inverse inv of f for use in the preinvn functions listed below.

void fmpz_preinvn_clear(fmpz_preinvn_t inv)
Clean up the resources used by a precomputed inverse created with the fmpz_preinvn_init() function.

void fmpz_fdiv_qr_preinvn(fmpz_t f, fmpz_t s, const fmpz_t g, const fmpz_t h, const fmpz_preinvn_t hinv)
As per fmpz_fdiv_qr(), but takes a precomputed inverse hinv of h constructed using fmpz_preinvn().
This function will be faster than fmpz_fdiv_qr_preinvn() when the number of limbs of h is at least PREINVN_CUTOFF.

void fmpz_pow_ui(fmpz_t f, const fmpz_t g, ulong x)
void fmpz_ui_pow_ui(fmpz_t f, ulong g, ulong x)
Sets f to g^x. Defines 0^0 = 1.

int fmpz_pow_fmpz(fmpz_t f, const fmpz_t g, const fmpz_t x)
Sets f to g^x. Defines 0^0 = 1. Return 1 for success and 0 for failure. The function throws only if x is negative.

void fmpz_powm_ui(fmpz_t f, const fmpz_t g, const fmpz_t e, const fmpz_t m)
Sets f to g^e mod m. If e = 0, sets f to 1.
Assumes that m ≠ 0, raises an abort signal otherwise.

slong fmpz_clog(const fmpz_t x, const fmpz_t b)
slong fmpz_clog_ui(const fmpz_t x, ulong b)
Returns \lceil \log_b x \rceil.
Assumes that x ≥ 1 and b ≥ 2 and that the return value fits into a signed slong.

slong fmpz_flog(const fmpz_t x, const fmpz_t b)
slong fmpz_flog_ui(const fmpz_t x, ulong b)
Returns \lfloor \log_b x \rfloor.
Assumes that x ≥ 1 and b ≥ 2 and that the return value fits into a signed slong.

double fmpz_dlog(const fmpz_t x)
Returns a double precision approximation of the natural logarithm of x.
The accuracy depends on the implementation of the floating-point logarithm provided by the C standard library. The result can typically be expected to have a relative error no greater than 1-2 bits.

int fmpz_sqrtmod(fmpz_t b, const fmpz_t a, const fmpz_t p)
If p is prime, set b to a square root of a modulo p if a is a quadratic residue modulo p and return 1, otherwise return 0.
If p is not prime the return value is with high probability 0, indicating that p is not prime, or a is not a square modulo p. If p is not prime and the return value is 1, the value of b is meaningless.
void \texttt{fmpz\_sqrt}(\texttt{fmpz\_t} f, const \texttt{fmpz\_t} g)

Sets $f$ to the integer part of the square root of $g$, where $g$ is assumed to be non-negative. If $g$ is negative, an exception is raised.

void \texttt{fmpz\_sqrrem}(\texttt{fmpz\_t} f, \texttt{fmpz\_t} r, const \texttt{fmpz\_t} g)

Sets $f$ to the integer part of the square root of $g$, where $g$ is assumed to be non-negative, and sets $r$ to the remainder, that is, the difference $g - f^2$. If $g$ is negative, an exception is raised. The behaviour is undefined if $f$ and $r$ are aliases.

int \texttt{fmpz\_is\_square}(const \texttt{fmpz\_t} f)

Returns nonzero if $f$ is a perfect square and zero otherwise.

int \texttt{fmpz\_root}(\texttt{fmpz\_t} r, const \texttt{fmpz\_t} f, \texttt{slong} n)

Set $r$ to the integer part of the $n$-th root of $f$. Requires that $n > 0$ and that if $n$ is even then $f$ be non-negative, otherwise an exception is raised. The function returns 1 if the root was exact, otherwise 0.

int \texttt{fmpz\_is\_perfect\_power}(\texttt{fmpz\_t} root, const \texttt{fmpz\_t} f)

If $f$ is a perfect power $r^k$ set root to $r$, and return $k$, otherwise return 0. Note that $-1, 0, 1$ are all considered perfect powers. No guarantee is made about $r$ or $k$ being the smallest possible value. Negative values of $f$ are permitted.

void \texttt{fmpz\_fac\_ui}(\texttt{fmpz\_t} f, \texttt{ulong} n)

Sets $f$ to the factorial $n!$ where $n$ is an \texttt{ulong}.

void \texttt{fmpz\_fib\_ui}(\texttt{fmpz\_t} f, \texttt{ulong} n)

Sets $f$ to the Fibonacci number $F_n$ where $n$ is an \texttt{ulong}.

void \texttt{fmpz\_bin\_uiui}(\texttt{fmpz\_t} f, \texttt{ulong} n, \texttt{ulong} k)

Sets $f$ to the binomial coefficient \binom{n}{k}.

void \texttt{fmpz\_rfac\_ui}(\texttt{fmpz\_t} r, const \texttt{fmpz\_t} x, \texttt{ulong} a, \texttt{ulong} b)

Sets $r$ to the rising factorial $(x+a)(x+a+1)(x+a+2)\cdots(x+b-1)$. Assumes $b > a$.

void \texttt{fmpz\_rfac\_uiui}(\texttt{fmpz\_t} r, \texttt{ulong} x, \texttt{ulong} k)

Sets $r$ to the rising factorial $x(x+1)(x+2)\cdots(x+k-1)$.

void \texttt{fmpz\_mul\_tdiv\_q\_2exp}(\texttt{fmpz\_t} f, const \texttt{fmpz\_t} g, \texttt{const fmpz\_t} h, \texttt{ulong} exp)

Sets $f$ to the product of $g$ and $h$ divided by $2^\text{exp}$, rounding down towards zero.

void \texttt{fmpz\_mul\_si\_tdiv\_q\_2exp}(\texttt{fmpz\_t} f, const \texttt{fmpz\_t} g, \texttt{slong} x, \texttt{ulong} exp)

Sets $f$ to the product of $g$ and $x$ divided by $2^\text{exp}$, rounding down towards zero.

4.2.10 Greatest common divisor

void \texttt{fmpz\_gcd\_ui}(\texttt{fmpz\_t} f, const \texttt{fmpz\_t} g, \texttt{ulong} h)

void \texttt{fmpz\_gcd}(\texttt{fmpz\_t} f, const \texttt{fmpz\_t} g, \texttt{const fmpz\_t} h)

Sets $f$ to the greatest common divisor of $g$ and $h$. The result is always positive, even if one of $g$ and $h$ is negative.

void \texttt{fmpz\_gcd3}(\texttt{fmpz\_t} f, const \texttt{fmpz\_t} a, \texttt{const fmpz\_t} b, \texttt{const fmpz\_t} c)

Sets $f$ to the greatest common divisor of $a$, $b$, and $c$. This is equivalent to calling \texttt{fmpz\_gcd} twice, but may be faster.
void \texttt{fmpz\_lcm}(\texttt{fmpz\_t f}, \texttt{const fmpz\_t g}, \texttt{const fmpz\_t h})

Sets $f$ to the least common multiple of $g$ and $h$. The result is always nonnegative, even if one of $g$ and $h$ is negative.

void \texttt{fmpz\_gcdinv}(\texttt{fmpz\_t d}, \texttt{fmpz\_t a}, \texttt{const fmpz\_t f}, \texttt{const fmpz\_t g})

Given integers $f,g$ with $0 \leq f < g$, computes the greatest common divisor $d = \gcd(f,g)$ and the modular inverse $a = f^{-1} \pmod{g}$, whenever $f \neq 0$.

Assumes that $d$ and $a$ are not aliased.

void \texttt{fmpz\_xgcd}(\texttt{fmpz\_t d}, \texttt{fmpz\_t a}, \texttt{fmpz\_t b}, \texttt{const fmpz\_t f}, \texttt{const fmpz\_t g})

Computes the extended GCD of $f$ and $g$, i.e. the values $a$ and $b$ such that $af + bg = d$, where $d = \gcd(f,g)$. Here $a$ will be the same as calling \texttt{fmpz\_gcdinv} when $f < g$ (or vice versa for $b$ when $g < f$).

To obtain the canonical solution to Bézout’s identity, call \texttt{fmpz\_xgcd\_canonical\_bezout} instead. This is also faster.

Assumes that there is no aliasing among the outputs.

void \texttt{fmpz\_xgcd\_canonical\_bezout}(\texttt{fmpz\_t d}, \texttt{fmpz\_t a}, \texttt{fmpz\_t b}, \texttt{const fmpz\_t f}, \texttt{const fmpz\_t g})

Computes the extended GCD $\text{xgcd}(f,g) = (d, a, b)$ such that the solution is the canonical solution to Bézout’s identity. We define the canonical solution to satisfy one of the following if one of the given conditions apply:

\begin{align*}
\text{xgcd}(\pm g, g) &= (|g|, 0, \text{sgn}(g)) \\
\text{xgcd}(f, 0) &= (|f|, \text{sgn}(f), 0) \\
\text{xgcd}(0, g) &= (|g|, 0, \text{sgn}(g)) \\
\text{xgcd}(f, \mp 1) &= (1, 0, \mp 1) \\
\text{xgcd}(\mp 1, g) &= (1, \mp 1, 0) \quad g \neq 0, \pm 1 \\
\text{xgcd}(\mp 2d, g) &= (d \cdot \frac{d-|g|}{\mp 2d}, \text{sgn}(g)) \\
\text{xgcd}(f, \mp 2d) &= (d, \text{sgn}(f), \frac{d-|g|}{\mp 2d}).
\end{align*}

If the pair $(f,g)$ does not satisfy any of these conditions, the solution $(d, a, b)$ will satisfy the following:

\[ |a| < \left| \frac{g}{2d} \right|, \quad |b| < \left| \frac{f}{2d} \right|. \]

Assumes that there is no aliasing among the outputs.

void \texttt{fmpz\_xgcd\_partial}(\texttt{fmpz\_t co2}, \texttt{fmpz\_t co1}, \texttt{fmpz\_t r2}, \texttt{fmpz\_t r1}, \texttt{const fmpz\_t L})

This function is an implementation of Lehmer extended GCD with early termination, as used in the \texttt{qfb} module. It terminates early when remainders fall below the specified bound. The initial values $r1$ and $r2$ are treated as successive remainders in the Euclidean algorithm and are replaced with the last two remainders computed. The values $co1$ and $co2$ are the last two cofactors and satisfy the identity $co2*r1 - co1*r2 = +/- r2\_orig$ upon termination, where $r2\_orig$ is the starting value of $r2$ supplied, and $r1$ and $r2$ are the final values.

Aliasing of inputs is not allowed. Similarly aliasing of inputs and outputs is not allowed.
4.2.11 Modular arithmetic

\texttt{slong \_fmpz_remove(fmpz\_t x, const fmpz\_t f, double finv)}

Removes all factors \( f \) from \( x \) and returns the number of such.

Assumes that \( x \) is non-zero, that \( f > 1 \) and that \( \text{finv} \) is the precomputed \texttt{double} inverse of \( f \) whenever \( f \) is a small integer and 0 otherwise.

Does not support aliasing.

\texttt{slong fmpz\_remove(fmpz\_t rop, const fmpz\_t op, const fmpz\_t f)}

Remove all occurrences of the factor \( f > 1 \) from the integer \( \text{op} \) and sets \( \text{rop} \) to the resulting integer.

If \( \text{op} \) is zero, sets \( \text{rop} \) to \( \text{op} \) and returns 0.

Returns an \texttt{abort} signal if any of the assumptions are violated.

\texttt{int fmpz\_invmod(fmpz\_t f, const fmpz\_t g, const fmpz\_t h)}

Sets \( f \) to the inverse of \( g \) modulo \( h \). The value of \( h \) may not be 0 otherwise an exception results. If the inverse exists the return value will be non-zero, otherwise the return value will be 0 and the value of \( f \) undefined. As a special case, we consider any number invertible modulo \( h = \pm 1 \), with inverse 0.

\texttt{void fmpz\_negmod(fmpz\_t f, const fmpz\_t g, const fmpz\_t h)}

Sets \( f \) to \(-g \) (mod \( h \)), assuming \( g \) is reduced modulo \( h \).

\texttt{int fmpz\_jacobi(const fmpz\_t a, const fmpz\_t n)}

Computes the Jacobi symbol \((\frac{a}{n})\) for any \( a \) and odd positive \( n \).

\texttt{int fmpz\_kronecker(const fmpz\_t a, const fmpz\_t n)}

Computes the Kronecker symbol \((\frac{a}{n})\) for any \( a \) and any \( n \).

\texttt{void fmpz\_divides\_mod\_list(fmpz\_t xstart, fmpz\_t xstride, fmpz\_t xlength, const fmpz\_t a, const fmpz\_t b, const fmpz\_t n)}

Set \( \text{xstart}, \text{xstride}, \) and \( \text{xlength} \) so that the solution set for \( x \) modulo \( n \) in \( a \times x = b \mod n \) is exactly \( \{\text{xstart} + \text{xstride}i | 0 \leq i < \text{xlength}\} \). This function essentially gives a list of possibilities for the fraction \( \frac{a}{b} \) modulo \( n \). The outputs may not be aliased, and \( n \) should be positive.

4.2.12 Bit packing and unpacking

\texttt{int fmpz\_bit\_pack(mp\_limb\_t \*arr, flint\_bitcnt\_t shift, flint\_bitcnt\_t bits, const fmpz\_t coeff, int negate, int borrow)}

Shifts the given coefficient to the left by \texttt{shift} bits and adds it to the integer in \texttt{arr} in a field of the given number of bits:

\begin{center}
\begin{tabular}{cccccccccc}
shift & bits & \multicolumn{10}{c}{
\begin{tabular}{c}
\hline
X & X & X & C & C & C & 0 & 0 & 0 & 0 & 0
\end{tabular}
}\end{tabular}
\end{center}

An optional borrow of 1 can be subtracted from \texttt{coeff} before it is packed. If \texttt{coeff} is negative after the borrow, then a borrow will be returned by the function.

The value of \texttt{shift} is assumed to be less than \texttt{FLINT\_BITS}. All but the first \texttt{shift} bits of \texttt{arr} are assumed to be zero on entry to the function.

The value of \texttt{coeff} may also be optionally (and notionally) negated before it is used, by setting the \texttt{negate} parameter to \(-1\).
int fmpz_bit_unpack(fmpz_t coeff, mp_limb_t *arr, flint_bitcnt_t shift, flint_bitcnt_t bits, int negate, int borrow)

A bit field of the given number of bits is extracted from arr, starting after shift bits, and placed into coeff. An optional borrow of 1 may be added to the coefficient. If the result is negative, a borrow of 1 is returned. Finally, the resulting coeff may be negated by setting the negate parameter to −1.

The value of shift is expected to be less than FLINT_BITS.

void fmpz_bit_unpack_unsigned(fmpz_t coeff, const mp_limb_t *arr, flint_bitcnt_t shift, flint_bitcnt_t bits)

A bit field of the given number of bits is extracted from arr, starting after shift bits, and placed into coeff.

The value of shift is expected to be less than FLINT_BITS.

4.2.13 Logic Operations

void fmpz_complement(fmpz_t r, const fmpz_t f)

The variable r is set to the ones-complement of f.

void fmpz_clrbit(fmpz_t f, ulong i)

Sets the ith bit in f to zero.

void fmpz_combit(fmpz_t f, ulong i)

Complements the ith bit in f.

void fmpz_and(fmpz_t r, const fmpz_t a, const fmpz_t b)

Sets r to the bit-wise logical and of a and b.

void fmpz_or(fmpz_t r, const fmpz_t a, const fmpz_t b)

Sets r to the bit-wise logical (inclusive) or of a and b.

void fmpz_xor(fmpz_t r, const fmpz_t a, const fmpz_t b)

Sets r to the bit-wise logical exclusive or of a and b.

ulong fmpz_popcnt(const fmpz_t a)

Returns the number of ‘1’ bits in the given Z (aka Hamming weight or population count). The return value is undefined if the input is negative.

4.2.14 Chinese remainding

The following functions can be used to reconstruct an integer from its residues modulo a set of prime numbers. The first two functions, fmpz_CRT_ui() and fmpz_CRT(), are easy to use and allow building the result one residue at a time, which is useful when the number of needed primes is not known in advance. The remaining functions support performing the modular reductions and reconstruction using balanced subdivision. This greatly improves efficiency for large integers but assumes that the basis of primes is known in advance. The user must precompute a comb structure and temporary working space with fmpz_comb_init() and fmpz_comb_temp_init(), and free this data afterwards. For simple demonstration programs showing how to use the CRT functions, see crt.c and multi_crt.c in the examples directory. The fmpz_multi_CRT class is similar to fmpz_multi_CRT_ui except that it performs error checking and works with arbitrary moduli.

void fmpz_CRT_ui(fmpz_t out, const fmpz_t r1, const fmpz_t m1, ulong r2, ulong m2, int sign)

Uses the Chinese Remainder Theorem to compute the unique integer 0 ≤ x < M (if sign = 0) or −M/2 < x ≤ M/2 (if sign = 1) congruent to r1 modulo m1 and r2 modulo m2, where M = m1 × m2. The result x is stored in out.

It is assumed that m1 and m2 are positive coprime integers.
If sign = 0, it is assumed that 0 ≤ r₁ < m₁ and 0 ≤ r₂ < m₂. Otherwise, it is assumed that
−m₁ ≤ r₁ < m₁ and 0 ≤ r₂ < m₂.

void \texttt{fmpz\_CRT}(\texttt{fmpz\_t} out, const \texttt{fmpz\_t} r1, const \texttt{fmpz\_t} m1, const \texttt{fmpz\_t} r2, const \texttt{fmpz\_t} m2, int sign)

Use the Chinese Remainder Theorem to set \texttt{out} to the unique value 0 ≤ x < M (if sign = 0) or
−M/2 < x ≤ M/2 (if sign = 1) congruent to \(r_1\) modulo \(m_1\) and \(r_2\) modulo \(m_2\), where \(M = m_1 \times m_2\).

It is assumed that \(m_1\) and \(m_2\) are positive coprime integers.

If sign = 0, it is assumed that 0 ≤ r₁ < m₁ and 0 ≤ r₂ < m₂. Otherwise, it is assumed that
−m₁ ≤ r₁ < m₁ and 0 ≤ r₂ < m₂.

void \texttt{fmpz\_multi\_mod\_ui}(\texttt{mp\_limb\_t} *out, const \texttt{fmpz\_t} in, const \texttt{fmpz\_comb\_t} comb,
\texttt{fmpz\_comb\_temp\_t} temp)

Reduces the multiprecision integer \texttt{in} modulo each of the primes stored in the \texttt{comb}
structure. The array \texttt{out} will be filled with the residues modulo these primes. The structure
\texttt{temp} is temporary space which must be provided by \texttt{fmpz\_comb\_temp\_init()} and cleared by
\texttt{fmpz\_comb\_temp\_clear()}.

void \texttt{fmpz\_multi\_CRT\_ui}(\texttt{fmpz\_t} output, \texttt{mp\_sring} residues, const \texttt{fmpz\_comb\_t} comb,
\texttt{fmpz\_comb\_temp\_t} ctemp, int sign)

This function takes a set of residues modulo the list of primes contained in the \texttt{comb}
structure and reconstructs a multiprecision integer modulo the product of the primes which has these residues
modulo the corresponding primes.

If \(N\) is the product of all the primes then \texttt{out} is normalised to be in the range \([0, N]\) if sign = 0
and the range \([-\lfloor(N - 1)/2\rfloor, N/2]\) if sign = 1. The array \texttt{temp} is temporary space which must be
provided by \texttt{fmpz\_comb\_temp\_init()} and cleared by \texttt{fmpz\_comb\_temp\_clear()}.

void \texttt{fmpz\_comb\_init}(\texttt{fmpz\_comb\_t} comb, \texttt{mp\_sring} primes, \texttt{slong} num\_primes)

Initialises a \texttt{comb} structure for multimodular reduction and recombination. The array \texttt{primes}
is assumed to contain \texttt{num\_primes} primes each of FLINT\_BITS - 1 bits. Modular reductions and
recombinations will be done modulo this list of primes. The \texttt{primes} array must not be \texttt{free’d}
until the \texttt{comb} structure is no longer required and must be cleared by the user.

void \texttt{fmpz\_comb\_temp\_init}(\texttt{fmpz\_comb\_temp\_t} temp, const \texttt{fmpz\_comb\_t} comb)

Creates temporary space to be used by multimodular and CRT functions based on an initialised
\texttt{comb} structure.

void \texttt{fmpz\_comb\_clear}(\texttt{fmpz\_comb\_t} comb)

Clears the given \texttt{comb} structure, releasing any memory it uses.

void \texttt{fmpz\_comb\_temp\_clear}(\texttt{fmpz\_comb\_temp\_t} temp)

Clears temporary space \texttt{temp} used by multimodular and CRT functions using the given \texttt{comb}
structure.

void \texttt{fmpz\_multi\_CRT\_init}(\texttt{fmpz\_multi\_CRT\_t} CRT)

Initialize CRT for Chinese remaindering.

int \texttt{fmpz\_multi\_CRT\_precompute}(\texttt{fmpz\_multi\_CRT\_t} CRT, const \texttt{fmpz\*} moduli, \texttt{slong} len)

Configure CRT for repeated Chinese remaindering of \texttt{moduli}. The number of moduli, \texttt{len},
should be positive. A return of 0 indicates that the compilation failed and future calls to
\texttt{fmpz\_multi\_CRT\_precomp()} will leave the output undefined. A return of 1 indicates that the
compilation was successful, which occurs if and only if either (1) \texttt{len} == 1 and \texttt{modulus} + 0 is
nonzero, or (2) no modulus is 0, 1, −1 and all moduli are pairwise relatively prime.

void \texttt{fmpz\_multi\_CRT\_precomp}(\texttt{fmpz\_t} output, const \texttt{fmpz\_multi\_CRT\_t} P, const \texttt{fmpz\*} inputs, int sign)

Set \texttt{output} to an integer of smallest absolute value that is congruent to \texttt{values} + \texttt{i} modulo the
moduli + \texttt{i} in \texttt{P}. 

4.2. \texttt{fmpz\_h} – integers
int fmpz_multi_CRT(fmpz_t output, const fmpz *moduli, const fmpz *values, slong len, int sign)

Perform the same operation as fmpz_multi_CRT_precomp() while internally constructing and destroying the precomputed data. All of the remarks in fmpz_multi_CRT_precompute() apply.

void fmpz_multi_CRT_clear(fmpz_multi_CRT_t P)

Free all space used by CRT.

4.2.15 Primality testing

int fmpz_is_strong_probabprime(const fmpz_t n, const fmpz_t a)

Returns 1 if n is a strong probable prime to base a, otherwise it returns 0.

int fmpz_is_probabprime_lucas(const fmpz_t n)

Performs a Lucas probable prime test with parameters chosen by Selfridge’s method A as per [BaiWag1980].

Return 1 if n is a Lucas probable prime, otherwise return 0. This function declares some composites probably prime, but no primes composite.

int fmpz_is_probabprime_BPSW(const fmpz_t n)

Perform a Baillie-PSW probable prime test with parameters chosen by Selfridge’s method A as per [BaiWag1980].

Return 1 if n is a Lucas probable prime, otherwise return 0.

There are no known composites passed as prime by this test, though infinitely many probably exist. The test will declare no primes composite.

int fmpz_is_probabprime(const fmpz_t p)

Performs some trial division and then some probabilistic primality tests. If p is definitely composite, the function returns 0, otherwise it is declared probably prime, i.e. prime for most practical purposes, and the function returns 1. The chance of declaring a composite prime is very small. Subsequent calls to the same function do not increase the probability of the number being prime.

int fmpz_is_prime_pseudosquare(const fmpz_t n)

Return 0 is n is composite. If n is too large (greater than about 94 bits) the function fails silently and returns -1, otherwise, if n is proven prime by the pseudosquares method, return 1.

Tests if n is a prime according to [Theorem 2.7] [LukPatWil1996].

We first factor N using trial division up to some limit B. In fact, the number of primes used in the trial factoring is at most FLINT_PSEUDOSQUARES_CUTOFF.

Next we compute N/B and find the next pseudosquare Lp above this value, using a static table as per https://oeis.org/A002189/b002189.txt.

As noted in the text, if p is prime then Step 3 will pass. This test rejects many composites, and so by this time we suspect that p is prime. If N is 3 or 7 modulo 8, we are done, and N is prime.

We now run a probable prime test, for which no known counterexamples are known, to reject any composites. We then proceed to prove N prime by executing Step 4. In the case that N is 1 modulo 8, if Step 4 fails, we extend the number of primes pi at Step 3 and hope to find one which passes Step 4. We take the test one past the largest p for which we have pseudosquares Lp tabulated, as this already corresponds to the next Lp which is bigger than 2^{64} and hence larger than any prime we might be testing.

As explained in the text, Condition 4 cannot fail if N is prime.

The possibility exists that the probable prime test declares a composite prime. However in that case an error is printed, as that would be of independent interest.
int fmpz_is_prime_pocklington(const fmpz_t F, fmpz_t R, const fmpz_t n, mp_ptr pm1, slong num_pm1)

Applies the Pocklington primality test. The test computes a product $F$ of prime powers which divide $n - 1$.

The function then returns either 0 if $n$ is definitely composite or it returns 1 if all factors of $n$ are $1 \pmod{F}$. Also in that case, $R$ is set to $(n - 1)/F$.

NB: a return value of 1 only proves $n$ prime if $F \geq \sqrt{n}$.

The function does not compute which primes divide $n - 1$. Instead, these must be supplied as an array pm1 of length num_pm1. It does not matter how many prime factors are supplied, but the more that are supplied, the larger $F$ will be.

There is a balance between the amount of time spent looking for factors of $n - 1$ and the usefulness of the output ($F$ may be as low as 2 in some cases).

A reasonable heuristic seems to be to choose limit to be some small multiple of $\log^3(n)/10$ (e.g. 1, 2, 5 or 10) depending on how long one is prepared to wait, then to trial factor up to the limit. (See fmpz_nm1_trial_factors.)

Requires $n$ to be odd.

void fmpz_nm1_trial_factors(const fmpz_t n, mp_ptr pm1, slong *num_pm1, ulong limit)

Trial factors $n - 1$ up to the given limit (approximately) and stores the factors in an array pm1 whose length is written out to num_pm1.

One can use $\log(n) + 2$ as a bound on the number of factors which might be produced (and hence on the length of the array that needs to be supplied).

int fmpz_is_prime_morrison(const fmpz_t F, fmpz_t R, const fmpz_t n, mp_ptr pp1, slong num_pp1)

Applies the Morrison $p + 1$ primality test. The test computes a product $F$ of primes which divide $n + 1$.

The function then returns either 0 if $n$ is definitely composite or it returns 1 if all factors of $n$ are $\pm 1 \pmod{F}$. Also in that case, $R$ is set to $(n + 1)/F$.

NB: a return value of 1 only proves $n$ prime if $F > \sqrt{n} + 1$.

The function does not compute which primes divide $n + 1$. Instead, these must be supplied as an array pp1 of length num_pp1. It does not matter how many prime factors are supplied, but the more that are supplied, the larger $F$ will be.

There is a balance between the amount of time spent looking for factors of $n + 1$ and the usefulness of the output ($F$ may be as low as 2 in some cases).

A reasonable heuristic seems to be to choose limit to be some small multiple of $\log^3(n)/10$ (e.g. 1, 2, 5 or 10) depending on how long one is prepared to wait, then to trial factor up to the limit. (See fmpz_np1_trial_factors.)

Requires $n$ to be odd and non-square.

void fmpz_np1_trial_factors(const fmpz_t n, mp_ptr pp1, slong *num_pp1, ulong limit)

Trial factors $n + 1$ up to the given limit (approximately) and stores the factors in an array pp1 whose length is written out to num_pp1.

One can use $\log(n) + 2$ as a bound on the number of factors which might be produced (and hence on the length of the array that needs to be supplied).

int fmpz_is_prime(const fmpz_t n)

Attempts to prove $n$ prime. If $n$ is proven prime, the function returns 1. If $n$ is definitely composite, the function returns 0.

This function calls n_is_prime() for $n$ that fits in a single word. For $n$ larger than one word, it tests divisibility by a few small primes and whether $n$ is a perfect square to rule out trivial
composites. For \( n \) up to about 81 bits, it then uses a strong probable prime test (Miller-Rabin test) with the first 13 primes as witnesses. This has been shown to prove primality [SorWeb2016].

For larger \( n \), it does a single base-2 strong probable prime test to eliminate most composite numbers. If \( n \) passes, it does a combination of Pocklington, Morrison and Brillhart, Lehmer, Selfridge tests. If any of these tests fails to give a proof, it falls back to performing an APRCL test.

The APRCL test could theoretically fail to prove that \( n \) is prime or composite. In that case, the program aborts. This is not expected to occur in practice.

```c
void fmpz_lucas_chain(fmpz_t Vm, fmpz_t Vm1, const fmpz_t A, const fmpz_t m, const fmpz_t n)
Given \( V_0 = 2 \), \( V_1 = A \) compute \( V_m, V_{m+1} \pmod n \) from the recurrences \( V_j = AV_{j-1} - V_{j-2} \pmod n \).
This is computed efficiently using \( V_{2j} = V_j^2 - 2 \pmod n \) and \( V_{2j+1} = V_jV_{j+1} - A \pmod n \).
No aliasing is permitted.
```

```c
void fmpz_lucas_chain_full(fmpz_t Vm, fmpz_t Vm1, const fmpz_t A, const fmpz_t B, const fmpz_t m, const fmpz_t n)
Given \( V_0 = 2 \), \( V_1 = A \) compute \( V_m, V_{m+1} \pmod n \) from the recurrences \( V_j = AV_{j-1} - BV_{j-2} \pmod n \).
This is computed efficiently using double and add formulas.
No aliasing is permitted.
```

```c
void fmpz_lucas_chain_double(fmpz_t U2m, fmpz_t U2m1, const fmpz_t U2m, const fmpz_t Un, const fmpz_t Un1, const fmpz_t A, const fmpz_t B, const fmpz_t m, const fmpz_t n)
Given \( U_m, U_{m+1} \pmod n \) compute \( U_{2m}, U_{2m+1} \pmod n \).
Aliasing of \( U_{2m+1} \) and \( U_m \) and aliasing of \( U_{2m+1} \) and \( U_{m+1} \) is permitted. No other aliasing is allowed.
```

```c
void fmpz_lucas_chain_add(fmpz_t Un, fmpz_t Un1, const fmpz_t U2m, const fmpz_t Un, const fmpz_t Un1, const fmpz_t A, const fmpz_t B, const fmpz_t m, const fmpz_t n)
Given \( U_m, U_{m+1} \pmod n \) and \( U_n, U_{n+1} \pmod n \) compute \( U_{m+n}, U_{m+n+1} \pmod n \).
Aliasing of \( U_{m+n} \) with \( U_m \) or \( U_n \) and aliasing of \( U_{m+n+1} \) with \( U_{m+1} \) or \( U_{n+1} \) is permitted. No other aliasing is allowed.
```

```c
void fmpz_lucas_chain_mul(fmpz_t Ukm, fmpz_t Ukm1, const fmpz_t Ukm, const fmpz_t Un, const fmpz_t Un1, const fmpz_t A, const fmpz_t B, const fmpz_t k, const fmpz_t m, const fmpz_t n)
Given \( U_m, U_{m+1} \pmod n \) compute \( U_{km}, U_{km+1} \pmod n \).
Aliasing of \( U_{km} \) and \( U_n \) and aliasing of \( U_{km+1} \) and \( U_{m+1} \) is permitted. No other aliasing is allowed.
```

```c
void fmpz_lucas_chain_VtoU(fmpz_t Um, fmpz_t Un, const fmpz_t Vm, const fmpz_t Vm1, const fmpz_t A, const fmpz_t B, const fmpz_t Dinv, const fmpz_t n)
Given \( V_m, V_{m+1} \pmod n \) compute \( U_m, U_{m+1} \pmod n \).
Aliasing of \( V_m \) and \( U_m \) and aliasing of \( V_{m+1} \) and \( U_{m+1} \) is permitted. No other aliasing is allowed.
```

```c
int fmpz_divisor_in_residue_class_lenstra(fmpz_t fac, const fmpz_t n, const fmpz_t r, const fmpz_t s)
If there exists a proper divisor of \( n \) which is \( r \pmod s \) for \( 0 < r < s < n \), this function returns 1 and sets \( fac \) to such a divisor. Otherwise the function returns 0 and the value of \( fac \) is undefined.
We require \( \gcd(r, s) = 1 \).
This is efficient if \( s^4 > n \).
```

```c
void fmpz_nextprime(fmpz_t res, const fmpz_t n, int proved)
Finds the next prime number larger than \( n \).
If \texttt{proved} is nonzero, then the integer returned is guaranteed to actually be prime. Otherwise if \( n \) fits in FLINT\_BITS - 3 bits \texttt{n\_nextprime} is called, and if not then the GMP \texttt{mpz\_nextprime} function is called which uses a BPSW test.

4.2.16 Special functions

\begin{verbatim}
void \texttt{fmpz\_primorial}(\texttt{fmpz\_t res, ulong n})
Sets \texttt{res} to \( n \) primorial or \( n\# \), the product of all prime numbers less than or equal to \( n \).

void \texttt{fmpz\_factor\_euler\_phi}(\texttt{fmpz\_t res, const fmpz\_factor\_t fac})
void \texttt{fmpz\_euler\_phi}(\texttt{fmpz\_t res, const fmpz\_t n})
Sets \texttt{res} to the Euler totient function \( \phi(n) \), counting the number of positive integers less than or equal to \( n \) that are coprime to \( n \). The factor version takes a precomputed factorisation of \( n \).

int \texttt{fmpz\_factor\_moebius\_mu(const fmpz\_factor\_t fac})
int \texttt{fmpz\_moebius\_mu(const fmpz\_t n})
Computes the Moebius function \( \mu(n) \), which is defined as \( \mu(n) = 0 \) if \( n \) has a prime factor of multiplicity greater than 1, \( \mu(n) = -1 \) if \( n \) has an odd number of distinct prime factors, and \( \mu(n) = 1 \) if \( n \) has an even number of distinct prime factors. By convention, \( \mu(0) = 0 \). The factor version takes a precomputed factorisation of \( n \).

void \texttt{fmpz\_factor\_divisor\_sigma}(\texttt{fmpz\_t res, ulong k, const fmpz\_factor\_t fac})
void \texttt{fmpz\_divisor\_sigma}(\texttt{fmpz\_t res, ulong k, const fmpz\_t n})
Sets \texttt{res} to \( \sigma_k(n) \), the sum of \( k \)th powers of all divisors of \( n \). The factor version takes a precomputed factorisation of \( n \).
\end{verbatim}

4.3 \texttt{fmpz\_vec.h} – vectors of integers

4.3.1 Memory management

\begin{verbatim}
\texttt{fmpz \_fmpz\_vec\_init(slong len})
Returns an initialised vector of \texttt{fmpz}'s of given length.

void \_fmpz\_vec\_clear(\texttt{fmpz \_vec, slong len})
Clears the entries of (\texttt{vec, len}) and frees the space allocated for \texttt{vec}.
\end{verbatim}

4.3.2 Randomisation

\begin{verbatim}
void \_fmpz\_vec\_randtest(\texttt{fmpz \_f, flint\_rand\_t state, slong len, flint\_bitcnt\_t bits})
Sets the entries of a vector of the given length to random integers with up to the given number of bits per entry.

void \_fmpz\_vec\_randtest\_unsigned(\texttt{fmpz \_f, flint\_rand\_t state, slong len, flint\_bitcnt\_t bits})
Sets the entries of a vector of the given length to random unsigned integers with up to the given number of bits per entry.
\end{verbatim}
4.3.3 Bit sizes and norms

`slong _fmpz_vec_max_bits(const fmpz *vec, slong len)`

If $b$ is the maximum number of bits of the absolute value of any coefficient of $vec$, then if any coefficient of $vec$ is negative, $-b$ is returned, else $b$ is returned.

`slong _fmpz_vec_max_bits_ref(const fmpz *vec, slong len)`

If $b$ is the maximum number of bits of the absolute value of any coefficient of $vec$, then if any coefficient of $vec$ is negative, $-b$ is returned, else $b$ is returned. This is a slower reference implementation of `_fmpz_vec_max_bits`.

`void _fmpz_vec_sum_max_bits(slong *sumabs, slong *maxabs, const fmpz *vec, slong len)`

Sets $sumabs$ to the bit count of the sum of the absolute values of the elements of $vec$. Sets $maxabs$ to the bit count of the maximum of the absolute values of the elements of $vec$.

`mp_size_t _fmpz_vec_max_limbs(const fmpz *vec, slong len)`

Returns the maximum number of limbs needed to store the absolute value of any entry in $(vec, len)$. If all entries are zero, returns zero.

`void _fmpz_vec_height(fmpz_t height, const fmpz *vec, slong len)`

Computes the height of $(vec, len)$, defined as the largest of the absolute values the coefficients. Equivalently, this gives the infinity norm of the vector. If $len$ is zero, the height is 0.

`slong _fmpz_vec_height_index(const fmpz *vec, slong len)`

Returns the index of an entry of maximum absolute value in the vector. The length must be at least 1.

4.3.4 Input and output

`int _fmpz_vec_fread(FILE *file, fmpz **vec, slong *len)`

Reads a vector from the stream `file` and stores it at `*vec`. The format is the same as the output format of `_fmpz_vec_fprint()`, followed by either any character or the end of the file.

The interpretation of the various input arguments depends on whether or not `*vec` is NULL:

If `*vec` == NULL, the value of `*len` on input is ignored. Once the length has been read from `file`, `*len` is set to that value and a vector of this length is allocated at `*vec`. Finally, `*len` coefficients are read from the input stream. In case of a file or parsing error, clears the vector and sets `*vec` and `*len` to NULL and 0, respectively.

Otherwise, if `*vec` != NULL, it is assumed that `(*vec, *len)` is a properly initialised vector. If the length on the input stream does not match `*len`, a parsing error is raised. Attempts to read the right number of coefficients from the input stream. In case of a file or parsing error, leaves the vector `(*vec, *len)` in its current state.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

`int _fmpz_vec_read(fmpz **vec, slong *len)`

Reads a vector from `stdin` and stores it at `*vec`.

For further details, see `_fmpz_vec_fread()`.

`int _fmpz_vec_fprint(FILE *file, const fmpz *vec, slong len)`

Prints the vector of given length to the stream `file`. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

`int _fmpz_vec_print(const fmpz *vec, slong len)`

Prints the vector of given length to `stdout`.

For further details, see `_fmpz_vec_fprint()`.
4.3.5 Conversions

void _fmpz_vec_get_nmod_vec(mp_ptr res, const fmpz *poly, slong len, nmod_t mod)
Reduce the coefficients of (poly, len) modulo the given modulus and set (res, len) to the result.

void _fmpz_vec_set_nmod_vec(fmpz *res, mp_srcptr poly, slong len, nmod_t mod)
Set the coefficients of (res, len) to the symmetric modulus of the coefficients of (poly, len), i.e.
convert the given coefficients modulo the given modulus n to their signed integer representatives
in the range \([-n/2, n/2)\).

void _fmpz_vec_get_fft(mp_limb_t **coeffs_f, const fmpz *coeffs_m, slong l, slong length)
Convert the vector of coeffs coeffs_m to an fft vector coeffs_f of the given length with l limbs
per coefficient with an additional limb for overflow.

void _fmpz_vec_set_fft(fmpz *coeffs_m, slong length, const mp_ptr *coeffs_f, slong limbs, slong sign)
Convert an fft vector coeffs_f of fully reduced Fermat numbers of the given length to a vector of fmpz's. Each is assumed to be the given number of limbs in length with an additional limb for overflow. If the output coefficients are to be signed then set sign, otherwise clear it. The resulting fmpz's will be in the range \([-n,n]\) in the signed case and in the range \([0,2n]\) in the unsigned case where \(n = 2^{(FLINT_BITS\times limbs - 1)}\).

slong _fmpz_vec_get_d_vec_2exp(double *appv, const fmpz *vec, slong len)
Export the array of len entries starting at the pointer vec to an array of doubles appv, each entry of which is notionally multiplied by a single returned exponent to give the original entry. The returned exponent is set to be the maximum exponent of all the original entries so that all the doubles in appv have a maximum absolute value of 1.0.

4.3.6 Assignment and basic manipulation

void _fmpz_vec_set(fmpz *vec1, const fmpz *vec2, slong len2)
Makes a copy of (vec2, len2) into vec1.

void _fmpz_vec_swap(fmpz *vec1, fmpz *vec2, slong len2)
Swaps the integers in (vec1, len2) and (vec2, len2).

void _fmpz_vec_zero(fmpz *vec, slong len)
Zeros the entries of (vec, len).

void _fmpz_vec_neg(fmpz *vec1, const fmpz *vec2, slong len2)
Negates (vec2, len2) and places it into vec1.

void _fmpz_vec_scalar_abs(fmpz *vec1, const fmpz *vec2, slong len2)
Takes the absolute value of entries in (vec2, len2) and places the result into vec1.

4.3.7 Comparison

int _fmpz_vec_equal(const fmpz *vec1, const fmpz *vec2, slong len)
Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

int _fmpz_vec_is_zero(const fmpz *vec, slong len)
Returns 1 if (vec, len) is zero, and 0 otherwise.

void _fmpz_vec_max(fmpz *vec1, const fmpz *vec2, const fmpz *vec3, slong len)
Sets vec1 to the pointwise maximum of vec2 and vec3.

void _fmpz_vec_max_inplace(fmpz *vec1, const fmpz *vec2, slong len)
Sets vec1 to the pointwise maximum of vec1 and vec2.
4.3.8 Sorting

```c
void _fmpz_vec_sort(fmpz *vec, slong len)
```

Sorts the coefficients of `vec` in ascending order.

4.3.9 Addition and subtraction

```c
void _fmpz_vec_add(fmpz *res, const fmpz *vec1, const fmpz *vec2, slong len2)
```

Sets `(res, len2)` to the sum of `(vec1, len2)` and `(vec2, len2)`.

```c
void _fmpz_vec_sub(fmpz *res, const fmpz *vec1, const fmpz *vec2, slong len2)
```

Sets `(res, len2)` to `(vec1, len2)` minus `(vec2, len2)`.

4.3.10 Scalar multiplication and division

```c
void _fmpz_vec_scalar_mul_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t x)
```

Sets `(vec1, len2)` to `(vec2, len2)` multiplied by `c`, where `c` is an `fmpz_t`.

```c
void _fmpz_vec_scalar_mul_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` multiplied by `c`, where `c` is a `slong`.

```c
void _fmpz_vec_scalar_mul_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` multiplied by `c`, where `c` is an `ulong`.

```c
void _fmpz_vec_scalar_mul_2exp(fmpz *vec1, const fmpz *vec2, slong len2, ulong exp)
```

Sets `(vec1, len2)` to `(vec2, len2)` multiplied by $2^\text{exp}$.

```c
void _fmpz_vec_scalar_divexact_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t x)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by `x`, where the division is assumed to be exact for every entry in `vec2`.

```c
void _fmpz_vec_scalar_divexact_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by `c`, where the division is assumed to be exact for every entry in `vec2`.

```c
void _fmpz_vec_scalar_divexact_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by `c`, where the division is assumed to be exact for every entry in `vec2`.

```c
void _fmpz_vec_scalar_fdiv_q_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by `c`, rounding down towards minus infinity whenever the division is not exact.

```c
void _fmpz_vec_scalar_fdiv_q_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by `c`, rounding down towards minus infinity whenever the division is not exact.

```c
void _fmpz_vec_scalar_fdiv_q_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by `c`, rounding down towards minus infinity whenever the division is not exact.

```c
void _fmpz_vec_scalar_fdiv_q_2exp(fmpz *vec1, const fmpz *vec2, slong len2, ulong exp)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by $2^{\text{exp}}$, rounding down towards minus infinity whenever the division is not exact.

```c
void _fmpz_vec_scalar_fdiv_r_2exp(fmpz *vec1, const fmpz *vec2, slong len2, ulong exp)
```

Sets `(vec1, len2)` to the remainder of `(vec2, len2)` divided by $2^{\text{exp}}$, rounding down the quotient towards minus infinity whenever the division is not exact.
void _fmpz_vec_scalar_tdiv_q_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t c)
Sets (vec1, len2) to (vec2, len2) divided by c, rounding towards zero whenever the division is not exact.

void _fmpz_vec_scalar_tdiv_q_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)
Sets (vec1, len2) to (vec2, len2) divided by c, rounding towards zero whenever the division is not exact.

void _fmpz_vec_scalar_tdiv_q_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)
Sets (vec1, len2) to (vec2, len2) divided by c, rounding towards zero whenever the division is not exact.

void _fmpz_vec_scalar_tdiv_q_2exp(fmpz *vec1, const fmpz *vec2, slong len2, ulong exp)
Sets (vec1, len2) to (vec2, len2) divided by $2^\text{exp}$, rounding down towards zero whenever the division is not exact.

void _fmpz_vec_scalar_addmul_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)
Adds (vec2, len2) times c to (vec1, len2).

void _fmpz_vec_scalar_addmul_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)
Adds (vec2, len2) times c to (vec1, len2).

void _fmpz_vec_scalar_addmul_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t c)
Adds (vec2, len2) times c to (vec1, len2), where c is a fmpz_t.

void _fmpz_vec_scalar_submul_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t x)
Subtracts (vec2, len2) times c from (vec1, len2), where c is a fmpz_t.

void _fmpz_vec_scalar_submul_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)
Subtracts (vec2, len2) times c from (vec1, len2), where c is a slong.

void _fmpz_vec_scalar_submul_si_2exp(fmpz *vec1, const fmpz *vec2, slong len2, slong c, ulong e)
Subtracts (vec2, len2) times c \times 2^e from (vec1, len2), where c is a slong.

4.3.11 Sums and products

void _fmpz_vec_sum(fmpz_t res, const fmpz *vec, slong len)
Sets res to the sum of the entries in (vec, len). Aliasing of res with the entries in vec is not permitted.

void _fmpz_vec_prod(fmpz_t res, const fmpz *vec, slong len)
Sets res to the product of the entries in (vec, len). Aliasing of res with the entries in vec is not permitted. Uses binary splitting.

4.3.12 Reduction mod $p$

void _fmpz_vec_scalar_mod_fmpz(fmpz *res, const fmpz *vec, slong len, const fmpz_t p)
Reduces all entries in (vec, len) modulo $p > 0$.

void _fmpz_vec_scalar_smod_fmpz(fmpz *res, const fmpz *vec, slong len, const fmpz_t p)
Reduces all entries in (vec, len) modulo $p > 0$, choosing the unique representative in $(-p/2, p/2]$. 

4.3. fmpz_vec.h – vectors of integers 145
4.3.13 Gaussian content

void _fmpz_vec_content(fmpz_t res, const fmpz *vec, slong len)

Sets res to the non-negative content of the entries in vec. The content of a zero vector, including the case when the length is zero, is defined to be zero.

void _fmpz_vec_content_chained(fmpz_t res, const fmpz *vec, slong len, const fmpz_t input)

Sets res to the non-negative content of input and the entries in vec. This is useful for calculating the common content of several vectors.

void _fmpz_vec_lcm(fmpz_t res, const fmpz *vec, slong len)

Sets res to the nonnegative least common multiple of the entries in vec. The least common multiple is zero if any entry in the vector is zero. The least common multiple of a length zero vector is defined to be one.

4.3.14 Dot product

void _fmpz_vec_dot_general_naive(fmpz_t res, const fmpz_t initial, int subtract, const fmpz *a, const fmpz *b, int reverse, slong len)

void _fmpz_vec_dot_general(fmpz_t res, const fmpz_t initial, int subtract, const fmpz *a, const fmpz_t b, int reverse, slong len)

Computes the dot product of the vectors a and b, setting res to $s + (-1)^{subtract} \sum_{i=0}^{len-1} a_i b_i$. The initial term s is optional and can be omitted by passing NULL (equivalently, s = 0). The parameter subtract must be 0 or 1. If the reverse flag is 1, the second vector is reversed.

Aliasing is allowed between res and initial but not between res and the entries of a and b.

The naive version is used for testing purposes.

void _fmpz_vec_dot(fmpz_t res, const fmpz *vec1, const fmpz *vec2, slong len2)

Sets res to the dot product of (vec1, len2) and (vec2, len2).

4.4 fmpz_factor.h – integer factorisation

4.4.1 Types, macros and constants

type fmpz_factor_struct

type fmpz_factor_t

4.4.2 Factoring integers

An integer may be represented in factored form using the fmpz_factor_t data structure. This consists of two fmpz vectors representing bases and exponents, respectively. Canonically, the bases will be prime numbers sorted in ascending order and the exponents will be positive. A separate int field holds the sign, which may be -1, 0 or 1.

void fmpz_factor_init(fmpz_factor_t factor)

Initialises an fmpz_factor_t structure.

void fmpz_factor_clear(fmpz_factor_t factor)

Clears an fmpz_factor_t structure.

void _fmpz_factor_append_ui(fmpz_factor_t factor, mp_limb_t p, ulong exp)

Append a factor p to the given exponent to the fmpz_factor_t structure factor.
void _fmpz_factor_append(fmpz_factor_t factor, const fmpz_t p, ulong exp)

Append a factor p to the given exponent to the fmpz_factor_t structure factor.

void fmpz_factor(fmpz_factor_t factor, const fmpz_t n)

Factors n into prime numbers. If n is zero or negative, the sign field of the factor object will be set accordingly.

int fmpz_factor_smooth(fmpz_factor_t factor, const fmpz_t n, slong bits, int proved)

Factors n into prime numbers up to approximately the given number of bits and possibly one additional cofactor, which may or may not be prime.

If the number is definitely factored fully, the return value is 1, otherwise the final factor (which may have exponent greater than 1) is composite and needs to be factored further.

If the number has a factor of around the given number of bits, there is at least a two-thirds chance of finding it. Smaller factors should be found with a much higher probability.

The amount of time spent factoring can be controlled by lowering or increasing bits. However, the quadratic sieve may be faster if bits is set to more than one third of the number of bits of n.

The function uses trial factoring up to bits = 15, followed by a primality test and a perfect power test to check if the factorisation is complete. If bits is at least 16, it proceeds to use the elliptic curve method to look for larger factors.

The behavior of primality testing is determined by the proved parameter:

If proved is set to 1 the function will prove all factors prime (other than the last factor, if the return value is 0).

If proved is set to 0, the function will only check that factors are probable primes.

If proved is set to −1, the function will not test primality after performing trial division. A perfect power test is still performed.

As an exception to the rules stated above, this function will call n_factor internally if n or the remainder after trial division is smaller than one word, guaranteeing a complete factorisation.

void fmpz_factor_si(fmpz_factor_t factor, slong n)

Like fmpz_factor, but takes a machine integer n as input.

int fmpz_factor_trial_range(fmpz_factor_t factor, const fmpz_t n, ulong start, ulong num_primes)

Factors n into prime factors using trial division. If n is zero or negative, the sign field of the factor object will be set accordingly.

The algorithm starts with the given start index in the flint_primes table and uses at most num_primes primes from that point.

The function returns 1 if n is completely factored, otherwise it returns 0.

int fmpz_factor_trial(fmpz_factor_t factor, const fmpz_t n, slong num_primes)

Factors n into prime factors using trial division. If n is zero or negative, the sign field of the factor object will be set accordingly.

The algorithm uses the given number of primes, which must be in the range [0, 3512]. An exception is raised if a number outside this range is passed.

The function returns 1 if n is completely factored, otherwise it returns 0.

The final entry in the factor struct is set to the cofactor after removing prime factors, if this is not 1.

void fmpz_factor_refine(fmpz_factor_t res, const fmpz_factor_t f)

Attempts to improve a partial factorization of an integer by “refining” the factorization f to a more complete factorization res whose bases are pairwise relatively prime.

This function does not require its input to be in canonical form, nor does it guarantee that the resulting factorization will be canonical.
void fmpz_factor_expand_iterative(fmpz_t n, const fmpz_factor_t factor)
Evaluates an integer in factored form back to an fmpz_t.

This currently exponentiates the bases separately and multiplies them together one by one, although much more efficient algorithms exist.

int fmpz_factor_pp1(fmpz_t factor, const fmpz_t n, ulong B1, ulong B2_sqrt, ulong c)
Use Williams’ \( p + 1 \) method to factor \( n \), using a prime bound in stage 1 of \( B_1 \) and a prime limit in stage 2 of at least the square of \( B_2 \text{-sqrt} \). If a factor is found, the function returns 1 and \( \text{factor} \) is set to the factor that is found. Otherwise, the function returns 0.

The value \( c \) should be a random value greater than 2. Successive calls to the function with different values of \( c \) give additional chances to factor \( n \) with roughly exponentially decaying probability of finding a factor which has been missed (if \( p + 1 \) or \( p - 1 \) is not smooth for any prime factors \( p \) of \( n \) then the function will not ever succeed).

int fmpz_factor_pollard_brent_single(fmpz_t p_factor, fmpz_t n_in, fmpz_t yi, fmpz_t ai, mp_limb_t max_iters)
Pollard Rho algorithm for integer factorization. Assumes that the \( n \) is not prime. \( \text{factor} \) is set as the factor if found. Takes as input the initial value \( y \), to start polynomial evaluation, and \( a \), the constant of the polynomial used. It is not assured that the factor found will be prime. Does not compute the complete factorization, just one factor. Returns the number of limbs of factor if factorization is successful (non trivial factor is found), else returns 0.

\( \text{max\_iters} \) is the number of iterations tried in process of finding the cycle. If the algorithm fails to find a non trivial factor in one call, it tries again (this time with a different set of random values).

int fmpz_factor_pollard_brent(fmpz_t factor, flint_rand_t state, fmpz_t n, mp_limb_t max_tries, mp_limb_t max_iters)
Pollard Rho algorithm for integer factorization. Assumes that the \( n \) is not prime. \( \text{factor} \) is set as the factor if found. It is not assured that the factor found will be prime. Does not compute the complete factorization, just one factor. Returns the number of limbs of factor if factorization is successful (non trivial factor is found), else returns 0.

\( \text{max\_iters} \) is the number of iterations tried in process of finding the cycle. If the algorithm fails to find a non trivial factor in one call, it tries again (this time with a different set of random values). This process is repeated a maximum of \( \text{max\_tries} \) times.

The algorithm used is a modification of the original Pollard Rho algorithm, suggested by Richard Brent. It can be found in the paper available at https://maths-people.anu.edu.au/~brent/pd/rpb051i.pdf

4.4.3 Input and output

int fmpz_factor_fprint(FILE *fs, const fmpz_factor_t factor)
int fmpz_factor_print(const fmpz_factor_t factor)
Prints the factorization \( \text{factor} \) into \( \text{fs} \) or stdout. If \( \text{factor} \) is zero, it prints 0. Else, it prints the factorization as \( f_{-1}^{e_{-1}} \times \ldots \times f_{-n}^{e_{-n}} \), where \( f_{-i} \) and \( e_{-i} \) are the \( i \)-th factor and exponent, where \( ^e_{-i} \) is omitted if \( e_i = 1 \). In particular, if \( \text{factor} \) is 1 or \( -1 \), it prints 1 or \( -1 \), respectively.

Returns the number of characters written to file stream.
4.4.4 Elliptic curve (ECM) method

Factoring of _fmpz_ integers using ECM

```c
void _fmpz_factor_ecm_init_(_ecm_t ecm_inf, _mp_limb_t sz)
```
Initializes the _ecm_t_ struct. This is needed in some functions and carries data between subsequent calls.

```c
void _fmpz_factor_ecm_clear_(_ecm_t ecm_inf)
```
Cleans the _ecm_t_ struct.

```c
void _fmpz_factor_ecm_addmod_(_mp_ptr a, _mp_ptr b, _mp_ptr c, _mp_ptr n, _mp_limb_t n_size)
```
Sets a to \((b + c) \mod n\). This is not a normal add mod function, it assumes \(n\) is normalized (highest bit set) and \(b\) and \(c\) are reduced modulo \(n\).

Used for arithmetic operations in _fmpz_factor_ecm_.

```c
void _fmpz_factor_ecm_submod_(_mp_ptr x, _mp_ptr a, _mp_ptr b, _mp_ptr n, _mp_limb_t n_size)
```
Sets \(x\) to \((a - b) \mod n\). This is not a normal subtract mod function, it assumes \(n\) is normalized (highest bit set) and \(b\) and \(c\) are reduced modulo \(n\).

Used for arithmetic operations in _fmpz_factor_ecm_.

```c
void _fmpz_factor_ecm_double_(_mp_ptr x, _mp_ptr z, _mp_ptr x0, _mp_ptr z0, _mp_ptr n, _ecm_t ecm_inf)
```
Sets the point \((x : z)\) to two times \((x_0 : z_0)\) modulo \(n\) according to the formula

\[
x = (x_0 + z_0)^2 \cdot (x_0 - z_0)^2 \mod n,
\]

\[
z = 4x_0z_0 \left((x_0 - z_0)^2 + 4a_2x_0z_0 \right) \mod n.
\]

_ecm_inf_ is used just to use temporary _mp_ptr_’s in the structure. This group doubling is valid only for points expressed in Montgomery projective coordinates.

```c
void _fmpz_factor_ecm_add_(_mp_ptr x, _mp_ptr z, _mp_ptr x1, _mp_ptr z1, _mp_ptr x2, _mp_ptr z2, _mp_ptr x0, _mp_ptr z0, _mp_ptr n, _ecm_t ecm_inf)
```
Sets the point \((x : z)\) to the sum of \((x_1 : z_1)\) and \((x_2 : z_2)\) modulo \(n\), given the difference \((x_0 : z_0)\) according to the formula

\[
x = 4z_0(x_1x_2 - z_1z_2)^2 \mod n,
\]

\[
z = 4x_0(x_2z_1 - x_1z_2)^2 \mod n.
\]

_ecm_inf_ is used just to use temporary _mp_ptr_’s in the structure. This group addition is valid only for points expressed in Montgomery projective coordinates.

```c
int _fmpz_factor_ecm_mul_montgomery_ladder_(_mp_ptr x, _mp_ptr z, _mp_ptr x0, _mp_ptr z0, _mp_limb_t k, _mp_ptr n, _ecm_t ecm_inf)
```
Montgomery ladder algorithm for scalar multiplication of elliptic points.

Sets the point \((x : z)\) to \(k(x_0 : z_0)\) modulo \(n\).

_ecm_inf_ is used just to use temporary _mp_ptr_’s in the structure. Valid only for points expressed in Montgomery projective coordinates.

```c
int _fmpz_factor_ecm_select_curve_(_mp_ptr f, _mp_ptr sigma, _mp_ptr n, _ecm_t ecm_inf)
```
Selects a random elliptic curve given a random integer _sigma_, according to Suyama’s parameterization. If the factor is found while selecting the curve, the number of limbs required to store the factor is returned, otherwise 0.

It could be possible that the selected curve is unsuitable for further computations, in such a case, -1 is returned.

Also selects the initial point \(x_0\), and the value of \((a + 2)/4\), where \(a\) is a curve parameter. Sets \(z_0\) as 1. All these are stored in the _ecm_t_ struct.

4.4. _fmpz_factor.h – integer factorisation
The curve selected is of Montgomery form, the points selected satisfy the curve and are projective coordinates.

```c
int fmpz_factor_ecm_stage_I(mp_ptr f, const mp_limb_t *prime_array, mp_limb_t num,
                           mp_limb_t B1, mp_ptr n, ecm_t ecm_inf)
```

Stage I implementation of the ECM algorithm.

- `f` is set as the factor if found. `num` is number of prime numbers ≤ the bound `B1`. `prime_array` is an array of first `B1` primes. `n` is the number being factored.
- If the factor is found, number of words required to store the factor is returned, otherwise 0.

```c
int fmpz_factor_ecm_stage_II(mp_ptr f, mp_limb_t B1, mp_limb_t B2, mp_limb_t P, mp_ptr n,
                           ecm_t ecm_inf)
```

Stage II implementation of the ECM algorithm.

- `f` is set as the factor if found. `B1`, `B2` are the two bounds. `P` is the primorial (approximately equal to \( \sqrt{B^2} \)). `n` is the number being factored.
- If the factor is found, number of words required to store the factor is returned, otherwise 0.

```c
int fmpz_factor_ecm(fmpz_t f, mp_limb_t curves, mp_limb_t B1, mp_limb_t B2, flint_rand_t state,
                    const fmpz_t n_in)
```

Outer wrapper function for the ECM algorithm. In case `f` can fit in a single unsigned word, a call to `n_factor_ecm` is made.

- The function calls stage I and II, and the precomputations (builds `prime_array` for stage I, `GCD_table` and `prime_table` for stage II).
- `f` is set as the factor if found. `curves` is the number of random curves being tried. `B1`, `B2` are the two bounds or stage I and stage II. `n` is the number being factored.
- If a factor is found in stage I, 1 is returned. If a factor is found in stage II, 2 is returned. If a factor is found while selecting the curve, −1 is returned. Otherwise 0 is returned.

### 4.5 fmpz_mat.h – matrices over the integers

The `fmpz_mat_t` data type represents dense matrices of multiprecision integers, implemented using `fmpz` vectors.

No automatic resizing is performed: in general, the user must provide matrices of correct dimensions for both input and output variables. Output variables are not allowed to be aliased with input variables unless otherwise noted.

- Matrices are indexed from zero: an \( m \times n \) matrix has rows of index 0, 1, \ldots, \( m - 1 \) and columns of index 0, 1, \ldots, \( n - 1 \). One or both of \( m \) and \( n \) may be zero.

- Elements of a matrix can be read or written using the `fmpz_mat_entry` macro, which returns a reference to the entry at a given row and column index. This reference can be passed as an input or output `fmpz_t` variable to any function in the `fmpz` module for direct manipulation.
4.5.1 Simple example

The following example creates the $2 \times 2$ matrix $A$ with value $2i + j$ at row $i$ and column $j$, computes $B = A^2$, and prints both matrices.

```c
#include "fmpz.h"
#include "fmpz_mat.h"

int main()
{
    long i, j;
    fmpz_mat_t A;
    fmpz_mat_t B;
    fmpz_mat_init(A, 2, 2);
    fmpz_mat_init(B, 2, 2);
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            fmpz_set_ui(fmpz_mat_entry(A, i, j), 2*i+j);
    fmpz_mat_mul(B, A, A);
    flint_printf("A = \
    ");
    fmpz_mat_print_pretty(A);
    flint_printf("A^2 = \
    ");
    fmpz_mat_print_pretty(B);
    fmpz_mat_clear(A);
    fmpz_mat_clear(B);
}
```

The output is:

```
A =
[[0 1]
 [2 3]]
A^2 =
[[2 3]
 [6 11]]
```

4.5.2 Types, macros and constants

- **type** `fmpz_mat_struct`
- **type** `fmpz_mat_t`

4.5.3 Memory management

- **void** `fmpz_mat_init(fmpz_mat_t mat, slong rows, slong cols)`
  - Initialises a matrix with the given number of rows and columns for use.
- **void** `fmpz_mat_clear(fmpz_mat_t mat)`
  - Clears the given matrix.
4.5.4 Basic assignment and manipulation

void fmpz_mat_set(fmpz_mat_t mat1, const fmpz_mat_t mat2)
    Sets mat1 to a copy of mat2. The dimensions of mat1 and mat2 must be the same.

void fmpz_mat_init_set(fmpz_mat_t mat, const fmpz_mat_t src)
   Initialises the matrix mat to the same size as src and sets it to a copy of src.

 slong fmpz_mat_nrows(const fmpz_mat_t mat)
 slong fmpz_mat_ncols(const fmpz_mat_t mat)
    Returns respectively the number of rows and columns of the matrix.

void fmpz_mat_swap(fmpz_mat_t mat1, fmpz_mat_t mat2)
    Swaps two matrices. The dimensions of mat1 and mat2 are allowed to be different.

void fmpz_mat_swap_entrywise(fmpz_mat_t mat1, fmpz_mat_t mat2)
    Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

fmpz *fmpz_mat_entry(const fmpz_mat_t mat, slong i, slong j)
    Returns a reference to the entry of mat at row i and column j. This reference can be passed as an input or output variable to any function in the fmpz module for direct manipulation.

    Both i and j must not exceed the dimensions of the matrix.

    This function is implemented as a macro.

void fmpz_mat_zero(fmpz_mat_t mat)
    Sets all entries of mat to 0.

void fmpz_mat_one(fmpz_mat_t mat)
    Sets mat to the unit matrix, having ones on the main diagonal and zeroes elsewhere. If mat is nonsquare, it is set to the truncation of a unit matrix.

void fmpz_mat_swap_rows(fmpz_mat_t mat, slong *perm, slong r, slong s)
    Swaps rows r and s of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

void fmpz_mat_swap_cols(fmpz_mat_t mat, slong *perm, slong r, slong s)
    Swaps columns r and s of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.

void fmpz_mat_invert_rows(fmpz_mat_t mat, slong *perm)
    Swaps rows i and r - i of mat for 0 <= i < r/2, where r is the number of rows of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

void fmpz_mat_invert_cols(fmpz_mat_t mat, slong *perm)
    Swaps columns i and c - i of mat for 0 <= i < c/2, where c is the number of columns of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.

4.5.5 Window

void fmpz_mat_window_init(fmpz_mat_t window, const fmpz_mat_t mat, slong r1, slong c1, slong r2, slong c2)
    Initializes the matrix window to be an r2 - r1 by c2 - c1 submatrix of mat whose (0,0) entry is the (r1, c1) entry of mat. The memory for the elements of window is shared with mat.

void fmpz_mat_window_clear(fmpz_mat_t window)
    Clears the matrix window and releases any memory that it uses. Note that the memory to the underlying matrix that window points to is not freed.
4.5.6 Random matrix generation

void \texttt{fmpz\_mat\_randbits} (\texttt{fmpz\_mat\_t mat, flint\_rand\_t state, flint\_bitcnt\_t bits})

Sets the entries of \texttt{mat} to random signed integers whose absolute values have the given number of binary bits.

void \texttt{fmpz\_mat\_randtest} (\texttt{fmpz\_mat\_t mat, flint\_rand\_t state, flint\_bitcnt\_t bits})

Sets the entries of \texttt{mat} to random signed integers whose absolute values have a random number of bits up to the given number of bits inclusive.

void \texttt{fmpz\_mat\_randintrel} (\texttt{fmpz\_mat\_t mat, flint\_rand\_t state, flint\_bitcnt\_t bits})

Sets \texttt{mat} to be a random integer relations matrix, with signed entries up to the given number of bits.

The number of columns of \texttt{mat} must be equal to one more than the number of rows. The format of the matrix is a set of random integers in the left hand column and an identity matrix in the remaining square submatrix.

void \texttt{fmpz\_mat\_randsimdioph} (\texttt{fmpz\_mat\_t mat, flint\_rand\_t state, flint\_bitcnt\_t bits, flint\_bitcnt\_t bits2})

Sets \texttt{mat} to a random simultaneous diophantine matrix.

The matrix must be square. The top left entry is set to $2^{\text{bits2}}$. The remainder of that row is then set to signed random integers of the given number of binary bits. The remainder of the first column is zero. Running down the rest of the diagonal are the values $2^{\text{bits}}$ with all remaining entries zero.

void \texttt{fmpz\_mat\_randntrulike} (\texttt{fmpz\_mat\_t mat, flint\_rand\_t state, flint\_bitcnt\_t bits, ulong q})

Sets a square matrix \texttt{mat} of even dimension to a random NTRU like matrix.

The matrix is broken into four square submatrices. The top left submatrix is set to the identity. The bottom left submatrix is set to the zero matrix. The bottom right submatrix is set to \texttt{q} times the identity matrix. Finally the top right submatrix has the following format. A random vector \texttt{h} of length \texttt{r/2} is created, with random signed entries of the given number of bits. Then entry \((i, j)\) of the submatrix is set to \(h[i + j \mod \text{r/2}]\).

void \texttt{fmpz\_mat\_randntrulike2} (\texttt{fmpz\_mat\_t mat, flint\_rand\_t state, flint\_bitcnt\_t bits, ulong q})

Sets a square matrix \texttt{mat} of even dimension to a random NTRU like matrix.

The matrix is broken into four square submatrices. The top left submatrix is set to \(q\) times the identity matrix. The top right submatrix is set to the zero matrix. The bottom right submatrix is set to the identity matrix. Finally the bottom left submatrix has the following format. A random vector \texttt{h} of length \texttt{r/2} is created, with random signed entries of the given number of bits. Then entry \((i, j)\) of the submatrix is set to \(h[i + j \mod \text{r/2}]\).

void \texttt{fmpz\_mat\_randajtai} (\texttt{fmpz\_mat\_t mat, flint\_rand\_t state, double alpha})

Sets a square matrix \texttt{mat} to a random ajtai matrix. The diagonal entries \((i, i)\) are set to a random entry in the range \([1, 2^b - 1]\) inclusive where \(b = \lceil (2r - i)^{\alpha} \rceil\) for some double parameter \(\alpha\). The entries below the diagonal in column \(i\) are set to a random entry in the range \((-2^b + 1, 2^b - 1)\) whilst the entries to the right of the diagonal in row \(i\) are set to zero.

int \texttt{fmpz\_mat\_randperdiag} (\texttt{fmpz\_mat\_t mat, flint\_rand\_t state, const fmpz \*diag, slong n})

Sets \texttt{mat} to a random permutation of the rows and columns of a given diagonal matrix. The diagonal matrix is specified in the form of an array of the \(n\) initial entries on the main diagonal.

The return value is 0 or 1 depending on whether the permutation is even or odd.

void \texttt{fmpz\_mat\_randrank} (\texttt{fmpz\_mat\_t mat, flint\_rand\_t state, slong rank, flint\_bitcnt\_t bits})

Sets \texttt{mat} to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the nonzero elements being random integers of the given bit size.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling \texttt{fmpz\_mat\_randops()}.
void fmpz_mat_randdet(fmpz_mat_t mat, flint_rand_t state, const fmpz_t det)
    Sets mat to a random sparse matrix with minimal number of nonzero entries such that its determinant has the given value.

    Note that the matrix will be zero if det is zero. In order to generate a non-zero singular matrix, the function fmpz_mat_randrank() can be used.

    The matrix can be transformed into a dense matrix with unchanged determinant by subsequently calling fmpz_mat_randops().

void fmpz_mat_randops(fmpz_mat_t mat, flint_rand_t state, slong count)
    Randomises mat by performing elementary row or column operations. More precisely, at most count random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, the determinant) unchanged.

4.5.7 Input and output

int fmpz_mat_fprint(FILE *file, const fmpz_mat_t mat)
    Prints the given matrix to the stream file. The format is the number of rows, a space, the number of columns, two spaces, then a space separated list of coefficients, one row after the other.

    In case of success, returns a positive value; otherwise, returns a non-positive value.

int fmpz_mat_fprint_pretty(FILE *file, const fmpz_mat_t mat)
    Prints the given matrix to the stream file. The format is an opening square bracket, then on each line a row of the matrix, followed by a closing square bracket. Each row is written as an opening square bracket followed by a space separated list of coefficients followed by a closing square bracket.

    In case of success, returns a positive value; otherwise, returns a non-positive value.

int fmpz_mat_print(const fmpz_mat_t mat)
    Prints the given matrix to the stream stdout. For further details, see fmpz_mat_fprint().

int fmpz_mat_print_pretty(const fmpz_mat_t mat)
    Prints the given matrix to stdout. For further details, see fmpz_mat_fprint_pretty().

int fmpz_mat_fread(FILE *file, fmpz_mat_t mat)
    Reads a matrix from the stream file, storing the result in mat. The expected format is the number of rows, a space, the number of columns, two spaces, then a space separated list of coefficients, one row after the other.

    In case of success, returns a positive number. In case of failure, returns a non-positive value.

int fmpz_mat_read(fmpz_mat_t mat)
    Reads a matrix from stdin, storing the result in mat.

    In case of success, returns a positive number. In case of failure, returns a non-positive value.

4.5.8 Comparison

int fmpz_mat_equal(const fmpz_mat_t mat1, const fmpz_mat_t mat2)
    Returns a non-zero value if mat1 and mat2 have the same dimensions and entries, and zero otherwise.

int fmpz_mat_is_zero(const fmpz_mat_t mat)
    Returns a non-zero value if all entries mat are zero, and otherwise returns zero.

int fmpz_mat_is_one(const fmpz_mat_t mat)
    Returns a non-zero value if mat is the unit matrix or the truncation of a unit matrix, and otherwise returns zero.
int fmpz_mat_is_empty(const fmpz_mat_t mat)
Returns a non-zero value if the number of rows or the number of columns in mat is zero, and otherwise returns zero.

int fmpz_mat_is_square(const fmpz_mat_t mat)
Returns a non-zero value if the number of rows is equal to the number of columns in mat, and otherwise returns zero.

int fmpz_mat_is_zero_row(const fmpz_mat_t mat, slong i)
Returns a non-zero value if row i of mat is zero.

int fmpz_mat_equal_col(fmpz_mat_t M, slong m, slong n)
Returns 1 if columns m and n of the matrix M are equal, otherwise returns 0.

int fmpz_mat_equal_row(fmpz_mat_t M, slong m, slong n)
Returns 1 if rows m and n of the matrix M are equal, otherwise returns 0.

4.5.9 Transpose

void fmpz_mat_transpose(fmpz_mat_t B, const fmpz_mat_t A)
Sets B to A\(T\), the transpose of A. Dimensions must be compatible. A and B are allowed to be the same object if A is a square matrix.

4.5.10 Concatenate

void fmpz_mat_concat_vertical(fmpz_mat_t res, const fmpz_mat_t mat1, const fmpz_mat_t mat2)
Sets res to vertical concatenation of (mat1, mat2) in that order. Matrix dimensions: mat1: \(m \times n\), mat2: \(k \times n\), res: \((m + k) \times n\).

void fmpz_mat_concat_horizontal(fmpz_mat_t res, const fmpz_mat_t mat1, const fmpz_mat_t mat2)
Sets res to horizontal concatenation of (mat1, mat2) in that order. Matrix dimensions: mat1: \(m \times n\), mat2: \(m \times k\), res: \(m \times (n + k)\).

4.5.11 Modular reduction and reconstruction

void fmpz_mat_get_nmod_mat(nmod_mat_t Amod, const fmpz_mat_t A)
Sets the entries of Amod to the entries of A reduced by the modulus of Amod.

void fmpz_mat_set_nmod_mat(fmpz_mat_t A, const nmod_mat_t Amod)
Sets the entries of Amod to the residues in Amod, normalised to the interval \(-m/2 <= r < m/2\) where \(m\) is the modulus.

void fmpz_mat_set_nmod_mat_unsigned(fmpz_mat_t A, const nmod_mat_t Amod)
Sets the entries of Amod to the residues in Amod, normalised to the interval \(0 <= r < m\) where \(m\) is the modulus.

void fmpz_mat_CRT_ui(fmpz_mat_t res, const fmpz_mat_t mat1, const fmpz_t m1, const nmod_mat_t mat2, int sign)
Given mat1 with entries modulo m and mat2 with modulus n, sets res to the CRT reconstruction modulo \(mn\) with entries satisfying \(-mn/2 <= c < mn/2\) (if sign = 1) or \(0 <= c < mn\) (if sign = 0).
void fmpz_mat_multi_mod_ui_precomp(nmod_mat_t *residues, slong nres, const fmpz_mat_t mat, const fmpz_comb_t comb, fmpz_comb_temp_t temp)

Sets each of the nres matrices in residues to mat reduced modulo the modulus of the respective matrix, given precomputed comb and comb_temp structures.

Note: fmpz.h must be included before fmpz_mat.h in order for this function to be declared.

void fmpz_mat_multi_mod_ui(nmod_mat_t *residues, slong nres, const fmpz_mat_t mat)

Sets each of the nres matrices in residues to mat reduced modulo the modulus of the respective matrix.

This function is provided for convenience purposes. For reducing or reconstructing multiple integer matrices over the same set of moduli, it is faster to use fmpz_mat_multi_mod_precomp.

void fmpz_mat_multi_CRT_ui_precomp(fmpz_mat_t mat, nmod_mat_t *const residues, slong nres, const fmpz_comb_t comb, fmpz_comb_temp_t temp, int sign)

Reconstructs mat from its images modulo the nres matrices in residues, given precomputed comb and comb_temp structures.

Note: fmpz.h must be included before fmpz_mat.h in order for this function to be declared.

void fmpz_mat_multi_CRT_ui(fmpz_mat_t mat, nmod_mat_t *const residues, slong nres, int sign)

Reconstructs mat from its images modulo the nres matrices in residues.

This function is provided for convenience purposes. For reducing or reconstructing multiple integer matrices over the same set of moduli, it is faster to use fmpz_mat_multi_CRT_ui_precomp().

4.5.12 Addition and subtraction

void fmpz_mat_add(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)

Sets C to the elementwise sum \( A + B \). All inputs must be of the same size. Aliasing is allowed.

void fmpz_mat_sub(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)

Sets C to the elementwise difference \( A - B \). All inputs must be of the same size. Aliasing is allowed.

void fmpz_mat_neg(fmpz_mat_t B, const fmpz_mat_t A)

Sets B to the elementwise negation of A. Both inputs must be of the same size. Aliasing is allowed.

4.5.13 Matrix-scalar arithmetic

void fmpz_mat_scalar_mul_si(fmpz_mat_t B, const fmpz_mat_t A, slong c)

void fmpz_mat_scalar_mul_ui(fmpz_mat_t B, const fmpz_mat_t A, ulong c)

void fmpz_mat_scalar_mul_fmpz(fmpz_mat_t B, const fmpz_mat_t A, const fmpz_t c)

Set B = A*c where A is an fmpz_mat_t and c is a scalar respectively of type slong, ulong, or fmpz_t. The dimensions of A and B must be compatible.

void fmpz_mat_scalar_addmul_si(fmpz_mat_t B, const fmpz_mat_t A, slong c)

void fmpz_mat_scalar_addmul_ui(fmpz_mat_t B, const fmpz_mat_t A, ulong c)

void fmpz_mat_scalar_addmul_fmpz(fmpz_mat_t B, const fmpz_mat_t A, const fmpz_t c)

Set B = A + A*c where A is an fmpz_mat_t and c is a scalar respectively of type slong, ulong, or fmpz_t. The dimensions of A and B must be compatible.

void fmpz_mat_scalar_submul_si(fmpz_mat_t B, const fmpz_mat_t A, slong c)

void fmpz_mat_scalar_submul_ui(fmpz_mat_t B, const fmpz_mat_t A, ulong c)
void fmpz_mat_scalar_submul_fmpz(fmpz_mat_t B, const fmpz_mat_t A, const fmpz_t c)
    Set \( B = B - A \cdot c \) where \( A \) is an fmpz_mat_t and \( c \) is a scalar respectively of type slong, ulong, or fmpz_t. The dimensions of \( A \) and \( B \) must be compatible.

void fmpz_mat_scalar_addmul_nmod_mat_ui(fmpz_mat_t B, const nmod_mat_t A, ulong c)
void fmpz_mat_scalar_addmul_nmod_mat_fmpz(fmpz_mat_t B, const nmod_mat_t A, const fmpz_t c)
    Set \( B = B + A \cdot c \) where \( A \) is an nmod_mat_t and \( c \) is a scalar respectively of type ulong or fmpz_t. The dimensions of \( A \) and \( B \) must be compatible.

void fmpz_mat_scalar_divexact_si(fmpz_mat_t B, const fmpz_mat_t A, slong c)
void fmpz_mat_scalar_divexact_ui(fmpz_mat_t B, const fmpz_mat_t A, ulong c)
void fmpz_mat_scalar_divexact_fmpz(fmpz_mat_t B, const fmpz_mat_t A, const fmpz_t c)
    Set \( A = B / c \), where \( B \) is an fmpz_mat_t and \( c \) is a scalar respectively of type slong, ulong, or fmpz_t, which is assumed to divide all elements of \( B \) exactly.

void fmpz_mat_scalar_mult_2exp(fmpz_mat_t B, const fmpz_mat_t A, ulong exp)
    Set the matrix \( B \) to the matrix \( A \), of the same dimensions, multiplied by \( 2^{\text{exp}} \).

void fmpz_mat_scalar_tdiv_q_2exp(fmpz_mat_t B, const fmpz_mat_t A, ulong exp)
    Set the matrix \( B \) to the matrix \( A \), of the same dimensions, divided by \( 2^{\text{exp}} \), rounding down towards zero.

void fmpz_mat_scalar_smod(fmpz_mat_t B, const fmpz_mat_t A, const fmpz_t P)
    Set the matrix \( B \) to the matrix \( A \), of the same dimensions, with each entry reduced modulo \( P \) in the symmetric moduli system. We require \( P > 0 \).

### 4.5.14 Matrix multiplication

void fmpz_mat_mul(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
    Sets \( C \) to the matrix product \( C = AB \). The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

This function automatically switches between classical and multimodular multiplication, based on a heuristic comparison of the dimensions and entry sizes.

void fmpz_mat_mul_classical(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
    Sets \( C \) to the matrix product \( C = AB \) computed using classical matrix algorithm.

The matrices must have compatible dimensions for matrix multiplication. No aliasing is allowed.

void fmpz_mat_mul_waksman(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
    Sets \( C \) to the matrix product \( C = AB \) computed using Waksman multiplication, which does only \( n^3/2 + O(n^2) \) products, but many additions. This is good for small matrices with large entries.

The matrices must have compatible dimensions for matrix multiplication. No aliasing is allowed.

void fmpz_mat_mul_strassen(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
    Sets \( C = AB \). Dimensions must be compatible for matrix multiplication. \( C \) is not allowed to be aliased with \( A \) or \( B \). Uses Strassen multiplication (the Strassen-Winograd variant).

void fmpz_mat_mul_multi_mod(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B, int sign, flint_bitcnt_t bits)
void fmpz_mat_mul_multi_mod(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
    Sets \( C \) to the matrix product \( C = AB \) computed using a multimodular algorithm. \( C \) is computed modulo several small prime numbers and reconstructed using the Chinese Remainder Theorem. This generally becomes more efficient than classical multiplication for large matrices.

The absolute value of the elements of \( C \) should be \(< 2^{\text{bits}}\), and sign should be 0 if the entries of \( C \) are known to be nonnegative and 1 otherwise. The function fmpz_mat_mul_multi_mod()
calculates a rigorous bound automatically. If the default bound is too pessimistic, 
\_fmpz\_mat\_mul\_multi\_mod() can be used with a custom bound.

The matrices must have compatible dimensions for matrix multiplication. No aliasing is allowed.

\begin{verbatim}
int fmpz_mat_mul_blas(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
Tries to set \( C = AB \) using BLAS and returns 1 for success and 0 for failure. Dimensions must be compatible for matrix multiplication. No aliasing is allowed. This function currently will fail if the matrices are empty, their dimensions are too large, or their max bits size is over one million bits.

void fmpz_mat_mul_fft(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
Aliasing is allowed.

void fmpz_mat_sqr(fmpz_mat_t B, const fmpz_mat_t A)
Sets \( B \) to the square of the matrix \( A \), which must be a square matrix. Aliasing is allowed. The function calls fmpz_mat_mul() for dimensions less than 12 and calls fmpz_mat_sqr_bodrato() for cases in which the latter is faster.

void fmpz_mat_sqr_bodrato(fmpz_mat_t B, const fmpz_mat_t A)
Sets \( B \) to the square of the matrix \( A \), which must be a square matrix. Aliasing is allowed. The Bodrato algorithm is described in [Bodrato2010]. It is highly efficient for squaring matrices which satisfy both the following conditions: (a) large elements, (b) dimensions less than 150.

void fmpz_mat_pow(fmpz_mat_t B, const fmpz_mat_t A, ulong e)
Sets \( B \) to the matrix \( A \) raised to the power \( e \), where \( A \) must be a square matrix. Aliasing is allowed.

void _fmpz_mat_mul_small(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
This internal function sets \( C \) to the matrix product \( C = AB \) computed using classical matrix algorithm assuming that all entries of \( A \) and \( B \) are small, that is, have bits \( \leq FLINT\_BITS - 2 \). No aliasing is allowed.

void _fmpz_mat_mul_double_word(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
This function is only for internal use and assumes that either:

- the entries of \( A \) and \( B \) are all nonnegative and strictly less than \( 2^{2*FLINT\_BITS} \), or
- the entries of \( A \) and \( B \) are all strictly less than \( 2^{2*FLINT\_BITS-1} \) in absolute value.

void fmpz_mat_mul_fmpz_vec(fmpz *c, const fmpz_mat_t A, const fmpz *b, slong blen)
void fmpz_mat_mul_fmpz_vec_ptr(fmpz *const *c, const fmpz_mat_t A, const fmpz *const *b, slong blen)
Compute a matrix-vector product of \( A \) and \( (b, blen) \) and store the result in \( c \). The vector \( (b, blen) \) is either truncated or zero-extended to the number of columns of \( A \). The number of entries written to \( c \) is always equal to the number of rows of \( A \).

void fmpz_mat_fmpz_vec_mul(fmpz *c, const fmpz *a, slong alen, const fmpz_mat_t B)
void fmpz_mat_fmpz_vec_mul_ptr(fmpz *const *c, const fmpz *const *a, slong alen, const fmpz_mat_t B)
Compute a vector-matrix product of \( (a, alen) \) and \( B \) and store the result in \( c \). The vector \( (a, alen) \) is either truncated or zero-extended to the number of rows of \( B \). The number of entries written to \( c \) is always equal to the number of columns of \( B \).
\end{verbatim}
4.5.15 Inverse

```c
int fmpz_mat_inv(fmpz_mat_t Ainv, fmpz_t den, const fmpz_mat_t A)
```
Sets \((A_{\text{inv}}, \text{den})\) to the inverse matrix of \(A\). Returns 1 if \(A\) is nonsingular and 0 if \(A\) is singular. Aliasing of \(A_{\text{inv}}\) and \(A\) is allowed.

The denominator is not guaranteed to be minimal, but is guaranteed to be a divisor of the determinant of \(A\).

This function uses a direct formula for matrices of size two or less, and otherwise solves for the identity matrix using fraction-free LU decomposition.

4.5.16 Kronecker product

```c
void fmpz_mat_kronecker_product(fmpz_mat_t C, const fmpz_mat_t A, const fmpz_mat_t B)
```
Sets \(C\) to the Kronecker product of \(A\) and \(B\).

4.5.17 Content

```c
void fmpz_mat_content(fmpz_t mat_gcd, const fmpz_mat_t A)
```
Sets \(\text{gcd}\) as the gcd of all the elements of the matrix \(A\). Returns 0 if the matrix is empty.

4.5.18 Trace

```c
void fmpz_mat_trace(fmpz_t trace, const fmpz_mat_t mat)
```
Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

4.5.19 Determinant

```c
void fmpz_mat_det(fmpz_t det, const fmpz_mat_t A)
```
Sets \(\text{det}\) to the determinant of the square matrix \(A\). The matrix of dimension 0 \(\times\) 0 is defined to have determinant 1.

This function automatically chooses between \(\text{fmpz_mat_det_cofactor()}\), \(\text{fmpz_mat_det_bareiss()}\), \(\text{fmpz_mat_det_modular()}\) and \(\text{fmpz_mat_det_modular_accelerated()}\) (with \(\text{proved} = 1\)), depending on the size of the matrix and its entries.

```c
void fmpz_mat_det_cofactor(fmpz_t det, const fmpz_mat_t A)
```
Sets \(\text{det}\) to the determinant of the square matrix \(A\) computed using direct cofactor expansion. This function only supports matrices up to size \(4 \times 4\).

```c
void fmpz_mat_det_bareiss(fmpz_t det, const fmpz_mat_t A)
```
Sets \(\text{det}\) to the determinant of the square matrix \(A\) computed using the Bareiss algorithm. A copy of the input matrix is row reduced using fraction-free Gaussian elimination, and the determinant is read off from the last element on the main diagonal.

```c
void fmpz_mat_det_modular(fmpz_t det, const fmpz_mat_t A, int proved)
```
Sets \(\text{det}\) to the determinant of the square matrix \(A\) (if \(\text{proved} = 1\)), or a probabilistic value for the determinant (\(\text{proved} = 0\)), computed using a multimodular algorithm.

The determinant is computed modulo several small primes and reconstructed using the Chinese Remainder Theorem. With \(\text{proved} = 1\), sufficiently many primes are chosen to satisfy the bound computed by \(\text{fmpz_mat_det_bound()}\). With \(\text{proved} = 0\), the determinant is considered determined if it remains unchanged modulo several consecutive primes (currently if their product exceeds \(2^{100}\)).
void fmpz_mat_det_modular_accelerated(fmpz_t det, const fmpz_mat_t A, int proved)

Sets \( \det \) to the determinant of the square matrix \( A \) (if \( \text{proved} = 1 \)), or a probabilistic value for the determinant (\( \text{proved} = 0 \)), computed using a multimodular algorithm.

This function uses the same basic algorithm as \texttt{fmpz_mat_det_modular}, but instead of computing \( \det(A) \) directly, it generates a divisor \( d \) of \( \det(A) \) and then computes \( x = \det(A)/d \) modulo several small primes not dividing \( d \). This typically accelerates the computation by requiring fewer primes for large matrices, since \( d \) with high probability will be nearly as large as the determinant. This trick is described in [AbbottBronsteinMulders1999].

void fmpz_mat_det_modular_given_divisor(fmpz_t det, const fmpz_mat_t A, const fmpz_t d, int proved)

Given a positive divisor \( d \) of \( \det(A) \), sets \( \det \) to the determinant of the square matrix \( A \) (if \( \text{proved} = 1 \)), or a probabilistic value for the determinant (\( \text{proved} = 0 \)), computed using a multimodular algorithm.

void fmpz_mat_det_bound(fmpz_t bound, const fmpz_mat_t A)

Sets \( \text{bound} \) to a nonnegative integer \( B \) such that \( |\det(A)| \leq B \). Assumes \( A \) to be a square matrix. The bound is computed from the Hadamard inequality \( |\det(A)| \leq \prod ||a_i||_2 \) where the product is taken over the rows \( a_i \) of \( A \).

void fmpz_mat_det_bound_nonzero(fmpz_t bound, const fmpz_mat_t A)

As per \texttt{fmpz_mat_det_bound()} but excludes zero columns. For use with non-square matrices.

void fmpz_mat_det_divisor(fmpz_t d, const fmpz_mat_t A)

Sets \( d \) to some positive divisor of the determinant of the given square matrix \( A \), if the determinant is nonzero. If \( |\det(A)| = 0 \), \( d \) will always be set to zero.

A divisor is obtained by solving \( Ax = b \) for an arbitrarily chosen right-hand side \( b \) using Dixon’s algorithm and computing the least common multiple of the denominators in \( x \). This yields a divisor \( d \) such that \( |\det(A)|/d \) is tiny with very high probability.

4.5.20 Transforms

void fmpz_mat_similarity(fmpz_mat_t A, slong r, fmpz_t d)

Applies a similarity transform to the \( n \times n \) matrix \( M \) in-place.

If \( P \) is the \( n \times n \) identity matrix the zero entries of whose row \( r \) (0-indexed) have been replaced by \( d \), this transform is equivalent to \( M = P^{-1}MP \).

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

4.5.21 Characteristic polynomial

void _fmpz_mat_charpoly_berkowitz(fmpz *cp, const fmpz_mat_t mat)

Sets \( (cp, n+1) \) to the characteristic polynomial of an \( n \times n \) square matrix.

void fmpz_mat_charpoly_berkowitz(fmpz_poly_t cp, const fmpz_mat_t mat)

Computes the characteristic polynomial of length \( n+1 \) of an \( n \times n \) square matrix. Uses an \( O(n^4) \) algorithm based on the method of Berkowitz.

void _fmpz_mat_charpoly_modular(fmpz *cp, const fmpz_mat_t mat)

Sets \( (cp, n+1) \) to the characteristic polynomial of an \( n \times n \) square matrix.

void fmpz_mat_charpoly_modular(fmpz_poly_t cp, const fmpz_mat_t mat)

Computes the characteristic polynomial of length \( n+1 \) of an \( n \times n \) square matrix. Uses a modular method based on an \( O(n^3) \) method over \( \mathbb{Z}/n\mathbb{Z} \).
void _fmpz_mat_charpoly(fmpz *cp, const fmpz_mat_t mat)
    Sets (cp, n+1) to the characteristic polynomial of an \( n \times n \) square matrix.
void fmpz_mat_charpoly(fmpz_poly_t cp, const fmpz_mat_t mat)
    Computes the characteristic polynomial of length \( n + 1 \) of an \( n \times n \) square matrix.

4.5.22 Minimal polynomial

slong _fmpz_mat_minpoly_modular(fmpz *cp, const fmpz_mat_t mat)
    Sets (cp, n+1) to the modular polynomial of an \( n \times n \) square matrix and returns its length.
void fmpz_mat_minpoly_modular(fmpz_poly_t cp, const fmpz_mat_t mat)
    Computes the minimal polynomial of an \( n \times n \) square matrix. Uses a modular method based on
    an average time \( O(n^3) \), worst case \( O(n^4) \) method over \( \mathbb{Z}/n\mathbb{Z} \).
slong _fmpz_mat_minpoly(fmpz *cp, const fmpz_mat_t mat)
    Sets cp to the minimal polynomial of an \( n \times n \) square matrix and returns its length.
void fmpz_mat_minpoly(fmpz_poly_t cp, const fmpz_mat_t mat)
    Computes the minimal polynomial of an \( n \times n \) square matrix.

4.5.23 Rank

slong fmpz_mat_rank(const fmpz_mat_t A)
    Returns the rank, that is, the number of linearly independent columns (equivalently, rows), of \( A \).
    The rank is computed by row reducing a copy of \( A \).

4.5.24 Column partitioning

int fmpz_mat_col_partition(slong *part, fmpz_mat_t M, int short_circuit)
    Returns the number \( p \) of distinct columns of \( M \) (or 0 if the flag short_circuit is set and this
    number is greater than the number of rows of \( M \)). The entries of array part are set to values in
    \([0, p]\) such that two entries of part are equal iff the corresponding columns of \( M \) are equal. This
    function is used in van Hoeij polynomial factoring.

4.5.25 Nonsingular solving

The following functions allow solving matrix-matrix equations \( AX = B \) where the system matrix \( A \)
    is square and has full rank. The solving is implicitly done over the field of rational numbers: except
    where otherwise noted, an integer matrix \( \hat{X} \) and a separate denominator \( d \) (den) are computed such that
    \( A(\hat{X}/d) = b \), equivalently such that \( AX = bd \) holds over the integers. No guarantee is made that the
    numerators and denominator are reduced to lowest terms, but the denominator is always guaranteed to
    be a divisor of the determinant of \( A \). If \( A \) is singular, den will be set to zero and the elements of the
    solution vector or matrix will have undefined values. No aliasing is allowed between arguments.
int fmpz_mat_solve(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
    Solves the equation \( AX = B \) for nonsingular \( A \). More precisely, computes \( (X, den) \) such that
    \( AX = B \times den \). Returns 1 if \( A \) is nonsingular and 0 if \( A \) is singular. The computed denominator
    will not generally be minimal.
This function uses Cramer’s rule for small systems and fraction-free LU decomposition followed by
    fraction-free forward and back substitution for larger systems.
    Note that for very large systems, it is faster to compute a modular solution using
    fmpz_mat_solve_dixon.

4.5. fmpz_mat.h – matrices over the integers
int _fmpz_mat_solve_dixon_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
Solves the equation $AX = B$ for nonsingular $A$. More precisely, computes $(X, \text{den})$ such that $AX = B \times \text{den}$. Returns 1 if $A$ is nonsingular and 0 if $A$ is singular. The computed denominator will not generally be minimal.

Uses the Dixon lifting algorithm with early termination once the lifting stabilises.

int fmpz_mat_solve_dixon_den(fmpz_mat_t X, const slong *perm, const fmpz_mat_t FFLU,
const fmpz_mat_t B)
Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation. If no impossible division is encountered, the function returns 1. This does not mean the system has a solution, however a return value of 0 can only occur if the system is insoluble.

If the return value is 1 and $r$ is the rank of the matrix $A$ whose FFLU we have, then the first $r$ rows of $p(A)y = p(b)d$ hold, where $d$ is the denominator of the FFLU. The remaining rows must be checked by the caller.

int fmpz_mat_solve_cramer(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
Solves the equation $AX = B$ for nonsingular $A$. More precisely, computes $(X, \text{den})$ such that $AX = B \times \text{den}$. Returns 1 if $A$ is nonsingular and 0 if $A$ is singular.

Uses Cramer’s rule. Only systems of size up to $3 \times 3$ are allowed.

void _fmpz_mat_solve_dixon(fmpz_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
Solves $AX = B$ given a nonsingular square matrix $A$ and a matrix $B$ of compatible dimensions, using a modular algorithm. In particular, Dixon’s $p$-adic lifting algorithm is used (currently a non-adaptive version). This is generally the preferred method for large dimensions.

More precisely, this function computes an integer $M$ and an integer matrix $X$ such that $AX = B \mod M$ and such that all the reduced numerators and denominators $n/d$ in $A^{-1}B$ satisfy the bounds $0 \leq |n| \leq N$ and $0 \leq d \leq D$.

int fmpz_mat_solve_fflu(fmpz_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
Solves the equation $AX = B$ for nonsingular $A$. More precisely, computes $(X, \text{den})$ such that $AX = B \times \text{den}$. Returns 1 if $A$ is nonsingular and 0 if $A$ is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

int fmpz_mat_solve_fflu_precomp(fmpz_mat_t X, const slong *perm, const fmpz_mat_t FFLU,
const fmpz_mat_t B)
Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation. If no impossible division is encountered, the function returns 1. This does not mean the system has a solution, however a return value of 0 can only occur if the system is insoluble.

If the return value is 1 and $r$ is the rank of the matrix $A$ whose FFLU we have, then the first $r$ rows of $p(A)y = p(b)d$ hold, where $d$ is the denominator of the FFLU. The remaining rows must be checked by the caller.

int fmpz_mat_solve_multi_mod_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
Solves the equation $AX = B$ for nonsingular $A$. More precisely, computes $(X, \text{den})$ such that $AX = B \times \text{den}$. Returns 1 if $A$ is nonsingular and 0 if $A$ is singular. The computed denominator will not generally be minimal.

Uses the Dixon lifting algorithm with early termination once the lifting stabilises.

int fmpz_mat_solve_bound(fmpz_t N, fmpz_t D, const fmpz_mat_t A, const fmpz_mat_t B)
Assuming that $A$ is nonsingular, computes integers $N$ and $D$ such that the reduced numerators and denominators $n/d$ in $A^{-1}B$ satisfy the bounds $0 \leq |n| \leq N$ and $0 \leq d \leq D$.

int fmpz_mat_solve_dixon(fmpz_mat_t X, fmpz_t M, const fmpz_mat_t A, const fmpz_mat_t B)
Solves $AX = B$ given a nonsingular square matrix $A$ and a matrix $B$ of compatible dimensions, using a modular algorithm. In particular, Dixon’s $p$-adic lifting algorithm is used (currently a non-adaptive version). This is generally the preferred method for large dimensions.

More precisely, this function computes an integer $M$ and an integer matrix $X$ such that $AX = B \mod M$ and such that all the reduced numerators and denominators of the elements $x = p/q$ in the full solution satisfy $2|p|q < M$. As such, the explicit rational solution matrix can be recovered uniquely by passing the output of this function to fmpz_mat_set_fmpz_mat_mod.

A nonzero value is returned if $A$ is nonsingular. If $A$ is singular, zero is returned and the values of the output variables will be undefined.

Aliasing between input and output matrices is allowed.

void _fmpz_mat_solve_dixon_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B, const nmod_mat_t Ainv, mp_limb_t p, const fmpz_t N, const fmpz_t D)
Solves the equation $AX = B$ for nonsingular $A$. More precisely, computes $(X, \text{den})$ such that $AX = B \times \text{den}$ using a $p$-adic algorithm for the supplied prime $p$. The values $N$ and $D$ are absolute value bounds for the numerator and denominator of the solution.

Uses the Dixon lifting algorithm with early termination once the lifting stabilises.

int fmpz_mat_solve_dixon_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
Solves the equation $AX = B$ for nonsingular $A$. More precisely, computes $(X, \text{den})$ such that $AX = B \times \text{den}$. Returns 1 if $A$ is nonsingular and 0 if $A$ is singular. The computed denominator will not generally be minimal.

Uses the Dixon lifting algorithm with early termination once the lifting stabilises.

int fmpz_mat_solve_multi_mod_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
Solves the equation $AX = B$ for nonsingular $A$. More precisely, computes $(X, \text{den})$ such that $AX = B \times \text{den}$. Returns 1 if $A$ is nonsingular and 0 if $A$ is singular. The computed denominator will not generally be minimal.

Uses a Chinese remainder algorithm with early termination once the lifting stabilises.

```c
int fmpz_mat_can_solve_multi_mod_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
```

Returns 1 if the system $AX = B$ can be solved. If so it computes $(X, \text{den})$ such that $AX = B \times \text{den}$. The computed denominator will not generally be minimal.

Uses a Chinese remainder algorithm.

Note that the matrices $A$ and $B$ may have any shape as long as they have the same number of rows.

```c
int fmpz_mat_can_solve_fflu(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
```

Returns 1 if the system $AX = B$ can be solved. If so it computes $(X, \text{den})$ such that $AX = B \times \text{den}$. The computed denominator will not generally be minimal.

Uses a fraction free LU decomposition algorithm.

Note that the matrices $A$ and $B$ may have any shape as long as they have the same number of rows.

```c
int fmpz_mat_can_solve(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
```

Returns 1 if the system $AX = B$ can be solved. If so it computes $(X, \text{den})$ such that $AX = B \times \text{den}$. The computed denominator will not generally be minimal.

Note that the matrices $A$ and $B$ may have any shape as long as they have the same number of rows.

### 4.5.26 Row reduction

```c
slong fmpz_mat_find_pivot_any(const fmpz_mat_t mat, slong start_row, slong end_row, slong c)
```

Attempts to find a pivot entry for row reduction. Returns a row index $r$ between `start_row` (inclusive) and `stop_row` (exclusive) such that column $c$ in `mat` has a nonzero entry on row $r$, or returns -1 if no such entry exists.

This implementation simply chooses the first nonzero entry it encounters. This is likely to be a nearly optimal choice if all entries in the matrix have roughly the same size, but can lead to unnecessary coefficient growth if the entries vary in size.

```c
slong fmpz_mat_fflu(fmpz_mat_t B, fmpz_t den, slong *perm, const fmpz_mat_t A, int rank_check)
```

Uses fraction-free Gaussian elimination to set $(B, \text{den})$ to a fraction-free LU decomposition of $A$ and returns the rank of $A$. Aliasing of $A$ and $B$ is allowed.

Pivot elements are chosen with `fmpz_mat_find_pivot_any`. If `perm` is non-NULL, the permutation of rows in the matrix will also be applied to `perm`.

If `rank_check` is set, the function aborts and returns 0 if the matrix is detected not to have full rank without completing the elimination.

The denominator `den` is set to $\pm \det(S)$ where $S$ is an appropriate submatrix of $A$ ($S = A$ if $A$ is square) and the sign is decided by the parity of the permutation. Note that the determinant is not generally the minimal denominator.

The fraction-free LU decomposition is defined in [NakTurWil1997].
sets (B, den) to the reduced row echelon form of A and returns the rank of A. Aliasing of A and B is allowed.

The algorithm used chooses between `fmpz_mat_rref_fflu` and `fmpz_mat_rref_mul` based on the dimensions of the input matrix.

Sets (B, den) to the reduced row echelon form of A and returns the rank of A. Aliasing of A and B is allowed.

The algorithm proceeds by first computing a row echelon form using `fmpz_mat_fflu`. Letting the upper part of this matrix be \((U|V)P\) where \(U\) is full rank upper triangular and \(P\) is a permutation matrix, we obtain the rref by setting \(V\) to \(U^{-1}V\) using back substitution. Scaling each completed row in the back substitution to the denominator \(den\), we avoid introducing new fractions. This strategy is equivalent to the fraction-free Gauss-Jordan elimination in [NakTurWil1997], but faster since only the part \(V\) corresponding to the null space has to be updated.

The denominator \(den\) is set to \(±\det(S)\) where \(S\) is an appropriate submatrix of \(A\) (\(S = A\) if \(A\) is square). Note that the determinant is not generally the minimal denominator.

Sets (B, den) to the reduced row echelon form of A and returns the rank of A. Aliasing of A and B is allowed.

The algorithm works by computing the reduced row echelon form of \(A\) modulo a prime \(p\) using `nmod_mat_rref`. The pivot columns and rows of this matrix will then define a non-singular submatrix of \(A\), nonsingular solving and matrix multiplication can then be used to determine the reduced row echelon form of the whole of \(A\). This procedure is described in [Stein2007].

Checks that the matrix \(A/den\) is in reduced row echelon form of rank \(rank\), returns 1 if so and 0 otherwise.

### 4.5.27 Modular gaussian elimination

Uses fraction-free Gauss-Jordan elimination to set \(A\) to its reduced row echelon form and returns the rank of \(A\). All computations are done modulo \(p\).

Pivot elements are chosen with `fmpz_mat_find_pivot_any`. If `perm` is non-NULL, the permutation of rows in the matrix will also be applied to `perm`.

### 4.5.28 Strong echelon form and Howell form

Transforms \(A\) such that \(A\) modulo \(mod\) is the strong echelon form of the input matrix modulo \(mod\). The Howell form and the strong echelon form are equal up to permutation of the rows, see [FieHof2014] for a definition of the strong echelon form and the algorithm used here.

\(A\) must have at least as many rows as columns.

Transforms \(A\) such that \(A\) modulo \(mod\) is the Howell form of the input matrix modulo \(mod\). For a definition of the Howell form see [StoMul1998]. The Howell form is computed by first putting \(A\) into strong echelon form and then ordering the rows.

\(A\) must have at least as many rows as columns.
4.5.29 Nullspace

\texttt{long fmpz\_mat\_nullspace(fmpz\_mat\_t B, const fmpz\_mat\_t A)}

Computes a basis for the right rational nullspace of $A$ and returns the dimension of the nullspace (or nullity). $B$ is set to a matrix with linearly independent columns and maximal rank such that $AB = 0$ (i.e. $Ab = 0$ for each column $b$ in $B$), and the rank of $B$ is returned.

In general, the entries in $B$ will not be minimal: in particular, the pivot entries in $B$ will generally differ from unity. $B$ must be allocated with sufficient space to represent the result (at most $n \times n$ where $n$ is the number of columns of $A$).

4.5.30 Echelon form

\texttt{long fmpz\_mat\_rref\_fraction\_free(long *perm, fmpz\_mat\_t B, fmpz\_t den, const fmpz\_mat\_t A)}

Computes an integer matrix $B$ and an integer $\text{den}$ such that $B / \text{den}$ is the unique row reduced echelon form (RREF) of $A$ and returns the rank, i.e. the number of nonzero rows in $B$.

Aliasing of $B$ and $A$ is allowed, with an in-place computation being more efficient. The size of $B$ must be the same as that of $A$.

The permutation order will be written to $\text{perm}$ unless this argument is \texttt{NULL}. That is, row $i$ of the output matrix will correspond to row $\text{perm}[i]$ of the input matrix.

The denominator will always be a divisor of the determinant of (some submatrix of) $A$, but is not guaranteed to be minimal or canonical in any other sense.

4.5.31 Hermite normal form

\texttt{void fmpz\_mat\_hnf(fmpz\_mat\_t H, const fmpz\_mat\_t A)}

Computes an integer matrix $H$ such that $H$ is the unique (row) Hermite normal form of $A$. The algorithm used is selected from the implementations in FLINT to be the one most likely to be optimal, based on the characteristics of the input matrix.

Aliasing of $H$ and $A$ is allowed. The size of $H$ must be the same as that of $A$.

\texttt{void fmpz\_mat\_hnf\_transform(fmpz\_mat\_t H, fmpz\_mat\_t U, const fmpz\_mat\_t A)}

Computes an integer matrix $H$ such that $H$ is the unique (row) Hermite normal form of $A$ along with the transformation matrix $U$ such that $UA = H$. The algorithm used is selected from the implementations in FLINT as per \texttt{fmpz\_mat\_hnf}.

Aliasing of $H$ and $A$ is allowed. The size of $H$ must be the same as that of $A$ and $U$ must be square of compatible dimension (having the same number of rows as $A$).

\texttt{void fmpz\_mat\_hnf\_classical(fmpz\_mat\_t H, const fmpz\_mat\_t A)}

Computes an integer matrix $H$ such that $H$ is the unique (row) Hermite normal form of $A$. The algorithm used is straightforward and is described, for example, in [Algorithm 2.4.4] [Coh1996].

Aliasing of $H$ and $A$ is allowed. The size of $H$ must be the same as that of $A$.

\texttt{void fmpz\_mat\_hnf\_xgcd(fmpz\_mat\_t H, const fmpz\_mat\_t A)}

Computes an integer matrix $H$ such that $H$ is the unique (row) Hermite normal form of $A$. The algorithm used is an improvement on the basic algorithm and uses extended gcds to speed up computation, this method is described, for example, in [Algorithm 2.4.5] [Coh1996].

Aliasing of $H$ and $A$ is allowed. The size of $H$ must be the same as that of $A$.  

4.5. fmpz\_mat.h – matrices over the integers
void \texttt{fmpz\_mat\_hnf\_modular}\( (fmpz\_mat\_t \, H, \text{const} \, fmpz\_mat\_t \, A, \text{const} \, fmpz\_t \, D) \)

Computes an integer matrix \( H \) such that \( H \) is the unique (row) Hermite normal form of the \( m \times n \) matrix \( A \), where \( A \) is assumed to be of rank \( n \) and \( D \) is known to be a positive multiple of the determinant of the non-zero rows of \( H \). The algorithm used here is due to Dornich, Kannan and Trotter [DomKanTro1987] and is also described in [Algorithm 2.4.8] [Coh1996].

Aliasing of \( H \) and \( A \) is allowed. The size of \( H \) must be the same as that of \( A \).

void \texttt{fmpz\_mat\_hnf\_modular\_eldiv}\( (fmpz\_mat\_t \, A, \text{const} \, fmpz\_t \, D) \)

Transforms the \( m \times n \) matrix \( A \) into Hermite normal form, where \( A \) is assumed to be of rank \( n \) and \( D \) is known to be a positive multiple of the largest elementary divisor of \( A \). The algorithm used here is described in [FieHof2014].

void \texttt{fmpz\_mat\_hnf\_minors}\( (fmpz\_mat\_t \, H, \text{const} \, fmpz\_mat\_t \, A) \)

Computes an integer matrix \( H \) such that \( H \) is the unique (row) Hermite normal form of the \( m \times n \) matrix \( A \), where \( A \) is assumed to be of rank \( n \). The algorithm used here is due to Kannan and Bachem [KanBac1979] and takes the principal minors to Hermite normal form in turn.

Aliasing of \( H \) and \( A \) is allowed. The size of \( H \) must be the same as that of \( A \).

void \texttt{fmpz\_mat\_hnf\_pernet\_stein}\( (fmpz\_mat\_t \, H, \text{const} \, fmpz\_mat\_t \, A, \text{flint\_rand\_t} \, \text{state}) \)

Computes an integer matrix \( H \) such that \( H \) is the unique (row) Hermite normal form of the \( m \times n \) matrix \( A \). The algorithm used here is due to Pernet and Stein [PernetStein2010].

Aliasing of \( H \) and \( A \) is allowed. The size of \( H \) must be the same as that of \( A \).

int \texttt{fmpz\_mat\_is\_in\_hnf}\( (\text{const} \, fmpz\_mat\_t \, A) \)

Checks that the given matrix is in Hermite normal form, returns 1 if so and 0 otherwise.

4.5.32 Smith normal form

void \texttt{fmpz\_mat\_snf}\( (fmpz\_mat\_t \, S, \text{const} \, fmpz\_mat\_t \, A) \)

Computes an integer matrix \( S \) such that \( S \) is the unique Smith normal form of \( A \). The algorithm used is selected from the implementations in FLINT to be the one most likely to be optimal, based on the characteristics of the input matrix.

Aliasing of \( S \) and \( A \) is allowed. The size of \( S \) must be the same as that of \( A \).

void \texttt{fmpz\_mat\_snf\_diagonal}\( (fmpz\_mat\_t \, S, \text{const} \, fmpz\_mat\_t \, A) \)

Computes an integer matrix \( S \) such that \( S \) is the unique Smith normal form of the diagonal matrix \( A \). The algorithm used simply takes gcds of pairs on the diagonal in turn until the Smith form is obtained.

Aliasing of \( S \) and \( A \) is allowed. The size of \( S \) must be the same as that of \( A \).

void \texttt{fmpz\_mat\_snf\_kannan\_bachem}\( (fmpz\_mat\_t \, S, \text{const} \, fmpz\_mat\_t \, A) \)

Computes an integer matrix \( S \) such that \( S \) is the unique Smith normal form of the diagonal matrix \( A \). The algorithm used here is due to Kannan and Bachem [KanBac1979].

Aliasing of \( S \) and \( A \) is allowed. The size of \( S \) must be the same as that of \( A \).

void \texttt{fmpz\_mat\_snf\_iliopoulos}\( (fmpz\_mat\_t \, S, \text{const} \, fmpz\_mat\_t \, A, \text{const} \, fmpz\_t \, \text{mod}) \)

Computes an integer matrix \( S \) such that \( S \) is the unique Smith normal form of the nonsingular \( n \times n \) matrix \( A \). The algorithm used is due to Iliopoulos [Iliopoulos1989].

Aliasing of \( S \) and \( A \) is allowed. The size of \( S \) must be the same as that of \( A \).

int \texttt{fmpz\_mat\_is\_in\_snf}\( (\text{const} \, fmpz\_mat\_t \, A) \)

Checks that the given matrix is in Smith normal form, returns 1 if so and 0 otherwise.
4.5.33 Special matrices

```c
void fmpz_mat_gram(fmpz_mat_t B, const fmpz_mat_t A)
```

Sets \( B \) to the Gram matrix of the \( m \)-dimensional lattice \( L \) in \( n \)-dimensional Euclidean space \( \mathbb{R}^n \) spanned by the rows of the \( m \times n \) matrix \( A \). Dimensions must be compatible. \( A \) and \( B \) are allowed to be the same object if \( A \) is a square matrix.

```c
int fmpz_mat_is_hadamard(const fmpz_mat_t H)
```

Returns nonzero iff \( H \) is a Hadamard matrix, meaning that it is a square matrix, only has entries that are \( \pm 1 \), and satisfies \( H^T = n H^{-1} \) where \( n \) is the matrix size.

```c
int fmpz_mat_hadamard(fmpz_mat_t H)
```

Attempts to set the matrix \( H \) to a Hadamard matrix, returning 1 if successful and 0 if unsuccessful.

A Hadamard matrix of size \( n \) can only exist if \( n \) is 1, 2, or a multiple of 4. It is not known whether a Hadamard matrix exists for every size that is a multiple of 4. This function uses the Paley construction, which succeeds for all \( n \) of the form \( n = 2^e \) or \( n = 2^e (q + 1) \) where \( q \) is an odd prime power. Orders \( n \) for which Hadamard matrices are known to exist but for which this construction fails are 92, 116, 156, … (OEIS A046116).

4.5.34 Conversions

```c
int fmpz_mat_get_d_mat(d_mat_t B, const fmpz_mat_t A)
```

Sets the entries of \( B \) as doubles corresponding to the entries of \( A \), rounding down towards zero if the latter cannot be represented exactly. The return value is -1 if any entry of \( A \) is too large to fit in the normal range of a double, and 0 otherwise.

```c
int fmpz_mat_get_d_mat_transpose(d_mat_t B, const fmpz_mat_t A)
```

Sets the entries of \( B \) as doubles corresponding to the entries of the transpose of \( A \), rounding down towards zero if the latter cannot be represented exactly. The return value is -1 if any entry of \( A \) is too large to fit in the normal range of a double, and 0 otherwise.

4.5.35 Cholesky Decomposition

```c
void fmpz_mat_is_spd(const fmpz_mat_t A)
```

Returns true iff \( A \) is symmetric and positive definite (in particular square).

We first attempt a numerical \( LDL^T \) decomposition using `arb_mat_ldl()` . If we cannot guarantee that \( A \) is positive definite, we use an exact method instead, computing the characteristic polynomial of \( A \) and applying Descartes’ rule of signs.

```c
void fmpz_mat_chol_d(d_mat_t R, const fmpz_mat_t A)
```

Computes \( R \), the Cholesky factor of a symmetric, positive definite matrix \( A \) using the Cholesky decomposition process. (Sets \( R \) such that \( A = RR^T \) where \( R \) is a lower triangular matrix.)

4.5.36 LLL

```c
int fmpz_mat_is_reduced(const fmpz_mat_t A, double delta, double eta)
```

```c
int fmpz_mat_is_reduced_gram(const fmpz_mat_t A, double delta, double eta)
```

Returns a non-zero value if the basis \( A \) is LLL-reduced with factor \((\text{delta}, \text{eta})\), and otherwise returns zero. The second version assumes \( A \) is the Gram matrix of the basis.

```c
int fmpz_mat_is_reduced_with_removal(const fmpz_mat_t A, double delta, double eta, const fmpz_t gs_B, int newd)
```

4.5. `fmpz_mat.h` – matrices over the integers
int fmpz_mat_is_reduced_gram_with_removal(const fmpz_mat_t A, double delta, double eta, const fmpz_t gs_B, int newd)

Returns a non-zero value if the basis $A$ is LLL-reduced with factor $(\delta, \eta)$ for each of the first $\text{newd}$ vectors and the squared Gram-Schmidt length of each of the remaining $i$-th vectors (where $i \geq \text{newd}$) is greater than $\text{gs}_B$, and otherwise returns zero. The second version assumes $A$ is the Gram matrix of the basis.

4.5.37 Classical LLL

void fmpz_mat_lll_original(fmpz_mat_t A, const fmpq_t delta, const fmpq_t eta)

Takes a basis $x_1, x_2, \ldots, x_m$ of the lattice $L \subset \mathbb{R}^n$ (as the rows of a $m \times n$ matrix $A$). The output is a $(\delta, \eta)$-reduced basis $y_1, y_2, \ldots, y_m$ of the lattice $L$ (as the rows of the same $m \times n$ matrix $A$).

4.5.38 Modified LLL

void fmpz_mat_lll_storjohann(fmpz_mat_t A, const fmpq_t delta, const fmpq_t eta)

Takes a basis $x_1, x_2, \ldots, x_m$ of the lattice $L \subset \mathbb{R}^n$ (as the rows of a $m \times n$ matrix $A$). The output is an $(\delta, \eta)$-reduced basis $y_1, y_2, \ldots, y_m$ of the lattice $L$ (as the rows of the same $m \times n$ matrix $A$). Uses a modified version of LLL, which has better complexity in terms of the lattice dimension, introduced by Storjohann.


4.6 fmpz_lll.h – LLL reduction

4.6.1 Parameter manipulation

These functions are used to initialise LLL context objects which are of the type fmpz_lll_t. These objects contain all information about the options governing the reduction using this module’s functions including the LLL parameters delta and eta, the representation type of the input matrix (whether it is a lattice basis or a Gram matrix), and the type of Gram matrix to be used during $L^2$ (approximate or exact).

void fmpz_lll_context_init_default(fmpz_lll_t fl)

Sets fl->delta, fl->eta, fl->rt and fl->gt to their default values, 0.99, 0.51, $Z_{\text{BASIS}}$ and APPROX respectively.

void fmpz_lll_context_init(fmpz_lll_t fl, double delta, double eta, rep_type rt, gram_type gt)

Sets fl->delta, fl->eta, fl->rt and fl->gt to delta, eta, rt and gt (given as input) respectively. delta and eta are the $L^2$ parameters. delta and eta must lie in the intervals $(0.25, 1)$ and $(0.5, \sqrt{\text{delta}})$ respectively. The representation type is input using rt and can have the values $Z_{\text{BASIS}}$ for a lattice basis and GRAM for a Gram matrix. The Gram type to be used during computation can be specified using gt which can assume the values APPROX and EXACT. Note that gt has meaning only when rt is $Z_{\text{BASIS}}$. 

168 Chapter 4. Integers
4.6.2 Random parameter generation

```c
void fmpz_lll_randtest(fmpz_lll_t fl, flint_rand_t state)
```

Sets $f1->\delta$ and $f1->\eta$ to random values in the interval $\left(0.25, 1\right)$ and $\left(0.5, \sqrt{\delta}\right)$ respectively. $f1->rt$ is set to $\text{GRAM}$ or $\text{Z\_BASIS}$ and $f1->gt$ is set to $\text{APPROX}$ or $\text{EXACT}$ in a pseudo random way.

4.6.3 Heuristic dot product

```c
double fmpz_lll_heuristic_dot(const double *vec1, const double *vec2, slong len2, const fmpz_mat_t B, slong k, slong j, slong exp_adj)
```

Computes the dot product of two vectors of doubles $\text{vec1}$ and $\text{vec2}$, which are respectively $\text{double}$ approximations (up to scaling by a power of 2) to rows $k$ and $j$ in the exact integer matrix $B$. If massive cancellation is detected an exact computation is made.

The exact computation is scaled by $2^{-\exp_{\text{adj}}}$, where $\exp_{\text{adj}} = r2 + r1$ where $r2$ is the exponent for row $j$ and $r1$ is the exponent for row $k$ (i.e. row $j$ is notionally thought of as being multiplied by $2^{r2}$, etc.).

The final dot product computed by this function is then notionally the return value times $2^{\exp_{\text{adj}}}$.

4.6.4 The various Babai’s

```c
int fmpz_lll_check_babai(int kappa, fmpz_mat_t B, fmpz_mat_t U, d_mat_t mu, d_mat_t r, double *s, d_mat_t appB, int *expo, fmpz_gram_t A, int a, int zeros, int kappamax, int n, const fmpz_lll_t fl)
```

Performs floating point size reductions of the $kappa$-th row of $B$ by all of the previous rows, uses $\text{d\_mats\_mu}$ and $\text{r}$ for storing the GSO data. $U$ is used to capture the unimodular transformations if it is not $\text{N\_ULL}$. The double array $s$ will contain the size of the $kappa$-th row if it were moved into position $i$. The $\text{d\_mat\_appB}$ is an approximation of $B$ with each row receiving an exponent stored in $\text{expo}$ which gets populated only when needed. The $\text{d\_mat\_A->appSP}$ is an approximation of the Gram matrix whose entries are scalar products of the rows of $B$ and is used when $f1->gt == \text{APPROX}$. When $f1->gt == \text{EXACT}$ the $\text{fmpz\_mat\_A->exactSP}$ (the exact Gram matrix) is used. The index $a$ is the smallest row index which will be reduced from the $kappa$-th row. $\text{index\_zeros}$ is the number of zero rows in the matrix. $\text{kappamax}$ is the highest index which has been size-reduced so far, and $n$ is the number of columns you want to consider. $\text{f1}$ is an LLL ($L^2$) context object. The output is the value -1 if the process fails (usually due to insufficient precision) or 0 if everything was successful. These descriptions will be true for the future Babai procedures as well.

```c
int fmpz_lll_check_babai_heuristic_d(int kappa, fmpz_mat_t B, fmpz_mat_t U, d_mat_t mu, d_mat_t r, double *s, d_mat_t appB, int *expo, fmpz_gram_t A, int a, int zeros, int kappamax, int n, const fmpz_lll_t fl)
```

Same as $fmpz\_lll\_check\_babai()$ but using the heuristic inner product rather than a purely floating point inner product. The heuristic will compute at full precision when there is cancellation.

```c
int fmpz_lll_check_babai_heuristic(int kappa, fmpz_mat_t B, fmpz_mat_t U, mpf_mat_t mu, mpf_mat_t r, mpf *s, mpf_mat_t appB, mpf_gram_t A, int a, int zeros, int kappamax, int n, mpf_t tmp, mpf_t rtmp, flint_bitcnt_t prec, const fmpz_lll_t fl)
```

This function is like the $\text{mpf}$ version of $fmpz\_lll\_check\_babai\_heuristic\_d()$. However, it also inherits some temporary $\text{mpf\_t}$ variables $\text{tmp}$ and $\text{rtmp}$.

```c
int fmpz_lll_advance_check_babai(int cur_kappa, int kappa, fmpz_mat_t B, fmpz_mat_t U, d_mat_t mu, d_mat_t r, double *s, d_mat_t appB, int *expo, fmpz_gram_t A, int a, int zeros, int kappamax, int n, const fmpz_lll_t fl)
```

4.6. $fmpz\_lll\_t$ – LLL reduction
This is a Babai procedure which is used when size reducing a vector beyond an index which LLL has reached. \texttt{cur\_kappa} is the index behind which we can assume \( B \) is LLL reduced, while \texttt{kappa} is the vector to be reduced. This procedure only size reduces the \texttt{kappa}-th row by vectors up to \texttt{cur\_kappa}, not \texttt{kappa} - 1.

\begin{verbatim}
int fmpz_lll_advance_check_babai_heuristic_d(int cur_kappa, int kappa, fmpz_mat_t B,
 fmpz_mat_t U, d_mat_t mu, d_mat_t r, double *s, d_mat_t appB, int *expo, fmpz_gram_t A,
 int a, int zeros, int kappamax, int n, const fmpz_lll_t fl)
\end{verbatim}

Same as \texttt{fmpz_lll_advance_check_babai()} but using the heuristic inner product rather than a purely floating point inner product. The heuristic will compute at full precision when there is cancellation.

4.6.5 Shift

\begin{verbatim}
int fmpz_lll_shift(const fmpz_mat_t B)
\end{verbatim}

Computes the largest number of non-zero entries after the diagonal in \( B \).

4.6.6 Varieties of LLL

These programs implement ideas from the book chapter [Stehle2010]. The list of function here that are heuristic in nature and may return with \( B \) unreduced - that is to say, not do their job - includes (but is not necessarily limited to):

- \texttt{fmpz\_lll\_d()}
- \texttt{fmpz\_lll\_d\_heuristic()}
- \texttt{fmpz\_lll\_d\_heuristic\_with\_removal()}
- \texttt{fmpz\_lll\_d\_with\_removal()}
- \texttt{fmpz\_lll\_d\_with\_removal\_knapsack()}

\begin{verbatim}
int fmpz\_lll\_d(fmpz\_mat\_t B, fmpz\_mat\_t U, const fmpz\_lll\_t fl)
\end{verbatim}

This is a mildly greedy version of floating point LLL using doubles only. It tries the fast version of the Babai algorithm (\texttt{fmpz\_lll\_check\_babai()}). If that fails, then it switches to the heuristic version (\texttt{fmpz\_lll\_check\_babai\_heuristic\_d()}) for only one loop and switches right back to the fast version. It reduces \( B \) in place. \( U \) is the matrix used to capture the unimodular transformations if it is not \texttt{NULL}. An exception is raised if \( U != NULL \) and \( U->r != d \), where \( d \) is the lattice dimension. \( fl \) is the context object containing information containing the LLL parameters delta and eta. The function can perform reduction on both the lattice basis as well as its Gram matrix. The type of lattice representation can be specified via the parameter \( fl->rt \). The type of Gram matrix to be used in computation (approximate or exact) can also be specified through the variable \( fl->gt \) (applies only if \( fl->rt == Z\_BASIS \)).

\begin{verbatim}
int fmpz\_lll\_d\_heuristic(fmpz\_mat\_t B, fmpz\_mat\_t U, const fmpz\_lll\_t fl)
\end{verbatim}

This LLL reduces \( B \) in place using doubles only. It is similar to \texttt{fmpz\_lll\_d()} but only uses the heuristic inner products which attempt to detect cancellations.

\begin{verbatim}
int fmpz\_lll\_mpf2(fmpz\_mat\_t B, fmpz\_mat\_t U, flint\_bitcnt\_t prec, const fmpz\_lll\_t fl)
\end{verbatim}

This is LLL using \texttt{mpf} with the given precision, \texttt{prec} for the underlying GSO. It reduces \( B \) in place like the other LLL functions. The \texttt{mpf2} in the function name refers to the way the \texttt{mpf\_t}'s are initialised.
int fmpz_lll_mpf(fmpz_mat_t B, fmpz_mat_t U, const fmpz_lll_t fl)

A wrapper of fmpz_lll_mpf2(). This currently begins with \( \text{prec} == D\_BITS \), then for the first 20 loops, increases the precision one limb at a time. After 20 loops, it doubles the precision each time. There is a proof that this will eventually work. The return value of this function is 0 if the LLL is successful or -1 if the precision maxes out before \( B \) is LLL-reduced.

int fmpz_lll_mpf2_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)

Same as fmpz_lll_d() but with a removal bound, \( gs_B \). The return value is the new dimension of \( B \) if removals are desired.

int fmpz_lll_d_heuristic_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)

Same as fmpz_lll_d_heuristic() but with a removal bound, \( gs_B \). The return value is the new dimension of \( B \) if removals are desired.

int fmpz_lll_mpf2_with_removal(fmpz_mat_t B, fmpz_mat_t U, flint_bitcnt_t prec, const fmpz_t gs_B, const fmpz_lll_t fl)

Same as fmpz_lll_mpf2() but with a removal bound, \( gs_B \). The return value is the new dimension of \( B \) if removals are desired.

int fmpz_lll_d_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)

Same as fmpz_lll_d() but with a removal bound, \( gs_B \). The return value is the new dimension of \( B \) if removals are desired.

int fmpz_lll_d_heuristic_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)

Same as fmpz_lll_d_heuristic() but with a removal bound, \( gs_B \). The return value is the new dimension of \( B \) if removals are desired.

int fmpz_lll_d_with_removal_knapsack(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)

This is floating point LLL specialized to knapsack-type lattices. It performs early size reductions occasionally which makes things faster in the knapsack case. Otherwise, it is similar to fmpz_lll_d_with_removal.

int fmpz_lll-wrapper_with_removal_knapsack(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)

A wrapper of the procedures implementing the base case LLL with the addition of the removal boundary. It begins with the greediest version (fmpz_lll_d_with_removal_knapsack()), then adapts to the version using heuristic inner products only (fmpz_lll_d_heuristic_with_removal_knapsack()) if \( fl\_rt == Z\_BASIS \) and \( fl\_gt == APPROX \), and finally to the mpf version (fmpz_lll_mpf2_with_removal_knapsack()) if needed.
lattices. It begins with the greediest version and the engine of this version, \( \text{fmpz\_lll\_d\_with\_removal\_knapsack()} \), then adapts to the version using heuristic inner products only \( \text{fmpz\_lll\_d\_heuristic\_with\_removal()} \) if \( \text{fl\_rt} == \mathbb{Z}_\text{BASIS} \) and \( \text{fl\_gt} == \text{APPROX} \), and finally to the mpf version \( \text{fmpz\_lll\_mpf\_with\_removal()} \) if needed.

### 4.6.7 ULLL

```c
int fmpz_lll_with_removal_ulll(fmpz_mat_t FM, fmpz_mat_t UM, slong new_size, const fmpz_t gs_B, const fmpz_lll_t fl)
```

ULLL is a new style of LLL which adjoins an identity matrix to the input lattice \( FM \), then scales the lattice down to \( new\_size \) bits and reduces this augmented lattice. This tends to be more stable numerically than traditional LLL which means higher dimensions can be attacked using doubles. In each iteration a new identity matrix is adjoined to the truncated lattice. \( UM \) is used to capture the unimodular transformations, while \( gs\_B \) and \( fl \) have the same role as in the previous routines. The function is optimised for factoring polynomials.

### 4.6.8 LLL-reducedness


```c
int fmpz_lll_is_reduced_d(const fmpz_mat_t B, const fmpz_lll_t fl)
int fmpz_lll_is_reduced_mpfr(const fmpz_mat_t B, const fmpz_lll_t fl, flint_bitcnt_t prec)
int fmpz_lll_is_reduced_d_with_removal(const fmpz_mat_t B, const fmpz_lll_t fl, const fmpz_t gs_B, int newd)
int fmpz_lll_is_reduced_mpfr_with_removal(const fmpz_mat_t B, const fmpz_lll_t fl, const fmpz_t gs_B, int newd, flint_bitcnt_t prec)
```

A non-zero return indicates the matrix is definitely reduced, that is, that * \( \text{fmpz\_mat\_is\_reduced()} \) or \( \text{fmpz\_mat\_is\_reduced\_gram()} \) (for the first two) * \( \text{fmpz\_mat\_is\_reduced\_with\_removal()} \) or \( \text{fmpz\_mat\_is\_reduced\_gram\_with\_removal()} \) (for the last two) return non-zero. A zero return value is inconclusive. The \( d \) variants are performed in machine precision, while the \( mpfr \) uses a precision of \( prec \) bits.

```c
int fmpz_lll_is_reduced(const fmpz_mat_t B, const fmpz_lll_t fl, flint_bitcnt_t prec)
int fmpz_lll_is_reduced_with_removal(const fmpz_mat_t B, const fmpz_lll_t fl, const fmpz_t gs_B, int newd, flint_bitcnt_t prec)
```

The return from these functions is always conclusive: the functions * \( \text{fmpz\_mat\_is\_reduced()} \) or \( \text{fmpz\_mat\_is\_reduced\_gram()} \) * \( \text{fmpz\_mat\_is\_reduced\_with\_removal()} \) or \( \text{fmpz\_mat\_is\_reduced\_gram\_with\_removal()} \) are optimized by calling the above heuristics first and returning right away if they give a conclusive answer.

### 4.6.9 Modified ULLL

```c
void fmpz_lll_storjohann_ulll(fmpz_mat_t FM, slong new_size, const fmpz_lll_t fl)
```

Performs ULLL using \( \text{fmpz\_mat\_lll\_storjohann()} \) as the LLL function.

**Note:** This function is currently not tested. Use at your own risk.
4.6.10 Main LLL functions

```c
void fmpz_lll(fmpz_mat_t B, fmpz_mat_t U, const fmpz_lll_t fl)
```
Reduces $B$ in place according to the parameters specified by the LLL context object $f1$.

This is the main LLL function which should be called by the user. It currently calls the ULLL algorithm (without removals). The ULLL function in turn calls a LLL wrapper which tries to choose an optimal LLL algorithm, starting with a version using just doubles (ULLL tries to maximise usage of this), then a heuristic LLL followed by a full precision floating point LLL if required.

$U$ is the matrix used to capture the unimodular transformations if it is not $NULL$. An exception is raised if $U != NULL$ and $U->r != d$, where $d$ is the lattice dimension. $f1$ is the context object containing information containing the LLL parameters delta and eta. The function can perform reduction on both the lattice basis as well as its Gram matrix. The type of lattice representation can be specified via the parameter $f1->rt$. The type of Gram matrix to be used in computation (approximate or exact) can also be specified through the variable $f1->gt$ (applies only if $f1->rt == Z_BASIS$).

```c
int fmpz_lll_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)
```
Reduces $B$ in place according to the parameters specified by the LLL context object $f1$ and removes vectors whose squared Gram-Schmidt length is greater than the bound $gs_B$. The return value is the new dimension of $B$ to be considered for further computation.

This is the main LLL with removals function which should be called by the user. Like `fmpz_lll` it calls ULLL, but it also sets the Gram-Schmidt bound to that supplied and does removals.

4.7 fmpz_poly.h – univariate polynomials over the integers

4.7.1 Introduction

The `fmpz_poly_t` data type represents elements of $\mathbb{Z}[x]$. The `fmpz_poly` module provides routines for memory management, basic arithmetic, and conversions from or to other types.

Each coefficient of an `fmpz_poly_t` is an integer of the FLINT `fmpz_t` type. There are two advantages of this model. Firstly, the `fmpz_t` type is memory managed, so the user can manipulate individual coefficients of a polynomial without having to deal with tedious memory management. Secondly, a coefficient of an `fmpz_poly_t` can be changed without changing the size of any of the other coefficients.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

4.7.2 Simple example

The following example computes the square of the polynomial $5x^3 - 1$.

```c
#include "fmpz_poly.h"

int main()
{
    fmpz_poly_t x, y;
    fmpz_poly_init(x);
    fmpz_poly_init(y);
    fmpz_poly_set_coeff_ui(x, 3, 5);
    fmpz_poly_set_coeff_si(x, 0, -1);
    fmpz_poly_mul(y, x, x);
    fmpz_poly_print(x); flint_printf("\n");
    fmpz_poly_print(y); flint_printf("\n");
}
```
(continues on next page)
fmpz_poly_clear(x);
fmpz_poly_clear(y);
}

The output is:

<table>
<thead>
<tr>
<th>4</th>
<th>-1</th>
<th>0</th>
<th>0</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-10</td>
</tr>
</tbody>
</table>

### 4.7.3 Definition of the fmpz_poly_t type

The `fmpz_poly_t` type is a typedef for an array of length 1 of `fmpz_poly_struct`'s. This permits passing parameters of type `fmpz_poly_t` by reference in a manner similar to the way GMP integers of type `mpz_t` can be passed by reference.

In reality one never deals directly with the `struct` and simply deals with objects of type `fmpz_poly_t`. For simplicity we will think of an `fmpz_poly_t` as a `struct`, though in practice to access fields of this `struct`, one needs to dereference first, e.g. to access the `length` field of an `fmpz_poly_t` called `poly1` one writes `poly1->length`.

An `fmpz_poly_t` is said to be normalised if either `length` is zero, or if the leading coefficient of the polynomial is non-zero. All `fmpz_poly` functions expect their inputs to be normalised, and unless otherwise specified they produce output that is normalised.

It is recommended that users do not access the fields of an `fmpz_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose, detailed below.

Functions in `fmpz_poly` do all the memory management for the user. One does not need to specify the maximum length or number of limbs per coefficient in advance before using a polynomial object. FLINT reallocates space automatically as the computation proceeds, if more space is required. Each coefficient is also managed separately, being resized as needed, independently of the other coefficients.

### 4.7.4 Types, macros and constants

```c
typedef struct fmpz_poly_struct {
    slong length;  /* Length (number of terms) of the polynomial */
    fmpz_t coefficients;  /* Array of coefficients */
} fmpz_poly_t;
```

### 4.7.5 Memory management

```c
void fmpz_poly_init(fmpz_poly_t poly)
    Initialises `poly` for use, setting its length to zero. A corresponding call to `fmpz_poly_clear()` must be made after finishing with the `fmpz_poly_t` to free the memory used by the polynomial.

void fmpz_poly_init2(fmpz_poly_t poly, slong alloc)
    Initialises `poly` with space for at least `alloc` coefficients and sets the length to zero. The allocated coefficients are all set to zero.

void fmpz_poly_realloc(fmpz_poly_t poly, slong alloc)
    Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` the polynomial is first truncated to length `alloc`.
```
void fmpz_poly_fit_length(fmpz_poly_t poly, slong len)
    
    If len is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least len coefficients. No data is lost when calling this function. The function efficiently deals with the case where fit_length is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void fmpz_poly_clear(fmpz_poly_t poly)
    
    Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

void _fmpz_poly_normalise(fmpz_poly_t poly)
    
    Sets the length of poly so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void _fmpz_poly_set_length(fmpz_poly_t poly, slong newlen)
    
    Demotes the coefficients of poly beyond newlen and sets the length of poly to newlen.

void fmpz_poly_attach_truncate(fmpz_poly_t trunc, const fmpz_poly_t poly, slong n)
    
    This function sets the uninitialised polynomial trunc to the low n coefficients of poly, or to poly if the latter doesn’t have n coefficients. The polynomial trunc not be cleared or used as the output of any Flint functions.

void fmpz_poly_attach_shift(fmpz_poly_t trunc, const fmpz_poly_t poly, slong n)
    
    This function sets the uninitialised polynomial trunc to the high coefficients of poly, i.e. the coefficients not among the low n coefficients of poly. If the latter doesn’t have n coefficients trunc is set to the zero polynomial. The polynomial trunc not be cleared or used as the output of any Flint functions.

4.7.6 Polynomial parameters

slong fmpz_poly_length(const fmpz_poly_t poly)
    
    Returns the length of poly. The zero polynomial has length zero.

slong fmpz_poly_degree(const fmpz_poly_t poly)
    
    Returns the degree of poly, which is one less than its length.

4.7.7 Assignment and basic manipulation

void fmpz_poly_set(fmpz_poly_t poly1, const fmpz_poly_t poly2)
    
    Sets poly1 to equal poly2.

void fmpz_poly_set_si(fmpz_poly_t poly, slong c)
    
    Sets poly to the signed integer c.

void fmpz_poly_set_ui(fmpz_poly_t poly, ulong c)
    
    Sets poly to the unsigned integer c.

void fmpz_poly_set_fmpz(fmpz_poly_t poly, const fmpz_t c)
    
    Sets poly to the integer c.

int _fmpz_poly_set_str(fmpz *poly, const char *str)
    
    Sets poly to the polynomial encoded in the null-terminated string str. Assumes that poly is allocated as a sufficiently large array suitable for the number of coefficients present in str.

    Returns 0 if no error occurred. Otherwise, returns a non-zero value, in which case the resulting value of poly is undefined. If str is not null-terminated, calling this method might result in a segmentation fault.
int fmpz_poly_set_str(fmpz_poly_t poly, const char *str)

Imports a polynomial from a null-terminated string. If the string \texttt{str} represents a valid polynomial returns 0, otherwise returns 1.

Returns 0 if no error occurred. Otherwise, returns a non-zero value, in which case the resulting value of \texttt{poly} is undefined. If \texttt{str} is not null-terminated, calling this method might result in a segmentation fault.

char *fmpz_poly_get_str(const fmpz_t poly, slong len)

Returns the plain FLINT string representation of the polynomial \texttt{(poly, len)}.

char *fmpz_poly_get_str(const fmpz_poly_t poly)

Returns the plain FLINT string representation of the polynomial \texttt{poly}.

char *fmpz_poly_get_str_pretty(const fmpz_t poly, slong len, const char *x)

Returns a pretty representation of the polynomial \texttt{(poly, len)} using the null-terminated string \texttt{x} as the variable name.

char *fmpz_poly_get_str_pretty(const fmpz_poly_t poly, const char *x)

Returns a pretty representation of the polynomial \texttt{poly} using the null-terminated string \texttt{x} as the variable name.

void fmpz_poly_zero(fmpz_poly_t poly)

Sets \texttt{poly} to the zero polynomial.

void fmpz_poly_one(fmpz_poly_t poly)

Sets \texttt{poly} to the constant polynomial one.

void fmpz_poly_zero_coeffs(fmpz_poly_t poly, slong i, slong j)

Sets the coefficients of \(x^i, \ldots, x^{j-1}\) to zero.

void fmpz_poly_swap(fmpz_poly_t poly1, fmpz_poly_t poly2)

Swaps \texttt{poly1} and \texttt{poly2}. This is done efficiently without copying data by swapping pointers, etc.

void _fmpz_poly_reverse(fmpz *res, const fmpz *poly, slong len, slong n)

Sets \texttt{(res, n)} to the reverse of \texttt{(poly, n)}, where \texttt{poly} is in fact an array of length \texttt{len}. Assumes that 0 < \texttt{len} \leq \texttt{n}. Supports aliasing of \texttt{res} and \texttt{poly}, but the behaviour is undefined in case of partial overlap.

void fmpz_poly_reverse(fmpz_poly_t res, const fmpz_poly_t poly, slong n)

This function considers the polynomial \texttt{poly} to be of length \texttt{n}, notionally truncating and zero padding if required, and reverses the result. Since the function normalises its result \texttt{res} may be of length less than \texttt{n}.

void fmpz_poly_truncate(fmpz_poly_t poly, slong newlen)

If the current length of \texttt{poly} is greater than \texttt{newlen}, it is truncated to have the given length. Discarded coefficients are not necessarily set to zero.

void fmpz_poly_set_trunc(fmpz_poly_t res, const fmpz_poly_t poly, slong n)

Sets \texttt{res} to a copy of \texttt{poly}, truncated to length \texttt{n}.
4.7.8 Randomisation

```c
void fmpz_poly_randtest(fmpz_poly_t f, flint_rand_t state, slong len, flint_bitcnt_t bits)
Sets f to a random polynomial with up to the given length and where each coefficient has up to
the given number of bits. The coefficients are signed randomly.
```

```c
void fmpz_poly_randtest_unsigned(fmpz_poly_t f, flint_rand_t state, slong len, flint_bitcnt_t bits)
Sets f to a random polynomial with up to the given length and where each coefficient has up to
the given number of bits.
```

```c
void fmpz_poly_randtest_not_zero(fmpz_poly_t f, flint_rand_t state, slong len, flint_bitcnt_t bits)
As for fmpz_poly_randtest() except that len and bits may not be zero and the polynomial
generated is guaranteed not to be the zero polynomial.
```

```c
void fmpz_poly_randtest_no_real_root(fmpz_poly_t p, flint_rand_t state, slong len, flint_bitcnt_t bits)
Sets p to a random polynomial without any real root, whose length is up to len and where each
coefficient has up to the given number of bits.
```

```c
void fmpz_poly_randtest_irreducible1(fmpz_poly_t pol, flint_rand_t state, slong len, mp_bitcnt_t bits)
```

```c
void fmpz_poly_randtest_irreducible2(fmpz_poly_t pol, flint_rand_t state, slong len, mp_bitcnt_t bits)
```

```c
void fmpz_poly_randtest_irreducible(fmpz_poly_t pol, flint_rand_t state, slong len, mp_bitcnt_t bits)
Sets p to a random irreducible polynomial, whose length is up to len and where each coefficient
has up to the given number of bits. There are two algorithms: irreducible1 generates an irreducible
polynomial modulo a random prime number and lifts it to the integers; irreducible2 generates a
random integer polynomial, factors it, and returns a random factor. The default function chooses
randomly between these methods.
```

4.7.9 Getting and setting coefficients

```c
void fmpz_poly_get_coeff_fmpz(fmpz_t x, const fmpz_poly_t poly, slong n)
Sets x to the n-th coefficient of poly. Coefficient numbering is from zero and if n is set to a value
beyond the end of the polynomial, zero is returned.
```

```c
slong fmpz_poly_get_coeff_si(const fmpz_poly_t poly, slong n)
Returns coefficient n of poly as a slong. The result is undefined if the value does not fit into
a slong. Coefficient numbering is from zero and if n is set to a value beyond the end of the
polynomial, zero is returned.
```

```c
ulong fmpz_poly_get_coeff_ui(const fmpz_poly_t poly, slong n)
Returns coefficient n of poly as an ulong. The result is undefined if the value does not fit into
an ulong. Coefficient numbering is from zero and if n is set to a value beyond the end of the
polynomial, zero is returned.
```

```c
fmpz *fmpz_poly_get_coeff_ptr(const fmpz_poly_t poly, slong n)
Returns a reference to the coefficient of \(x^n\) in the polynomial, as an fmpz *. This function is
provided so that individual coefficients can be accessed and operated on by functions in the fmpz
module. This function does not make a copy of the data, but returns a reference to the actual
coefficient.
Returns NULL when n exceeds the degree of the polynomial.
This function is implemented as a macro.
```
flint Documentation, Release 3.1.0-dev

**fmpz**

*flmpz_poly_lead*(const *fmpz_poly_t* poly)

Returns a reference to the leading coefficient of the polynomial, as an *fmpz*. This function is provided so that the leading coefficient can be easily accessed and operated on by functions in the *fmpz* module. This function does not make a copy of the data, but returns a reference to the actual coefficient.

Returns NULL when the polynomial is zero.

This function is implemented as a macro.

**void fmpz_poly_set_coeff_fmpz**(fmpz_poly_t poly, slong n, const fmpz_t x)

Sets coefficient *n* of *poly* to the *fmpz* value *x*. Coefficient numbering starts from zero and if *n* is beyond the current length of *poly* then the polynomial is extended and zero coefficients inserted if necessary.

**void fmpz_poly_set_coeff_si**(fmpz_poly_t poly, slong n, slong x)

Sets coefficient *n* of *poly* to the *slong* value *x*. Coefficient numbering starts from zero and if *n* is beyond the current length of *poly* then the polynomial is extended and zero coefficients inserted if necessary.

**void fmpz_poly_set_coeff_ui**(fmpz_poly_t poly, slong n, ulong x)

Sets coefficient *n* of *poly* to the *ulong* value *x*. Coefficient numbering starts from zero and if *n* is beyond the current length of *poly* then the polynomial is extended and zero coefficients inserted if necessary.

### 4.7.10 Comparison

**int fmpz_poly_equal**(const fmpz_poly_t poly1, const fmpz_poly_t poly2)

Returns 1 if *poly1* is equal to *poly2*, otherwise returns 0. The polynomials are assumed to be normalised.

**int fmpz_poly_equal_trunc**(const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)

Return 1 if *poly1* and *poly2*, notionally truncated to length *n* are equal, otherwise return 0.

**int fmpz_poly_is_zero**(const fmpz_poly_t poly)

Returns 1 if the polynomial is zero and 0 otherwise.

This function is implemented as a macro.

**int fmpz_poly_is_one**(const fmpz_poly_t poly)

Returns 1 if the polynomial is one and 0 otherwise.

**int fmpz_poly_is_unit**(const fmpz_poly_t poly)

Returns 1 if the polynomial is the constant polynomial ±1, and 0 otherwise.

**int fmpz_poly_is_gen**(const fmpz_poly_t poly)

Returns 1 if the polynomial is the degree 1 polynomial *x*, and 0 otherwise.

### 4.7.11 Addition and subtraction

**void _fmpz_poly_add**(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)

Sets *res* to the sum of (*poly1*, *len1*) and (*poly2*, *len2*). It is assumed that *res* has sufficient space for the longer of the two polynomials.

**void fmpz_poly_add**(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)

Sets *res* to the sum of *poly1* and *poly2*.
void fmpz_poly_add_series(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
Notionally truncate poly1 and poly2 to length n and then set res to the sum.

void _fmpz_poly_sub(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
Sets res to (poly1, len1) minus (poly2, len2). It is assumed that res has sufficient space for the longer of the two polynomials.

void fmpz_poly_sub(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
Sets res to poly1 minus poly2.

void fmpz_poly_sub_series(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
Notionally truncate poly1 and poly2 to length n and then set res to the sum.

void fmpz_poly_neg(fmpz_poly_t res, const fmpz_poly_t poly)
Sets res to -poly.

4.7.12 Scalar absolute value, multiplication and division

void fmpz_poly_scalar_abs(fmpz_poly_t res, const fmpz_poly_t poly)
Sets poly1 to the polynomial whose coefficients are the absolute value of those of poly2.

void fmpz_poly_scalar_mul_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t x)
Sets poly1 to poly2 times x.

void fmpz_poly_scalar_mul_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
Sets poly1 to poly2 times the signed slong x.

void fmpz_poly_scalar_mul_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
Sets poly1 to poly2 times the ulong x.

void fmpz_poly_scalar_mul_2exp(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong exp)
Sets poly1 to poly2 times $2^{exp}$.

void fmpz_poly_scalar_addmul_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
Sets poly1 to poly1 + x * poly2.

void fmpz_poly_scalar_addmul_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
Sets poly1 to poly1 + x * poly2.

void fmpz_poly_scalar_fdiv_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t x)
Sets poly1 to poly2 divided by the fmpz_t x, rounding coefficients down toward $-\infty$.

void fmpz_poly_scalar_fdiv_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
Sets poly1 to poly2 divided by the slong x, rounding coefficients down toward $-\infty$.

void fmpz_poly_scalar_fdiv_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
Sets poly1 to poly2 divided by the ulong x, rounding coefficients down toward $-\infty$.

void fmpz_poly_scalar_fdiv_2exp(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong exp)
Sets poly1 to poly2 divided by $2^{\text{exp}}$, rounding coefficients down toward $-\infty$.

void fmpz_poly_scalar_tdiv_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t x)
Sets poly1 to poly2 divided by the fmpz_t x, rounding coefficients toward 0.
void fmpz_poly_scalar_tdiv_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
Sets poly1 to poly2 divided by the slong x, rounding coefficients toward 0.

void fmpz_poly_scalar_tdiv_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
Sets poly1 to poly2 divided by the ulong x, rounding coefficients toward 0.

void fmpz_poly_scalar_tdiv_2exp(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
Sets poly1 to poly2 divided by $2^x$, rounding coefficients toward 0.

void fmpz_poly_scalar_divexact_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t x)
Sets poly1 to poly2 divided by the fmpz_t x, assuming the division is exact for every coefficient.

void fmpz_poly_scalar_divexact_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
Sets poly1 to poly2 divided by the slong x, assuming the coefficient is exact for every coefficient.

void fmpz_poly_scalar_divexact_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
Sets poly1 to poly2 divided by the ulong x, assuming the coefficient is exact for every coefficient.

void fmpz_poly_scalar_mod_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t p)
Sets poly1 to poly2, reducing each coefficient modulo $p > 0$.

void fmpz_poly_scalar_mmod_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t p)
Sets poly1 to poly2, symmetrically reducing each coefficient modulo $p > 0$, that is, choosing the unique representative in the interval $(-p/2, p/2]$.

slong _fmpz_poly_remove_content_2exp(fmpz *pol, slong len)
Remove the 2-content of pol and return the number $k$ that is the maximal non-negative integer so that $2^k$ divides all coefficients of the polynomial. For the zero polynomial, 0 is returned.

void _fmpz_poly_scale_2exp(fmpz *pol, slong len, slong k)
Scale (pol, len) to $p(2^k X)$ in-place and divide by the 2-content (so that the gcd of coefficients is odd). If $k$ is negative the polynomial is multiplied by $2^{|k|}$.

### 4.7.13 Bit packing

void _fmpz_poly_bit_pack(mp_ptr arr, const fmpz_poly_t poly, slong len, flint_bitcnt_t bit_size, int negate)
Packs the coefficients of poly into bitfields of the given bit_size, negating the coefficients before packing if negate is set to $-1$.

int _fmpz_poly_bit_unpack(fmpz *poly, slong len, mp_srcptr arr, flint_bitcnt_t bit_size, int negate)
Unpacks the polynomial of given length from the array as packed into fields of the given bit_size, finally negating the coefficients if negate is set to $-1$. Returns borrow, which is nonzero if a leading term with coefficient $\pm 1$ should be added at position len of poly.

void _fmpz_poly_bit_unpack_unsigned(fmpz *poly, slong len, mp_srcptr arr, flint_bitcnt_t bit_size)
Unpacks the polynomial of given length from the array as packed into fields of the given bit_size. The coefficients are assumed to be unsigned.

void fmpz_poly_bit_pack(fmpz_poly_t f, const fmpz_poly_t poly, flint_bitcnt_t bit_size)
Packs poly into bitfields of size bit_size, writing the result to f. The sign of f will be the same as that of the leading coefficient of poly.

void fmpz_poly_bit_unpack(fmpz_poly_t poly, const fmpz_poly_t poly, flint_bitcnt_t bit_size)
Unpacks the polynomial with signed coefficients packed into fields of size bit_size as represented by the integer f.
void `fmpz_poly_bit_unpack_unsigned(fmpz_poly_t poly, const fmpz_t f, flint_bitcnt_t bit_size)`

Unpacks the polynomial with unsigned coefficients packed into fields of size `bit_size` as represented by the integer `f`. It is required that `f` is nonnegative.

### 4.7.14 Multiplication

void `_fmpz_poly_mul_classical(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)`

Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2).

Assumes `len1` and `len2` are positive. Allows zero-padding of the two input polynomials. No aliasing of inputs with outputs is allowed.

void `fmpz_poly_mul_classical(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)`

Sets `res` to the product of `poly1` and `poly2`, computed using the classical or schoolbook method.

void `_fmpz_poly_mullow_classical(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)`

Sets `(res, n)` to the first `n` coefficients of `(poly1, len1)` multiplied by `(poly2, len2).

Assumes `0 < n <= len1 + len2 - 1`. Assumes neither `len1` nor `len2` is zero.

void `fmpz_poly_mullow_classical(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)`

Sets `res` to the first `n` coefficients of `poly1 * poly2`.

void `_fmpz_poly_mulhigh_classical(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong start)`

Sets the first `start` coefficients of `res` to zero and the remainder to the corresponding coefficients of `(poly1, len1) * (poly2, len2).

Assumes `start <= len1 + len2 - 1`. Assumes neither `len1` nor `len2` is zero.

void `fmpz_poly_mulhigh_classical(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong start)`

Sets the first `start` coefficients of `res` to zero and the remainder to the corresponding coefficients of the product of `poly1` and `poly2`.

void `_fmpz_poly_mulmid_classical(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)`

Sets `res` to the middle `len1 - len2 + 1` coefficients of the product of `(poly1, len1)` and `(poly2, len2)`, i.e. the coefficients from degree `len2 - 1` to `len1 - 1` inclusive. Assumes that `len1 >= len2 > 0`.

void `fmpz_poly_mulmid_classical(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)`

Sets `res` to the middle `len(poly1) - len(poly2) + 1` coefficients of `poly1 * poly2`, i.e. the coefficient from degree `len2 - 1` to `len1 - 1` inclusive. Assumes that `len1 >= len2`.

void `_fmpz_poly_mul_karatsuba(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)`

Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2). Assumes `len1 >= len2 > 0`. Allows zero-padding of the two input polynomials. No aliasing of inputs with outputs is allowed.

void `fmpz_poly_mul_karatsuba(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)`

Sets `res` to the product of `poly1` and `poly2`. 

---

**4.7. `fmpz_poly.h` – univariate polynomials over the integers**
void _fmpz_poly_mullow_karatsuba_n(fmpz *res, const fmpz *poly1, const fmpz *poly2, slong n)
Sets res to the product of poly1 and poly2 and truncates to the given length. It is assumed that poly1 and poly2 are precisely the given length, possibly zero padded. Assumes n is not zero.

void fmpz_poly_mullow_karatsuba_n(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
Sets res to the product of poly1 and poly2 and truncates to the given length.

void _fmpz_poly_mulhigh_karatsuba_n(fmpz *res, const fmpz *poly1, const fmpz *poly2, slong len)
Sets res to the product of poly1 and poly2 and truncates at the top to the given length. The first len - 1 coefficients are set to zero. It is assumed that poly1 and poly2 are precisely the given length, possibly zero padded. Assumes len is not zero.

void fmpz_poly_mulhigh_karatsuba_n(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong len)
Sets the first len - 1 coefficients of the result to zero and the remaining coefficients to the corresponding coefficients of the product of poly1 and poly2. Assumes poly1 and poly2 are at most of the given length.

void _fmpz_poly_mul_KS(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
Sets (res, len1 + len2 - 1) to the product of (poly1, len1) and (poly2, len2). Places no assumptions on len1 and len2. Allows zero-padding of the two input polynomials. Supports aliasing of inputs and outputs.

void fmpz_poly_mul_KS(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
Sets res to the product of poly1 and poly2.

void _fmpz_poly_mullow_KS(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)
Sets (res, n) to the lowest n coefficients of the product of (poly1, len1) and (poly2, len2). Assumes that len1 and len2 are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between res, poly1 and poly2.

void fmpz_poly_mullow_KS(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
Sets res to the lowest n coefficients of the product of poly1 and poly2.

void _fmpz_poly_mul_SS(fmpz *output, const fmpz *input1, slong length1, const fmpz *input2, slong length2)
Sets (output, length1 + length2 - 1) to the product of (input1, length1) and (input2, length2).
We must have length1 > 1 and length2 > 1. Allows zero-padding of the two input polynomials. Supports aliasing of inputs and outputs.

void fmpz_poly_mul_SS(fmpz_poly_t output, const fmpz_poly_t input1, const fmpz_poly_t input2)
Sets res to the product of poly1 and poly2. Uses the Schönhage-Strassen algorithm.

void _fmpz_poly_mullow_SS(fmpz *output, const fmpz *input1, slong length1, const fmpz *input2, slong length2, slong n)
Sets (res, n) to the lowest n coefficients of the product of (poly1, len1) and (poly2, len2). Assumes that len1 and len2 are positive, but does allow for the polynomials to be zero-padded. We must have len1 > 1 and len2 > 1. Assumes n is positive. Supports aliasing between res, poly1 and poly2.

void fmpz_poly_mullow_SS(fmpz_poly_t output, const fmpz_poly_t input1, const fmpz_poly_t input2, slong n)
Sets res to the lowest n coefficients of the product of poly1 and poly2.
void _fmpz_poly_mul(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
Sets (res, len1 + len2 - 1) to the product of (poly1, len1) and (poly2, len2). Assumes len1 == len2 > 0. Allows zero-padding of the two input polynomials. Does not support aliasing between the inputs and the output.

void fmpz_poly_mul(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
Sets res to the product of poly1 and poly2. Chooses an optimal algorithm from the choices above.

void _fmpz_poly_mullow(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)
Sets (res, n) to the lowest \( n \) coefficients of the product of (poly1, len1) and (poly2, len2). Assumes len1 == len2 > 0 and 0 < n <= len1 + len2 - 1. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

void fmpz_poly_mullow(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
Sets res to the lowest \( n \) coefficients of the product of poly1 and poly2.

void fmpz_poly_mulhigh_n(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
Sets the high \( n \) coefficients of res to the high \( n \) coefficients of the product of poly1 and poly2, assuming the latter are precisely \( n \) coefficients in length, zero padded if necessary. The remaining \( n - 1 \) coefficients may be arbitrary.

void _fmpz_poly_mulhigh(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong start)
Sets all but the low \( n \) coefficients of res to the corresponding coefficients of the product of poly1 of length len1 and poly2 of length len2, the remaining coefficients being arbitrary. It is assumed that len1 == len2 > 0 and that 0 < n < len1 + len2 - 1. Aliasing of inputs is not permitted.

4.7.15 FFT precached multiplication

void fmpz_poly_mul_SS_precache_init(fmpz_poly_mul_precache_t pre, slong len1, slong bits1, const fmpz_poly_t poly2)
Precompute the FFT of poly2 to enable repeated multiplication of poly2 by polynomials whose length does not exceed len1 and whose number of bits per coefficient does not exceed bits1. The value bits1 may be negative, i.e. it may be the result of calling fmpz_poly_max_bits. The function only considers the absolute value of bits1.

Suppose len2 is the length of poly2 and len = len1 + len2 - 1 is the maximum output length of a polynomial multiplication using pre. Then internally len is rounded up to a power of two, \( 2^n \) say. The truncated FFT algorithm is used to smooth performance but note that it can only do this in the range \( (2^{n-1}, 2^n] \). Therefore, it may be more efficient to recompute pre for cases where the output length will fall below \( 2^n - 1 \). Otherwise the implementation will zero pad them up to that length.

Note that the Schoenhage-Strassen algorithm is only efficient for polynomials with relatively large coefficients relative to the length of the polynomials.

Also note that there are no restrictions on the polynomials. In particular the polynomial whose FFT is being precached does not have to be either longer or shorter than the polynomials it is to be multiplied by.

void fmpz_poly_mul_SS_precache_clear(fmpz_poly_mul_precache_t pre)
Clear the space allocated by fmpz_poly_mul_SS_precache_init.
void _fmpz_poly_mullow_SS_precache(fmpz *output, const fmpz *input1, slong len1, fmpz_poly_mul_precache_t pre, slong trunc)

Write into output the first trunc coefficients of the polynomial (input1, len1) by the polynomial whose FFT was precached by fmpz_poly_mul_SS_precache_init and stored in pre.

For performance reasons it is recommended that all polynomials be truncated to at most trunc coefficients if possible.

void fmpz_poly_mullow_SS_precache(fmpz_poly_t res, const fmpz_poly_t poly1, fmpz_poly_mul_precache_t pre, slong n)

Set res to the product of poly1 by the polynomial whose FFT was precached by fmpz_poly_mul_SS_precache_init (and stored in pre). The result is truncated to n coefficients (and normalised).

There are no restrictions on the length of poly1 other than those given in the call to fmpz_poly_mul_SS_precache_init.

void fmpz_poly_mul_SS_precache(fmpz_poly_t res, const fmpz_poly_t poly1, fmpz_poly_mul_precache_t pre)

Set res to the product of poly1 by the polynomial whose FFT was precached by fmpz_poly_mul_SS_precache_init (and stored in pre). There are no restrictions on the length of poly1 other than those given in the call to fmpz_poly_mul_SS_precache_init.

4.7.16 Squaring

void _fmpz_poly_sqr_KS(fmpz *rop, const fmpz *op, slong len)

Sets (rop, 2*len - 1) to the square of (op, len), assuming that len > 0.

Supports zero-padding in (op, len). Does not support aliasing.

void fmpz_poly_sqr_KS(fmpz_poly_t rop, const fmpz_poly_t op)

Sets rop to the square of the polynomial op using Kronecker segmentation.

void _fmpz_poly_sqr_karatsuba(fmpz *rop, const fmpz *op, slong len)

Sets (rop, 2*len - 1) to the square of (op, len), assuming that len > 0.

Supports zero-padding in (op, len). Does not support aliasing.

void fmpz_poly_sqr_karatsuba(fmpz_poly_t rop, const fmpz_poly_t op)

Sets rop to the square of the polynomial op using the Karatsuba multiplication algorithm.

void _fmpz_poly_sqr_classical(fmpz *rop, const fmpz *op, slong len)

Sets (rop, 2*len - 1) to the square of (op, len), assuming that len > 0.

Supports zero-padding in (op, len). Does not support aliasing.

void fmpz_poly_sqr_classical(fmpz_poly_t rop, const fmpz_poly_t op)

Sets rop to the square of the polynomial op using the classical or schoolbook method.

void _fmpz_poly_sqr(fmpz *rop, const fmpz *op, slong len)

Sets (rop, 2*len - 1) to the square of (op, len), assuming that len > 0.

Supports zero-padding in (op, len). Does not support aliasing.

void fmpz_poly_sqr(fmpz_poly_t rop, const fmpz_poly_t op)

Sets rop to the square of the polynomial op.
void _fmpz_poly_sqrlow_KS(fmpz *res, const fmpz *poly, slong len, slong n)
Sets (res, n) to the lowest n coefficients of the square of (poly, len).
Assumes that len is positive, but does allow for the polynomial to be zero-padded. The polynomial may be zero, too. Assumes n is positive. Supports aliasing between res and poly.

void fmpz_poly_sqrlow_KS(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
Sets res to the lowest n coefficients of the square of poly.

void _fmpz_poly_sqrlow_karatsuba_n(fmpz *res, const fmpz *poly, slong n)
Sets (res, n) to the square of (poly, n) truncated to length n, which is assumed to be positive. Allows for poly to be zero-padded.

void fmpz_poly_sqrlow_karatsuba_n(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
Sets res to the square of poly and truncates to the given length.

void _fmpz_poly_sqrlow_classical(fmpz *res, const fmpz *poly, slong len, slong n)
Sets (res, n) to the first n coefficients of the square of (poly, len). Assumes that 0 < n <= 2 * len - 1.

void fmpz_poly_sqrlow_classical(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
Sets res to the first n coefficients of the square of poly.

void _fmpz_poly_sqrlow(fmpz *res, const fmpz *poly, slong len, slong n)
Sets (res, n) to the lowest n coefficients of the square of (poly, len). Assumes len1 >= len2 > 0 and 0 < n <= 2 * len - 1. Allows for zero-padding in the input. Does not support aliasing between the input and the output.

void fmpz_poly_sqrlow(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
Sets res to the lowest n coefficients of the square of poly.

4.7.17 Powering

void _fmpz_poly_pow_multinomial(fmpz *res, const fmpz *poly, slong len, ulong e)
Computes res = poly^e. This uses the J.C.P. Miller pure recurrence as follows:
If ℓ is the index of the lowest non-zero coefficient in poly, as a first step this method zeros out
the lowest ℓe coefficients of res. The recurrence above is then used to compute the remaining coefficients.
Assumes len > 0, e > 0. Does not support aliasing.

void fmpz_poly_pow_multinomial(fmpz_poly_t res, const fmpz_poly_t poly, ulong e)
Computes res = poly^e using a generalisation of binomial expansion called the J.C.P. Miller pure
recurrence [1], [2]. If e is zero, returns one, so that in particular 0^0 = 1.
The formal statement of the recurrence is as follows. Write the input polynomial as
\[ P(x) = p_0 + p_1 x + \cdots + p_m x^m \]
with \( p_0 \neq 0 \) and let
\[ P(x)^n = a(n, 0) + a(n, 1)x + \cdots + a(n, mn)x^{mn}. \]
Then \( a(n, 0) = p_0^n \) and, for all \( 1 \leq k \leq mn \),
\[ a(n, k) = (kp_0)^{-1} \sum_{i=1}^{m} p_i ((n + 1)i - k)a(n, k - i). \]

void \_fmpz\_poly\_pow\_binomial(fmpz *res, const fmpz *poly, ulong e)
Computes res = poly\(^e\) when poly is of length 2, using binomial expansion.
Assumes \(e > 0\). Does not support aliasing.

void \*fmpz\_poly\_pow\_binomial(fmpz\_poly\_t res, const fmpz\_poly\_t poly, ulong e)
Computes res = poly\(^e\) when poly is of length 2, using binomial expansion.
If the length of poly is not 2, raises an exception and aborts.

void \_fmpz\_poly\_pow\_addchains(fmpz *res, const fmpz *poly, slong len, const int *a, int n)
Given a star chain \(1 = a_0 < a_1 < \cdots < a_n = e\) computes res = poly\(^e\).
A star chain is an addition chain \(1 = a_0 < a_1 < \cdots < a_n\) such that, for all \(i > 0\), \(a_i = a_{i-1} + a_j\) for some \(j < i\).
Assumes that \(e > 2\), or equivalently \(n > 1\), and \(len > 0\). Does not support aliasing.

void \*fmpz\_poly\_pow\_addchains(fmpz\_poly\_t res, const fmpz\_poly\_t poly, ulong e)
Computes res = poly\(^e\) using addition chains whenever \(0 \leq e \leq 148\).
If \(e > 148\), raises an exception and aborts.

void \_fmpz\_poly\_pow\_binexp(fmpz *res, const fmpz *poly, slong len, ulong e)
Sets res = poly\(^e\) using left-to-right binary exponentiation as described on p. 461 of \[Km1997\].
Assumes that \(len > 0\), \(e > 1\). Assumes that res is an array of length at least \(e\*(len - 1) + 1\).
Does not support aliasing.

void \*fmpz\_poly\_pow\_binexp(fmpz\_poly\_t res, const fmpz\_poly\_t poly, ulong e)
Computes res = poly\(^e\) using the binary exponentiation algorithm. If \(e\) is zero, returns one, so that in particular \(0^0 = 1\).

void \_fmpz\_poly\_pow\_small(fmpz *res, const fmpz *poly, slong len, ulong e)
Sets res = poly\(^e\) whenever \(0 \leq e \leq 4\).
Assumes that \(len > 0\) and that res is an array of length at least \(e\*(len - 1) + 1\). Does not support aliasing.

void \*fmpz\_poly\_pow(fmpz *res, const fmpz *poly, slong len, ulong e)
Sets res = poly\(^e\), assuming that \(e\), \(len > 0\) and that res has space for \(e\*(len - 1) + 1\) coefficients. Does not support aliasing.

void \*fmpz\_poly\_pow(fmpz\_poly\_t res, const fmpz\_poly\_t poly, ulong e)
Computes res = poly\(^e\). If \(e\) is zero, returns one, so that in particular \(0^0 = 1\).

void \_fmpz\_poly\_pow\_trunc(fmpz *res, const fmpz *poly, ulong e, slong n)
Sets (res, n) to (poly, n) raised to the power \(e\) and truncated to length \(n\).
Assumes that \(e, n > 0\). Allows zero-padding of (poly, n). Does not support aliasing of any inputs and outputs.

void \*fmpz\_poly\_pow\_trunc(fmpz\_poly\_t res, const fmpz\_poly\_t poly, ulong e, slong n)
Notationally raises poly to the power \(e\), truncates the result to length \(n\) and writes the result in res. This is computed much more efficiently than simply powering the polynomial and truncating.
Thus, if \(n = 0\) the result is zero. Otherwise, whenever \(e = 0\) the result will be the constant polynomial equal to 1.
This function can be used to raise power series to a power in an efficient way.
4.7.18 Shifting

void _fmpz_poly_shift_left(fmpz *res, const fmpz *poly, slong len, slong n)
Sets (res, len + n) to (poly, len) shifted left by n coefficients.
Inserts zero coefficients at the lower end. Assumes that len and n are positive, and that res fits len + n elements. Supports aliasing between res and poly.

void fmpz_poly_shift_left(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
Sets res to poly shifted left by n coeffs. Zero coefficients are inserted.

void _fmpz_poly_shift_right(fmpz *res, const fmpz *poly, slong len, slong n)
Sets (res, len - n) to (poly, len) shifted right by n coefficients.
Assumes that len and n are positive, that len > n, and that res fits len - n elements. Supports aliasing between res and poly, although in this case the top coefficients of poly are not set to zero.

void fmpz_poly_shift_right(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
Sets res to poly shifted right by n coefficients. If n is equal to or greater than the current length of poly, res is set to the zero polynomial.

4.7.19 Bit sizes and norms

ulong fmpz_poly_max_limbs(const fmpz_poly_t poly)
Returns the maximum number of limbs required to store the absolute value of coefficients of poly. If poly is zero, returns 0.

slong fmpz_poly_max_bits(const fmpz_poly_t poly)
Computes the maximum number of bits b required to store the absolute value of coefficients of poly. If all the coefficients of poly are non-negative, b is returned, otherwise −b is returned.

void fmpz_poly_height(fmpz_t height, const fmpz_poly_t poly)
Computes the height of poly, defined as the largest of the absolute values of the coefficients of poly. Equivalently, this gives the infinity norm of the coefficients. If poly is zero, the height is 0.

void _fmpz_poly_2norm(fmpz_t res, const fmpz *poly, slong len)
Sets res to the Euclidean norm of (poly, len), that is, the integer square root of the sum of the squares of the coefficients of poly.

void fmpz_poly_2norm(fmpz_t res, const fmpz_poly_t poly)
Sets res to the Euclidean norm of poly, that is, the integer square root of the sum of the squares of the coefficients of poly.

mp_limb_t _fmpz_poly_2norm_normalised_bits(const fmpz *poly, slong len)
Returns an upper bound on the number of bits of the normalised Euclidean norm of (poly, len), i.e. the number of bits of the Euclidean norm divided by the absolute value of the leading coefficient. The returned value will be no more than 1 bit too large.
This is used in the computation of the Landau-Mignotte bound.
It is assumed that len > 0. The result only makes sense if the leading coefficient is nonzero.
4.7.20 Greatest common divisor

```c
void _fmpz_poly_gcd_subresultant(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)

Computes the greatest common divisor (res, len2) of (poly1, len1) and (poly2, len2), assuming len1 >= len2 > 0. The result is normalised to have positive leading coefficient. Aliasing between res, poly1 and poly2 is supported.

void fmpz_poly_gcd_subresultant(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)

Computes the greatest common divisor res of poly1 and poly2, normalised to have non-negative leading coefficient.

This function uses the subresultant algorithm as described in Algorithm 3.3.1 of [Coh1996].

int _fmpz_poly_gcd_heuristic(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)

Computes the greatest common divisor (res, len2) of (poly1, len1) and (poly2, len2), assuming len1 >= len2 > 0. The result is normalised to have positive leading coefficient. Aliasing between res, poly1 and poly2 is not supported. The function may not always succeed in finding the GCD. If it fails, the function returns 0, otherwise it returns 1.

int fmpz_poly_gcd_heuristic(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)

Computes the greatest common divisor res of poly1 and poly2, normalised to have non-negative leading coefficient. The function may not always succeed in finding the GCD. If it fails, the function returns 0, otherwise it returns 1.

This function uses the heuristic GCD algorithm (GCDHEU). The basic strategy is to remove the content of the polynomials, pack them using Kronecker segmentation (given a bound on the size of the coefficients of the GCD) and take the integer GCD. Unpack the result and test divisibility.

void _fmpz_poly_gcd_modular(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)

Computes the greatest common divisor (res, len2) of (poly1, len1) and (poly2, len2), assuming len1 >= len2 > 0. The result is normalised to have positive leading coefficient. Assumes that res has space for len2 coefficients. Aliasing between res, poly1 and poly2 is not supported.

void fmpz_poly_gcd_modular(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)

Computes the greatest common divisor res of poly1 and poly2, normalised to have non-negative leading coefficient. This function uses the modular GCD algorithm. The basic strategy is to remove the content of the polynomials, reduce them modulo sufficiently many primes and do CRT reconstruction until some bound is reached (or we can prove with trial division that we have the GCD).

void _fmpz_poly_gcd(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)

Computes the greatest common divisor res of (poly1, len1) and (poly2, len2), assuming len1 >= len2 > 0. The result is normalised to have positive leading coefficient.

void fmpz_poly_gcd(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)

Computes the greatest common divisor res of poly1 and poly2, normalised to have non-negative leading coefficient.

void _fmpz_poly_xgcd_modular(fmpz_t r, fmpz *s, fmpz *t, const fmpz *f, slong len1, const fmpz *g, slong len2)

Set r to the resultant of (f, len1) and (g, len2). If the resultant is zero, the function returns immediately. Otherwise it finds polynomials s and t such that s*f + t*g = r. The length of s
will be no greater than \texttt{len2} and the length of \(t\) will be no greater than \texttt{len1} (both are zero padded if necessary).

It is assumed that \(\texttt{len1} >= \texttt{len2} > 0\). No aliasing of inputs and outputs is permitted.

The function assumes that \(f\) and \(g\) are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

Uses a multimodular algorithm. The resultant is first computed and extended GCDs modulo various primes \(p\) are computed and combined using CRT. When the CRT stabilises the resulting polynomials are simply reduced modulo further primes until a proven bound is reached.

\begin{verbatim}
void fmpz_poly_xgcd_modular (fmpz_t r, fmpz_poly_t s, fmpz_poly_t t, const fmpz_poly_t f, const fmpz_poly_t g)
Set \(r\) to the resultant of \(f\) and \(g\). If the resultant is zero, the function then returns immediately, otherwise \(s\) and \(t\) are found such that \(s*f + t*g = r\).

The function assumes that \(f\) and \(g\) are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

Uses the multimodular algorithm.
\end{verbatim}

\begin{verbatim}
void _fmpz_poly_xgcd (fmpz_t r, fmpz *s, fmpz *t, const fmpz *f, slong len1, const fmpz *g, slong len2)
Set \(r\) to the resultant of \((f, \texttt{len1})\) and \((g, \texttt{len2})\). If the resultant is zero, the function returns immediately, otherwise \(s\) and \(t\) are found such that \(s*f + t*g = r\). The length of \(s\) will be no greater than \texttt{len2} and the length of \(t\) will be no greater than \texttt{len1} (both are zero padded if necessary).

The function assumes that \(f\) and \(g\) are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

It is assumed that \(\texttt{len1} >= \texttt{len2} > 0\). No aliasing of inputs and outputs is permitted.
\end{verbatim}

\begin{verbatim}
void fmpz_poly_xgcd (fmpz_t r, fmpz_poly_t s, fmpz_poly_t t, const fmpz_poly_t f, const fmpz_poly_t g)
Set \(r\) to the resultant of \(f\) and \(g\). If the resultant is zero, the function then returns immediately, otherwise \(s\) and \(t\) are found such that \(s*f + t*g = r\).

The function assumes that \(f\) and \(g\) are primitive (have Gaussian content equal to 1). The result is undefined otherwise.
\end{verbatim}

\begin{verbatim}
void _fmpz_poly_lcm (fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
Sets \((\texttt{res, len1 + len2 - 1})\) to the least common multiple of the two polynomials \((\texttt{poly1, len1})\) and \((\texttt{poly2, len2})\), normalised to have non-negative leading coefficient.

Assumes that \(\texttt{len1} >= \texttt{len2} > 0\).

Does not support aliasing.
\end{verbatim}

\begin{verbatim}
void fmpz_poly_lcm (fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
Sets \(\texttt{res}\) to the least common multiple of the two polynomials \(\texttt{poly1}\) and \(\texttt{poly2}\), normalised to have non-negative leading coefficient.

If either of the two polynomials is zero, sets \(\texttt{res}\) to zero.

This ensures that the equality

\[
f g = \gcd(f, g) \text{lcm}(f, g)
\]

holds up to sign.
\end{verbatim}

\begin{verbatim}
void _fmpz_poly_resultant_modular (fmpz_t res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
Sets \(\texttt{res}\) to the resultant of \((\texttt{poly1, len1})\) and \((\texttt{poly2, len2})\), assuming that \(\texttt{len1} >= \texttt{len2} > 0\).
\end{verbatim}
void \texttt{fmpz\_poly\_resultant\_modular} (\texttt{fmpz\_t res}, \texttt{const fmpz\_poly\_t poly1}, \texttt{const fmpz\_poly\_t poly2})

Computes the resultant of poly1 and poly2.

For two non-zero polynomials \( f(x) = a_m x^m + \cdots + a_0 \) and \( g(x) = b_n x^n + \cdots + b_0 \) of degrees \( m \) and \( n \), the resultant is defined to be

\[
a_m n b_n m \prod_{(x,y):f(x)=g(y)=0} (x-y).
\]

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

This function uses the modular algorithm described in [Col1971].

void \texttt{fmpz\_poly\_resultant\_modular\_div} (\texttt{fmpz\_t res}, \texttt{const fmpz\_poly\_t poly1}, \texttt{const fmpz\_poly\_t poly2}, \texttt{const fmpz\_poly\_t div}, \texttt{slong nbits})

Computes the resultant of poly1 and poly2 divided by div using a slight modification of the above function. It is assumed that the resultant is exactly divisible by div and the result res has at most nbits bits. This bypasses the computation of general bounds.

void \texttt{fmpz\_poly\_resultant\_euclidean} (\texttt{fmpz\_t res}, \texttt{const fmpz \* poly1}, \texttt{slong len1}, \texttt{const fmpz \* poly2}, \texttt{slong len2})

Sets res to the resultant of (poly1, len1) and (poly2, len2), assuming that len1 >= len2 > 0.

void \texttt{fmpz\_poly\_resultant\_euclidean} (\texttt{fmpz\_t res}, \texttt{const fmpz\_poly\_t poly1}, \texttt{const fmpz\_poly\_t poly2})

Computes the resultant of poly1 and poly2.

For two non-zero polynomials \( f(x) = a_m x^m + \cdots + a_0 \) and \( g(x) = b_n x^n + \cdots + b_0 \) of degrees \( m \) and \( n \), the resultant is defined to be

\[
a_m n b_n m \prod_{(x,y):f(x)=g(y)=0} (x-y).
\]

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

This function uses the algorithm described in Algorithm 3.3.7 of [Coh1996].

void \texttt{fmpz\_poly\_resultant} (\texttt{fmpz\_t res}, \texttt{const fmpz \* poly1}, \texttt{slong len1}, \texttt{const fmpz \* poly2}, \texttt{slong len2})

Sets res to the resultant of (poly1, len1) and (poly2, len2), assuming that len1 >= len2 > 0.

void \texttt{fmpz\_poly\_resultant} (\texttt{fmpz\_t res}, \texttt{const fmpz\_poly\_t poly1}, \texttt{const fmpz\_poly\_t poly2})

Computes the resultant of poly1 and poly2.

For two non-zero polynomials \( f(x) = a_m x^m + \cdots + a_0 \) and \( g(x) = b_n x^n + \cdots + b_0 \) of degrees \( m \) and \( n \), the resultant is defined to be

\[
a_m n b_n m \prod_{(x,y):f(x)=g(y)=0} (x-y).
\]

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

\section*{4.7.21 Discriminant}

void \texttt{fmpz\_poly\_discriminant} (\texttt{fmpz\_t res}, \texttt{const fmpz \* poly}, \texttt{slong len})

Set res to the discriminant of (poly, len). Assumes len > 1.

void \texttt{fmpz\_poly\_discriminant} (\texttt{fmpz\_t res}, \texttt{const fmpz\_poly\_t poly})

Set res to the discriminant of poly. We normalise the discriminant so that \(\text{disc}(f) = (-1)^{(n(n-1)/2)} \text{res}(f,f')/\text{lcf}(f)\), thus \(\text{disc}(f) = \text{lcf}(f)^{2(n-2)} \prod_{i<j} (r_i - r_j)^2\), where lcf(f) is the leading coefficient of f, n is the degree of f and r_i are the roots of f.
4.7.22 Gaussian content

void _fmpz_poly_content(fmpz_t res, const fmpz *poly, slong len)
Sets res to the non-negative content of (poly, len). Aliasing between res and the coefficients of poly is not supported.

void fmpz_poly_content(fmpz_t res, const fmpz_poly_t poly)
Sets res to the non-negative content of poly. The content of the zero polynomial is defined to be zero. Supports aliasing, that is, res is allowed to be one of the coefficients of poly.

void _fmpz_poly_primitive_part(fmpz *res, const fmpz *poly, slong len)
Sets (res, len) to (poly, len) divided by the content of (poly, len), and normalises the result to have non-negative leading coefficient.
Assumes that (poly, len) is non-zero. Supports aliasing of res and poly.

void fmpz_poly_primitive_part(fmpz_poly_t res, const fmpz_poly_t poly)
Sets res to poly divided by the content of poly, and normalises the result to have non-negative leading coefficient. If poly is zero, sets res to zero.

4.7.23 Square-free

int _fmpz_poly_is_squarefree(const fmpz *poly, slong len)
Returns whether the polynomial (poly, len) is square-free.

int fmpz_poly_is_squarefree(const fmpz_poly_t poly)
Returns whether the polynomial poly is square-free. A non-zero polynomial is defined to be square-free if it has no non-unit square factors. We also define the zero polynomial to be square-free.

Returns 1 if the length of poly is at most 2. Returns whether the discriminant is zero for quadratic polynomials. Otherwise, returns whether the greatest common divisor of poly and its derivative has length 1.

4.7.24 Euclidean division

int _fmpz_poly_divrem_basecase(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB, int exact)
Computes (Q, lenA - lenB + 1), (R, lenA) such that A = BQ + R and each coefficient of R beyond lenB is reduced modulo the leading coefficient of B. If the leading coefficient of B is ±1 or the division is exact, this is the same thing as division over Q.
Assumes that len(A), len(B) > 0. Allows zero-padding in (A, lenA). R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.
If the flag exact is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B. This feature is useful for series division and for divisibility testing (upon testing the remainder).
For ordinary use set the flag exact to 0. In this case, no checks or early aborts occur and the function always returns 1.

void fmpz_poly_divrem_basecase(fmpz_poly_t Q, fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B)
Computes Q, R such that A = BQ + R and each coefficient of R beyond len(B) − 1 is reduced modulo the leading coefficient of B. If the leading coefficient of B is ±1 or the division is exact, this is the same thing as division over Q. An exception is raised if B is zero.
int _fmpz_poly_divrem_divconquer_recursive(fmpz *Q, fmpz *BQ, fmpz *W, const fmpz *A, const fmpz *B, slong lenB, int exact)

Computes \((Q, lenB), (BQ, 2 \text{ lenB} - 1)\) such that \(BQ = B \times Q\) and \(A = BQ + R\) where each coefficient of \(R\) beyond \(\text{len}(B) - 1\) is reduced modulo the leading coefficient of \(B\). We assume that \(\text{len}(A) = 2\text{len}(B) - 1\). If the leading coefficient of \(B\) is \(\pm1\) or the division is exact, this is the same as division over \(\mathbb{Q}\).

Assumes \(\text{len}(B) > 0\). Allows zero-padding in \((A, \text{lenA})\). Requires a temporary array \((W, 2 \text{ lenB} - 1)\). No aliasing of input and output operands is allowed.

This function does not read the bottom \(\text{len}(B) - 1\) coefficients from \(A\), which means that they might not even need to exist in allocated memory.

If the flag \text{exact} is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of \(B\). This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag \text{exact} to 0. In this case, no checks or early aborts occur and the function always returns 1.

int _fmpz_poly_divrem_divconquer(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB, int exact)

Computes \((Q, \text{lenA} - \text{lenB} + 1), (R, \text{lenA})\) such that \(A = BQ + R\) and each coefficient of \(R\) beyond \(\text{len}(B) - 1\) is reduced modulo the leading coefficient of \(B\). If the leading coefficient of \(B\) is \(\pm1\) or the division is exact, this is the same as division over \(\mathbb{Q}\).

Assumes \(\text{len}(A) \geq \text{len}(B) > 0\). Allows zero-padding in \((A, \text{lenA})\). No aliasing of input and output operands is allowed.

If the flag \text{exact} is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of \(B\). This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag \text{exact} to 0. In this case, no checks or early aborts occur and the function always returns 1.
void _fmpz_poly_div_basecase(fmpz *Q, fmpz *R, const fmpz *A, const fmpz *B, slong lenA, const fmpz *B, slong lenB, int exact)

Computes the quotient \(Q\), \(\text{len}A - \text{len}B + 1\) of \((A, \text{len}A)\) divided by \((B, \text{len}B)\).

Notationally, computes \(Q, R\) such that \(A = BQ + R\) and each coefficient of \(R\) beyond \(\text{len}(B) - 1\) is reduced modulo the leading coefficient of \(B\). If the leading coefficient of \(B\) is \(\pm 1\) or the division is exact, this is the same as division over \(\mathbb{Q}\). An exception is raised if \(B\) is zero.

Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of \(B\). This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag \textbf{exact} to 0. In this case, no checks or early aborts occur and the function always returns 1.

void _fmpz_poly_div_basecase(fmpz *Q, const fmpz *A, const fmpz *B)

Computes the quotient \(Q\) of \(A\) divided by \(B\).

Notationally, computes \(Q, R\) such that \(A = BQ + R\) and each coefficient of \(R\) beyond \(\text{len}(B) - 1\) is reduced modulo the leading coefficient of \(B\). If the leading coefficient of \(B\) is \(\pm 1\) or the division is exact, this is the same as division over \(\mathbb{Q}\). An exception is raised if \(B\) is zero.

int _fmpz_poly_divremlow_divconquer_recursive(fmpz *Q, fmpz *BQ, const fmpz *A, const fmpz *B, slong lenB, int exact)

Divide and conquer division of \((A, 2 \text{ len}B - 1)\) by \((B, \text{len}B)\), computing only the bottom \(\text{len}(B) - 1\) coefficients of \(BQ\).

Assumes \(\text{len}(B) > 0\). Requires \(BQ\) to have length at least \(2\text{len}(B) - 1\), although only the bottom \(\text{len}(B) - 1\) coefficients will carry meaningful output. Does not support any aliasing. Allows zero-padding in \(A\), but not in \(B\).

If the flag \textbf{exact} is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of \(B\). This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag \textbf{exact} to 0. In this case, no checks or early aborts occur and the function always returns 1.

int _fmpz_poly_div_divconquer_recursive(fmpz *Q, fmpz *temp, const fmpz *A, const fmpz *B, slong lenB, int exact)

Recursive short division in the balanced case.

Computes the quotient \((Q, \text{len}B)\) of \((A, 2 \text{ len}B - 1)\) upon division by \((B, \text{len}B)\). Requires \(\text{len}(B) > 0\). Needs a temporary array \textbf{temp} of length \(2\text{len}(B) - 1\). Does not support any aliasing.

For further details, see [Mul2000].

If the flag \textbf{exact} is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this
does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B. This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

```c
int _fmpz_poly_div_basecase(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB, int exact)
```

Computes the quotient \( Q \), \( \text{len}(A) - \text{len}(B) + 1 \) of \( (A, \text{len}(A)) \) upon division by \( (B, \text{len}(B)) \). Assumes that \( \text{len}(A) \geq \text{len}(B) > 0 \). Does not support aliasing.

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B. This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

```c
void fmpz_poly_div_basecase(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B, int exact)
```

Computes the quotient \( Q \) of \( A \) divided by \( B \).

Notationally, computes \( Q, R \) such that \( A = BQ + R \) and each coefficient of \( R \) beyond \( \text{len}(B) - 1 \) is reduced modulo the leading coefficient of \( B \).

If the leading coefficient of \( B \) is \( \pm 1 \) or the division is exact, this is the same as division over \( \mathbb{Q} \). An exception is raised if \( B \) is zero.

```c
int _fmpz_poly_div(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB, int exact)
```

Computes the quotient \( Q \), \( \text{len}(A) - \text{len}(B) + 1 \) of \( (A, \text{len}(A)) \) divided by \( (B, \text{len}(B)) \).

Notationally, computes \( Q, R \) such that \( A = BQ + R \) and each coefficient of \( R \) beyond \( \text{len}(B) - 1 \) is reduced modulo the leading coefficient of \( B \). If the leading coefficient of \( B \) is \( \pm 1 \) or the division is exact, this is the same as division over \( \mathbb{Q} \). An exception is raised if \( B \) is zero.

Assumes \( \text{len}(A) \geq \text{len}(B) > 0 \). Allows zero-padding in \( (A, \text{len}(A)) \). Aliasing of input and output operands is not allowed.

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B. This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

```c
void fmpz_poly_div(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B)
```

Computes the quotient \( Q \) of \( A \) divided by \( B \).

Notationally, computes \( Q, R \) such that \( A = BQ + R \) and each coefficient of \( R \) beyond \( \text{len}(B) - 1 \) is reduced modulo the leading coefficient of \( B \). If the leading coefficient of \( B \) is \( \pm 1 \) or the division is exact, this is the same as division over \( \mathbb{Q} \). An exception is raised if \( B \) is zero.

```c
void _fmpz_poly_rem_basecase(fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB)
```

Computes the remainder \( (R, \text{len}(A)) \) of \( (A, \text{len}(A)) \) upon division by \( (B, \text{len}(B)) \).

Notationally, computes \( Q, R \) such that \( A = BQ + R \) and each coefficient of \( R \) beyond \( \text{len}(B) - 1 \) is reduced modulo the leading coefficient of \( B \). If the leading coefficient of \( B \) is \( \pm 1 \) or the division is exact, this is the same thing as division over \( \mathbb{Q} \).

Assumes that \( \text{len}(A),\text{len}(B) > 0 \). Allows zero-padding in \( (A, \text{len}(A)) \). \( R \) and \( A \) may be aliased, but apart from this no aliasing of input and output operands is allowed.
void `fmpz_poly_rem_basecase(fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B)`
Computes the remainder \( R \) of \( A \) upon division by \( B \).

Notationally, computes \( Q, R \) such that \( A = BQ + R \) and each coefficient of \( R \) beyond \( \text{len}(B) - 1 \) is reduced modulo the leading coefficient of \( B \). If the leading coefficient of \( B \) is \( \pm 1 \) or the division is exact, this is the same as division over \( Q \). An exception is raised if \( B \) is zero.

void `_fmpz_poly_rem(fmpz *R, const fmpz *A, long lenA, const fmpz *B, long lenB)`
Computes the remainder \((R, \text{lenA})\) of \((A, \text{lenA})\) upon division by \((B, \text{lenB})\).

Notationally, computes \( Q, R \) such that \( A = BQ + R \) and each coefficient of \( R \) beyond \( \text{len}(B) - 1 \) is reduced modulo the leading coefficient of \( B \). If the leading coefficient of \( B \) is \( \pm 1 \) or the division is exact, this is the same thing as division over \( Q \).

Assumes that \( \text{len}(A) \geq \text{len}(B) > 0 \). Allows zero-padding in \((A, \text{lenA})\). Aliasing of input and output operands is not allowed.

void `fmpz_poly_rem(fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B)`
Computes the remainder \( R \) of \( A \) upon division by \( B \).

Notationally, computes \( Q, R \) such that \( A = BQ + R \) and each coefficient of \( R \) beyond \( \text{len}(B) - 1 \) is reduced modulo the leading coefficient of \( B \). If the leading coefficient of \( B \) is \( \pm 1 \) or the division is exact, this is the same as division over \( Q \). An exception is raised if \( B \) is zero.

void `_fmpz_poly_div_root(fmpz *Q, const fmpz *A, long len, const fmpz_t c)`
Computes the quotient \((Q, \text{len}-1)\) of \((A, \text{len})\) upon division by \( x - c \).

Supports aliasing of \( Q \) and \( A \), but the result is undefined in case of partial overlap.

void `fmpz_poly_div_root(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_t c)`
Computes the quotient \((Q, \text{len}-1)\) of \((A, \text{len})\) upon division by \( x - c \).

### 4.7.25 Division with precomputed inverse

void `_fmpz_poly_preinvert(fmpz *B_inv, const fmpz *B, long n)`
Given a monic polynomial \( B \) of length \( n \), compute a precomputed inverse \( B_{\text{inv}} \) of length \( n \) for use in the functions below. No aliasing of \( B \) and \( B_{\text{inv}} \) is permitted. We assume \( n \) is not zero.

void `fmpz_poly_preinvert(fmpz_poly_t B_inv, const fmpz_poly_t B)`
Given a monic polynomial \( B \), compute a precomputed inverse \( B_{\text{inv}} \) for use in the functions below. An exception is raised if \( B \) is zero.

void `_fmpz_poly_div_preinv(fmpz *Q, const fmpz *A, long len1, const fmpz *B, const fmpz *B_inv, slong len2)`
Given a precomputed inverse \( B_{\text{inv}} \) of the polynomial \( B \) of length \( \text{len2} \), compute the quotient \( Q \) of \( A \) by \( B \). We assume the length \( \text{len1} \) of \( A \) is at least \( \text{len2} \). The polynomial \( Q \) must have space for \( \text{len1} - \text{len2} + 1 \) coefficients. No aliasing of operands is permitted.

void `fmpz_poly_div_preinv(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B, const fmpz_poly_t B_inv)`
Given a precomputed inverse \( B_{\text{inv}} \) of the polynomial \( B \), compute the quotient \( Q \) of \( A \) by \( B \). Aliasing of \( B \) and \( B_{\text{inv}} \) is not permitted.

void `_fmpz_poly_divrem_preinv(fmpz *Q, fmpz *A, long len1, const fmpz *B, const fmpz *B_inv, slong len2)`
Given a precomputed inverse \( B_{\text{inv}} \) of the polynomial \( B \) of length \( \text{len2} \), compute the quotient \( Q \) of \( A \) by \( B \). The remainder is then placed in \( A \). We assume the length \( \text{len1} \) of \( A \) is at least \( \text{len2} \). The polynomial \( Q \) must have space for \( \text{len1} - \text{len2} + 1 \) coefficients. No aliasing of operands is permitted.
Given a precomputed inverse \( B_{\text{inv}} \) of the polynomial \( B \), compute the quotient \( Q \) of \( A \) by \( B \) and the remainder \( R \). Aliasing of \( B \) and \( B_{\text{inv}} \) is not permitted.

**Flint Documentation, Release 3.1.0-dev**
4.7.27 Division mod p

void fmpz_poly_divlow_smodp(fmpz *res, const fmpz_poly_t f, const fmpz_poly_t g, const fmpz_t p, slong n)

Compute the \(n\) lowest coefficients of \(f\) divided by \(g\), assuming the division is exact modulo \(p\). The computed coefficients are reduced modulo \(p\) using the symmetric remainder system. We require \(f\) to be at least \(n\) in length. The function can handle trailing zeroes, but the low nonzero coefficient of \(g\) must be coprime to \(p\). This is a bespoke function used by factoring.

void fmpz_poly_divhigh_smodp(fmpz *res, const fmpz_poly_t f, const fmpz_poly_t g, const fmpz_t p, slong n)

Compute the \(n\) highest coefficients of \(f\) divided by \(g\), assuming the division is exact modulo \(p\). The computed coefficients are reduced modulo \(p\) using the symmetric remainder system. We require \(f\) to be as output by fmpz_poly_mulhigh_n given polynomials \(g\) and a polynomial of length \(n\) as inputs. The leading coefficient of \(g\) must be coprime to \(p\). This is a bespoke function used by factoring.

4.7.28 Power series division

void _fmpz_poly_inv_series_basecase(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n)

Computes the first \(n\) terms of the inverse power series of \((Q, \text{len}Q)\) using a recurrence. Assumes that \(n \geq 1\) and that \(Q\) has constant term \(\pm 1\). Does not support aliasing.

void fmpz_poly_inv_series_basecase(fmpz_poly_t Qinv, const fmpz_poly_t Q, slong n)

Computes the first \(n\) terms of the inverse power series of \(Q\) using a recurrence, assuming that \(Q\) has constant term \(\pm 1\) and \(n \geq 1\).

void _fmpz_poly_inv_series_newton(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n)

Computes the first \(n\) terms of the inverse power series of \((Q, \text{len}Q)\) using Newton iteration. Assumes that \(n \geq 1\) and that \(Q\) has constant term \(\pm 1\). Does not support aliasing.

void fmpz_poly_inv_series_newton(fmpz_poly_t Qinv, const fmpz_poly_t Q, slong n)

Computes the first \(n\) terms of the inverse power series of \(Q\) using Newton iteration, assuming \(Q\) has constant term \(\pm 1\) and \(n \geq 1\).

void _fmpz_poly_inv_series(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n)

Computes the first \(n\) terms of the inverse power series of \((Q, \text{len}Q)\). Assumes that \(n \geq 1\) and that \(Q\) has constant term \(\pm 1\). Does not support aliasing.

void fmpz_poly_inv_series(fmpz_poly_t Qinv, const fmpz_poly_t Q, slong n)

Computes the first \(n\) terms of the inverse power series of \(Q\), assuming \(Q\) has constant term \(\pm 1\) and \(n \geq 1\).

void _fmpz_poly_div_series_basecase(fmpz *Q, const fmpz *A, slong Alen, const fmpz *B, slong Blen, slong n)

void _fmpz_poly_div_series_divconquer(fmpz *Q, const fmpz *A, slong Alen, const fmpz *B, slong Blen, slong n)

void _fmpz_poly_div_series(fmpz *Q, const fmpz *A, slong Alen, const fmpz *B, slong Blen, slong n)

Divides \((A, \text{Alen})\) by \((B, \text{Blen})\) as power series over \(\mathbb{Z}\), assuming \(B\) has constant term \(\pm 1\) and \(n \geq 1\). Aliasing is not supported.

void fmpz_poly_div_series_basecase(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B, slong n)
#### 4.7.29 Pseudo division

**void fmpz_poly_pseudo_divrem_basecase** *(fmpz *Q, fmpz *R, ulong *d, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_preinvn_t inv)*

If $\ell$ is the leading coefficient of $B$, then computes $Q$, $R$ such that $\ell^d A = QB + R$. This function is used for simulating division over $\mathbb{Q}$. Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that $Q$ can fit $\text{len}(A) - \text{len}(B) + 1$ coefficients, and that $R$ can fit $\text{len}(A)$ coefficients. Supports aliasing of $(R, \text{lenA})$ and $(A, \text{lenA})$. But other than this, no aliasing of the inputs and outputs is supported.

An optional precomputed inverse of the leading coefficient of $B$ from $\text{fmpz_preinvn_init}$ can be supplied. Otherwise $\text{inv}$ should be `NULL`.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

**void fmpz_poly_pseudo_divrem_basecase** *(fmpz_poly_t Q, fmpz_poly_t R, ulong *d, const fmpz_poly_t A, const fmpz_poly_t B)***

If $\ell$ is the leading coefficient of $B$, then computes $Q$, $R$ such that $\ell^d A = QB + R$. This function is used for simulating division over $\mathbb{Q}$. Computes $(Q, \text{lenA} - \text{lenB} + 1)$, $(R, \text{lenA})$ such that $\ell^d A = BQ + R$, only setting the bottom $\text{len}(B) - 1$ coefficients of $R$ to their correct values. The remaining top coefficients of $(R, \text{lenA})$ may be arbitrary.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in $(A, \text{lenA})$. No aliasing of input and output operands is allowed.

An optional precomputed inverse of the leading coefficient of $B$ from $\text{fmpz_preinvn_init}$ can be supplied. Otherwise $\text{inv}$ should be `NULL`.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

**void fmpz_poly_pseudo_divrem_divconquer** *(fmpz_poly_t Q, fmpz_poly_t R, ulong *d, const fmpz_poly_t A, const fmpz_poly_t B)***

Computes $Q$, $R$, and $d$ such that $\ell^d A = BQ + R$, where $R$ has length less than the length of $B$ and $\ell$ is the leading coefficient of $B$. An exception is raised if $B$ is zero.

**void fmpz_poly_pseudo_divrem_cohen** *(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB)***

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that $Q$ can fit $\text{len}(A) - \text{len}(B) + 1$ coefficients, and that $R$ can fit $\text{len}(A)$ coefficients. Supports aliasing of $(R, \text{lenA})$ and $(A, \text{lenA})$. But other than this, no aliasing of the inputs and outputs is supported.

**void fmpz_poly_pseudo_divrem_cohen** *(fmpz_poly_t Q, fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B)***
This is a variant of `fmpz_poly_pseudo_divrem` which computes polynomials $Q$ and $R$ such that $\ell^d A = B Q + R$. However, the value of $d$ is fixed at $\max \{0, \text{len}(A) - \text{len}(B) + 1\}$.

This function is faster when the remainder is not well behaved, i.e. where it is not expected to be close to zero. Note that this function is not asymptotically fast. It is efficient only for short polynomials, e.g. when $\text{len}(B) < 32$.

```c
void _fmpz_poly_pseudo_rem_cohen(fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB)
```

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that $R$ can fit $\text{len}(A)$ coefficients. Supports aliasing of $(R, \text{len}A)$ and $(A, \text{len}A)$. But other than this, no aliasing of the inputs and outputs is supported.

```c
void fmpz_poly_pseudo_rem_cohen(fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B)
```

This is a variant of `fmpz_poly_pseudo_rem()` which computes polynomials $Q$ and $R$ such that $\ell^d A = B Q + R$, but only returns $R$. However, the value of $d$ is fixed at $\max \{0, \text{len}(A) - \text{len}(B) + 1\}$.

This function is faster when the remainder is not well behaved, i.e. where it is not expected to be close to zero. Note that this function is not asymptotically fast. It is efficient only for short polynomials, e.g. when $\text{len}(B) < 32$.

This function uses the algorithm described in Algorithm 3.1.2 of [Coh1996].

```c
void _fmpz_poly_pseudo_divrem(fmpz *Q, fmpz *R, ulong *d, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_preinvn_t inv)
```

If $\ell$ is the leading coefficient of $B$, then computes $(Q, \text{len}A - \text{len}B + 1)$, $(R, \text{len}B - 1)$ and $d$ such that $\ell^d A = B Q + R$. This function is used for simulating division over $Q$.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that $Q$ can fit $\text{len}(A) - \text{len}(B) + 1$ coefficients, and that $R$ can fit $\text{len}(A)$ coefficients, although on exit only the bottom $\text{len}(B)$ coefficients will carry meaningful data.

Supports aliasing of $(R, \text{len}A)$ and $(A, \text{len}A)$. But other than this, no aliasing of the inputs and outputs is supported.

An optional precomputed inverse of the leading coefficient of $B$ from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be NULL.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

```c
void fmpz_poly_pseudo_divrem(fmpz_poly_t Q, fmpz_poly_t R, ulong *d, const fmpz_poly_t A, const fmpz_poly_t B)
```

Computes $Q, R,$ and $d$ such that $\ell^d A = B Q + R$.

```c
void _fmpz_poly_pseudo_div(fmpz *Q, ulong *d, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_preinvn_t inv)
```

Pseudo-division, only returning the quotient.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

```c
void fmpz_poly_pseudo_div(fmpz_poly_t Q, ulong *d, const fmpz_poly_t A, const fmpz_poly_t B)
```

Pseudo-division, only returning the quotient.

```c
void _fmpz_poly_pseudo_rem(fmpz *R, ulong *d, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_preinvn_t inv)
```

Pseudo-division, only returning the remainder.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

```c
void fmpz_poly_pseudo_rem(fmpz_poly_t R, ulong *d, const fmpz_poly_t A, const fmpz_poly_t B)
```

Pseudo-division, only returning the remainder.
4.7.30 Derivative

void _fmpz_poly_derivative(fmpz *rpoly, const fmpz *poly, slong len)
Sets (rpoly, len - 1) to the derivative of (poly, len). Also handles the cases where len is 0 or 1 correctly. Supports aliasing of rpoly and poly.

void fmpz_poly_derivative(fmpz_poly_t res, const fmpz_poly_t poly)
Sets res to the derivative of poly.

void _fmpz_poly_nth_derivative(fmpz *rpoly, const fmpz *poly, ulong n, slong len)
Sets (rpoly, len - n) to the nth derivative of (poly, len). Also handles the cases where len <= n correctly. Supports aliasing of rpoly and poly.

void fmpz_poly_nth_derivative(fmpz_poly_t res, const fmpz_poly_t poly, ulong n)
Sets res to the nth derivative of poly.

4.7.31 Evaluation

void _fmpz_poly_evaluate_divconquer_fmpz(fmpz_t res, const fmpz *f, slong len, const fmpz_t a)
Evaluates the polynomial (f, len) at the integer a using a divide and conquer approach. Assumes that the length of the polynomial is at least one. Allows zero padding. Does not allow aliasing between res and x.

void fmpz_poly_evaluate_divconquer_fmpz(fmpz_t res, const fmpz_poly_t f, const fmpz_t a)
Evaluates the polynomial f at the integer a using a divide and conquer approach.

void _fmpz_poly_evaluate_horner_fmpz(fmpz_t res, const fmpz *f, slong len, const fmpz_t a)
Evaluates the polynomial f at the integer a using Horner’s rule, and sets res to the result.

As expected, aliasing between res and a is supported. However, res may not be aliased with a coefficient of f.

void _fmpz_poly_evaluate_fmpz(fmpz_t res, const fmpz *f, slong len, const fmpz_t a)
Evaluates the polynomial f at the integer a and sets res to the result. Aliasing between res and a or any of the coefficients of f is not supported.

void fmpz_poly_evaluate_fmpz(fmpz_t res, const fmpz_poly_t f, const fmpz_t a)
Evaluates the polynomial f at the integer a and sets res to the result.

As expected, aliasing between res and a is supported. However, res may not be aliased with a coefficient of f.

void _fmpz_poly_evaluate_divconquer_fmpq(fmpz_t rnum, fmpz_t rden, const fmpz *f, slong len, const fmpz_t anum, const fmpz_t aden)
Evaluates the polynomial (f, len) at the rational (anum, aden) using a divide and conquer approach, and sets (rnum, rden) to the result in lowest terms. Assumes that the length of the polynomial is at least one.

Aliasing between (rnum, rden) and (anum, aden) or any of the coefficients of f is not supported.

void fmpz_poly_evaluate_divconquer_fmpq(fmpq_t res, const fmpz_poly_t f, const fmpq_t a)
Evaluates the polynomial f at the rational a using a divide and conquer approach, and sets res to the result.
void _fmpz_poly_evaluate_horner_fmpq(fmpz_t rnum, fmpz_t rden, const fmpz *f, slong len, const fmpz_t anum, const fmpz_t aden)

Evaluates the polynomial \( f \), at the rational \( (anum, aden) \) using Horner’s rule, and sets \( (rnum, rden) \) to the result in lowest terms.

Aliasing between \( (rnum, rden) \) and \( (anum, aden) \) or any of the coefficients of \( f \) is not supported.

void fmpz_poly_evaluate_horner_fmpq(fmpq_t res, const fmpz_poly_t f, const fmpq_t a)

Evaluates the polynomial \( f \) at the rational \( a \) using Horner’s rule, and sets \( res \) to the result.

void _fmpz_poly_evaluate_fmpq(fmpz_t rnum, fmpz_t rden, const fmpz *f, slong len, const fmpz_t anum, const fmpz_t aden)

Evaluates the polynomial \( f \), at the rational \( (anum, aden) \) and sets \( (rnum, rden) \) to the result in lowest terms.

Aliasing between \( (rnum, rden) \) and \( (anum, aden) \) or any of the coefficients of \( f \) is not supported.

void fmpz_poly_evaluate_fmpq(fmpq_t res, const fmpz_poly_t f, const fmpq_t a)

Evaluates the polynomial \( f \) at the rational \( a \), and sets \( res \) to the result.

mp_limb_t _fmpz_poly_evaluate_mod(const fmpz *poly, slong len, mp_limb_t a, mp_limb_t n, mp_limb_t ninv)

Evaluates \( (poly, len) \) at the value \( a \) modulo \( n \) and returns the result. The last argument \( ninv \) must be set to the precomputed inverse of \( n \), which can be obtained using the function \( n\_preinvert\_limb() \).

mp_limb_t fmpz_poly_evaluate_mod(const fmpz_poly_t poly, mp_limb_t a, mp_limb_t n)

Evaluates \( poly \) at the value \( a \) modulo \( n \) and returns the result.

void fmpz_poly_evaluate_fmpz_vec(fmpz *res, const fmpz_poly_t f, const fmpz *a, slong n)

Evaluates \( f \) at the \( n \) values given in the vector \( f \), writing the results to \( res \).

double _fmpz_poly_evaluate_horner_d(const fmpz *poly, slong n, double d)

Evaluate \( poly \) at the double \( d \). No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in Flint for quick and dirty evaluations of polynomials with all coefficients positive.

double fmpz_poly_evaluate_horner_d(const fmpz_poly_t poly, slong n, double d)

Evaluate \( poly \) at the double \( d \). No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in Flint for quick and dirty evaluations of polynomials with all coefficients positive.

double _fmpz_poly_evaluate_horner_d_2exp(slong *exp, const fmpz *poly, slong n, double d)

Evaluate \( poly \) at \( d*2^{exp} \). Return the result as a double and an exponent \( exp \) combination. No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in Flint for quick and dirty evaluations of polynomials with all coefficients positive.

double fmpz_poly_evaluate_horner_d_2exp(slong *exp, const fmpz_poly_t poly, double d)

Evaluate \( poly \) at the double \( d \). Return the result as a double and an exponent \( exp \) combination. No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in Flint for quick and dirty evaluations of polynomials with all coefficients positive.

double _fmpz_poly_evaluate_horner_d_2exp2(slong *exp, const fmpz *poly, slong n, double d, slong dexp)

Evaluate \( poly \) at \( d*2^{dexp} \). Return the result as a double and an exponent \( exp \) combination. No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in Flint for quick and dirty evaluations of polynomials with all coefficients positive.
4.7.32 Newton basis

**void _fmpz_poly_monomial_to_newton**(*fmpz* poly, const *fmpz* *roots*, *slong* n)

Converts (poly, n) in-place from its coefficients given in the standard monomial basis to the Newton basis for the roots $r_0, r_1, \ldots, r_{n-2}$. In other words, this determines output coefficients $c_i$ such that $c_0 + c_1(x - r_0) + c_2(x - r_0)(x - r_1) + \cdots + c_{n-1}(x - r_0)(x - r_1)\cdots(x - r_{n-2})$ is equal to the input polynomial. Uses repeated polynomial division.

**void _fmpz_poly_newton_to_monomial**(*fmpz* poly, const *fmpz* *roots*, *slong* n)

Converts (poly, n) in-place from its coefficients given in the Newton basis for the roots $r_0, r_1, \ldots, r_{n-2}$ to the standard monomial basis. In other words, this evaluates $c_0 + c_1(x - r_0) + c_2(x - r_0)(x - r_1) + \cdots + c_{n-1}(x - r_0)(x - r_1)\cdots(x - r_{n-2})$ where $c_i$ are the input coefficients for poly. Uses Horner’s rule.

4.7.33 Interpolation

**void fmpz_poly_interpolate_fmpz_vec**(*fmpz_poly_t* poly, const *fmpz* *xs*, const *fmpz* *ys*, *slong* n)

Sets poly to the unique interpolating polynomial of degree at most $n - 1$ satisfying $f(x_i) = y_i$ for every pair $x_i, y_i$ in xs and ys, assuming that this polynomial has integer coefficients. If an interpolating polynomial with integer coefficients does not exist, a FLINT_INEXACT exception is thrown.

It is assumed that the $x$ values are distinct.

4.7.34 Composition

**void _fmpz_poly_compose_horner**(*fmpz* *res, const *fmpz* *poly1*, *slong* len1, const *fmpz* *poly2*, *slong* len2)

Sets res to the composition of (poly1, len1) and (poly2, len2).

Assumes that res has space for $(len1 - 1)\times(len2 - 1) + 1$ coefficients. Assumes that poly1 and poly2 are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

**void fmpz_poly_compose_horner**(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets res to the composition of poly1 and poly2. To be more precise, denoting res, poly1, and poly2 by $f$, $g$, and $h$, sets $f(t) = g(h(t))$.

This implementation uses Horner’s method.

**void _fmpz_poly_compose_divconquer**(*fmpz* *res, const *fmpz* *poly1*, *slong* len1, const *fmpz* *poly2*, *slong* len2)

Computes the composition of (poly1, len1) and (poly2, len2) using a divide and conquer approach and places the result into res, assuming res can hold the output of length $(len1 - 1) \times (len2 - 1) + 1$.

Assumes len1, len2 > 0. Does not support aliasing between res and any of (poly1, len1) and (poly2, len2).

**void fmpz_poly_compose_divconquer**(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets res to the composition of poly1 and poly2. To be precise about the order of composition, denoting res, poly1, and poly2 by $f$, $g$, and $h$, respectively, sets $f(t) = g(h(t))$. 

202 Chapter 4. Integers
void \texttt{fmpz\_poly\_compose}\( (\texttt{fmpz}*\texttt{res}, \texttt{const fmpz}*\texttt{poly1}, \texttt{slong}\ \texttt{len1}, \texttt{const fmpz}*\texttt{poly2}, \texttt{slong}\ \texttt{len2}) \)

Sets \texttt{res} to the composition of \((\texttt{poly1}, \texttt{len1})\) and \((\texttt{poly2}, \texttt{len2})\).

Assumes that \texttt{res} has space for \((\texttt{len1}-1)*\texttt{(len2}-1) + 1\) coefficients. Assumes that \texttt{poly1} and \texttt{poly2} are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

void \texttt{fmpz\_poly\_compose}\( (\texttt{fmpz\_poly\_t}\ \texttt{res}, \texttt{const fmpz\_poly\_t}\ \texttt{poly1}, \texttt{const fmpz\_poly\_t}\ \texttt{poly2}) \)

Sets \texttt{res} to the composition of \texttt{poly1} and \texttt{poly2}. To be precise about the order of composition, denoting \texttt{res}, \texttt{poly1}, and \texttt{poly2} by \(f\), \(g\), and \(h\), respectively, sets \(f(t) = g(h(t))\).

### 4.7.35 Inflation and deflation

void \texttt{fmpz\_poly\_inflate}\( (\texttt{fmpz\_poly\_t}\ \texttt{result}, \texttt{const fmpz\_poly\_t}\ \texttt{input}, \texttt{ulong}\ \texttt{inflation}) \)

Sets \texttt{result} to the inflated polynomial \(p(x^n)\) where \(p\) is given by \texttt{input} and \(n\) is given by \texttt{inflation}.

void \texttt{fmpz\_poly\_deflate}\( (\texttt{fmpz\_poly\_t}\ \texttt{result}, \texttt{const fmpz\_poly\_t}\ \texttt{input}, \texttt{ulong}\ \texttt{deflation}) \)

Sets \texttt{result} to the deflated polynomial \(p(x^{1/n})\) where \(p\) is given by \texttt{input} and \(n\) is given by \texttt{deflation}. Requires \(n > 0\).

ulong \texttt{fmpz\_poly\_deflation}\( (\texttt{const fmpz\_poly\_t}\ \texttt{input}) \)

Returns the largest integer by which \texttt{input} can be deflated. As special cases, returns 0 if \texttt{input} is the zero polynomial and 1 if \texttt{input} is a constant polynomial.

### 4.7.36 Taylor shift

void \texttt{fmpz\_poly\_taylor\_shift\_horner}\( (\texttt{fmpz}\ \texttt{poly}, \texttt{const fmpz\_t}\ \texttt{c}, \texttt{slong}\ \texttt{n}) \)

Performs the Taylor shift composing \texttt{poly} by \(x + c\) in-place. Uses an efficient version Horner’s rule.

void \texttt{fmpz\_poly\_taylor\_shift\_horner}\( (\texttt{fmpz\_poly\_t}\ \texttt{g}, \texttt{const fmpz\_poly\_t}\ \texttt{f}, \texttt{const fmpz\_t}\ \texttt{c}) \)

Performs the Taylor shift composing \texttt{f} by \(x + c\).

void \texttt{fmpz\_poly\_taylor\_shift\_divconquer}\( (\texttt{fmpz}\ \texttt{poly}, \texttt{const fmpz\_t}\ \texttt{c}, \texttt{slong}\ \texttt{n}) \)

Performs the Taylor shift composing \texttt{poly} by \(x + c\) in-place. Uses the divide-and-conquer polynomial composition algorithm.

void \texttt{fmpz\_poly\_taylor\_shift\_divconquer}\( (\texttt{fmpz\_poly\_t}\ \texttt{g}, \texttt{const fmpz\_poly\_t}\ \texttt{f}, \texttt{const fmpz\_t}\ \texttt{c}) \)

Performs the Taylor shift composing \texttt{f} by \(x + c\). Uses the divide-and-conquer polynomial composition algorithm.

void \texttt{fmpz\_poly\_taylor\_shift\_multi\_mod}\( (\texttt{fmpz}\ \texttt{poly}, \texttt{const fmpz\_t}\ \texttt{c}, \texttt{slong}\ \texttt{n}) \)

Performs the Taylor shift composing \texttt{poly} by \(x + c\) in-place. Uses a multimodular algorithm, distributing the computation across \texttt{flint\_get\_num\_threads()} threads.

void \texttt{fmpz\_poly\_taylor\_shift\_multi\_mod}\( (\texttt{fmpz\_poly\_t}\ \texttt{g}, \texttt{const fmpz\_poly\_t}\ \texttt{f}, \texttt{const fmpz\_t}\ \texttt{c}) \)

Performs the Taylor shift composing \texttt{f} by \(x + c\). Uses a multimodular algorithm, distributing the computation across \texttt{flint\_get\_num\_threads()} threads.

void \texttt{fmpz\_poly\_taylor\_shift}\( (\texttt{fmpz}\ \texttt{poly}, \texttt{const fmpz\_t}\ \texttt{c}, \texttt{slong}\ \texttt{n}) \)

Performs the Taylor shift composing \texttt{poly} by \(x + c\) in-place.

void \texttt{fmpz\_poly\_taylor\_shift}\( (\texttt{fmpz\_poly\_t}\ \texttt{g}, \texttt{const fmpz\_poly\_t}\ \texttt{f}, \texttt{const fmpz\_t}\ \texttt{c}) \)

Performs the Taylor shift composing \texttt{f} by \(x + c\).
4.7.37 Power series composition

void _fmpz_poly_compose_series_horner(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)

Sets res to the composition of poly1 and poly2 modulo $x^n$, where the constant term of poly2 is required to be zero.

Assumes that len1, len2, n > 0, that len1, len2 <= n, and that (len1-1) * (len2-1) + 1 <= n, and that res has space for n coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses the Horner scheme.

void fmpz_poly_compose_series_horner(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)

Sets res to the composition of poly1 and poly2 modulo $x^n$, where the constant term of poly2 is required to be zero.

This implementation uses the Horner scheme.

void _fmpz_poly_compose_series_brent_kung(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)

Sets res to the composition of poly1 and poly2 modulo $x^n$, where the constant term of poly2 is required to be zero.

Assumes that len1, len2, n > 0, that len1, len2 <= n, and that (len1-1) * (len2-1) + 1 <= n, and that res has space for n coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses Brent-Kung algorithm 2.1 [BrentKung1978].

void fmpz_poly_compose_series_brent_kung(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)

Sets res to the composition of poly1 and poly2 modulo $x^n$, where the constant term of poly2 is required to be zero.

This implementation uses Brent-Kung algorithm 2.1 [BrentKung1978].

void _fmpz_poly_compose_series(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)

Sets res to the composition of poly1 and poly2 modulo $x^n$, where the constant term of poly2 is required to be zero.

Assumes that len1, len2, n > 0, that len1, len2 <= n, and that (len1-1) * (len2-1) + 1 <= n, and that res has space for n coefficients. Does not support aliasing between any of the inputs and the output.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs.

void fmpz_poly_compose_series(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)

Sets res to the composition of poly1 and poly2 modulo $x^n$, where the constant term of poly2 is required to be zero.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs.
4.7.38 Power series reversion

```c
void _fmpz_poly_revert_series(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n)
(void fmpz_poly_revert_series(fmpz_poly_t Qinv, const fmpz_poly_t Q, slong n))
```

Sets Qinv to the compositional inverse or reversion of Q as a power series, i.e. computes \( Q^{-1}(x) \) such that \( Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \mod x^n \). It is required that \( Q_0 = 0 \) and \( Q_1 = \pm 1 \).

Wraps \_gr_poly_revert_series() which chooses automatically between various algorithms.

4.7.39 Square root

```c
int _fmpz_poly_sqrtrem_classical(fmpz *res, fmpz *r, const fmpz *poly, slong len)
```

Returns 1 if \((poly, len)\) can be written in the form \( A^2 + R \) where \( \deg(R) < \deg(poly) \), otherwise returns 0. If it can be so written, \((res, m - 1)\) is set to \( A \) and \((res, m)\) is set to \( R \), where \( m = \deg(poly)/2 + 1 \).

For efficiency reasons, \( r \) must have room for \( len \) coefficients, and may alias \( poly \).

```c
int fmpz_poly_sqrtrem_classical(fmpz_poly_t b, fmpz_poly_t r, const fmpz_poly_t a)
```

If \( a \) can be written as \( b^2 + r \) with \( \deg(r) < \deg(a)/2 \), return 1 and set \( b \) and \( r \) appropriately. Otherwise return 0.

```c
int _fmpz_poly_sqrtrem_divconquer(fmpz *res, fmpz *r, const fmpz *poly, slong len, fmpz *temp)
```

Returns 1 if \((poly, len)\) can be written in the form \( A^2 + R \) where \( \deg(R) < \deg(poly) \), otherwise returns 0. If it can be so written, \((res, m - 1)\) is set to \( A \) and \((res, m)\) is set to \( R \), where \( m = \deg(poly)/2 + 1 \).

For efficiency reasons, \( r \) must have room for \( len \) coefficients, and may alias \( poly \). Temporary space of \( len \) coefficients is required.

```c
int fmpz_poly_sqrtrem_divconquer(fmpz_poly_t b, fmpz_poly_t r, const fmpz_poly_t a)
```

If \( a \) can be written as \( b^2 + r \) with \( \deg(r) < \deg(a)/2 \), return 1 and set \( b \) and \( r \) appropriately. Otherwise return 0.

```c
int _fmpz_poly_sqrt_classical(fmpz *res, const fmpz *poly, slong len, int exact)
```

If \( exact \) is 1 and \((poly, len)\) is a perfect square, sets \((res, len / 2 + 1)\) to the square root of poly with positive leading coefficient and returns 1. Otherwise returns 0.

If \( exact \) is 0, allows a remainder after the square root, which is not computed.

This function first uses various tests to detect nonsquares quickly. Then, it computes the square root iteratively from top to bottom, requiring \( O(n^2) \) coefficient operations.

```c
int fmpz_poly_sqrt_classical(fmpz_poly_t b, const fmpz_poly_t a)
```

If \( a \) is a perfect square, sets \( b \) to the square root of \( a \) with positive leading coefficient and returns 1. Otherwise returns 0.

```c
int _fmpz_poly_sqrt_KS(fmpz *res, const fmpz *poly, slong len)
```

Heuristic square root. If the return value is \(-1\), the function failed, otherwise it succeeded and the following applies.

If \((poly, len)\) is a perfect square, sets \((res, len / 2 + 1)\) to the square root of poly with positive leading coefficient and returns 1. Otherwise returns 0.

This function first uses various tests to detect nonsquares quickly. Then, it computes the square root iteratively from top to bottom.

```c
int fmpz_poly_sqrt_KS(fmpz_poly_t b, const fmpz_poly_t a)
```

Heuristic square root. If the return value is \(-1\), the function failed, otherwise it succeeded and the following applies.

If \( a \) is a perfect square, sets \( b \) to the square root of \( a \) with positive leading coefficient and returns 1. Otherwise returns 0.
int _fmpz_poly_sqrt_divconquer(fmpz *res, const fmpz *poly, slong len, int exact)
   If exact is 1 and (poly, len) is a perfect square, sets (res, len / 2 + 1) to the square root of poly with positive leading coefficient and returns 1. Otherwise returns 0.
   If exact is 0, allows a remainder after the square root, which is not computed.
   This function first uses various tests to detect nonsquares quickly. Then, it computes the square root iteratively from top to bottom.

int fmpz_poly_sqrt_divconquer(fmpz_poly_t b, const fmpz_poly_t a)
   If a is a perfect square, sets b to the square root of a with positive leading coefficient and returns 1. Otherwise returns 0.

int _fmpz_poly_sqrt(fmpz *res, const fmpz *poly, slong len)
   If (poly, len) is a perfect square, sets (res, len / 2 + 1) to the square root of poly with positive leading coefficient and returns 1. Otherwise returns 0.

int fmpz_poly_sqrt(fmpz_poly_t b, const fmpz_poly_t a)
   If a is a perfect square, sets b to the square root of a with positive leading coefficient and returns 1. Otherwise returns 0.

int _fmpz_poly_sqrt_series(fmpz *res, const fmpz *poly, slong len, slong n)
   Set (res, n) to the square root of the series (poly, n), if it exists, and return 1, otherwise, return 0.
   If the valuation of poly is not zero, res is zero padded to make up for the fact that the square root may not be known to precision n.

int fmpz_poly_sqrt_series(fmpz_poly_t b, const fmpz_poly_t a, slong n)
   Set b to the square root of the series a, where the latter is taken to be a series of precision n. If such a square root exists, return 1, otherwise, return 0.
   Note that if the valuation of a is not zero, b will not have precision n. It is given only to the precision to which the square root can be computed.

4.7.40 Power sums

void _fmpz_poly_power_sums_naive(fmpz *res, const fmpz *poly, slong len, slong n)
   Compute the (truncated) power sums series of the monic polynomial (poly,len) up to length n using Newton identities.

void fmpz_poly_power_sums_naive(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
   Compute the (truncated) power sum series of the monic polynomial poly up to length n using Newton identities.

void fmpz_poly_power_sums(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
   Compute the (truncated) power sums series of the monic polynomial poly up to length n. That is the power series whose coefficient of degree i is the sum of the i-th power of all (complex) roots of the polynomial poly.

void _fmpz_poly_power_sums_to_poly(fmpz *res, const fmpz *poly, slong len)
   Compute the (monic) polynomial given by its power sums series (poly,len).

void fmpz_poly_power_sums_to_poly(fmpz_poly_t res, const fmpz_poly_t Q)
   Compute the (monic) polynomial given its power sums series (Q).
4.7.41 Signature

```c
void _fmpz_poly_signature(slong *r1, slong *r2, const fmpz_poly *poly, slong len)
```

Computes the signature \((r_1, r_2)\) of the polynomial \((\text{poly}, \text{len})\). Assumes that the polynomial is squarefree over \(\mathbb{Q}\).

```c
void fmpz_poly_signature(slong *r1, slong *r2, const fmpz_poly_t poly)
```

Computes the signature \((r_1, r_2)\) of the polynomial \(\text{poly}\), which is assumed to be square-free over \(\mathbb{Q}\). The values of \(r_1\) and \(2r_2\) are the number of real and complex roots of the polynomial, respectively.

For convenience, the zero polynomial is allowed, in which case the output is \((0, 0)\).

If the polynomial is not square-free, the behaviour is undefined and an exception may be raised.

This function uses the algorithm described in Algorithm 4.1.11 of [Coh1996].

4.7.42 Hensel lifting

```c
void fmpz_poly_hensel_build_tree(slong *link, fmpz_poly_t *v, fmpz_poly_t *w, const nmod_poly_factor_t *fac)
```

Initialises and builds a Hensel tree consisting of two arrays \(v, w\) of polynomials and an array of links, called \(\text{link}\).

The caller supplies a set of \(r\) local factors (in the factor structure \(\text{fac}\)) of some polynomial \(F\) over \(\mathbb{Z}\). They also supply two arrays of initialised polynomials \(v\) and \(w\), each of length \(2r - 2\) and an array \(\text{link}\), also of length \(2r - 2\).

We will have five arrays: a \(v\) of \(\text{fmpz\_poly\_t}\)'s and a \(V\) of \(\text{nmod\_poly\_t}\)'s and also a \(w\) and a \(W\) and \(\text{link}\). Here’s the idea: we sort each leaf and node of a factor tree by degree, in fact choosing to multiply the two smallest factors, then the next two smallest (factors or products) etc. until a tree is made. The tree will be stored in the \(v\)'s. The first two elements of \(v\) will be the smallest modular factors, the last two elements of \(v\) will multiply to form \(F\) itself. Since \(v\) will be rearranging the original factors we will need to be able to recover the original order. For this we use the array \(\text{link}\) which has nonnegative even numbers and negative numbers. It is an array of \(\text{slongs}\) which aligns with \(V\) and \(v\) if \(\text{link}\) has a negative number in spot \(j\) that means \(V_j\) is an original modular factor which has been lifted, if \(\text{link}[j]\) is a nonnegative even number then \(V_j\) stores a product of the two entries at \(V[\text{link}[j]]\) and \(V[\text{link}[j]+1]\). \(W\) and \(w\) play the role of the extended GCD, at \(V_0, V_2, V_4, \text{etc.}\) we have a new product, \(W_0, W_2, W_4, \text{etc.}\) are the XGCD cofactors of the \(V\)'s. For example, \(V_0W_0 + V_1W_1 \equiv 1 \pmod{p^\ell}\) for some \(\ell\). These will be lifted along with the entries in \(V\). It is not enough to just lift each factor, we have to lift the entire tree and the tree of XGCD cofactors.

```c
void fmpz_poly_hensel_lift(fmpz_poly_t G, fmpz_poly_t H, fmpz_poly_t A, fmpz_poly_t B, const fmpz_poly_t f, const fmpz_poly_t g, const fmpz_poly_t h, const fmpz_poly_t p, const fmpz_poly_t p1)
```

This is the main Hensel lifting routine, which performs a Hensel step from polynomials \(p\) to polynomials \(P = pp_1\). One starts with polynomials \(f, g, h\) such that \(f = gh \pmod{p}\). The polynomials \(a, b\) satisfy \(ag + bh = 1 \pmod{p}\).

The lifting formulae are

\[
G = \left(\frac{f - gh}{p}\right)b \pmod{g} + g
\]

\[
H = \left(\frac{f - gh}{p}\right)a \pmod{h} + h
\]

\[
B = \left(\frac{1 - aG - bH}{p}\right)b \pmod{g} + b
\]

\[
A = \left(\frac{1 - aG - bH}{p}\right)a \pmod{h} + a
\]
Upon return we have \( AG + BH = 1 \) (mod \( P \)) and \( f = GH \) (mod \( P \)), where \( G = g \) (mod \( p \)) etc.

We require that \( 1 < p_1 \leq p \) and that the input polynomials \( f, g, h \) have degree at least 1 and that the input polynomials \( a \) and \( b \) are non-zero.

The output arguments \( G, H, A, B \) may only be aliased with the input arguments \( g, h, a, b \), respectively.

```c
void fmpz_poly_hensel_lift_without_inverse(fmpz_poly_t Gout, fmpz_poly_t Hout, const fmpz_poly_t f, const fmpz_poly_t g, const fmpz_poly_t h, const fmpz_poly_t a, const fmpz_poly_t b, const fmpz_poly_t p, const fmpz_poly_t p1)
```

Given polynomials such that \( f = gh \) (mod \( p \)) and \( ag + bh = 1 \) (mod \( p \)), lifts only the factors \( g \) and \( h \) modulo \( P = pp_1 \).

See `fmpz_poly_hensel_lift()`.  

```c
void fmpz_poly_hensel_lift_only_inverse(fmpz_poly_t Aout, fmpz_poly_t Bout, const fmpz_poly_t G, const fmpz_poly_t H, const fmpz_poly_t a, const fmpz_poly_t b, const fmpz_poly_t p, const fmpz_poly_t p1)
```

Given polynomials such that \( f = gh \) (mod \( p \)) and \( ag + bh = 1 \) (mod \( p \)), lifts only the cofactors \( a \) and \( b \) modulo \( P = pp_1 \).

See `fmpz_poly_hensel_lift()`.  

```c
void fmpz_poly_hensel_lift_tree_recursive(slong *link, fmpz_poly_t *v, fmpz_poly_t *w, fmpz_poly_t f, slong j, slong inv, const fmpz_poly_t p0, const fmpz_poly_t p1)
```

Takes a current Hensel tree \((\text{link}, v, w)\) and a pair \((j, j+1)\) of entries in the tree and lifts the tree from mod \( p_0 \) to mod \( P = pp_0 p_1 \), where \( 1 < p_1 \leq p_0 \).

Set \( \text{inv} \) to \(-1\) if restarting Hensel lifting, \( 0 \) if stopping and \( 1 \) otherwise.

Here \( f = gh \) is the polynomial whose factors we are trying to lift. We will have that \( v[j] \) is the product of \( v[\text{link}[j]] \) and \( v[\text{link}[j] + 1] \) as described above.

Does support aliasing of \( f \) with one of the polynomials in the lists \( v \) and \( w \). But the polynomials in these two lists are not allowed to be aliases of each other.

```c
void fmpz_poly_hensel_lift_tree(slong *link, fmpz_poly_t *v, fmpz_poly_t *w, fmpz_poly_t f, slong r, const fmpz_poly_t p, slong e0, slong e1, slong inv)
```

Computes \( p_0 = p^{e_0} \) and \( p_1 = p^{e_1-e_0} \) for a small prime \( p \) and \( P = p^{e_1} \).

If we aim to lift to \( p^h \) then \( f \) is the polynomial whose factors we wish to lift, made monic mod \( p^h \). As usual, \((\text{link}, v, w)\) is an initialised tree.

This starts the recursion on lifting the product tree for lifting from \( p^{e_0} \) to \( p^{e_1} \). The value of \( \text{inv} \) corresponds to that given for the function `fmpz_poly_hensel_lift_tree_recursive()`.

Set \( r \) to the number of local factors of \( f \).

In terms of the notation, above \( P = p^{e_1} \), \( p_0 = p^{e_0} \) and \( p_1 = p^{e_1-e_0} \).

Assumes that \( f \) is monic.

Assumes that \( 1 < p_1 \leq p_0 \), that is, \( 0 < e_1 \leq e_0 \).

```c
slong _fmpz_poly_hensel_start_lift(fmpz_poly_factor_t lifted_fac, slong *link, fmpz_poly_t *v, fmpz_poly_t *w, const fmpz_poly_t f, const nmod_poly_factor_t local_fac, slong N)
```

This function takes the local factors in \( \text{local_fac} \) and Hensel lifts them until they are known mod \( p^N \), where \( N \geq 1 \).

These lifted factors will be stored (in the same ordering) in \( \text{lifted_fac} \). It is assumed that \( \text{link} \), \( v \), and \( w \) are initialized arrays of \( fmpz_poly_t \)'s with at least \( 2 + r - 2 \) entries and that \( r \geq 2 \). This
is done outside of this function so that you can keep them for restarting Hensel lifting later. The product of local factors must be squarefree.

The return value is an exponent which must be passed to the function 
\texttt{fmpz\_poly\_hensel\_continue\_lift()} as \texttt{prev\_exp} if the Hensel lifting is to be resumed.

Currently, supports the case when \( N = 1 \) for convenience, although it is preferable in this case to simply iterate over the local factors and convert them to polynomials over \( \mathbb{Z} \).

\begin{verbatim}
slong \_fmpz\_poly\_hensel\_continue\_lift(fmpz\_poly\_factor\_t lifted\_fac, slong *link, fmpz\_poly\_t \*v, fmpz\_poly\_t \*w, const fmpz\_poly\_t f, slong prev, slong curr, slong N, const fmpz\_t p)
\end{verbatim}

This function restarts a stopped Hensel lift. It lifts from \( \texttt{curr} \) to \( N \). It also requires \texttt{prev} (to lift the cofactors) given as the return value of the function \texttt{fmpz\_poly\_hensel\_start\_lift()} or the function \texttt{fmpz\_poly\_hensel\_continue\_lift()}. The current lifted factors are supplied in \texttt{lifted\_fac} and upon return are updated there. As usual \texttt{link}, \texttt{v}, and \texttt{w} describe the current Hensel tree, \( r \) is the number of local factors and \( p \) is the small prime modulo whose power we are lifting to. It is required that \texttt{curr} be at least 1 and that \( N > \texttt{curr} \).

Currently, supports the case when \texttt{prev} and \texttt{curr} are equal.

\begin{verbatim}
void fmpz\_poly\_hensel\_lift\_once(fmpz\_poly\_factor\_t lifted\_fac, const fmpz\_poly\_t f, const nmod\_poly\_factor\_t local\_fac, slong N)
\end{verbatim}

This function does a Hensel lift. It lifts local factors stored in \texttt{local\_fac} of \( f \) to \( p^N \), where \( N \geq 2 \). The lifted factors will be stored in \texttt{lifted\_fac}. This lift cannot be restarted. This function is a convenience function intended for end users. The product of local factors must be squarefree.

### 4.7.43 Input and output

The functions in this section are not intended to be particularly fast. They are intended mainly as a debugging aid.

For the string output functions there are two variants. The first uses a simple string representation of polynomials which prints only the length of the polynomial and the integer coefficients, whilst the latter variant, appended with \_pretty, uses a more traditional string representation of polynomials which prints a variable name as part of the representation.

The first string representation is given by a sequence of integers, in decimal notation, separated by white space. The first integer gives the length of the polynomial; the remaining integers are the coefficients. For example \( 5x^3 - x + 1 \) is represented by the string "4 1 -1 0 5", and the zero polynomial is represented by "0". The coefficients may be signed and arbitrary precision.

The string representation of the functions appended by \_pretty includes only the non-zero terms of the polynomial, starting with the one of highest degree. Each term starts with a coefficient, prepended with a sign, followed by the character *, followed by a variable name, which must be passed as a string parameter to the function, followed by a caret ^ followed by a non-negative exponent.

If the sign of the leading coefficient is positive, it is omitted. Also the exponents of the degree 1 and 0 terms are omitted, as is the variable and the * character in the case of the degree 0 coefficient. If the coefficient is plus or minus one, the coefficient is omitted, except for the sign.

Some examples of the \_pretty representation are:

\begin{verbatim}
5\*x^3+7\*x-4
x^2+3
-x^4+2\*x-1
x+1
5
\end{verbatim}
int _fmpz_poly_print(const fmpz *poly, slong len)
    Prints the polynomial (poly, len) to stdout.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fmpz_poly_print(const fmpz_poly_t poly)
    Prints the polynomial to stdout.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fmpz_poly_print_pretty(const fmpz *poly, slong len, const char *x)
    Prints the pretty representation of (poly, len) to stdout, using the string x to represent the indeterminate.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fmpz_poly_print_pretty(const fmpz_poly_t poly, const char *x)
    Prints the pretty representation of poly to stdout, using the string x to represent the indeterminate.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fmpz_poly_fprint(FILE *file, const fmpz *poly, slong len)
    Prints the polynomial (poly, len) to the stream file.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fmpz_poly_fprint(FILE *file, const fmpz_poly_t poly)
    Prints the polynomial to the stream file.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fmpz_poly_fprint_pretty(FILE *file, const fmpz *poly, slong len, const char *x)
    Prints the pretty representation of (poly, len) to the stream file, using the string x to represent the indeterminate.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fmpz_poly_fprint_pretty(FILE *file, const fmpz_poly_t poly, const char *x)
    Prints the pretty representation of poly to the stream file, using the string x to represent the indeterminate.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fmpz_poly_read(fmpz_poly_t poly)
    Reads a polynomial from stdin, storing the result in poly.
    In case of success, returns a positive number. In case of failure, returns a non-positive value.

int fmpz_poly_read_pretty(fmpz_poly_t poly, char **x)
    Reads a polynomial in pretty format from stdin.
    For further details, see the documentation for the function fmpz_poly_fread_pretty().

int fmpz_poly_fread(FILE *file, fmpz_poly_t poly)
    Reads a polynomial from the stream file, storing the result in poly.
    In case of success, returns a positive number. In case of failure, returns a non-positive value.

int fmpz_poly_fread_pretty(FILE *file, fmpz_poly_t poly, char **x)
    Reads a polynomial from the file file and sets poly to this polynomial. The string *x is set to the variable name that is used in the input.
    Returns a positive value, equal to the number of characters read from the file, in case of success. Returns a non-positive value in case of failure, which could either be a read error or the indicator of a malformed input.
4.7.44 Modular reduction and reconstruction

void fmpz_poly_get_nmod_poly(nmod_poly_t Amod, const fmpz_poly_t A)
  Sets the coefficients of Amod to the coefficients in A, reduced by the modulus of Amod.

void fmpz_poly_set_nmod_poly(fmpz_poly_t A, const nmod_poly_t Amod)
  Sets the coefficients of A to the residues in Amod, normalised to the interval \(-m/2 \leq r < m/2\) where \(m\) is the modulus.

void fmpz_poly_set_nmod_poly_unsigned(fmpz_poly_t A, const nmod_poly_t Amod)
  Sets the coefficients of A to the residues in Amod, normalised to the interval \(0 \leq r < m\) where \(m\) is the modulus.

void _fmpz_poly_CRT_ui_precomp(fmpz *res, const fmpz *poly1, slong len1, const fmpz_t m1, mp_srcptr poly2, slong len2, mp_limb_t m2, mp_limb_t m2inv, fmpz_t m1m2, mp_limb_t c, int sign)
  Sets the coefficients in res to the CRT reconstruction modulo \(m_1m_2\) of the residues (poly1, len1) and (poly2, len2) which are images modulo \(m_1\) and \(m_2\) respectively. The caller must supply the precomputed product of the input moduli as \(m_1m_2\), the inverse of \(m_1\) modulo \(m_2\) as c, and the precomputed inverse of \(m_2\) (in the form computed by n_preinvert_limb) as m2inv.

If sign = 0, residues \(0 \leq r < m_1m_2\) are computed, while if sign = 1, residues \(-m_1m_2/2 \leq r < m_1m_2/2\) are computed.

Coefficients of res are written up to the maximum of len1 and len2.

void _fmpz_poly_CRT_ui(fmpz *res, const fmpz *poly1, slong len1, const fmpz_t m1, mp_srcptr poly2, slong len2, mp_limb_t m2, mp_limb_t m2inv, int sign)
  This function is identical to _fmpz_poly_CRT_ui_precomp, apart from automatically computing \(m_1m_2\) and c. It also aborts if c cannot be computed.

void fmpz_poly_CRT_ui(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_t m, const nmod_poly_t poly2, int sign)
  Given poly1 with coefficients modulo m and poly2 with modulus n, sets res to the CRT reconstruction modulo \(mn\) with coefficients satisfying \(-mn/2 \leq c < mn/2\) (if sign = 1) or \(0 \leq c < mn\) (if sign = 0).

4.7.45 Products

void _fmpz_poly_product_roots_fmpz_vec(fmpz *poly, const fmpz *xs, slong n)
  Sets (poly, \(n + 1\)) to the monic polynomial which is the product of \((x - x_0)(x - x_1) \cdots (x - x_{n-1})\), the roots \(x_i\) being given by xs.

Aliasing of the input and output is not allowed.

void fmpz_poly_product_roots_fmpz_vec(fmpz_poly_t poly, const fmpz *xs, slong n)
  Sets poly to the monic polynomial which is the product of \((x - x_0)(x - x_1) \cdots (x - x_{n-1})\), the roots \(x_i\) being given by xs.

void _fmpz_poly_product_roots_fmpq_vec(fmpz *poly, const fmpq *xs, slong n)
  Sets (poly, \(n + 1\)) to the product of \((p_0x - p_0)(q_1x - p_1) \cdots (q_{n-1}x - p_{n-1})\), the roots \(p_i/q_i\) being given by xs.

void fmpz_poly_product_roots_fmpq_vec(fmpz_poly_t poly, const fmpq *xs, slong n)
  Sets poly to the polynomial which is the product of \((p_0x - p_0)(q_1x - p_1) \cdots (q_{n-1}x - p_{n-1})\), the roots \(p_i/q_i\) being given by xs.
4.7.46 Roots

void _fmpz_poly_bound_roots(fmpz_t bound, const fmpz *poly, slong len)

void fmpz_poly_bound_roots(fmpz_t bound, const fmpz_poly_t poly)

Computes a nonnegative integer bound that bounds the absolute value of all complex roots of poly.

Uses Fujiwara’s bound

\[ 2 \max \left( \left| \frac{a_{n-1}}{a_n} \right|, \left| \frac{a_{n-2}}{a_n} \right|^{\frac{1}{2}}, \ldots, \left| \frac{a_1}{a_n} \right|^{\frac{1}{n}}, \left| \frac{a_0}{2a_n} \right|^{\frac{1}{n}} \right) \]

where the coefficients of the polynomial are \( a_0, \ldots, a_n \).

void _fmpz_poly_num_real_roots_sturm(slong *n_neg, slong *n_pos, const fmpz *pol, slong len)

Sets \( n_{\text{neg}} \) and \( n_{\text{pos}} \) to the number of negative and positive roots of the polynomial \( (\text{pol}, \text{len}) \) using Sturm sequence. The Sturm sequence is computed via subresultant remainders obtained by repeated call to the function _fmpz_poly_pseudo_rem_cohen.

The polynomial is assumed to be squarefree, of degree larger than 1 and with non-zero constant coefficient.

slong fmpz_poly_num_real_roots_sturm(const fmpz_poly_t pol)

Returns the number of real roots of the squarefree polynomial \( \text{pol} \) using Sturm sequence.

The polynomial is assumed to be squarefree.

slong _fmpz_poly_num_real_roots(const fmpz *pol, slong len)

Returns the number of real roots of the squarefree polynomial \( \text{pol}, \text{len} \).

The polynomial is assumed to be squarefree.

4.7.47 Minimal polynomials

void _fmpz_poly_cyclotomic(fmpz *a, ulong n, mp_ptr factors, slong num_factors, ulong phi)

Sets \( a \) to the lower half of the cyclotomic polynomial \( \Phi_n(x) \), given \( n \geq 3 \) which must be squarefree.

A precomputed array containing the prime factors of \( n \) must be provided, as well as the value of the Euler totient function \( \phi(n) \) as phi. If \( n \) is even, 2 must be the first factor in the list.

The degree of \( \Phi_n(x) \) is exactly \( \phi(n) \). Only the low \( \phi(n) + 1 \) coefficients are written; the high coefficients can be obtained afterwards by copying the low coefficients in reverse order, since \( \Phi_n(x) \) is a palindrome for \( n \neq 1 \).

We use the sparse power series algorithm described as Algorithm 4 [ArnoldMonagan2011]. The algorithm is based on the identity

\[ \Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)}. \]

Treating the polynomial as a power series, the multiplications and divisions can be done very cheaply using repeated additions and subtractions. The complexity is \( O(2^k \phi(n)) \) where \( k \) is the number of prime factors in \( n \).

To improve efficiency for small \( n \), we treat the fmpz coefficients as machine integers when there is no risk of overflow. The following bounds are given in Table 6 of [ArnoldMonagan2011]:

For \( n < 10163195 \), the largest coefficient in any \( \Phi_n(x) \) has 27 bits, so machine arithmetic is safe on 32 bits.
For $n < 169828113$, the largest coefficient in any $\Phi_n(x)$ has 60 bits, so machine arithmetic is safe on 64 bits.

Further, the coefficients are always $\pm 1$ or 0 if there are exactly two prime factors, so in this case machine arithmetic can be used as well.

Finally, we handle two special cases: if there is exactly one prime factor $n = p$, then $\Phi_n(x) = 1 + x + x^2 + \ldots + x^{n-1}$, and if $n = 2m$, we use $\Phi_n(x) = \Phi_m(-x)$ to fall back to the case when $n$ is odd.

```c
void fmpz_poly_chebyshev_u(fmpz_poly_t poly, ulong n)
```

Sets poly to the $n$-th Chebyshev polynomial of the first kind $T_n(x)$, defined by $T_n(x) = \cos(n \cos^{-1}(x))$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence.

```c
void _fmpz_poly_chebyshev_t(fmpz_poly_t poly, ulong n)
```

Sets poly to the Chebyshev polynomial of the first kind $T_n(x)$, defined by $T_n(x) = \cos(n \cos^{-1}(x))$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence.

```c
void fmpz_poly_chebyshev_u(fmpz_poly_t poly, ulong n)
```

Sets poly to the Chebyshev polynomial of the first kind $U_n(x)$, defined by $(n+1)U_n(x) = T_{n+1}(x)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence.

```c
void _fmpz_poly_legendre_pt(fmpz_poly_t poly, ulong n)
```

Sets coeffs to the coefficient array of the shifted Legendre polynomial $\tilde{P}_n(x)$, defined by $\tilde{P}_n(x) = P_n(2x - 1)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence. The length of the array will be $n+1$. See fmpz_poly for the Legendre polynomials.

### 4.7.48 Orthogonal polynomials

```c
ulong fmpz_poly_is_cyclotomic(const fmpz *poly, ulong len)
```

If poly is a cyclotomic polynomial, returns the index $n$ of this cyclotomic polynomial. If poly is not a cyclotomic polynomial, returns 0.

```c
ulong _fmpz_poly_is_cyclotomic(fmpz *poly, ulong n)
```

```c
void fmpz_poly_is_cyclotomic(const fmpz_poly_t poly)
```

If poly is a cyclotomic polynomial, returns the index $n$ of this cyclotomic polynomial. If poly is not a cyclotomic polynomial, returns 0.

```c
void _fmpz_poly_cos_minpoly(fmpz *coeffs, ulong n)
```

```c
void fmpz_poly_cos_minpoly(fmpz_poly_t poly, ulong n)
```

Sets poly to the minimal polynomial of $\cos(2\pi/n)$. For suitable choice of $n$, this gives the minimal polynomial of $2\cos(a\pi)$ or $2\sin(a\pi)$ for any rational $a$.

The cosine is multiplied by a factor two since this gives a monic polynomial with integer coefficients. One can obtain the minimal polynomial for $\cos(2\pi/n)$ by making the substitution $x \to x/2$.

For $n > 2$, the degree of the polynomial is $\varphi(n)/2$. For $n = 1, 2$, the degree is 1. For $n = 0$, we define the output to be the constant polynomial 1.

See [WaktinsZeitlin1993].

```c
void _fmpz_poly_swinnerton_dyer(fmpz *coeffs, ulong n)
```

```c
void fmpz_poly_swinnerton_dyer(fmpz_poly_t poly, ulong n)
```

Sets poly to the Swinnerton-Dyer polynomial $S_n$, defined as the integer polynomial $S_n = \prod(x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \ldots \pm \sqrt{p_n})$ where $p_n$ denotes the $n$-th prime number and all combinations of signs are taken. This polynomial has degree $2^n$ and is irreducible over the integers (it is the minimal polynomial of $\sqrt{2} + \ldots + \sqrt{p_n}$).

### 4.7. fmpz_poly.h – univariate polynomials over the integers
void fmpz_poly_legendre_pt(fmpz_poly_t poly, ulong n)
    Sets poly to the shifted Legendre polynomial \( \tilde{P}_n(x) \), defined by \( \tilde{P}_n(x) = P_n(2x - 1) \), for \( n \geq 0 \). The coefficients are calculated using a hypergeometric recurrence. See fmpq_poly for the Legendre polynomials.

void _fmpz_poly_hermite_h(fmpz *coeffs, ulong n)
    Sets coeffs to the coefficient array of the Hermite polynomial \( H_n(x) \), defined by \( H_n'(x) = 2^n H_n - 1(x) \), for \( n \geq 0 \). The coefficients are calculated using a hypergeometric recurrence. The length of the array will be \( n+1 \).

void fmpz_poly_hermite_h(fmpz_poly_t poly, ulong n)
    Sets poly to the Hermite polynomial \( H_n(x) \), defined by \( H_n'(x) = 2^n H_n - 1(x) \), for \( n \geq 0 \). The coefficients are calculated using a hypergeometric recurrence.

void _fmpz_poly_hermite_he(fmpz *coeffs, ulong n)
    Sets coeffs to the coefficient array of the Hermite polynomial \( H_e_n(x) \), defined by \( H_e_n(x) = 2^{-\frac{m}{2}} H_n \left( \frac{x}{\sqrt{2}} \right) \), for \( n \geq 0 \). The coefficients are calculated using a hypergeometric recurrence. The length of the array will be \( n+1 \).

void fmpz_poly_hermite_he(fmpz_poly_t poly, ulong n)
    Sets poly to the Hermite polynomial \( H_e_n(x) \), defined by \( H_e_n(x) = 2^{-\frac{m}{2}} H_n \left( \frac{x}{\sqrt{2}} \right) \), for \( n \geq 0 \). The coefficients are calculated using a hypergeometric recurrence.

4.7.49 Fibonacci polynomials

void _fmpz_poly_fibonacci(fmpz *coeffs, ulong n)
    Sets coeffs to the coefficient array of the \( n \)-th Fibonacci polynomial. The coefficients are calculated using a hypergeometric recurrence.

void fmpz_poly_fibonacci(fmpz_poly_t poly, ulong n)
    Sets poly to the \( n \)-th Fibonacci polynomial. The coefficients are calculated using a hypergeometric recurrence.

4.7.50 Eulerian numbers and polynomials

Eulerian numbers are the coefficients to the Eulerian polynomials

\[
A_n(x) = \sum_{m=0}^{n} A(n, m)x^m,
\]

where the Eulerian polynomials are defined by the exponential generating function

\[
x - 1 \over x - e^{(x-1)t} = \sum_{n=0}^\infty A_n(x) {t^n \over n!}.
\]

The Eulerian numbers can be expressed explicitly via the formula

\[
A(n, m) = \sum_{k=0}^{m+1} (-1)^k {n+1 \choose k}(m+1-k)^n.
\]

Note: Not to be confused with Euler numbers and polynomials.

void arith_eulerian_polynomial(fmpz_poly_t res, ulong n)
    Sets res to the Eulerian polynomial \( A_n(x) \), where we define \( A_0(x) = 1 \). The polynomial is calculated via a recursive relation.
4.7.51 Modular forms and q-series

```c
void _fmpz_poly_eta_qexp(fmpz *f, slong r, slong len)
void fmpz_poly_eta_qexp(fmpz_poly_t f, slong r, slong n)
```

Sets \( f \) to the \( q \)-expansion to length \( n \) of the Dedekind eta function (without the leading factor \( q^{1/24} \)) raised to the power \( r \), i.e. \( (q^{-1/24}\eta(q))^r = \prod_{k=1}^{\infty}(1-q^k)^r \).

In particular, \( r = -1 \) gives the generating function of the partition function \( p(k) \), and \( r = 24 \) gives, after multiplication by \( q \), the modular discriminant \( \Delta(q) \) which generates the Ramanujan tau function \( \tau(k) \).

This function uses sparse formulas for \( r = 1, 2, 3, 4, 6 \) and otherwise reduces to one of those cases using power series arithmetic.

```c
void _fmpz_poly_theta_qexp(fmpz *f, slong r, slong len)
void fmpz_poly_theta_qexp(fmpz_poly_t f, slong r, slong n)
```

Sets \( f \) to the \( q \)-expansion to length \( n \) of the Jacobi theta function raised to the power \( r \), i.e. \( \vartheta(q)^r \) where \( \vartheta(q) = 1 + 2 \sum_{k=1}^{\infty} q^{k^2} \).

This function uses sparse formulas for \( r = 1, 2 \) and otherwise reduces to those cases using power series arithmetic.

4.7.52 CLD bounds

```c
void fmpz_poly_CLD_bound(fmpz_t res, const fmpz_poly_t f, slong n)
```

Compute a bound on the \( n \) coefficient of \( fg'/g \) where \( g \) is any factor of \( f \).

4.8 fmpz_poly_mat.h – matrices of polynomials over the integers

The \( fmpz_poly_mat_t \) data type represents matrices whose entries are integer polynomials.

The \( fmpz_poly_mat_t \) type is defined as an array of \( fmpz_poly_mat_struct \)'s of length one. This permits passing parameters of type \( fmpz_poly_mat_t \) by reference.

An integer polynomial matrix internally consists of a single array of \( fmpz_poly_struct \)'s, representing a dense matrix in row-major order. This array is only directly indexed during memory allocation and deallocation. A separate array holds pointers to the start of each row, and is used for all indexing. This allows the rows of a matrix to be permuted quickly by swapping pointers.

Matrices having zero rows or columns are allowed.

The shape of a matrix is fixed upon initialisation. The user is assumed to provide input and output variables whose dimensions are compatible with the given operation.

4.8.1 Simple example

The following example constructs the matrix \( \begin{pmatrix} 2x + 1 & x \\ 1 - x & -1 \end{pmatrix} \) and computes its determinant.

```c
#include "fmpz_poly.h"
#include "fmpz_poly_mat.h"

int main()
{
    fmpz_poly_mat_t A;
    fmpz_poly_t P;

    (continues on next page)
```c
fmpz_poly_mat_init(A, 2, 2);
fmpz_poly_init(P);

fmpz_poly_set_str(fmpz_poly_mat_entry(A, 0, 0), "2 1 2");
fmpz_poly_set_str(fmpz_poly_mat_entry(A, 0, 1), "2 0 1");
fmpz_poly_set_str(fmpz_poly_mat_entry(A, 1, 0), "2 1 -1");
fmpz_poly_set_str(fmpz_poly_mat_entry(A, 1, 1), "1 -1");

fmpz_poly_mat_det(P, A);
fmpz_poly_print_pretty(P, "x");

fmpz_poly_clear(P);
fmpz_poly_mat_clear(A);
```

The output is:

```
x^2-3*x-1
```

### 4.8.2 Types, macros and constants

type `fmpz_poly_mat_struct`

type `fmpz_poly_mat_t`

### 4.8.3 Memory management

```c
void fmpz_poly_mat_init(fmpz_poly_mat_t mat, slong rows, slong cols)
    Initialises a matrix with the given number of rows and columns for use.

void fmpz_poly_mat_init_set(fmpz_poly_mat_t mat, const fmpz_poly_mat_t src)
    Initialises a matrix mat of the same dimensions as src, and sets it to a copy of src.

void fmpz_poly_mat_clear(fmpz_poly_mat_t mat)
    Frees all memory associated with the matrix. The matrix must be reinitialised if it is to be used again.
```

### 4.8.4 Basic properties

```c
slong fmpz_poly_mat_nrows(const fmpz_poly_mat_t mat)
    Returns the number of rows in mat.

slong fmpz_poly_mat_ncols(const fmpz_poly_mat_t mat)
    Returns the number of columns in mat.
```
4.8.5 Basic assignment and manipulation

`fmpz_poly_struct *fmpz_poly_mat_entry(const fmpz_poly_mat_t mat, slong i, slong j)`

Gives a reference to the entry at row `i` and column `j`. The reference can be passed as an input or output variable to any `fmpz_poly` function for direct manipulation of the matrix element. No bounds checking is performed.

`void fmpz_poly_mat_set(fmpz_poly_mat_t mat1, const fmpz_poly_mat_t mat2)`
Sets `mat1` to a copy of `mat2`.

`void fmpz_poly_mat_swap(fmpz_poly_mat_t mat1, fmpz_poly_mat_t mat2)`
Swaps `mat1` and `mat2` efficiently.

`void fmpz_poly_mat_swap_entrywise(fmpz_poly_mat_t mat1, fmpz_poly_mat_t mat2)`
Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

4.8.6 Input and output

`void fmpz_poly_mat_print(const fmpz_poly_mat_t mat, const char *x)`
Prints the matrix `mat` to standard output, using the variable `x`.

4.8.7 Random matrix generation

`void fmpz_poly_mat_randtest(fmpz_poly_mat_t mat, flint_rand_t state, slong len, flint_bitcnt_t bits)`
This is equivalent to applying `fmpz_poly_randtest` to all entries in the matrix.

`void fmpz_poly_mat_randtest_unsigned(fmpz_poly_mat_t mat, flint_rand_t state, slong len, flint_bitcnt_t bits)`
This is equivalent to applying `fmpz_poly_randtest_unsigned` to all entries in the matrix.

`void fmpz_poly_mat_randtest_sparse(fmpz_poly_mat_t A, flint_rand_t state, slong len, flint_bitcnt_t bits, float density)`
Creates a random matrix with the amount of nonzero entries given approximately by the `density` variable, which should be a fraction between 0 (most sparse) and 1 (most dense).
The nonzero entries will have random lengths between 1 and `len`.

4.8.8 Special matrices

`void fmpz_poly_mat_zero(fmpz_poly_mat_t mat)`
Sets `mat` to the zero matrix.

`void fmpz_poly_mat_one(fmpz_poly_mat_t mat)`
Sets `mat` to the unit or identity matrix of given shape, having the element 1 on the main diagonal and zeros elsewhere. If `mat` is nonsquare, it is set to the truncation of a unit matrix.
4.8.9 Basic comparison and properties

int fmpz_poly_mat_equal(const fmpz_poly_mat_t mat1, const fmpz_poly_mat_t mat2)
    
    Returns nonzero if mat1 and mat2 have the same shape and all their entries agree, and returns zero otherwise.

int fmpz_poly_mat_is_zero(const fmpz_poly_mat_t mat)
    
    Returns nonzero if all entries in mat are zero, and returns zero otherwise.

int fmpz_poly_mat_is_one(const fmpz_poly_mat_t mat)
    
    Returns nonzero if all entries of mat on the main diagonal are the constant polynomial 1 and all remaining entries are zero, and returns zero otherwise. The matrix need not be square.

int fmpz_poly_mat_is_empty(const fmpz_poly_mat_t mat)
    
    Returns a non-zero value if the number of rows or the number of columns in mat is zero, and otherwise returns zero.

int fmpz_poly_mat_is_square(const fmpz_poly_mat_t mat)
    
    Returns a non-zero value if the number of rows is equal to the number of columns in mat, and otherwise returns zero.

4.8.10 Norms

slong fmpz_poly_mat_max_bits(const fmpz_poly_mat_t A)
    
    Returns the maximum number of bits among the coefficients of the entries in A, or the negative of that value if any coefficient is negative.

slong fmpz_poly_mat_max_length(const fmpz_poly_mat_t A)
    
    Returns the maximum polynomial length among all the entries in A.

4.8.11 Transpose

void fmpz_poly_mat_transpose(fmpz_poly_mat_t B, const fmpz_poly_mat_t A)
    
    Sets B to Aᵀ.

4.8.12 Evaluation

void fmpz_poly_mat_evaluate_fmpz(fmpz_mat_t B, const fmpz_poly_mat_t A, const fmpz_t x)
    
    Sets the fmpz_mat_t B to A evaluated entrywise at the point x.

4.8.13 Arithmetic

void fmpz_poly_mat_scalar_mul_fmpz_poly(fmpz_poly_mat_t B, const fmpz_poly_mat_t A, const fmpz_poly_t c)
    
    Sets B to A multiplied entrywise by the polynomial c.

void fmpz_poly_mat_scalar_mul_fmpz(fmpz_poly_mat_t B, const fmpz_poly_mat_t A, const fmpz_t c)
    
    Sets B to A multiplied entrywise by the integer c.

void fmpz_poly_mat_add(fmpz_poly_mat_t C, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
    
    Sets C to the sum of A and B. All matrices must have the same shape. Aliasing is allowed.
void fmpz_poly_mat_sub(fmpz_poly_mat_t C, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
Sets C to the sum of A and B. All matrices must have the same shape. Aliasing is allowed.

void fmpz_poly_mat_neg(fmpz_poly_mat_t B, const fmpz_poly_mat_t A)
Sets B to the negation of A. The matrices must have the same shape. Aliasing is allowed.

void fmpz_poly_mat_mul(fmpz_poly_mat_t C, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
Sets C to the matrix product of A and B. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and KS multiplication.

void fmpz_poly_mat_mul_classical(fmpz_poly_mat_t C, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
Sets C to the matrix product of A and B, computed using the classical algorithm. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

void fmpz_poly_mat_mul_KS(fmpz_poly_mat_t C, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
Sets C to the matrix product of A and B, computed using Kronecker segmentation. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

void fmpz_poly_mat_mullow(fmpz_poly_mat_t C, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B, slong len)
Sets C to the matrix product of A and B, truncating each entry in the result to length len. Uses classical matrix multiplication. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

void fmpz_poly_mat_sqr(fmpz_poly_mat_t B, const fmpz_poly_mat_t A)
Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function automatically chooses between classical and KS squaring.

void fmpz_poly_mat_sqr_classical(fmpz_poly_mat_t B, const fmpz_poly_mat_t A)
Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function uses direct formulas for very small matrices, and otherwise classical matrix multiplication.

void fmpz_poly_mat_sqr_KS(fmpz_poly_mat_t B, const fmpz_poly_mat_t A)
Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function uses Kronecker segmentation.

void fmpz_poly_mat_sqrlow(fmpz_poly_mat_t B, const fmpz_poly_mat_t A, slong len)
Sets B to the square of A, which must be a square matrix, truncating all entries to length len. Aliasing is allowed. This function uses direct formulas for very small matrices, and otherwise classical matrix multiplication.

void fmpz_poly_mat_pow(fmpz_poly_mat_t B, const fmpz_poly_mat_t A, ulong exp)
Sets B to A raised to the power exp, where A is a square matrix. Uses exponentiation by squaring. Aliasing is allowed.

void fmpz_poly_mat_pow_trunc(fmpz_poly_mat_t B, const fmpz_poly_mat_t A, ulong exp, slong len)
Sets B to A raised to the power exp, truncating all entries to length len, where A is a square matrix. Uses exponentiation by squaring. Aliasing is allowed.

void fmpz_poly_mat_prod(fmpz_poly_mat_t res, fmpz_poly_mat_t *const factors, slong n)
Sets res to the product of the n matrices given in the vector factors, all of which must be square and of the same size. Uses binary splitting.
4.8.14 Row reduction

```c
slong fmpz_poly_mat_find_pivot_any(const fmpz_poly_mat_t mat, slong start_row, slong end_row, slong c)
```

Attempts to find a pivot entry for row reduction. Returns a row index \( r \) between \( \text{start\_row} \) (inclusive) and \( \text{stop\_row} \) (exclusive) such that column \( c \) in \( \text{mat} \) has a nonzero entry on row \( r \), or returns -1 if no such entry exists.

This implementation simply chooses the first nonzero entry it encounters. This is likely to be a nearly optimal choice if all entries in the matrix have roughly the same size, but can lead to unnecessary coefficient growth if the entries vary in size.

```c
slong fmpz_poly_mat_find_pivot_partial(const fmpz_poly_mat_t mat, slong start_row, slong end_row, slong c)
```

Attempts to find a pivot entry for row reduction. Returns a row index \( r \) between \( \text{start\_row} \) (inclusive) and \( \text{stop\_row} \) (exclusive) such that column \( c \) in \( \text{mat} \) has a nonzero entry on row \( r \), or returns -1 if no such entry exists.

This implementation searches all the rows in the column and chooses the nonzero entry of smallest degree. If there are several entries with the same minimal degree, it chooses the entry with the smallest coefficient bit bound. This heuristic typically reduces coefficient growth when the matrix entries vary in size.

```c
slong fmpz_poly_mat_fflu(fmpz_poly_mat_t B, fmpz_poly_t den, slong *perm, const fmpz_poly_mat_t A, int rank_check)
```

Uses fraction-free Gaussian elimination to set \( (B, \text{den}) \) to a fraction-free LU decomposition of \( A \) and returns the rank of \( A \). Aliasing of \( A \) and \( B \) is allowed.

Pivot elements are chosen with \texttt{fmpz\_poly\_mat\_find\_pivot\_partial}. If \( \text{perm} \) is non-NULL, the permutation of rows in the matrix will also be applied to \( \text{perm} \).

If \( \text{rank\_check} \) is set, the function aborts and returns 0 if the matrix is detected not to have full rank without completing the elimination.

The denominator \( \text{den} \) is set to \( \pm \det(A) \), where the sign is decided by the parity of the permutation. Note that the determinant is not generally the minimal denominator.

```c
slong fmpz_poly_mat_rref(fmpz_poly_mat_t B, fmpz_poly_t den, const fmpz_poly_mat_t A)
```

Sets \( (B, \text{den}) \) to the reduced row echelon form of \( A \) and returns the rank of \( A \). Aliasing of \( A \) and \( B \) is allowed.

The denominator \( \text{den} \) is set to \( \pm \det(A) \). Note that the determinant is not generally the minimal denominator.

4.8.15 Trace

```c
void fmpz_poly_mat_trace(fmpz_poly_t trace, const fmpz_poly_mat_t mat)
```

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.
### 4.8.16 Determinant and rank

```c
void fmpz_poly_mat_det(fmpz_poly_t det, const fmpz_poly_mat_t A)
```
- Sets `det` to the determinant of the square matrix `A`. Uses a direct formula, fraction-free LU decomposition, or interpolation, depending on the size of the matrix.

```c
void fmpz_poly_mat_det_fflu(fmpz_poly_t det, const fmpz_poly_mat_t A)
```
- Sets `det` to the determinant of the square matrix `A`. The determinant is computed by performing a fraction-free LU decomposition on a copy of `A`.

```c
void fmpz_poly_mat_det_interpolate(fmpz_poly_t det, const fmpz_poly_mat_t A)
```
- Sets `det` to the determinant of the square matrix `A`. The determinant is computed by determining a bound `n` for its length, evaluating the matrix at `n` distinct points, computing the determinant of each integer matrix, and forming the interpolating polynomial.

```c
slong fmpz_poly_mat_rank(const fmpz_poly_mat_t A)
```
- Returns the rank of `A`. Performs fraction-free LU decomposition on a copy of `A`.

### 4.8.17 Inverse

```c
int fmpz_poly_mat_inv(fmpz_poly_mat_t Ainv, fmpz_poly_t den, const fmpz_poly_mat_t A)
```
- Sets `(Ainv, den)` to the inverse matrix of `A`. Returns 1 if `A` is nonsingular and 0 if `A` is singular. Aliasing of `Ainv` and `A` is allowed.
  - More precisely, `det` will be set to the determinant of `A` and `Ainv` will be set to the adjugate matrix of `A`. Note that the determinant is not necessarily the minimal denominator.
  - Uses fraction-free LU decomposition, followed by solving for the identity matrix.

### 4.8.18 Nullspace

```c
slong fmpz_poly_mat_nullspace(fmpz_poly_mat_t res, const fmpz_poly_mat_t mat)
```
- Computes the right rational nullspace of the matrix `mat` and returns the nullity.
  - More precisely, assume that `mat` has rank `r` and nullity `n`. Then this function sets the first `n` columns of `res` to linearly independent vectors spanning the nullspace of `mat`. As a result, we always have `rank(res) = n`, and `mat × res` is the zero matrix.
  - The computed basis vectors will not generally be in a reduced form. In general, the polynomials in each column vector in the result will have a nontrivial common GCD.

### 4.8.19 Solving

```c
int fmpz_poly_mat_solve(fmpz_poly_mat_t X, fmpz_t den, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
```
- Solves the equation `AX = B` for nonsingular `A`. More precisely, computes `(X, den)` such that `AX = B × den`. Returns 1 if `A` is nonsingular and 0 if `A` is singular. The computed denominator will not generally be minimal.
  - Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```c
int fmpz_poly_mat_solve_fflu(fmpz_poly_mat_t X, fmpz_t den, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
```
- Solves the equation `AX = B` for nonsingular `A`. More precisely, computes `(X, den)` such that `AX = B × den`. Returns 1 if `A` is nonsingular and 0 if `A` is singular. The computed denominator will not generally be minimal.
  - Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.
void \texttt{fmpz\_poly\_mat\_solve\_fflu\_precomp}(fmpz\_mat\_t X, const \texttt{slong} *perm, const fmpz\_mat\_t FFLU, const fmpz\_mat\_t B) 

Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation.

4.9 \texttt{fmpz\_poly\_factor.h} – factorisation of polynomials over the integers

4.9.1 Types, macros and constants

\begin{itemize}
\item \texttt{type fmpz\_poly\_factor\_struct}
\item \texttt{type fmpz\_poly\_factor\_t}
\end{itemize}

4.9.2 Memory management

\begin{itemize}
\item \texttt{void fmpz\_poly\_factor\_init}\,(fmpz\_poly\_factor\_t fac)
\quad Initialises a new factor structure.
\item \texttt{void fmpz\_poly\_factor\_init2}\,(fmpz\_poly\_factor\_t fac, \texttt{slong} alloc)
\quad Initialises a new factor structure, providing space for at least alloc factors.
\item \texttt{void fmpz\_poly\_factor\_reallocate}\,(fmpz\_poly\_factor\_t fac, \texttt{slong} alloc)
\quad Reallocates the factor structure to provide space for precisely alloc factors.
\item \texttt{void fmpz\_poly\_factor\_fit\_length}\,(fmpz\_poly\_factor\_t fac, \texttt{slong} len)
\quad Ensures that the factor structure has space for at least len factors. This functions takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.
\item \texttt{void fmpz\_poly\_factor\_clear}\,(fmpz\_poly\_factor\_t fac)
\quad Releases all memory occupied by the factor structure.
\end{itemize}

4.9.3 Manipulating factors

\begin{itemize}
\item \texttt{void fmpz\_poly\_factor\_set}\,(fmpz\_poly\_factor\_t res, const fmpz\_poly\_factor\_t fac)
\quad Sets res to the same factorisation as fac.
\item \texttt{void fmpz\_poly\_factor\_insert}\,(fmpz\_poly\_factor\_t fac, const fmpz\_t p, \texttt{slong} e)
\quad Adds the primitive polynomial \(p^e\) to the factorisation fac.
\quad Assumes that \(\deg(p) \geq 2\) and \(e \neq 0\).
\item \texttt{void fmpz\_poly\_factor\_concat}\,(fmpz\_poly\_factor\_t res, const fmpz\_poly\_factor\_t fac)
\quad Concatenates two factorisations.
\quad This is equivalent to calling \texttt{fmpz\_poly\_factor\_insert()} repeatedly with the individual factors of fac.
\quad Does not support aliasing between res and fac.
\end{itemize}
4.9.4 Input and output

```c
void fmpz_poly_factor_print(const fmpz_poly_factor_t fac)
    Prints the entries of fac to standard output.
```

4.9.5 Factoring algorithms

```c
void fmpz_poly_factor_squarefree(fmpz_poly_factor_t fac, const fmpz_poly_t F)
    Takes as input a polynomial F and a freshly initialized factor structure fac. Updates fac to
    contain a factorization of F into (not necessarily irreducible) factors that themselves have no
    repeated factors. None of the returned factors will have the same exponent. That is we return g_i
    and unique e_i such that

    \[ F = c \prod_i g_i^{e_i} \]

    where c is the signed content of F and gcd(g_i, g_i') = 1.
```

```c
void fmpz_poly_factor_zassenhaus_recombination(fmpz_poly_factor_t final_fac, const
                                             fmpz_poly_factor_t lifted_fac, const
                                             fmpz_poly_t F, const fmpz_t P, slong exp)
    Takes as input a factor structure lifted_fac containing a squarefree factorization of the polynomial
    F mod p. The algorithm does a brute force search for irreducible factors of F over the integers,
    and each factor is raised to the power exp.

    The impact of the algorithm is to augment a factorization of F^exp to the factor structure
    final_fac.
```

```c
void _fmpz_poly_factor_zassenhaus(fmpz_poly_factor_t final_fac, slong exp, const
                                 fmpz_poly_t f, slong cutoff, int use_van_hoeij)
    This is the internal wrapper of Zassenhaus.

    It will attempt to find a small prime such that f modulo p has a minimal number of factors. If it
    cannot find a prime giving less than cutoff factors it aborts. Then it decides a p-adic precision to
    lift the factors to, Hensel lifts, and finally calls Zassenhaus recombination.

    Assumes that \text{len}(f) \geq 2.

    Assumes that f is primitive.

    Assumes that the constant coefficient of f is non-zero. Note that this can be easily achieved by
    taking out factors of the form x^k before calling this routine.

    If the final flag is set, the function will use the van Hoeij factorisation algorithm with gradual
    feeding and mod 2^k data truncation to find factors when the number of local factors is large.
```

```c
void fmpz_poly_factor_zassenhaus(fmpz_poly_factor_t final_fac, const fmpz_poly_t F)
    A wrapper of the Zassenhaus factoring algorithm, which takes as input any polynomial F, and
    stores a factorization in final_fac.

    The complexity will be exponential in the number of local factors we find for the components of a
    squarefree factorization of F.
```

```c
void _fmpz_poly_factor_quadratic(fmpz_poly_factor_t fac, const fmpz_poly_t f, slong exp)
void _fmpz_poly_factor_cubic(fmpz_poly_factor_t fac, const fmpz_poly_t f, slong exp)
    Inserts the factorisation of the quadratic (resp. cubic) polynomial f into fac with multiplicity exp.
    This function requires that the content of f has been removed, and does not update the content of
    fac. The factorization is calculated over \( \mathbb{R} \) or \( \mathbb{Q}_2 \) and then tested over \( \mathbb{Z} \).
```

```c
void fmpz_poly_factor(fmpz_poly_factor_t final_fac, const fmpz_poly_t F)
    A wrapper of the Zassenhaus and van Hoeij factoring algorithms, which takes as input any poly-
    nomial F, and stores a factorization in final_fac.
```
4.10 fmpz_mpoly.h – multivariate polynomials over the integers

The exponents follow the mpoly interface. A coefficient may be referenced as a fmpz *.

4.10.1 Types, macros and constants

type fmpz_mpoly_struct
A structure holding a multivariate integer polynomial.

type fmpz_mpoly_t
An array of length 1 of fmpz_mpoly_struct.

type fmpz_mpoly_ctx_struct
Context structure representing the parent ring of an fmpz_mpoly.

type fmpz_mpoly_ctx_t
An array of length 1 of fmpz_mpoly_ctx_struct.

4.10.2 Context object

void fmpz_mpoly_ctx_init(fmpz_mpoly_ctx_t ctx, slong nvars, const ordering_t ord)
Initialise a context object for a polynomial ring with the given number of variables and the given ordering. The possibilities for the ordering are ORD_LEX, ORD_DEGLEX and ORD_DEGREVLEX.

slong fmpz_mpoly_ctx_nvars(const fmpz_mpoly_ctx_t ctx)
Return the number of variables used to initialize the context.

ordering_t fmpz_mpoly_ctx_ord(const fmpz_mpoly_ctx_t ctx)
Return the ordering used to initialize the context.

void fmpz_mpoly_ctx_clear(fmpz_mpoly_ctx_t ctx)
Release up any space allocated by ctx.

4.10.3 Memory management

void fmpz_mpoly_init(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
Initialise A for use with the given and initialised context object. Its value is set to zero.

void fmpz_mpoly_init2(fmpz_mpoly_t A, slong alloc, const fmpz_mpoly_ctx_t ctx)
Initialise A for use with the given and initialised context object. Its value is set to zero. It is allocated with space for alloc terms and at least MPOLY_MIN_BITS bits for the exponents.

void fmpz_mpoly_init3(fmpz_mpoly_t A, slong alloc, flint_bitcnt_t bits, const fmpz_mpoly_ctx_t ctx)
Initialise A for use with the given and initialised context object. Its value is set to zero. It is allocated with space for alloc terms and bits bits for the exponents.

void fmpz_mpoly_fit_length(fmpz_mpoly_t A, slong len, const fmpz_mpoly_ctx_t ctx)
Ensure that A has space for at least len terms.

void fmpz_mpoly_fit_bits(fmpz_mpoly_t A, flint_bitcnt_t bits, const fmpz_mpoly_ctx_t ctx)
Ensure that the exponent fields of A have at least bits bits.

void fmpz_mpoly_realloc(fmpz_mpoly_t A, slong alloc, const fmpz_mpoly_ctx_t ctx)
Reallocation A to have space for alloc terms. Assumes the current length of the polynomial is not greater than alloc.
void **fmpz_mpoly_clear(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Release any space allocated for A.

4.10.4 Input/Output

The variable strings in x start with the variable of most significance at index 0. If x is NULL, the variables are named x1, x2, etc.

char *fmpz_mpoly_get_str_pretty(const fmpz_mpoly_t A, const char **x, const fmpz_mpoly_ctx_t ctx)
    Return a string, which the user is responsible for cleaning up, representing A, given an array of variable strings x.

int fmpz_mpoly_fprint_pretty(FILE *file, const fmpz_mpoly_t A, const char **x, const fmpz_mpoly_ctx_t ctx)
    Print a string representing A to file.

int fmpz_mpoly_print_pretty(const fmpz_mpoly_t A, const char **x, const fmpz_mpoly_ctx_t ctx)
    Print a string representing A to stdout.

int fmpz_mpoly_set_str_pretty(fmpz_mpoly_t A, const char *str, const char **x, const fmpz_mpoly_ctx_t ctx)
    Set A to the polynomial in the null-terminates string str given an array x of variable strings. If parsing str fails, A is set to zero, and −1 is returned. Otherwise, 0 is returned. The operations +, −, *, and / are permitted along with integers and the variables in x. The character ^ must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

4.10.5 Basic manipulation

void fmpz_mpoly_gen(fmpz_mpoly_t A, slong var, const fmpz_mpoly_ctx_t ctx)
    Set A to the variable of index var, where var = 0 corresponds to the variable with the most significance with respect to the ordering.

int fmpz_mpoly_is_gen(const fmpz_mpoly_t A, slong var, const fmpz_mpoly_ctx_t ctx)
    If var ≥ 0, return 1 if A is equal to the var-th generator, otherwise return 0. If var < 0, return 1 if the polynomial is equal to any generator, otherwise return 0.

void fmpz_mpoly_set(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)
    Set A to B.

int fmpz_mpoly_equal(const fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)
    Return 1 if A is equal to B, else return 0.

void fmpz_mpoly_swap(fmpz_mpoly_t poly1, fmpz_mpoly_t poly2, const fmpz_mpoly_ctx_t ctx)
    Efficiently swap A and B.

int fmpz_mpoly_fits_small(const fmpz *poly, slong len)
    Return 1 if the array of coefficients of length len consists entirely of values that are small fmpz values, i.e. of at most FLINT_BITS - 2 bits plus a sign bit.

slong fmpz_mpoly_max_bits(const fmpz_mpoly_t A)
    Computes the maximum number of bits b required to represent the absolute values of the coefficients of A. If all of the coefficients are positive, b is returned, otherwise −b is returned.
4.10.6 Constants

```c
int fmpz_mpoly_is_fmpz(const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Return 1 if A is a constant, else return 0.

void fmpz_mpoly_get_fmpz(fmpz_t c, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Assuming that A is a constant, set c to this constant. This function throws if A is not a constant.

void fmpz_mpoly_set_fmpz(fmpz_mpoly_t A, const fmpz_t c, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_set_ui(fmpz_mpoly_t A, ulong c, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_set_si(fmpz_mpoly_t A, slong c, const fmpz_mpoly_ctx_t ctx)
    Set A to the constant c.

void fmpz_mpoly_zero(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Set A to the constant 0.

void fmpz_mpoly_one(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Set A to the constant 1.

int fmpz_mpoly_equal_fmpz(const fmpz_mpoly_t A, const fmpz_t c, const fmpz_mpoly_ctx_t ctx)
    Return 1 if A is equal to the constant c, else return 0.

int fmpz_mpoly_equal_ui(const fmpz_mpoly_t A, ulong c, const fmpz_mpoly_ctx_t ctx)

int fmpz_mpoly_equal_si(const fmpz_mpoly_t A, slong c, const fmpz_mpoly_ctx_t ctx)
    Return 1 if A is equal to the constant c, else return 0.

int fmpz_mpoly_is_zero(const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Return 1 if A is the constant 0, else return 0.

int fmpz_mpoly_is_one(const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Return 1 if A is the constant 1, else return 0.
```

4.10.7 Degrees

```c
int fmpz_mpoly_degrees_fit_si(const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Return 1 if the degrees of A with respect to each variable fit into an slong, otherwise return 0.

void fmpz_mpoly_degrees_fmpz(fmpz **degs, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_degrees_si(slong *degs, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Set degs to the degrees of A with respect to each variable. If A is zero, all degrees are set to −1.

void fmpz_mpoly_degree_fmpz(fmpz_t deg, const fmpz_mpoly_t A, slong var, const fmpz_mpoly_ctx_t ctx)

slong fmpz_mpoly_degree_si(const fmpz_mpoly_t A, slong var, const fmpz_mpoly_ctx_t ctx)
    Either return or set deg to the degree of A with respect to the variable of index var. If A is zero, the
degree is defined to be −1.

int fmpz_mpoly_total_degree_fits_si(const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Return 1 if the total degree of A fits into an slong, otherwise return 0.

void fmpz_mpoly_total_degree_fmpz(fmpz_t tdeg, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)

slong fmpz_mpoly_total_degree_si(const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    Either return or set tdeg to the total degree of A. If A is zero, the total degree is defined to be −1.

void fmpz_mpoly_used_vars(int *used, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
    For each variable index i, set used[i] to nonzero if the variable of index i appears in A and to
zero otherwise.
```
4.10.8 Coefficients

void fmpz_mpoly_get_coeff_fmpz_monomial(fmpz_t c, const fmpz_mpoly_t A, const fmpz_mpoly_t M, const fmpz_mpoly_ctx_t ctx)

Assuming that \( M \) is a monomial, set \( c \) to the coefficient of the corresponding monomial in \( A \). This function throws if \( M \) is not a monomial.

void fmpz_mpoly_set_coeff_fmpz_monomial(fmpz_mpoly_t poly, const fmpz_t c, const fmpz_mpoly_t poly2, const fmpz_mpoly_ctx_t ctx)

Assuming that \( M \) is a monomial, set the coefficient of the corresponding monomial in \( A \) to \( c \). This function throws if \( M \) is not a monomial.

void fmpz_mpoly_get_coeff_fmpz_fmpz(fmpz_t c, const fmpz_mpoly_t A, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)

ulong fmpz_mpoly_get_coeff_ui_fmpz(const fmpz_mpoly_t A, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)

slong fmpz_mpoly_get_coeff_si_fmpz(const fmpz_mpoly_t A, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_get_coeff_fmpz_ui(fmpz_t c, const fmpz_mpoly_t A, const ulong *exp, const fmpz_mpoly_ctx_t ctx)

ulong fmpz_mpoly_getcoeff_ui_ui(const fmpz_mpoly_t A, const ulong *exp, const fmpz_mpoly_ctx_t ctx)

slong fmpz_mpoly_get_coeff_si_ui(const fmpz_mpoly_t A, const ulong *exp, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_set_coeff_fmpz_ui(fmpz_mpoly_t A, const fmpz_t c, const ulong *exp, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_set_coeff_ui_ui(fmpz_mpoly_t A, ulong c, const ulong *exp, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_set_coeff_si_ui(fmpz_mpoly_t A, slong c, const ulong *exp, const fmpz_mpoly_ctx_t ctx)

Set the coefficient of the monomial with exponent vector \( exp \) to \( c \).

void fmpz_mpoly_get_coeff_vars_ui(fmpz_mpoly_t C, const fmpz_mpoly_t A, const slong *vars, const ulong *exp, const fmpz_mpoly_ctx_t ctx)

Set \( C \) to the coefficient of \( A \) with respect to the variables in \( vars \) with powers in the corresponding array \( exps \). Both \( vars \) and \( exps \) point to array of length \( length \). It is assumed that \( 0 < length \leq nvars(A) \) and that the variables in \( vars \) are distinct.
4.10.9 Comparison

```c
int fmpz_mpoly_cmp(const fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx) {
    // Return 1 (resp. -1, or 0) if A is after (resp. before, same as) B in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when A and B are both monomials.
}
```

4.10.10 Conversion

```c
int fmpz_mpoly_is_fmpz_poly(const fmpz_mpoly_t A, slong var, const fmpz_mpoly_ctx_t ctx) {
    // Return whether A is a univariate polynomial in the variable with index var.
}
```

```c
int fmpz_mpoly_get_fmpz_poly(fmpz_poly_t A, const fmpz_mpoly_t B, slong var, const fmpz_mpoly_ctx_t ctx) {
    // If B is a univariate polynomial in the variable with index var, set A to this polynomial and return 1; otherwise return 0.
}
```

```c
void fmpz_mpoly_set_fmpz_poly(fmpz_mpoly_t A, const fmpz_poly_t B, slong var, const fmpz_mpoly_ctx_t ctx) {
    // Set A to the univariate polynomial B in the variable with index var.
}
```

4.10.11 Container operations

These functions deal with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw.

```c
fmpz *fmpz_mpoly_term_coeff_ref(fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx) {
    // Return a reference to the coefficient of index i of A.
}
```

```c
int fmpz_mpoly_is_canonical(const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx) {
    // Return 1 if A is in canonical form. Otherwise, return 0. To be in canonical form, all of the terms must have nonzero coefficient, and the terms must be sorted from greatest to least.
}
```

```c
slong fmpz_mpoly_length(const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx) {
    // Return the number of terms in A. If the polynomial is in canonical form, this will be the number of nonzero coefficients.
}
```

```c
void fmpz_mpoly_resize(fmpz_mpoly_t A, slong new_length, const fmpz_mpoly_ctx_t ctx) {
    // Set the length of A to new_length. Terms are either deleted from the end, or new zero terms are appended.
}
```

```c
void fmpz_mpoly_get_term_coeff_fmpz(fmpz_t c, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx) {
    // Either return or set c to the coefficient of the term of index i.
}
```

```c
ulong fmpz_mpoly_get_term_coeff_ui(const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx) {
    // Either return or set c to the coefficient of the term of index i.
}
```

```c
slong fmpz_mpoly_get_term_coeff_si(const fmpz_mpoly_t poly, slong i, const fmpz_mpoly_ctx_t ctx) {
    // Either return or set c to the coefficient of the term of index i.
}
```

```c
void fmpz_mpoly_set_term_coeff_fmpz(fmpz_mpoly_t A, slong i, const fmpz_t c, const fmpz_mpoly_ctx_t ctx) {
    // Either return or set c to the coefficient of the term of index i.
}
```

```c
void fmpz_mpoly_set_term_coeff_ui(fmpz_mpoly_t A, slong i, ulong c, const fmpz_mpoly_ctx_t ctx) {
    // Either return or set c to the coefficient of the term of index i.
}
```
**Flint Documentation, Release 3.1.0-dev**

```c
void fmpz_mpoly_set_term_coeff_si(fmpz_mpoly_t A, slong i, slong c, const fmpz_mpoly_ctx_t ctx)

Set the coefficient of the term of index \(i\) to \(c\).
```

```c
int fmpz_mpoly_term_exp_fits_si(const fmpz_mpoly_t poly, slong i, const fmpz_mpoly_ctx_t ctx)
int fmpz_mpoly_term_exp_fits_ui(const fmpz_mpoly_t poly, slong i, const fmpz_mpoly_ctx_t ctx)

Return 1 if all entries of the exponent vector of the term of index \(i\) fit into an `slong` (resp. a `ulong`). Otherwise, return 0.
```

```c
void fmpz_mpoly_get_term_exp_fmpz(fmpz **exp, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_get_term_exp_ui(ulong *exp, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_get_term_exp_si(slong *exp, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)

Set `exp` to the exponent vector of the term of index \(i\). The _ui (resp. _si) version throws if any entry does not fit into a `ulong` (resp. `slong`).
```

```c
ulong fmpz_mpoly_get_term_var_exp_ui(const fmpz_mpoly_t A, slong i, slong var, const fmpz_mpoly_ctx_t ctx)
slong fmpz_mpoly_get_term_var_exp_si(const fmpz_mpoly_t A, slong i, slong var, const fmpz_mpoly_ctx_t ctx)

Return the exponent of the variable \(\text{var}\) of the term of index \(i\). This function throws if the exponent does not fit into a `ulong` (resp. `slong`).
```

```c
void fmpz_mpoly_set_term_exp_fmpz(fmpz_mpoly_t A, slong i, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_set_term_exp_ui(fmpz_mpoly_t A, slong i, const ulong *exp, const fmpz_mpoly_ctx_t ctx)
Set the exponent vector of the term of index \(i\) to \(exp\).
```

```c
void fmpz_mpoly_get_term(fmpz_mpoly_t M, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_get_term_monomial(fmpz_mpoly_t M, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)
Set \(M\) to the term of index \(i\) in \(A\). The coefficient of \(M\) will be one.
```

```c
void fmpz_mpoly_set_term_monomial(fmpz_mpoly_t M, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)

Set \(M\) to the monomial of the term of index \(i\) in \(A\).
```

```c
void fmpz_mpoly_push_term_fmpz_fmpz(fmpz_mpoly_t A, const fmpz_t c, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_push_term_fmpz_ffmpz(fmpz_mpoly_t A, const fmpz_t c, const fmpz *exp, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_push_term_ui_fmpz(fmpz_mpoly_t A, ulong c, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_push_term_ui_ffmpz(fmpz_mpoly_t A, ulong c, const fmpz *exp, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_push_term_si_fmpz(fmpz_mpoly_t A, slong c, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_push_term_si_ffmpz(fmpz_mpoly_t A, slong c, const fmpz *exp, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_push_term_fmpz_ui(fmpz_mpoly_t A, const fmpz_t c, const ulong *exp, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_push_term_ui_ui(fmpz_mpoly_t A, ulong c, const ulong *exp, const fmpz_mpoly_ctx_t ctx)
```

### 4.10. fmpz_mpoly.h – multivariate polynomials over the integers

229
void fmpz_mpoly_push_term_si_ui(fmpz_mpoly_t A, slong c, const ulong *exp, const fmpz_mpoly_ctx_t ctx)

Append a term to \( A \) with coefficient \( c \) and exponent vector \( \text{exp} \). This function runs in constant average time.

void fmpz_mpoly_sort_terms(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)

Sort the terms of \( A \) into the canonical ordering dictated by the ordering in \( \text{ctx} \). This function simply reorders the terms: It does not combine like terms, nor does it delete terms with coefficient zero. This function runs in linear time in the size of \( A \).

void fmpz_mpoly_combine_like_terms(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)

Combine adjacent like terms in \( A \) and delete terms with coefficient zero. If the terms of \( A \) were sorted to begin with, the result will be in canonical form. This function runs in linear time in the size of \( A \).

void fmpz_mpoly_reverse(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)

Set \( A \) to the reversal of \( B \).

4.10.12 Random generation

void fmpz_mpoly_randtest_bound(fmpz_mpoly_t A, flint_rand_t state, slong length, mp_limb_t coeff_bits, ulong exp_bound, const fmpz_mpoly_ctx_t ctx)

Generate a random polynomial with length up to \( \text{length} \) and exponents in the range \([0, \text{exp_bound} - 1]\). The exponents of each variable are generated by calls to n_randint(state, \text{exp_bound}).

void fmpz_mpoly_randtest_bounds(fmpz_mpoly_t A, flint_rand_t state, slong length, mp_limb_t coeff_bits, ulong *exp_bounds, const fmpz_mpoly_ctx_t ctx)

Generate a random polynomial with length up to \( \text{length} \) and exponents in the range \([0, \text{exp_bounds}[i] - 1]\). The exponents of the variable of index \( i \) are generated by calls to n_randint(state, \text{exp_bounds}[i]).

void fmpz_mpoly_randtest_bits(fmpz_mpoly_t A, flint_rand_t state, slong length, mp_limb_t coeff_bits, mp_limb_t exp_bits, const fmpz_mpoly_ctx_t ctx)

Generate a random polynomial with length up to the given length and exponents whose packed form does not exceed the given bit count. The parameter \( \text{coeff_bits} \) to the three functions \text{fmpz_mpoly_randtest_{bound|bounds|bits}} \) is merely a suggestion for the approximate bit count of the resulting signed coefficients. The function \text{fmpz_mpoly_max_bits()} \) will give the exact bit count of the result.

4.10.13 Addition/Subtraction

void fmpz_mpoly_add_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t c, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_add_ui(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong c, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_add_si(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong c, const fmpz_mpoly_ctx_t ctx)

Set \( A \) to \( B + c \). If \( A \) and \( B \) are aliased, this function will probably run quickly.

void fmpz_mpoly_sub_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t c, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_sub_ui(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong c, const fmpz_mpoly_ctx_t ctx)
void `fmpz_mpoly_sub_si`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, `slong` c, const `fmpz_mpoly_ctx_t` ctx)

Set A to B − c. If A and B are aliased, this function will probably run quickly.

void `fmpz_mpoly_add`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, const `fmpz_mpoly_t` C, const `fmpz_mpoly_ctx_t` ctx)

Set A to B + C. If A and B are aliased, this function might run in time proportional to the size of C.

void `fmpz_mpoly_sub`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, const `fmpz_mpoly_t` C, const `fmpz_mpoly_ctx_t` ctx)

Set A to B − C. If A and B are aliased, this function might run in time proportional to the size of C.

### 4.10.14 Scalar operations

void `fmpz_mpoly_neg`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, const `fmpz_mpoly_ctx_t` ctx)

Set A to −B.

void `fmpz_mpoly_scalar_mul_fmpz`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, const `fmpz_t` c, const `fmpz_mpoly_ctx_t` ctx)

void `fmpz_mpoly_scalar_mul_ui`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, `ulong` c, const `fmpz_mpoly_ctx_t` ctx)

void `fmpz_mpoly_scalar_mul_si`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, `slong` c, const `fmpz_mpoly_ctx_t` ctx)

Set A to B × c.

void `fmpz_mpoly_scalar_fmma`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, const `fmpz_t` c, const `fmpz_t` D, const `fmpz_t` e, const `fmpz_mpoly_ctx_t` ctx)

Sets A to B × c + D × e.

void `fmpz_mpoly_scalar_divexact_fmpz`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, const `fmpz_t` c, const `fmpz_mpoly_ctx_t` ctx)

void `fmpz_mpoly_scalar_divexact_ui`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, `ulong` c, const `fmpz_mpoly_ctx_t` ctx)

void `fmpz_mpoly_scalar_divexact_si`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, `slong` c, const `fmpz_mpoly_ctx_t` ctx)

Set A to B divided by c. The division is assumed to be exact.

int `fmpz_mpoly_scalar_divides_fmpz`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, const `fmpz_t` c, const `fmpz_mpoly_ctx_t` ctx)

int `fmpz_mpoly_scalar_divides_ui`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, `ulong` c, const `fmpz_mpoly_ctx_t` ctx)

int `fmpz_mpoly_scalar_divides_si`(`fmpz_mpoly_t` A, const `fmpz_mpoly_t` B, `slong` c, const `fmpz_mpoly_ctx_t` ctx)

If B is divisible by c, set A to the exact quotient and return 1, otherwise set A to zero and return 0.
4.10.15 Differentiation/Integration

void fmpz_mpoly_derivative(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong var, const fmpz_mpoly_ctx_t ctx)

Set A to the derivative of B with respect to the variable of index var.

void fmpz_mpoly_integral(fmpz_mpoly_t A, fmpz_t scale, const fmpz_mpoly_t B, slong var, const fmpz_mpoly_ctx_t ctx)

Set A and scale so that A is an integral of scale \times B with respect to the variable of index var, where scale is positive and as small as possible.

4.10.16 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

int fmpz_mpoly_evaluate_all_fmpz(fmpz_t ev, const fmpz_mpoly_t A, fmpz *const *vals, const fmpz_mpoly_ctx_t ctx)

Set ev to the evaluation of A where the variables are replaced by the corresponding elements of the array vals. Return 1 for success and 0 for failure.

int fmpz_mpoly_evaluate_one_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong var, const fmpz_t val, const fmpz_mpoly_ctx_t ctx)

Set A to the evaluation of B where the variable of index var is replaced by val. Return 1 for success and 0 for failure.

int fmpz_mpoly_compose_fmpz_poly(fmpz_poly_t A, const fmpz_mpoly_t B, fmpz_poly_struct *const *C, const fmpz_mpoly_ctx_t ctxB)

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C. The context object of B is ctxB. Return 1 for success and 0 for failure.

int fmpz_mpoly_compose_fmpz_mpoly_geobucket(fmpz_mpoly_t A, const fmpz_mpoly_t B, fmpz_mpoly_struct *const *C, const fmpz_mpoly_ctx_t ctxB, const fmpz_mpoly_ctx_t ctxAC)

int fmpz_mpoly_compose_fmpz_mpoly_horner(fmpz_mpoly_t A, const fmpz_mpoly_t B, fmpz_mpoly_struct *const *C, const fmpz_mpoly_ctx_t ctxB, const fmpz_mpoly_ctx_t ctxAC)

int fmpz_mpoly_compose_fmpz_mpoly(fmpz_mpoly_t A, const fmpz_mpoly_t B, fmpz_mpoly_struct *const *C, const fmpz_mpoly_ctx_t ctxB, const fmpz_mpoly_ctx_t ctxAC)

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C. Both A and the elements of C have context object ctxAC, while B has context object ctxB. The length of the array C is the number of variables in ctxB. Neither A nor B is allowed to alias any other polynomial. Return 1 for success and 0 for failure. The main method attempts to perform the calculation using matrices and chooses heuristically between the geobucket and horner methods if needed.

void fmpz_mpoly_compose_fmpz_mpoly_gen(fmpz_mpoly_t A, const fmpz_mpoly_t B, const slong *c, const fmpz_mpoly_ctx_t ctxB, const fmpz_mpoly_ctx_t ctxAC)

Set A to the evaluation of B where the variable of index i in ctxB is replaced by the variable of index c[i] in ctxAC. The length of the array C is the number of variables in ctxB. If any c[i] is negative, the corresponding variable of B is replaced by zero. Otherwise, it is expected that c[i] is less than the number of variables in ctxAC.
4.10.17 Multiplication

```c
void fmpz_mpoly_mul(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const fmpz_mpoly_ctx_t ctx);
void fmpz_mpoly_mul_threaded(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const fmpz_mpoly_ctx_t ctx, slong thread_limit);
void fmpz_mpoly_mul_johnson(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const fmpz_mpoly_ctx_t ctx);
void fmpz_mpoly_mul_heap_threaded(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const fmpz_mpoly_ctx_t ctx);
```

Set \( A \) to \( B \times C \).

```c
int fmpz_mpoly_mul_array(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const fmpz_mpoly_ctx_t ctx);
int fmpz_mpoly_mul_array_threaded(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const fmpz_mpoly_ctx_t ctx);
int fmpz_mpoly_mul_dense(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const fmpz_mpoly_ctx_t ctx);
```

Try to set \( A \) to \( B \times C \) using arrays. If the return is 0, the operation was unsuccessful. Otherwise, it was successful and the return is 1. The first version always uses one thread.

4.10.18 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

```c
int fmpz_mpoly_pow_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t k, const fmpz_mpoly_ctx_t ctx);
int fmpz_mpoly_pow_ui(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong k, const fmpz_mpoly_ctx_t ctx);
```

Set \( A \) to \( B \) raised to the \( k \)-th power. Return 1 for success and 0 for failure.

4.10.19 Division

```c
int fmpz_mpoly_divides(fmpz_mpoly_t Q, const fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx);
void fmpz_mpoly_divrem(fmpz_mpoly_t Q, const fmpz_mpoly_t R, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx);
void fmpz_mpoly_quasidivrem(fmpz_t scale, fmpz_mpoly_t Q, fmpz_mpoly_t R, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx);
```

Set \( Q \) and \( R \) to the quotient and remainder of \( A \) divided by \( B \). The monomials in \( R \) divisible by the leading monomial of \( B \) will have coefficients reduced modulo the absolute value of the leading coefficient of \( B \). Note that this function is not very useful if the leading coefficient \( B \) is not a unit. No monomials in \( R \) will be divisible by the leading monomial of \( B \).
void \texttt{fmpz_mpoly\_div}(fmpz\_mpoly\_t Q, const fmpz\_mpoly\_t A, const fmpz\_mpoly\_t B, const fmpz\_mpoly\_ctx\_t ctx)

Perform the operation of \texttt{fmpz\_mpoly\_divrem()} and discard \( R \). Note that this function is not very useful if the division is not exact and the leading coefficient \( B \) is not a unit.

void \texttt{fmpz\_mpoly\_quasidiv}(fmpz\_t scale, fmpz\_mpoly\_t Q, const fmpz\_mpoly\_t A, const fmpz\_mpoly\_t B, const fmpz\_mpoly\_ctx\_t ctx)

Perform the operation of \texttt{fmpz\_mpoly\_quasidivrem()} and discard \( R \).

void \texttt{fmpz\_mpoly\_divrem\_ideal}(fmpz\_mpoly\_struct **Q, fmpz\_mpoly\_t R, const fmpz\_mpoly\_t A, fmpz\_mpoly\_struct *const *B, slong len, const fmpz\_mpoly\_ctx\_t ctx)

This function is as per \texttt{fmpz\_mpoly\_divrem()} except that it takes an array of divisor polynomials \( B \) and it returns an array of quotient polynomials \( Q \). The number of divisor (and hence quotient) polynomials is given by \( \text{len} \). Note that this function is not very useful if there is no unit among the leading coefficients in the array \( B \).

void \texttt{fmpz\_mpoly\_quasidivrem\_ideal}(fmpz\_t scale, fmpz\_mpoly\_struct **Q, fmpz\_mpoly\_t R, const fmpz\_mpoly\_t A, fmpz\_mpoly\_struct *const *B, slong len, const fmpz\_mpoly\_ctx\_t ctx)

This function is as per \texttt{fmpz\_mpoly\_quasidivrem()} except that it takes an array of divisor polynomials \( B \) and it returns an array of quotient polynomials \( Q \). The number of divisor (and hence quotient) polynomials is given by \( \text{len} \).

4.10.20 Greatest Common Divisor

void \texttt{fmpz\_mpoly\_term\_content}(fmpz\_mpoly\_t M, const fmpz\_mpoly\_t A, const fmpz\_mpoly\_ctx\_t ctx)

Set \( M \) to the GCD of the terms of \( A \). If \( A \) is zero, \( M \) will be zero. Otherwise, \( M \) will be a monomial with positive coefficient.

int \texttt{fmpz\_mpoly\_content\_vars}(fmpz\_mpoly\_t g, const fmpz\_mpoly\_t A, slong *vars, slong vars\_length, const fmpz\_mpoly\_ctx\_t ctx)

Set \( g \) to the GCD of the coefficients of \( A \) when viewed as a polynomial in the variables \( \text{vars} \). Return 1 for success and 0 for failure. Upon success, \( g \) will be independent of the variables \( \text{vars} \).

int \texttt{fmpz\_mpoly\_gcd}(fmpz\_mpoly\_t G, const fmpz\_mpoly\_t A, const fmpz\_mpoly\_t B, const fmpz\_mpoly\_ctx\_t ctx)

Try to set \( G \) to the GCD of \( A \) and \( B \) with positive leading coefficient. The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and \( G \) is left untouched.

int \texttt{fmpz\_mpoly\_gcd\_cofactors}(fmpz\_mpoly\_t G, fmpz\_mpoly\_t Abar, fmpz\_mpoly\_t Bbar, const fmpz\_mpoly\_t A, const fmpz\_mpoly\_t B, const fmpz\_mpoly\_ctx\_t ctx)

Do the operation of \texttt{fmpz\_mpoly\_gcd()} and also compute \( Abar = A/G \) and \( Bbar = B/G \) if successful.

int \texttt{fmpz\_mpoly\_gcd\_brown}(fmpz\_mpoly\_t G, const fmpz\_mpoly\_t A, const fmpz\_mpoly\_t B, const fmpz\_mpoly\_ctx\_t ctx)

int \texttt{fmpz\_mpoly\_gcd\_hensel}(fmpz\_mpoly\_t G, const fmpz\_mpoly\_t A, const fmpz\_mpoly\_t B, const fmpz\_mpoly\_ctx\_t ctx)

int \texttt{fmpz\_mpoly\_gcd\_subresultant}(fmpz\_mpoly\_t G, const fmpz\_mpoly\_t A, const fmpz\_mpoly\_t B, const fmpz\_mpoly\_ctx\_t ctx)

int \texttt{fmpz\_mpoly\_gcd\_zippel}(fmpz\_mpoly\_t G, const fmpz\_mpoly\_t A, const fmpz\_mpoly\_t B, const fmpz\_mpoly\_ctx\_t ctx)
int fmpz_mpoly_gcd_zippel2(fmpz_mpoly_t G, const fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)

Try to set $G$ to the GCD of $A$ and $B$ using various algorithms.

int fmpz_mpoly_resultant(fmpz_mpoly_t R, const fmpz_mpoly_t A, const fmpz_mpoly_t B, slong var, const fmpz_mpoly_ctx_t ctx)

Try to set $R$ to the resultant of $A$ and $B$ with respect to the variable of index $var$.

int fmpz_mpoly_discriminant(fmpz_mpoly_t D, const fmpz_mpoly_t A, slong var, const fmpz_mpoly_ctx_t ctx)

Try to set $D$ to the discriminant of $A$ with respect to the variable of index $var$.

void fmpz_mpoly_primitive_part(fmpz_mpoly_t res, const fmpz_mpoly_t f, const fmpz_mpoly_ctx_t ctx)

Sets $res$ to the primitive part of $f$, obtained by dividing out the content of all coefficients and normalizing the leading coefficient to be positive. The zero polynomial is unchanged.

### 4.10.21 Square Root

int fmpz_mpoly_sqrt_heap(fmpz_mpoly_t Q, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx, int check)

If $A$ is a perfect square return 1 and set $Q$ to the square root with positive leading coefficient. Otherwise return 0 and set $Q$ to the zero polynomial. If $check = 0$ the polynomial is assumed to be a perfect square. This can be significantly faster, but it will not detect non-squares with any reliability, and in the event of being passed a non-square the result is meaningless.

int fmpz_mpoly_sqrt(fmpz_mpoly_t q, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)

If $A$ is a perfect square return 1 and set $Q$ to the square root with positive leading coefficient. Otherwise return 0 and set $Q$ to zero.

int fmpz_mpoly_is_square(const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)

Return 1 if $A$ is a perfect square, otherwise return 0.

### 4.10.22 Univariate Functions

An fmpz_mpoly_univar_t holds a univariate polynomial in some main variable with fmpz_mpoly_t coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of $\mathbb{Z}[x_1, \ldots, x_m]$ as an element of $(\mathbb{Z}[x_1, \ldots, x_{v-1}, x_{v+1}, \ldots, x_m])[x_v]$ and vice versa.

void fmpz_mpoly_univar_init(fmpz_mpoly_univar_t A, const fmpz_mpoly_ctx_t ctx)

Initialize $A$.

void fmpz_mpoly_univar_clear(fmpz_mpoly_univar_t A, const fmpz_mpoly_ctx_t ctx)

Clear $A$.

void fmpz_mpoly_univar_swap(fmpz_mpoly_univar_t A, fmpz_mpoly_univar_t B, const fmpz_mpoly_ctx_t ctx)

Swap $A$ and $B$.

void fmpz_mpoly_to_univar(fmpz_mpoly_univar_t A, const fmpz_mpoly_t B, slong var, const fmpz_mpoly_ctx_t ctx)

Set $A$ to a univariate form of $B$ by pulling out the variable of index $var$. The coefficients of $A$ will still belong to the content $ctx$ but will not depend on the variable of index $var$. 

4.10. fmpz_mpoly.h – multivariate polynomials over the integers 235
void fmpz_mpoly_from_univar(fmpz_mpoly_t A, const fmpz_mpoly_univar_t B, slong var, const fmpz_mpoly_ctx_t ctx)

Set A to the normal form of B by putting in the variable of index var. This function is undefined if the coefficients of B depend on the variable of index var.

int fmpz_mpoly_univar_degree_fits_si(const fmpz_mpoly_univar_t A, const fmpz_mpoly_ctx_t ctx)

Return 1 if the degree of A with respect to the main variable fits an slong. Otherwise, return 0.

slong fmpz_mpoly_univar_length(const fmpz_mpoly_univar_t A, const fmpz_mpoly_ctx_t ctx)

Return the number of terms in A with respect to the main variable.

slong fmpz_mpoly_univar_get_term_exp_si(fmpz_mpoly_univar_t A, slong i, const fmpz_mpoly_ctx_t ctx)

Return the exponent of the term of index i of A.

void fmpz_mpoly_univar_get_term_coeff(fmpz_mpoly_t c, const fmpz_mpoly_univar_t A, slong i, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_univar_swap_term_coeff(fmpz_mpoly_t c, fmpz_mpoly_univar_t A, slong i, const fmpz_mpoly_ctx_t ctx)

Set (resp. swap) c to (resp. with) the coefficient of the term of index i of A.

4.10.23 Internal Functions

void fmpz_mpoly_inflate(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz *shift, const fmpz *stride, const fmpz_mpoly_ctx_t ctx)

Apply the function e -> shift[v] + stride[v]*e to each exponent e corresponding to the variable v. It is assumed that each shift and stride is not negative.

void fmpz_mpoly_deflate(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz *shift, const fmpz *stride, const fmpz_mpoly_ctx_t ctx)

Apply the function e -> (e - shift[v])/stride[v] to each exponent e corresponding to the variable v. If any stride[v] is zero, the corresponding numerator e - shift[v] is assumed to be zero, and the quotient is defined as zero. This allows the function to undo the operation performed by fmpz_mpoly_inflate() when possible.

void fmpz_mpoly_deflation(fmpz *shift, fmpz *stride, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)

For each variable v let S_v be the set of exponents appearing on v. Set shift[v] to min(S_v) and set stride[v] to gcd(S - min(S_v)). If A is zero, all shifts and strides are set to zero.

void fmpz_mpoly_pow_fps(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong k, const fmpz_mpoly_ctx_t ctx)

Set A to B raised to the k-th power, using the Monagan and Pearce FPS algorithm. It is assumed that B is not zero and k ≥ 2.

slong _fmpz_mpoly_divides_array(fmpz **poly1, ulong **exp1, slong *alloc, const fmpz *poly2, const ulong *exp2, slong len2, const fmpz *poly3, const ulong *exp3, slong len3, slong *mults, slong num, slong bits)

Use dense array exact division to set (poly1, exp1, alloc) to (poly2, exp3, len2) divided by (poly3, exp3, len3) in num variables, given a list of multipliers to tightly pack exponents and a number of bits for the fields of the exponents of the result. The array “mults” is a list of bases to be used in encoding the array indices from the exponents. The function reallocates its output, hence the double indirection, and returns the length of its output if the quotient is exact, or zero if not. It is assumed that poly2 is not zero. No aliasing is allowed.
int fmpz_mpoly_divides_array(fmpz_mpoly_t poly1, const fmpz_mpoly_t poly2, const fmpz_mpoly_t poly3, const fmpz_mpoly_ctx_t ctx)

Set poly1 to poly2 divided by poly3, using a big dense array to accumulate coefficients, and return 1 if the quotient is exact. Otherwise, return 0 if the quotient is not exact. If the array will be larger than some internally set parameter, the function fails silently and returns −1 so that some other method may be called. This function is most efficient on dense inputs. Note that the function fmpz_mpoly_div_monagan_pearce below may be much faster if the quotient is known to be exact.

Note: This function is only defined if the machine is known to be strongly ordered during the configuration. To check whether this function is defined during compilation-time, use the C preprocessor macro #ifdef fmpz_mpoly_divides_heap_threaded.

Note that, if the system is known to be strongly ordered, the underlying algorithm for this function is utilized in fmpz_mpoly_divides(). Hence, you may find it easier to use this function instead if the C preprocessor is not available.

slong fmpz_mpoly_div_monagan_pearce(fmpz **polyq, ulong **expq, slong *allocq, const fmpz *poly2, const ulong *exp2, slong len2, const fmpz *poly3, const ulong *exp3, slong len3, ulong bits, slong N, const mp_limb_t *cmpmask)

Set (poly1, exp1, alloc) to (poly2, exp3, len2) divided by (poly3, exp3, len3) and return 1 if the quotient is exact. Otherwise return 0. The function assumes exponent vectors that each fit in N words, and are packed into fields of the given number of bits. Assumes input polys are nonzero. Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce. No aliasing is allowed.

4.10. fmpz_mpoly.h – multivariate polynomials over the integers

void fmpz_mpoly_div_monagan_pearce(fmpz_mpoly_t polyq, const fmpz_mpoly_t poly2, const fmpz_mpoly_ctx_t ctx)

Set polyq to the quotient of poly2 by poly3, discarding the remainder (with notional remainder coefficients reduced modulo the leading coefficient of poly3). Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce. This function is exceptionally efficient if the division is known to be exact.
void fmpz_mpoly_divrem_monagan_pearce(slong *lenr, fmpz **polyq, ulong **expq, slong *allocq, fmpz **polyr, ulong **expr, slong *alocr, const fmpz *poly2, const ulong *exp2, slong len2, const fmpz *poly3, const ulong *exp3, slong len3, slong bits, slong N, const mp_limb_t *cmpmask)

Set (polyq, expq, allocq) and (polyr, expr, alocr) to the quotient and remainder of (poly2, exp2, len2) by (poly3, exp3, len3) (with remainder coefficients reduced modulo the leading coefficient of (poly3, exp3, len3)), and return the length of the quotient. The function reallocates its outputs, hence the double direction. The function assumes the exponent vectors all fit in N words. The exponent vectors are assumed to have fields with the given number of bits. Assumes input polynomials are nonzero. Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce. No aliasing is allowed.

void fmpz_mpoly_divrem_monagan_pearce(fmpz_mpoly_t q, fmpz_mpoly_t r, const fmpz_mpoly_t poly2, const fmpz_mpoly_t poly3, const fmpz_mpoly_ctx_t ctx)

Set polyq and polyr to the quotient and remainder of poly2 divided by poly3 (with remainder coefficients reduced modulo the leading coefficient of poly3). Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce.

slong _fmpz_mpoly_divrem_array(slong *lenr, fmpz **polyq, ulong **expq, slong *allocq, fmpz **polyr, ulong **expr, slong *alocr, const fmpz *poly2, const ulong *exp2, slong len2, const fmpz *poly3, const ulong *exp3, slong len3, slong bits, slong N, const mp_limb_t *cmpmask)

Use dense array division to set (polyq, expq, allocq) and (polyr, expr, alocr) to the quotient and remainder of (poly2, exp2, len2) divided by (poly3, exp3, len3) in num variables, given a list of multipliers to tightly pack exponents and a number of bits for the fields of the exponents of the result. The function reallocates its outputs, hence the double direction. The array mults is a list of bases to be used in encoding the array indices from the exponents. The function returns the length of the quotient. It is assumed that the input polynomials are not zero. No aliasing is allowed.

int fmpz_mpoly_divrem_array(fmpz_mpoly_t q, fmpz_mpoly_t r, const fmpz_mpoly_t poly2, const fmpz_mpoly_t poly3, const fmpz_mpoly_ctx_t ctx)

Set polyq and polyr to the quotient and remainder of poly2 divided by poly3 (with remainder coefficients reduced modulo the leading coefficient of poly3). The function is implemented using dense arrays, and is efficient when the inputs are fairly dense. If the array will be larger than some internally set parameter, the function silently returns 0 so that another function can be called, otherwise it returns 1.

void fmpz_mpoly_quasidivrem_heap(fmpz_t scale, fmpz_mpoly_t q, fmpz_mpoly_t r, const fmpz_mpoly_t poly2, const fmpz_mpoly_t poly3, const fmpz_mpoly_ctx_t ctx)

Set scale, q and r so that scale*poly2 = q*poly3 + r and no monomial in r is divisible by the leading monomial of poly3, where scale is positive and as small as possible. This function throws an exception if poly3 is zero or if an exponent overflow occurs.

slong _fmpz_mpoly_divrem_ideal_monagan_pearce(fmpz_mpoly_struct **polyq, fmpz **polyr, ulong **expr, slong *alocr, const fmpz *poly2, const ulong *exp2, slong len2, fmpz_mpoly_struct *const *poly3, ulong *const *exp3, slong len, slong N, slong bits, const fmpz_mpoly_ctx_t ctx, const mp_limb_t *cmpmask)

This function is as per _fmpz_mpoly_divrem_monagan_pearce except that it takes an array of divisor polynomials poly3 and an array of repacked exponent arrays exp3, which may alias the exponent arrays of poly3, and it returns an array of quotient polynomials polyq. The number of divisor (and hence quotient) polynomials is given by len. The function computes polynomials q_i such that r = a - \sum_{i=0}^{len-1} q_i b_i, where the q_i are the quotient polynomials and the b_i are the divisor polynomials.
This function is as per \texttt{fmpz_mpoly_divrem_monagan_pearce} except that it takes an array of divisor polynomials \texttt{poly3}, and it returns an array of quotient polynomials \texttt{q}. The number of divisor (and hence quotient) polynomials is given by \texttt{len}. The function computes polynomials $q_i = q[i]$ such that $poly2 = r + \sum_{i=0}^{len-1} q_i b_i$, where $b_i = poly3[i]$.

### 4.10.24 Vectors

- **\texttt{fmpz_mpoly_vec_struct}**
- **\texttt{fmpz_mpoly_vec_t}**
  - A type holding a vector of \texttt{fmpz_mpoly_t}.

\texttt{fmpz_mpoly_vec_entry(v, i)}

Macro for accessing the entry at position $i$ in \texttt{v}.

\texttt{void fmpz_mpoly_vec_init(fmpz_mpoly_vec_t vec, slong len, const fmpz_mpoly_ctx_t ctx)}

Initializes \texttt{vec} to a vector of length \texttt{len}, setting all entries to the zero polynomial.

\texttt{void fmpz_mpoly_vec_clear(fmpz_mpoly_vec_t vec, const fmpz_mpoly_ctx_t ctx)}

Clears \texttt{vec}, freeing its allocated memory.

\texttt{void fmpz_mpoly_vec_print(const fmpz_mpoly_vec_t vec, const fmpz_mpoly_ctx_t ctx)}

Prints \texttt{vec} to standard output.

\texttt{void fmpz_mpoly_vec_swap(fmpz_mpoly_vec_t x, fmpz_mpoly_vec_t y, const fmpz_mpoly_ctx_t ctx)}

Swaps \texttt{x} and \texttt{y} efficiently.

\texttt{void fmpz_mpoly_vec_fit_length(fmpz_mpoly_vec_t vec, slong len, const fmpz_mpoly_ctx_t ctx)}

Allocates room for \texttt{len} entries in \texttt{vec}.

\texttt{void fmpz_mpoly_vec_set(fmpz_mpoly_vec_t dest, const fmpz_mpoly_vec_t src, const fmpz_mpoly_ctx_t ctx)}

Sets \texttt{dest} to a copy of \texttt{src}.

\texttt{void fmpz_mpoly_vec_append(fmpz_mpoly_vec_t vec, const fmpz_mpoly_t f, const fmpz_mpoly_ctx_t ctx)}

Appends \texttt{f} to the end of \texttt{vec}.

\texttt{slong fmpz_mpoly_vec_insert_unique(fmpz_mpoly_vec_t vec, const fmpz_mpoly_t f, const fmpz_mpoly_ctx_t ctx)}

Inserts \texttt{f} without duplication into \texttt{vec} and returns its index. If this polynomial already exists, \texttt{vec} is unchanged. If this polynomial does not exist in \texttt{vec}, it is appended.

\texttt{void fmpz_mpoly_vec_set_length(fmpz_mpoly_vec_t vec, slong len, const fmpz_mpoly_ctx_t ctx)}

Sets the length of \texttt{vec} to \texttt{len}, truncating or zero-extending as needed.

\texttt{void fmpz_mpoly_vec_randtest_not_zero(fmpz_mpoly_vec_t vec, flint_rand_t state, slong len, slong poly_len, slong bits, ulong exp_bound, fmpz_mpoly_ctx_t ctx)}

Sets \texttt{vec} to a random vector with exactly \texttt{len} entries, all nonzero, with random parameters defined by \texttt{poly_len}, \texttt{bits} and \texttt{exp_bound}.

\texttt{void fmpz_mpoly_vec_set_primitive_unique(fmpz_mpoly_vec_t res, const fmpz_mpoly_vec_t src, const fmpz_mpoly_ctx_t ctx)}

Sets \texttt{res} to a vector containing all polynomials in \texttt{src} reduced to their primitive parts, without duplication. The zero polynomial is skipped if present. The output order is arbitrary.
4.10.25 Ideals and Gröbner bases

The following methods deal with ideals in \( \mathbb{Q}[X_1, \ldots, X_n] \). We use primitive integer polynomials as normalised generators in place of monic rational polynomials.

```c
void fmpz_mpoly_spoly(fmpz_mpoly_t res, const fmpz_mpoly_t f, const fmpz_mpoly_t g, const fmpz_mpoly_ctx_t ctx)
```

Sets \( \text{res} \) to the \( S \)-polynomial of \( f \) and \( g \), scaled to an integer polynomial by computing the LCM of the leading coefficients.

```c
void fmpz_mpoly_reduction_primitive_part(fmpz_mpoly_t res, const fmpz_mpoly_t f, const fmpz_mpoly_vec_t vec, const fmpz_mpoly_ctx_t ctx)
```

Sets \( \text{res} \) to the primitive part of the reduction (remainder of multivariate quasidivision with remainder) with respect to the polynomials \( \text{vec} \).

```c
int fmpz_mpoly_vec_is_groebner(const fmpz_mpoly_vec_t G, const fmpz_mpoly_vec_t F, const fmpz_mpoly_ctx_t ctx)
```

If \( F \) is \( \text{NULL} \), checks if \( G \) is a Gröbner basis. If \( F \) is not \( \text{NULL} \), checks if \( G \) is a Gröbner basis for \( F \).

```c
int fmpz_mpoly_vec_is_autoreduced(const fmpz_mpoly_vec_t F, const fmpz_mpoly_ctx_t ctx)
```

Checks whether the vector \( F \) is autoreduced (or inter-reduced).

```c
void fmpz_mpoly_vec_autoreduction(fmpz_mpoly_vec_t H, const fmpz_mpoly_vec_t F, const fmpz_mpoly_ctx_t ctx)
```

Sets \( \text{H} \) to the autoreduction (inter-reduction) of \( F \).

```c
void fmpz_mpoly_vec_autoreduction_groebner(fmpz_mpoly_vec_t H, const fmpz_mpoly_vec_t G, const fmpz_mpoly_ctx_t ctx)
```

Sets \( \text{H} \) to the autoreduction (inter-reduction) of \( G \). Assumes that \( G \) is a Gröbner basis. This produces a reduced Gröbner basis, which is unique (up to the sort order of the entries in the vector).

```c
void fmpz_mpoly_buchberger_naive(fmpz_mpoly_vec_t G, const fmpz_mpoly_vec_t F, const fmpz_mpoly_ctx_t ctx)
```

Sets \( \text{G} \) to a Gröbner basis for \( F \), computed using a naive implementation of Buchberger’s algorithm.

```c
int fmpz_mpoly_buchberger_naive_with_limits(fmpz_mpoly_vec_t G, const fmpz_mpoly_vec_t F, slong ideal_len_limit, slong poly_len_limit, slong poly_bits_limit, const fmpz_mpoly_ctx_t ctx)
```

As \( \text{fmpz_mpoly_buchberger_naive()} \), but halts if during the execution of Buchberger’s algorithm the length of the ideal basis set exceeds \( \text{ideal_len_limit} \), the length of any polynomial exceeds \( \text{poly_len_limit} \), or the size of the coefficients of any polynomial exceeds \( \text{poly_bits_limit} \). Returns 1 for success and 0 for failure. On failure, \( \text{G} \) is a valid basis for \( F \) but it might not be a Gröbner basis.

4.10.26 Special polynomials

```c
void fmpz_mpoly_symmetric_gens(fmpz_mpoly_t res, ulong k, slong *vars, slong n, const fmpz_mpoly_ctx_t ctx)
```

```c
void fmpz_mpoly_symmetric(fmpz_mpoly_t res, ulong k, const fmpz_mpoly_ctx_t ctx)
```

Sets \( \text{res} \) to the elementary symmetric polynomial \( e_k(X_1, \ldots, X_n) \).

The \text{gens} version takes \( X_1, \ldots, X_n \) to be the subset of generators given by \( \text{vars} \) and \( n \). The indices in \( \text{vars} \) start from zero. Currently, the indices in \( \text{vars} \) must be distinct.
4.11 \texttt{fmpz_mpoly_factor.h} – factorisation of multivariate polynomials over the integers

4.11.1 Types, macros and constants

\begin{itemize}
  \item \textit{type} \texttt{fmpz_mpoly_factor_struct}  
    A struct for holding a factored integer polynomial. There is a single constant and a product of bases to corresponding exponents.
  \item \textit{type} \texttt{fmpz_mpoly_factor_t}  
    An array of length 1 of \texttt{fmpz_mpoly_factor_struct}.
\end{itemize}

4.11.2 Memory management

\begin{itemize}
  \item \texttt{void fmpz_mpoly_factor_init(fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)}  
    Initialise \texttt{f}.
  \item \texttt{void fmpz_mpoly_factor_clear(fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)}  
    Clear \texttt{f}.
\end{itemize}

4.11.3 Basic manipulation

\begin{itemize}
  \item \texttt{void fmpz_mpoly_factor_swap(fmpz_mpoly_factor_t f, fmpz_mpoly_factor_t g, const fmpz_mpoly_ctx_t ctx)}  
    Efficiently swap \texttt{f} and \texttt{g}.
  \item \texttt{slong fmpz_mpoly_factor_length(const fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)}  
    Return the length of the product in \texttt{f}.
  \item \texttt{void fmpz_mpoly_factor_get_constant_fmpz(fmpz_t c, const fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)}  
    Set \texttt{c} to the constant of \texttt{f}.
  \item \texttt{void fmpz_mpoly_factor_get_base(fmpz_mpoly_t B, const fmpz_mpoly_factor_t f, slong i, const fmpz_mpoly_ctx_t ctx)}  
    Set \texttt{B} to (resp. swap) the base of the term of index \texttt{i} in \texttt{f}.
  \item \texttt{slong fmpz_mpoly_factor_get_exp_si(fmpz_mpoly_factor_t f, slong i, const fmpz_mpoly_ctx_t ctx)}  
    Return the exponent of the term of index \texttt{i} in \texttt{f}. It is assumed to fit an slong.
  \item \texttt{void fmpz_mpoly_factor_sort(fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)}  
    Sort the product of \texttt{f} first by exponent and then by base.
\end{itemize}
4.11.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and $f$ is undefined. None of these functions multiply $f$ by $A$: $f$ is simply set to a factorisation of $A$, and thus these functions should not depend on the initial value of the output $f$.

```c
int fmpz_mpolynomial_factor_squarefree(fmpz_mpolynomial_t f, const fmpz_mpolynomial_t A, const fmpz_mpolynomial_ctx_t ctx)
```

Set $f$ to a factorization of $A$ where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call `fmpz_mpolynomial_factor_sort()` and then multiply the bases with the desired exponent.

```c
int fmpz_mpolynomial_factor(fmpz_mpolynomial_t f, const fmpz_mpolynomial_t A, const fmpz_mpolynomial_ctx_t ctx)
```

Set $f$ to a factorization of $A$ where the bases are irreducible.

4.12 long_extras.h – support functions for signed word arithmetic

4.12.1 Properties

```c
size_t z_sizeinbase(slong n, int b)
```

Returns the number of digits in the base $b$ representation of the absolute value of the integer $n$.

Assumes that $b \geq 2$.

4.12.2 Checked Arithmetic

```c
int z_mul_checked(slong *a, slong b, slong c)
```

Set *a to $b$ times $c$ and return 1 if the product overflowed. Otherwise, return 0.

4.12.3 Random functions

```c
mp_limb_signed_t z_randtest(flint_rand_t state)
```

Returns a pseudo random number with a random number of bits, from 0 to FLINT_BITS. The probability of the special values 0, ±1, COEFF_MAX, COEFF_MIN, WORD_MAX and WORD_MIN is increased.

This random function is mainly used for testing purposes.

```c
mp_limb_signed_t z_randtest_not_zero(flint_rand_t state)
```

As for `z_randtest(state)`, but does not return 0.

```c
mp_limb_signed_t z_randint(flint_rand_t state, mp_limb_t limit)
```

Returns a pseudo random number of absolute value less than `limit`. If `limit` is zero or exceeds WORD_MAX, it is interpreted as WORD_MAX.
4.12.4 Modular arithmetic

```c
int z_kronecker(slong a, slong n)
```

Return the Kronecker symbol $\left( \frac{a}{n} \right)$ for any $a$ and any $n$.

4.13 `longlong.h` – support functions for multi-word arithmetic

4.13.1 Auxiliary asm macros

```c
umul_ppmm(high_prod, low_prod, multiplier, multiplicand)
```

Multiplies two single limb integers `MULTIPLIER` and `MULTIPLICAND`, and generates a two limb product in `HIGH_PROD` and `LOW_PROD`.

```c
smul_ppmm(high_prod, low_prod, multiplier, multiplicand)
```

As for `umul_ppmm()` but the numbers are signed.

```c
udiv_qrnnd(quotient, remainder, high_numerator, low_numerator, denominator)
```

Divides an unsigned integer, composed by the limb integers `HIGH_NUMERATOR` and `LOW_NUMERATOR`, by `DENOMINATOR` and places the quotient in `QUOTIENT` and the remainder in `REMAINDER`. `HIGH_NUMERATOR` must be less than `DENOMINATOR` for correct operation.

```c
div_qrnnd(quotient, remainder, high_numerator, low_numerator, denominator)
```

As for `udiv_qrnnd()` but the numbers are signed. The quotient is rounded towards 0. Note that as the quotient is signed it must lie in the range $[-2^63, 2^63)$.

```c
flint_clz(x)
```

Returns the number of zero-bits from the msb to the first non-zero bit in the limb `x`. This is the number of steps `x` needs to be shifted left to set the msb. If `x` is 0 then the return value is undefined.

```c
flint_ctz(x)
```

As for `flint_clz()`, but counts from the least significant end. If `x` is zero then the return value is undefined.

```c
add_ssaaaa(high_sum, low_sum, high_addend_1, low_addend_1, high_addend_2, low_addend_2)
```

Adds two limb integers, composed by `HIGH_ADDEND_1` and `LOW_ADDEND_1`, and `HIGH_ADDEND_2` and `LOW_ADDEND_2`, respectively. The result is placed in `HIGH_SUM` and `LOW_SUM`. Overflow, i.e. carry out, is not stored anywhere, and is lost.

```c
add_sssaaaaaa(high_sum, mid_sum, low_sum, high_addend_1, mid_addend_1, low_addend_1, high_addend_2, mid_addend_2, low_addend_2)
```

Adds two three limb integers. Carry out is lost.

```c
sub_ddmmss(high_difference, low_difference, high_minuend, low_minuend, high_subtrahend, low_subtrahend)
```

Subtracts two limb integers, composed by `HIGH_MINUEND_1` and `LOW_MINUEND_1`, and `HIGH_SUBTRAHEND_2` and `LOW_SUBTRAHEND_2`, respectively. The result is placed in `HIGH_DIFFERENCE` and `LOW_DIFFERENCE`. Overflow, i.e. carry out is not stored anywhere, and is lost.

```c
sub_dddmmmssss(high_diff, mid_diff, low_diff, high_minuend_1, mid_minuend_1, low_minuend_1, high_subtrahend_2, mid_subtrahend_2, low_subtrahend_2)
```

Subtracts two three limb integers. Borrow out is lost.

```c
byte_swap(x)
```

Swap the order of the bytes in the word `x`, i.e. most significant byte becomes least significant byte, etc.
invert_limb(invxl, xl)
        Deprecated: see n_preinvert_limb_prenorm().

udiv_qrnde_preinv(q, r, nh, nl, d, di)
        As for udiv_qrnde() but takes a precomputed inverse di as computed by invert_limb(). The
        algorithm, in terms of the theorem above, is:

        \[
        \begin{align*}
        \text{nadj} & = n1 \ast (d-B/2) + n0 \\
        xh, xl & = (n2+n1) \ast (m-B) \\
        xh, xl & += \text{nadj} + n2 \ast B \quad (xh, xl = n2 \ast B + (n2+n1) \ast (m-B) + n1 \ast (d-B/2) + n0) \\
        \_q1 & = B - xh - 1 \\
        xh, xl & = \_q1 \ast d + nh, nl - B \ast d = nh, nl - \_q1 \ast d - d \quad \text{so that } xh = 0 \text{ or } -1 \\
        r & = xl + xh \ast d \quad \text{where } xh \text{ is } 0 \text{ if } \_q1 \text{ is off by } 1, \text{ otherwise } -1 \\
        q & = xh - \_q1 = xh + 1 + n2
        \end{align*}
        \]

4.14 mpn_extras.h – support functions for limb arrays

4.14.1 Macros

MPN_NORM(a, an)
        Normalise (a, an) so that either an is zero or a[an - 1] is nonzero.

MPN_SWAP(a, an, b, bn)
        Swap (a, an) and (b, bn), i.e. swap pointers and sizes.

4.14.2 Utility functions

void flint_mpn_debug(mp_srcptr x, mp_size_t xsize)
        Prints debug information about (x, xsize) to stdout. In particular, this will print binary repre-
        sentations of all the limbs.

int flint_mpn_zero_p(mp_srcptr x, mp_size_t xsize)
        Returns 1 if all limbs of (x, xsize) are zero, otherwise 0.

mp_limb_t flint_mpn_sumdiff_n(mp_ptr s, mp_ptr d, mp_srcptr x, mp_srcptr y, mp_size_t n)
        Simultaneously computes the sum s and difference d of (x, n) and (y, n), returning carry mul-
        tiplied by two plus borrow.

4.14.3 Multiplication

mp_limb_t flint_mpn_mul(mp_ptr z, mp_srcptr x, mp_size_t xn, mp_srcptr y, mp_size_t yn)
        Sets (z, xn+yn) to the product of (x, xn) and (y, yn) and returns the top limb of the result.
        We require xn \geq yn \geq 1 and that z is not aliased with either input operand. This function uses
        FFT multiplication if the operands are large enough and otherwise calls mpn_mul.

void flint_mpn_mul_n(mp_ptr z, mp_srcptr x, mp_srcptr y, mp_size_t n)
        Sets z to the product of (x, n) and (y, n). We require n \geq 1 and that z is not aliased with either input
        operand. This function uses FFT multiplication if the operands are large enough and otherwise
        calls mpn_mul_n.

void flint_mpn_sqr(mp_ptr z, mp_srcptr x, mp_size_t n)
        Sets z to the square of (x, n). We require n \geq 1 and that z is not aliased with either input
        operand. This function uses FFT multiplication if the operands are large enough and otherwise
        calls mpn_sqr.
Given not-necessarily-normalized $x_1$ and $x_2$ of length $n > 0$ and output $y$ of length $n$, try to compute $y = a_1 \cdot x_1 - a_2 \cdot x_2$. Return the normalized length of $y$ if $y \geq 0$ and $y$ fits into $n$ limbs. Otherwise, return $-1$. $y$ may alias $x_1$ but is not allowed to alias $x_2$.

Sets $\{rp, n\}$ to the “high part” of $\{xp, n\}$ times $\{yp, n\}$ and returns the $n$-th least significant limb, in the sense that the multiplication of the $n - 1$ lower limbs are never accounted for. Hence, the multiplication is typically non-exact for sizes larger than one. The highest error is $n + 2$ ULP in the returned limb.

Note: This function may not exist on processors not supporting the ADX instruction set.

### 4.14.4 Divisibility

**int flint_mpn_divisible_1_odd(mp_srcptr x, mp_size_t xsize, mp_limb_t d)**

Expression determining whether $(x, xsize)$ is divisible by the $mp\_limb\_t\ d$ which is assumed to be odd-valued and at least 3.

This function is implemented as a macro.

**mp_size_t flint_mpn_divexact_1(mp_ptr x, mp_size_t xsize, mp_limb_t d)**

Divides $x$ once by a known single-limb divisor, returns the new size.

**mp_size_t flint_mpn_remove_2exp(mp_ptr x, mp_size_t xsize, flint_bitcnt_t *bits)**

Divides $(x, xsize)$ by $2^n$ where $n$ is the number of trailing zero bits in $x$. The new size of $x$ is returned, and $n$ is stored in the bits argument. $x$ may not be zero.

**mp_size_t flint_mpn_remove_power_ascending(mp_ptr x, mp_size_t xsize, mp_ptr p, mp_size_t psize, ulong *exp)**

Divides $(x, xsize)$ by the largest power $n$ of $(p, psize)$ that is an exact divisor of $x$. The new size of $x$ is returned, and $n$ is stored in the $exp$ argument. $x$ may not be zero, and $p$ must be greater than 2.

This function works by testing divisibility by ascending squares $p, p^2, p^4, p^8, \ldots$, making it efficient for removing potentially large powers. Because of its high overhead, it should not be used as the first stage of trial division.

**int flint_mpn_factor_trial(mp_srcptr x, mp_size_t xsize, slong start, slong stop)**

Searches for a factor of $(x, xsize)$ among the primes in positions $start, \ldots, stop-1$ of $flint\_primes$. Returns $i$ if $flint\_primes[i]$ is a factor, otherwise returns 0 if no factor is found. It is assumed that $start \geq 1$.

**int flint_mpn_factor_trial_tree(slong *factors, mp_srcptr x, mp_size_t xsize, slong num_primes)**

Searches for a factor of $(x, xsize)$ among the primes in positions approximately in the range $0, \ldots, num\_primes - 1$ of $flint\_primes$.

Returns the number of prime factors found and fills $factors$ with their indices in $flint\_primes$. It is assumed that $num\_primes$ is in the range $0, \ldots, 3512$.

If the input fits in a small $fmpz$ the number is fully factored instead.

The algorithm used is a tree based gcd with a product of primes, the tree for which is cached globally (it is threadsafe).
4.14.5 Division

```c
int flint_mpn_divides(mp_ptr q, mp_srcptr array1, mp_size_t limbs1, mp_srcptr arrayg,
                      mp_size_t limbsg, mp_ptr temp)
```

If (arrayg, limbsg) divides (array1, limbs1) then (q, limbs1 - limbsg + 1) is set to the quotient and 1 is returned, otherwise 0 is returned. The temporary space temp must have space for limbsg limbs.

Assumes limbs1 >= limbsg > 0.

```c
mp_limb_t flint_mpn_preinv1(mp_limb_t d, mp_limb_t d2)
```

Computes a precomputed inverse from the leading two limbs of the divisor b, n to be used with the preinv1 functions. We require the most significant bit of b, n to be 1.

```c
mp_limb_t flint_mpn_divrem_preinv1(mp_ptr q, mp_ptr a, mp_size_t m, mp_srcptr b,
                                   mp_size_t n, mp_limb_t dinv)
```

Divide a, m by b, n, returning the high limb of the quotient (which will either be 0 or 1), storing the remainder in-place in a, n and the rest of the quotient in q, m - n. We require the most significant bit of b, n to be 1. dinv must be computed from b[n - 1], b[n - 2] by flint_mpn_preinv1. We also require m >= n >= 2.

```c
void flint_mpn_mulmod_preinv1(mp_ptr r, mp_srcptr a, mp_srcptr b, mp_size_t n, mp_srcptr d,
                             mp_limb_t dinv, ulong norm)
```

Given a normalised integer d with precomputed inverse dinv provided by flint_mpn_preinv1, computes ab (mod d) and stores the result in r. Each of a, b and r is expected to have n limbs of space, with zero padding if necessary.

The value norm is provided for convenience. If a, b and d have been shifted left by norm bits so that d is normalised, then r will be shifted right by norm bits so that it has the same shift as all the inputs.

We require a and b to be reduced modulo n before calling the function.

```c
void flint_mpn_preinvn(mp_ptr dinv, mp_srcptr d, mp_size_t n)
```

Compute an n limb precomputed inverse dinv of the n limb integer d.

We require that d is normalised, i.e. with the most significant bit of the most significant limb set.

```c
void flint_mpn_mod_preinvn(mp_ptr r, mp_srcptr a, mp_size_t m, mp_srcptr d, mp_size_t n,
                          mp_srcptr dinv)
```

Given a normalised integer d of n limbs, with precomputed inverse dinv provided by flint_mpn_preinvn and integer a of m limbs, computes a (mod d) and stores the result in-place in the lower n limbs of a. The remaining limbs of a are destroyed.

We require m >= n. No aliasing of a with any of the other operands is permitted.

Note that this function is not always as fast as ordinary division.

```c
mp_limb_t flint_mpn_divrem_preinvn(mp_ptr q, mp_ptr r, mp_srcptr a, mp_size_t m,
                                   mp_srcptr d, mp_size_t n, mp_srcptr dinv)
```

Given a normalised integer d with precomputed inverse dinv provided by flint_mpn_preinvn, computes the quotient of a by d and stores the result in q and the remainder in the lower n limbs of a. The remaining limbs of a are destroyed.

The value q is expected to have space for m - n limbs and we require m >= n. No aliasing is permitted between q and a or between these and any of the other operands.

Note that this function is not always as fast as ordinary division.

```c
void flint_mpn_mulmod_preinvn(mp_ptr r, mp_srcptr a, mp_srcptr b, mp_size_t n, mp_srcptr d,
                            mp_srcptr dinv, ulong norm)
```

Given a normalised integer d with precomputed inverse dinv provided by flint_mpn_preinvn,
computes $ab \pmod{d}$ and stores the result in $r$. Each of $a$, $b$ and $r$ is expected to have $n$ limbs of space, with zero padding if necessary.

The value $\text{norm}$ is provided for convenience. If $a$, $b$ and $d$ have been shifted left by $\text{norm}$ bits so that $d$ is normalised, then $r$ will be shifted right by $\text{norm}$ bits so that it has the same shift as all the inputs.

We require $a$ and $b$ to be reduced modulo $n$ before calling the function.

Note that this function is not always as fast as ordinary division.

### 4.14.6 GCD

```c
mp_size_t flint_mpn_gcd_full2(mp_ptr arrayg, mp_srcptr array1, mp_size_t limbs1, mp_srcptr array2, mp_size_t limbs2, mp_ptr temp)
```

Sets $(arrayg, retvalue)$ to the gcd of $(array1, limbs1)$ and $(array2, limbs2)$.

The only assumption is that neither $\text{limbs1}$ nor $\text{limbs2}$ is zero.

The function must be supplied with $\text{limbs1} + \text{limbs2}$ limbs of temporary space, or NULL must be passed to $\text{temp}$ if the function should allocate its own space.

```c
mp_size_t flint_mpn_gcd_full(mp_ptr arrayg, mp_srcptr array1, mp_size_t limbs1, mp_srcptr array2, mp_size_t limbs2)
```

Sets $(arrayg, retvalue)$ to the gcd of $(array1, limbs1)$ and $(array2, limbs2)$.

The only assumption is that neither $\text{limbs1}$ nor $\text{limbs2}$ is zero.

### 4.14.7 Random Number Generation

```c
void flint_mpn_rrandom(mp_limb_t *rp, gmp_randstate_t state, mp_size_t n)
```

Generates a random number with $n$ limbs and stores it on $\text{rp}$. The number it generates will tend to have long strings of zeros and ones in the binary representation.

Useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs.

```c
void flint_mpn_urandomb(mp_limb_t *rp, gmp_randstate_t state, flint_bitcnt_t n)
```

Generates a uniform random number of $n$ bits and stores it on $\text{rp}$.

### 4.15 aprcl.h – APRCL primality testing

This module implements the rigorous APRCL primality test, suitable for integers up to a few thousand digits.

The APR-CL test uses the Jacobi sums that belong to $\mathbb{Z}[\zeta]/(n)$, so we have $\text{unity_zp}$ struct and some useful operations. $\text{unity_zp}$ is just a wrapper over $\text{fmpz_mod_poly}$ with additional fields.

Also provides Gauss sum test, which is not very useful in practice, but can be useful for people who want to see an implementation of these. Gauss sums belong $\mathbb{Z}[\zeta_q, \zeta_p]/(n)$ and implemented in $\text{unity_zpq}$ struct.

Authors:

- Vladimir Glazachev (Google Summer of Code, 2015)
4.15.1 Primality test functions

```c
int aprcl_is_prime(const fmpz_t n)
Tests n for primality using the APRCL test. This is the same as aprcl_is_prime_jacobi().
```

```c
int aprcl_is_prime_jacobi(const fmpz_t n)
If n is prime returns 1; otherwise returns 0. The algorithm is well described in “Implementation of a New Primality Test” by H. Cohen and A.K. Lenstra and “A Course in Computational Algebraic Number Theory” by H. Cohen.

It is theoretically possible that this function fails to prove that n is prime. In this event, flint_abort() is called. To handle this condition, the _aprcl_is_prime_jacobi() function can be used.
```

```c
int aprcl_is_prime_gauss(const fmpz_t n)
If n is prime returns 1; otherwise returns 0. Uses the cyclotomic primality testing algorithm described in “Four primality testing algorithms” by Rene Schoof. The minimum required numbers s and R are computed automatically.

By default R ≥ 180. In some cases this function fails to prove that n is prime. This means that we select a too small R value. In this event, flint_abort() is called. To handle this condition, the _aprcl_is_prime_jacobi() function can be used.
```

```c
primality_test_status _aprcl_is_prime_jacobi(const fmpz_t n, const aprcl_config config)
Jacobi sum test for n. Possible return values: PRIME, COMPOSITE and UNKNOWN (if we cannot prove primality).
```

```c
primality_test_status _aprcl_is_prime_gauss(const fmpz_t n, const aprcl_config config)
Tests n for primality with fixed config. Possible return values: PRIME, COMPOSITE and PROBABPRIME (if we cannot prove primality).
```

```c
int aprcl_is_prime_gauss_min_R(const fmpz_t n, ulong R)
Same as aprcl_is_prime_gauss() with fixed minimum value of R.
```

```c
int aprcl_is_prime_final_division(const fmpz_t n, const fmpz_t s, ulong r)
Returns 0 if for some \(a = n^k mod s\), where \(k \in [1, r - 1]\), we have that \(a | n\); otherwise returns 1.
```

4.15.2 Configuration functions

```c
type _aprcl_config

type aprcl_config
Holds precomputed parameters.
```

```c
void aprcl_config_gauss_init(aprcl_config conf, const fmpz_t n)
Computes the s and R values used in the cyclotomic primality test, \(s^2 > n\) and 
\[s = \prod_{q \mid R, q \text{ prime}} q\]. Also stores factors of R and s.
```

```c
void aprcl_config_gauss_init_min_R(aprcl_config conf, const fmpz_t n, ulong R)
Computes the s with fixed minimum R such that \(a^R \equiv 1 \mod s\) for all integers a coprime to s.
```

```c
void aprcl_config_gauss_clear(aprcl_config conf)
Clears the given aprcl_config element. It must be reinitialised in order to be used again.
```

```c
ulong aprcl_R_value(const fmpz_t n)
Returns a precomputed R value for APRCL, such that the corresponding s value is greater than \(\sqrt{n}\). The maximum stored value 6983776800 allows to test numbers up to 6000 digits.
```
void *aprcl_config_jacobi_init(aprcl_config conf, const fmpz_t n)

Computes the $s$ and $R$ values used in the cyclotomic primality test, $s^2 > n$ and $a^R \equiv 1 \mod s$ for all $a$ coprime to $s$. Also stores factors of $R$ and $s$.

void *aprcl_config_jacobi_clear(aprcl_config conf)

Clears the given aprcl_config element. It must be reinitialised in order to be used again.

### 4.15.3 Cyclotomic arithmetic

This code implements arithmetic in cyclotomic rings.

#### Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type _unity_zp</td>
<td></td>
</tr>
<tr>
<td>type unity_zp</td>
<td>Represents an element of $\mathbb{Z}[[\zeta_p^e]]/(n)$ as an fmpz_mod_poly_t reduced modulo a cyclotomic polynomial.</td>
</tr>
<tr>
<td>type _unity_zpq</td>
<td></td>
</tr>
<tr>
<td>type unity_zpq</td>
<td>Represents an element of $\mathbb{Z}[[\zeta_q,\zeta_p]]/(n)$ as an array of fmpz_mod_poly_t.</td>
</tr>
</tbody>
</table>

#### Memory management

void unity_zp_init(unity_zp f, ulong p, ulong exp, const fmpz_t n)

Initializes $f$ as an element of $\mathbb{Z}[[\zeta_p^e]]/(n)$.

void unity_zp_clear(unity_zp f)

Clears the given element. It must be reinitialised in order to be used again.

void unity_zp_copy(unity_zp f, const unity_zp g)

Sets $f$ to $g$. $f$ and $g$ must be initialized with same $p$ and $n$.

void unity_zp_swap(unity_zp f, unity_zp q)

Swaps $f$ and $g$. $f$ and $g$ must be initialized with same $p$ and $n$.

void unity_zp_set_zero(unity_zp f)

Sets $f$ to zero.

#### Comparison

slong unity_zp_is_unity(unity_zp f)

If $f = \zeta^h$ returns $h$; otherwise returns -1.

int unity_zp_equal(unity_zp f, unity_zp g)

Returns nonzero if $f = g$ reduced by the $p^e p$-th cyclotomic polynomial.
Coefficient management

void unity_zp_coeff_set_fmpz(unity_zp f, ulong ind, const fmpz_t x)
Sets the coefficient of $\zeta^{\text{ind}}$ to $x$. $\text{ind}$ must be less than $p^{\text{exp}}$.

void unity_zp_coeff_add_fmpz(unity_zp f, ulong ind, const fmpz_t x)
Adds $x$ to the coefficient of $\zeta^{\text{ind}}$. $x$ must be less than $n$. $\text{ind}$ must be less than $p^{\text{exp}}$.

void unity_zp_coeff_inc(unity_zp f, ulong ind)
Increments the coefficient of $\zeta^{\text{ind}}$. $\text{ind}$ must be less than $p^{\text{exp}}$.

void unity_zp_coeff_dec(unity_zp f, ulong ind)
Decrements the coefficient of $\zeta^{\text{ind}}$. $\text{ind}$ must be less than $p^{\text{exp}}$.

Scalar multiplication

void unity_zp_mul_scalar_ui(unity_zp f, const unity_zp g, ulong s)
Sets $f$ to $s \cdot g$. $f$ and $g$ must be initialized with same $p$, $\text{exp}$ and $n$.

Addition and multiplication

void unity_zp_add(unity_zp f, const unity_zp g, const unity_zp h)
Sets $f$ to $g + h$. $f$, $g$ and $h$ must be initialized with same $p$, $\text{exp}$ and $n$.

void unity_zp_mul(unity_zp f, const unity_zp g, const unity_zp h)
Sets $f$ to $g \cdot h$. $f$, $g$ and $h$ must be initialized with same $p$, $\text{exp}$ and $n$.

void unity_zp_sqr(unity_zp f, const unity_zp g)
Sets $f$ to $g \cdot g$. $f$, $g$ and $h$ must be initialized with same $p$, $\text{exp}$ and $n$.

void unity_zp_mul_inplace(unity_zp f, const unity_zp g, const unity_zp h, fmpz_t *t)
Sets $f$ to $g \cdot h$. If $p^{\text{exp}} = 3, 4, 5, 7, 8, 9, 11, 16$ special multiplication functions are used. The preallocated array $t$ of $\text{fmpz}$ is used for all computations in this case. $f$, $g$ and $h$ must be initialized with same $p$, $\text{exp}$ and $n$.

void unity_zp_sqr_inplace(unity_zp f, const unity_zp g, fmpz_t *t)
Sets $f$ to $g \cdot g$. If $p^{\text{exp}} = 3, 4, 5, 7, 8, 9, 11, 16$ special multiplication functions are used. The preallocated array $t$ of $\text{fmpz}$ is used for all computations in this case. $f$ and $g$ must be initialized with same $p$, $\text{exp}$ and $n$.

Powering functions

void unity_zp_pow_fmpz(unity_zp f, const unity_zp g, const fmpz_t pow)
Sets $f$ to $g^{\text{pow}}$. $f$ and $g$ must be initialized with same $p$, $\text{exp}$ and $n$.

void unity_zp_pow_ui(unity_zp f, const unity_zp g, ulong pow)
Sets $f$ to $g^{\text{pow}}$. $f$ and $g$ must be initialized with same $p$, $\text{exp}$ and $n$.

ulong _unity_zp_pow_select_k(const fmpz_t n)
Returns the smallest integer $k$ satisfying $\log(n) < (k(k + 1)2^k)/(2^{k+1} - k - 2) + 1$

void unity_zp_pow_2k_fmpz(unity_zp f, const unity_zp g, const fmpz_t pow)
Sets $f$ to $g^{\text{pow}}$ using the $2^k$-ary exponentiation method. $f$ and $g$ must be initialized with same $p$, $\text{exp}$ and $n$. 
void unity_zp_pow_2k_ui(unity_zp f, const unity_zp g, ulong pow)
    Sets $f$ to $g^{pow}$ using the $2^k$-ary exponentiation method. $f$ and $g$ must be initialized with the same $p$, $\exp$ and $n$.

void unity_zp_pow_sliding_fmpz(unity_zp f, unity_zp g, const fmpz_t pow)
    Sets $f$ to $g^{pow}$ using the sliding window exponentiation method. $f$ and $g$ must be initialized with the same $p$, $\exp$ and $n$.

Cyclotomic reduction

void _unity_zp_reduce_cyclotomic_divmod(unity_zp f)
void _unity_zp_reduce_cyclotomic(unity_zp f)
    Sets $f = f \mod \Phi_{\exp}$, where $\Phi_{\exp}$ is the $\exp$-th cyclotomic polynomial. $g$ must be reduced by $x^{\exp} - 1$ poly. $f$ and $g$ must be initialized with the same $p$, $\exp$ and $n$.

void unity_zp_reduce_cyclotomic(unity_zp f, const unity_zp g)
    Sets $f = g \mod \Phi_{\exp}$. $\Phi_{\exp}$ is the $\exp$-th cyclotomic polynomial.

Automorphism and inverse

void unity_zp_aut(unity_zp f, const unity_zp g, ulong x)
    Sets $f = \sigma_x(g)$, the automorphism $\sigma_x(\zeta) = \zeta^x$. $f$ and $g$ must be initialized with the same $p$, $\exp$ and $n$.

void unity_zp_aut_inv(unity_zp f, const unity_zp g, ulong x)
    Sets $f = \sigma^{-1}_x(g)$, so $\sigma_x(f) = g$. $g$ must be reduced by $\Phi_{\exp}$. $f$ and $g$ must be initialized with the same $p$, $\exp$ and $n$.

Jacobi sum

Here $\chi_{p,q}$ is the character defined by $\chi_{p,q}(g^x) = \zeta_{p,q}^x$, where $g$ is a primitive root modulo $q$.

void unity_zp_jacobi_sum_pq(unity_zp f, ulong q, ulong p)
    Sets $f$ to the Jacobi sum $J(p, q) = j(\chi_{p,q}, \chi_{p,q})$.

void unity_zp_jacobi_sum_2q_one(unity_zp f, ulong q)
    Sets $f$ to the Jacobi sum $J_2(q) = j(\chi_{2,q}^{2^k-3}, \chi_{2,q}^{2^k-3})^2$.

void unity_zp_jacobi_sum_2q_two(unity_zp f, ulong q)
    Sets $f$ to the Jacobi sum $J_3(1) = j(\chi_{2,q}, \chi_{2,q}, \chi_{2,q}) = J(2, q) \cdot J(\chi_{2,q}, \chi_{2,q})$.

Extended rings

void unity_zpq_init(unity_zpq f, ulong q, ulong p, const fmpz_t n)
    Initializes $f$ as an element of $\mathbb{Z}[\zeta_q, \zeta_p]/(n)$.

void unity_zpq_clear(unity_zpq f)
    Clears the given element. It must be reinitialized in order to be used again.

void unity_zpq_copy(unity_zpq f, const unity_zpq g)
    Sets $f$ to $g$. $f$ and $g$ must be initialized with the same $p$, $q$ and $n$.

void unity_zpq_swap(unity_zpq f, unity_zpq q)
    Swaps $f$ and $g$. $f$ and $g$ must be initialized with the same $p$, $q$ and $n$. 
int unity_zpq_equal(const unity_zpq f, const unity_zpq g)
   Returns nonzero if \( f = g \).

void unity_zpq_coeff_set_fmpz(unity_zpq f, slong i, slong j, const fmpz_t x)
   Sets the coefficient of \( \zeta_p^i \zeta_q^j \) to \( x \). \( i \) must be less than \( q \) and \( j \) must be less than \( p \).

void unity_zpq_coeff_set_ui(unity_zpq f, slong i, slong j, ulong x)
   Sets the coefficient of \( \zeta_p^i \zeta_q^j \) to \( x \). \( i \) must be less than \( q \) and \( j \) must be less than \( p \).

void unity_zpq_coeff_add(unity_zpq f, slong i, slong j, const fmpz_t x)
   Adds \( x \) to the coefficient of \( \zeta_p^i \zeta_q^j \). \( x \) must be less than \( n \).

void unity_zpq_add(unity_zpq f, const unity_zpq g, const unity_zpq h)
   Sets \( f \) to \( g + h \). \( f \), \( g \) and \( h \) must be initialized with same \( q \), \( p \) and \( n \).

void unity_zpq_mul(unity_zpq f, const unity_zpq g, const unity_zpq h)
   Sets the \( f \) to \( g \cdot h \). \( f \), \( g \) and \( h \) must be initialized with same \( q \), \( p \) and \( n \).

void _unity_zpq_mul_unity_p(unity_zpq f)
   Sets \( f = f \cdot \zeta_p \).

void unity_zpq_mul_unity_p_pow(unity_zpq f, const unity_zpq g, slong k)
   Sets \( f \) to \( g^k \cdot \zeta_p \).

void unity_zpq_pow(unity_zpq f, const unity_zpq g, const fmpz_t p)
   Sets \( f \) to \( g^p \). \( f \) and \( g \) must be initialized with same \( p \), \( q \) and \( n \).

void unity_zpq_pow_ui(unity_zpq f, const unity_zpq g, ulong p)
   Sets \( f \) to \( g^p \). \( f \) and \( g \) must be initialized with same \( p \), \( q \) and \( n \).

void unity_zpq_gauss_sum(unity_zpq f, ulong q, ulong p)
   Sets \( f = \tau(x_{p, q}) \).

void unity_zpq_gauss_sum_sigma_pow(unity_zpq f, ulong q, ulong p)
   Sets \( f = \sigma^n(x_{p, q}) \).

4.16 arith.h – arithmetic and special functions

This module implements arithmetic functions, number-theoretic and combinatorial special number sequences and polynomials.

4.16.1 Primorials

void arith_primorial(fmpz_t res, slong n)
   Sets \( res \) to \( n \) primorial or \( n\# \), the product of all prime numbers less than or equal to \( n \).

4.16.2 Harmonic numbers

void _arith_harmonic_number(fmpz_t num, fmpz_t den, slong n)
void arith_harmonic_number(fmpz_t x, slong n)
   These are aliases for the functions in the fmpq module.
4.16.3 Stirling numbers

void \texttt{arith\_stirling\_number\_1u(fmpz\_t s, ulong n, ulong k)}

void \texttt{arith\_stirling\_number\_1(fmpz\_t s, ulong n, ulong k)}

void \texttt{arith\_stirling\_number\_2(fmpz\_t s, ulong n, ulong k)}

Sets $s$ to $S(n, k)$ where $S(n, k)$ denotes an unsigned Stirling number of the first kind $|S_1(n, k)|$, a signed Stirling number of the first kind $S_1(n, k)$, or a Stirling number of the second kind $S_2(n, k)$. The Stirling numbers are defined using the generating functions

\[ x(n) = \sum_{k=0}^{n} S_1(n, k) x^k \]
\[ x^{(n)} = \sum_{k=0}^{n} |S_1(n, k)| x^k \]
\[ x^n = \sum_{k=0}^{n} S_2(n, k) x^{(k)} \]

where $x^{(n)} = x(x-1)(x-2) \cdots (x-n+1)$ is a falling factorial and $x^{(n)} = x(x+1)(x+2) \cdots (x+n-1)$ is a rising factorial. $S(n, k)$ is taken to be zero if $n < 0$ or $k < 0$.

These three functions are useful for computing isolated Stirling numbers efficiently. To compute a range of numbers, the vector or matrix versions should generally be used.

void \texttt{arith\_stirling\_number\_1u\_vec(fmpz\_t row, ulong n, slong klen)}

void \texttt{arith\_stirling\_number\_1\_vec(fmpz\_t row, ulong n, slong klen)}

void \texttt{arith\_stirling\_number\_2\_vec(fmpz\_t row, ulong n, slong klen)}

Computes the row of Stirling numbers $S(n, 0), S(n, 1), S(n, 2), \ldots, S(n, klen-1)$.

To compute a full row, this function can be called with $klen = n+1$. It is assumed that $klen$ is at most $n+1$.

void \texttt{arith\_stirling\_number\_1u\_vec\_next(fmpz\_t row, const fmpz\_t prev, slong n, slong klen)}

void \texttt{arith\_stirling\_number\_1\_vec\_next(fmpz\_t row, const fmpz\_t prev, slong n, slong klen)}

void \texttt{arith\_stirling\_number\_2\_vec\_next(fmpz\_t row, const fmpz\_t prev, slong n, slong klen)}

Given the vector $\texttt{prev}$ containing a row of Stirling numbers $S(n-1, 0), S(n-1, 1), S(n-1, 2), \ldots, S(n-1, klen-1)$, computes and stores in the row argument $S(n, 0), S(n, 1), S(n, 2), \ldots, S(n, klen-1)$.

If $klen$ is greater than $n$, the output ends with $S(n, n) = 1$ followed by $S(n, n+1) = S(n, n+2) = \ldots = 0$. In this case, the input only needs to have length $n-1$; only the input entries up to $S(n-1, n-2)$ are read.

The row and prev arguments are permitted to be the same, meaning that the row will be updated in-place.

void \texttt{arith\_stirling\_matrix\_1u(fmpz\_mat\_t mat)}

void \texttt{arith\_stirling\_matrix\_1(fmpz\_mat\_t mat)}

void \texttt{arith\_stirling\_matrix\_2(fmpz\_mat\_t mat)}

For an arbitrary $m$-by-$n$ matrix, writes the truncation of the infinite Stirling number matrix:

<table>
<thead>
<tr>
<th>row 0</th>
<th>row 1</th>
<th>row 2</th>
<th>row 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(0,0)$</td>
<td>$S(1,0)$, $S(1,1)$</td>
<td>$S(2,0)$, $S(2,1)$, $S(2,2)$</td>
<td>$S(3,0)$, $S(3,1)$, $S(3,2)$, $S(3,3)$</td>
</tr>
</tbody>
</table>
up to row $m - 1$ and column $n - 1$ inclusive. The upper triangular part of the matrix is zeroed.
For any $n$, the $S_1$ and $S_2$ matrices thus obtained are inverses of each other.

### 4.16.4 Bell numbers

**void arith_bell_number (fmpz_t b, ulong n)**

**void arith_bell_number_dobinski (fmpz_t res, ulong n)**

Sets $b$ to the Bell number $B_n$, defined as the number of partitions of a set with $n$ members. Equivalently, $B_n = \sum_{k=0}^{n} S_2(n, k)$ where $S_2(n, k)$ denotes a Stirling number of the second kind.

The default version automatically selects between table lookup, Dobinski’s formula, and the multimodular algorithm.

The **dobinski** version evaluates a precise truncation of the series $B_n = e^{-1} \sum_{k=0}^{\infty} \frac{k^n}{k!}$ (Dobinski’s formula). In fact, we compute $P = N! \sum_{k=0}^{N} \frac{k^n}{k!}$ and $Q = N! \sum_{k=0}^{N} \frac{1}{k!} \approx N! e$ and evaluate $B_n = \lfloor P/Q \rfloor$, avoiding the use of floating-point arithmetic.

The **multi_mod** version computes the result modulo several limb-size primes and reconstructs the integer value using the fast Chinese remainder algorithm. A bound for the number of needed primes is computed using **arith_bell_number_size**.

**void arith_bell_number_vec (fmpz *b, slong n)**

**void arith_bell_number_vec_recursive (fmpz *b, slong n)**

**void arith_bell_number_vec_multi_mod (fmpz *b, slong n)**

Sets $b$ to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive. The **recursive** version uses the $O(n^3 \log n)$ triangular recurrence, while the **multi_mod** version implements multimodular evaluation of the exponential generating function, running in time $O(n^2 \log O(1) \log n)$. The default version chooses an algorithm automatically.

**mp_limb_t arith_bell_number_nmod (ulong n, nmod_t mod)**

Computes the Bell number $B_n$ modulo an integer given by **mod**.

After handling special cases, we use the formula

$$B_n = \sum_{k=0}^{n} \frac{(n-k)^n}{(n-k)!} \sum_{j=0}^{k} \frac{(-1)^j}{j!}.$$

We arrange the operations in such a way that we only have to multiply (and not divide) in the main loop. As a further optimisation, we use sieving to reduce the number of powers that need to be evaluated. This results in $O(n)$ memory usage.

If the divisions by factorials are impossible, we fall back to calling **arith_bell_number_nmod_vec** and reading the last coefficient.

**void arith_bell_number_nmod_vec (mp_ptr b, slong n, nmod_t mod)**

**void arith_bell_number_nmod_vec_recursive (mp_ptr b, slong n, nmod_t mod)**

**void arith_bell_number_nmod_vec_ogf (mp_ptr b, slong n, nmod_t mod)**

**int arith_bell_number_nmod_vec_series (mp_ptr b, slong n, nmod_t mod)**

Sets $b$ to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive modulo an integer given by **mod**.

The **recursive** version uses the $O(n^2)$ triangular recurrence. The **ogf** version expands the ordinary generating function using binary splitting, which is $O(n \log^2 n)$.

The **series** version uses the exponential generating function $\sum_{k=0}^{\infty} \frac{B_k}{k!} x^k = \exp(e^x - 1)$, running in $O(n \log n)$. This only works if division by $n!$ is possible, and the function returns whether it is successful. All other versions support any modulus.

The default version of this function selects an algorithm automatically.
4.16.5 Bernoulli numbers and polynomials

void _arith_bernoulli_number(fmpz_t num, fmpz_t den, ulong n)

Sets (num, den) to the reduced numerator and denominator of the n-th Bernoulli number.

void arith_bernoulli_number(fmpz_t x, ulong n)

Sets x to the n-th Bernoulli number. This function is equivalent to _arith_bernoulli_number apart from the output being a single fmpz_t variable.

void _arith_bernoulli_number_vec(fmpz *num, fmpz *den, slong n)

Sets the elements of num and den to the reduced numerators and denominators of the Bernoulli numbers B_0, B_1, B_2, ..., B_{n-1} inclusive. This function automatically chooses between the recursive, zeta and multi_mod algorithms according to the size of n.

void arith_bernoulli_number_vec(fmpz *x, slong n)

Sets the x to the vector of Bernoulli numbers B_0, B_1, B_2, ..., B_{n-1} inclusive. This function is equivalent to _arith_bernoulli_number_vec apart from the output being a single fmpz vector.

void arith_bernoulli_number_denom(fmpz_t den, ulong n)

Sets den to the reduced denominator of the n-th Bernoulli number B_n. For even n, the denominator is computed as the product of all primes p for which p - 1 divides n; this property is a consequence of the von Staudt-Clausen theorem. For odd n, the denominator is trivial (den is set to 1 whenever B_n = 0). The initial sequence of values smaller than 2^{32} are looked up directly from a table.

double arith_bernoulli_number_size(ulong n)

Returns b such that |B_n| < 2^{b}, using the inequality |B_n| < \left(\frac{4n!}{(2\pi)^n}\right)^{n!} and n! ≤ (n + 1)^{n+1}e^{-n}. No special treatment is given to odd n. Accuracy is not guaranteed if n > 10^{14}.

void arith_bernoulli_polynomial(fmpz_poly_t poly, ulong n)

Sets poly to the Bernoulli polynomial of degree n, B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k} where B_k is a Bernoulli number. This function basically calls arith_bernoulli_number_vec and then rescales the coefficients efficiently.

void _arith_bernoulli_number_vec_recursive(fmpz *num, fmpz *den, slong n)

Sets the elements of num and den to the reduced numerators and denominators of B_0, B_1, B_2, ..., B_{n-1} inclusive.

The first few entries are computed using arith_bernoulli_number, and then Ramanujan’s recursive formula expressing B_m as a sum over B_k for k congruent to m modulo 6 is applied repeatedly.

To avoid costly GCDs, the numerators are transformed internally to a common denominator and all operations are performed using integer arithmetic. This makes the algorithm fast for small n, say n < 1000. The common denominator is calculated directly as the primorial of n + 1.

void _arith_bernoulli_number_vec_multi_mod(fmpz *num, fmpz *den, slong n)

Sets the elements of num and den to the reduced numerators and denominators of B_0, B_1, B_2, ..., B_{n-1} inclusive. Uses the generating function

\[
\frac{x^2}{\cosh(x) - 1} = \sum_{k=0}^{\infty} \frac{(2 - 4k)B_{2k}}{(2k)!} x^{2k}
\]

which is evaluated modulo several limb-size primes using nmod_poly arithmetic to yield the numerators of the Bernoulli numbers after multiplication by the denominators and CRT reconstruction.
This formula, given (incorrectly) in [BuhlerCrandallSompolski1992], saves about half of the time compared to the usual generating function $x/(e^x - 1)$ since the odd terms vanish.

### 4.16.6 Euler numbers and polynomials

Euler numbers are the integers $E_n$ defined by $\frac{1}{\cosh(t)} = \sum_{n=0}^{\infty} \frac{E_n}{n!} t^n$. With this convention, the odd-indexed numbers are zero and the even ones alternate signs, viz. $E_0, E_1, E_2, \ldots = 1, 0, -1, 0, 5, 0, -61, 0, 1385, 0, \ldots$. The corresponding Euler polynomials are defined by $\frac{2}{e^{tx} + 1} = \sum_{n=0}^{\infty} \frac{E_n(x)}{n!} t^n$.

```c
void arith_euler_number(fmpz_t res, ulong n)
```
Sets res to the Euler number $E_n$.

```c
void arith_euler_number_vec(fmpz *res, ulong n)
```
Computes the Euler numbers $E_0, E_1, \ldots, E_{n-1}$ for $n \geq 0$ and stores the result in res, which must be an initialised fmpz vector of sufficient size.

This function evaluates the even-index $E_k$ modulo several limb-size primes using the generating function and nmod_poly arithmetic. A tight bound for the number of needed primes is computed using arith_euler_number_size, and the final integer values are recovered using balanced CRT reconstruction.

```c
double arith_euler_number_size(ulong n)
```
Returns $b$ such that $|E_n| < 2^b$, using the inequality $|E_n| < \frac{2^{n+2} n!}{\pi^{n+1}}$ and $n! \leq (n+1)^{n+1} e^{-n}$. No special treatment is given to odd $n$. Accuracy is not guaranteed if $n > 10^{14}$.

```c
void arith_euler_polynomial(fmpq_poly_t poly, ulong n)
```
Sets poly to the Euler polynomial $E_n(x)$. Uses the formula

$$E_n(x) = \frac{2}{n+1} \left( B_{n+1}(x) - 2^{n+1} B_{n+1}(\frac{x}{2}) \right),$$

with the Bernoulli polynomial $B_{n+1}(x)$ evaluated once using bernoulli_polynomial and then rescaled.

### 4.16.7 Multiplicative functions

```c
void arith_euler_phi(fmpz_t res, const fmpz_t n)
```

```c
int arith_moebius_mu(const fmpz_t n)
```

```c
void arith_divisor_sigma(fmpz_t res, ulong k, const fmpz_t n)
```
These are aliases for the functions in the fmpz module.

```c
void arith_divisors(fmpz_poly_t res, const fmpz_t n)
```
Set the coefficients of the polynomial res to the divisors of $n$, including 1 and $n$ itself, in ascending order.

```c
void arith_ramanujan_tau(fmpz_t res, const fmpz_t n)
```
Sets res to the Ramanujan tau function $\tau(n)$ which is the coefficient of $q^n$ in the series expansion of $f(q) = q \prod_{k \geq 1} (1 - q^k)^4$.

We factor $n$ and use the identity $\tau(pq) = \tau(p)\tau(q)$ along with the recursion $\tau(p^{r+1}) = \tau(p)\tau(p^r) - p^{11}\tau(p^{r-1})$ for prime powers.

The base values $\tau(p)$ are obtained using the function arith_ramanujan_tau_series(). Thus the speed of arith_ramanujan_tau() depends on the largest prime factor of $n$.

Future improvement: optimise this function for small $n$, which could be accomplished using a lookup table or by calling arith_ramanujan_tau_series() directly.
void \texttt{arith\_ramanujan\_tau\_series}(\texttt{fmpz\_poly\_t} \texttt{res}, \texttt{slong} \texttt{n})

Sets \texttt{res} to the polynomial with coefficients $\tau(0), \tau(1), \ldots, \tau(n - 1)$, giving the initial $n$ terms in the series expansion of $f(q) = q \prod_{k \geq 1} (1 - q^k)^{24}$.

We use the theta function identity

$$f(q) = q \left( \sum_{k \geq 0} (-1)^k (2k + 1) q^{k(k+1)/2} \right)^8$$

which is evaluated using three squarings. The first squaring is done directly since the polynomial is very sparse at this point.

4.16.8 Landau’s function

void \texttt{arith\_landau\_function\_vec}(\texttt{fmpz *} \texttt{res}, \texttt{slong} \texttt{len})

Computes the first \texttt{len} values of Landau’s function $g(n)$ starting with $g(0)$. Landau’s function gives the largest order of an element of the symmetric group $S_n$.

Implements the “basic algorithm” given in [DelegliseNicolasZimmermann2009]. The running time is $O(n^{3/2}/\sqrt{\log n})$.

4.16.9 Dedekind sums

void \texttt{arith\_dedekind\_sum\_naive}(\texttt{fmpq} \texttt{s}, const \texttt{fmpz} \texttt{h}, const \texttt{fmpz} \texttt{k})
double \texttt{arith\_dedekind\_sum\_coprime\_d}(\texttt{double} \texttt{h}, \texttt{double} \texttt{k})
void \texttt{arith\_dedekind\_sum\_coprime\_large}(\texttt{fmpq} \texttt{s}, const \texttt{fmpz} \texttt{h}, const \texttt{fmpz} \texttt{k})
void \texttt{arith\_dedekind\_sum\_coprime}(\texttt{fmpq} \texttt{s}, const \texttt{fmpz} \texttt{h}, const \texttt{fmpz} \texttt{k})
void \texttt{arith\_dedekind\_sum}(\texttt{fmpq} \texttt{s}, const \texttt{fmpz} \texttt{h}, const \texttt{fmpz} \texttt{k})

These are aliases for the functions in the \texttt{fmpq} module.

4.16.10 Number of partitions

void \texttt{arith\_number\_of\_partitions\_vec}(\texttt{fmpz *} \texttt{res}, \texttt{slong} \texttt{len})

Computes first \texttt{len} values of the partition function $p(n)$ starting with $p(0)$. Uses inversion of Euler’s pentagonal series.

void \texttt{arith\_number\_of\_partitions\_nmod\_vec}(\texttt{mp\_ptr} \texttt{res}, \texttt{slong} \texttt{len}, \texttt{nmod\_t} \texttt{mod})

Computes first \texttt{len} values of the partition function $p(n)$ starting with $p(0)$, modulo the modulus defined by \texttt{mod}. Uses inversion of Euler’s pentagonal series.

void \texttt{trig\_prod\_init}(\texttt{trig\_prod\_t} \texttt{prod})

Initializes \texttt{prod}. This is an inline function only.

void \texttt{arith\_hrr\_expsum\_factored}(\texttt{trig\_prod\_t} \texttt{prod}, \texttt{mp\_limb\_t} \texttt{k}, \texttt{mp\_limb\_t} \texttt{n})

Symbolically evaluates the exponential sum

$$A_k(n) = \sum_{h=0}^{k-1} \exp \left( \pi i \left[ s(h, k) - \frac{2hn}{k} \right] \right)$$

appearing in the Hardy-Ramanujan-Rademacher formula, where $s(h, k)$ is a Dedekind sum.

Rather than evaluating the sum naively, we factor $A_k(n)$ into a product of cosines based on the prime factorisation of $k$. This process is based on the identities given in [Whiteman1956].

The special \texttt{trig\_prod\_t} structure \texttt{prod} represents a product of cosines of rational arguments, multiplied by an algebraic prefactor. It must be pre-initialised with \texttt{trig\_prod\_init}.
This function assumes that $24k$ and $24n$ do not overflow a single limb. If $n$ is larger, it can be pre-reduced modulo $k$, since $A_k(n)$ only depends on the value of $n \mod k$.

```c
void arith_number_of_partitions_mpfr(mpfr_t x, ulong n)
```

Sets the pre-initialised MPFR variable $x$ to the exact value of $p(n)$. The value is computed using the Hardy-Ramanujan-Rademacher formula.

The precision of $x$ will be changed to allow $p(n)$ to be represented exactly. The interface of this function may be updated in the future to allow computing an approximation of $p(n)$ to smaller precision.

The Hardy-Ramanujan-Rademacher formula is given with error bounds in [Rademacher1937]. We evaluate it in the form

$$p(n) = \sum_{k=1}^{N} B_k(n)U(C/k) + R(n, N)$$

where

$$U(x) = \cosh(x) + \frac{\sinh(x)}{x}, \quad C = \frac{\pi}{6} \sqrt{24n - 1}$$

$$B_k(n) = \sqrt{\frac{3}{k}} \frac{4}{24n - 1} A_k(n)$$

and where $A_k(n)$ is a certain exponential sum. The remainder satisfies

$$|R(n, N)| < \frac{44\pi^2}{225\sqrt{3}} N^{-1/2} + \frac{\pi\sqrt{2}}{75} \left( \frac{N}{n - 1} \right)^{1/2} \sinh \left( \pi \sqrt{\frac{2}{3} \sqrt{n} \frac{N}{n - 1}} \right).$$

We choose $N$ such that $|R(n, N)| < 0.25$, and a working precision at term $k$ such that the absolute error of the term is expected to be less than $0.25/N$. We also use a summation variable with increased precision, essentially making additions exact. Thus the sum of errors adds up to less than 0.5, giving the correct value of $p(n)$ when rounding to the nearest integer.

The remainder estimate at step $k$ provides an upper bound for the size of the $k$-th term. We add $\log_2 N$ bits to get low bits in the terms below $0.25/N$ in magnitude.

Using `arith_hrr_expsum_factored`, each $B_k(n)$ evaluation is broken down to a product of cosines of exact rational multiples of $\pi$. We transform all angles to $[0, \pi/4)$ for optimal accuracy.

Since the evaluation of each term involves only $O(\log k)$ multiplications and evaluations of trigonometric functions of small angles, the relative rounding error is at most a few bits. We therefore just add an additional $\log_2(C/k)$ bits for the $U(x)$ when $x$ is large. The cancellation of terms in $U(x)$ is of no concern, since Rademacher’s bound allows us to terminate before $x$ becomes small.

This analysis should be performed in more detail to give a rigorous error bound, but the precision currently implemented is almost certainly sufficient, not least considering that Rademacher’s remainder bound significantly overshoots the actual values.

To improve performance, we switch to doubles when the working precision becomes small enough. We also use a separate accumulator variable which gets added to the main sum periodically, in order to avoid costly updates of the full-precision result when $n$ is large.

```c
void arith_number_of_partitions(fmpz_t x, ulong n)
```

Sets $x$ to $p(n)$, the number of ways that $n$ can be written as a sum of positive integers without regard to order.

This function uses a lookup table for $n < 128$ (where $p(n) < 2^{32}$), and otherwise calls `arith_number_of_partitions_mpfr`.
4.16.11 Sums of squares

void arith_sum_of_squares(fmpz_t r, ulong k, const fmpz_t n)

Sets \( r \) to the number of ways \( r_k(n) \) in which \( n \) can be represented as a sum of \( k \) squares.

If \( k = 2 \) or \( k = 4 \), we write \( r_k(n) \) as a divisor sum.

Otherwise, we either recurse on \( k \) or compute the theta function expansion up to \( O(x^{n+1}) \) and read off the last coefficient. This is generally optimal.

void arith_sum_of_squares_vec(fmpz *r, ulong k, slong n)

For \( i = 0, 1, \ldots, n - 1 \), sets \( r_i \) to the number of representations of \( i \) a sum of \( k \) squares, \( r_k(i) \). This effectively computes the \( q \)-expansion of \( \vartheta_3(q) \) raised to the \( k \)-th power, i.e.

\[
\vartheta_3^k(q) = \left( \sum_{i=-\infty}^{\infty} q^{i^2} \right)^k.
\]

4.17 fft.h – Schoenhage-Strassen FFT

4.17.1 Split/combine FFT coefficients

mp_size_t fft_split_limbs(mp_limb_t **poly, mp_srcptr limbs, mp_size_t total_limbs, mp_size_t coeff_limbs, mp_size_t output_limbs)

Split an integer \((\text{limbs, total_limbs})\) into coefficients of length \(\text{coeff_limbs}\) limbs and store as the coefficients of \(\text{poly}\) which are assumed to have space for \(\text{output_limbs} + 1\) limbs per coefficient. The coefficients of the polynomial do not need to be zeroed before calling this function, however the number of coefficients written is returned by the function and any coefficients beyond this point are not touched.

mp_size_t fft_split_bits(mp_limb_t **poly, mp_srcptr limbs, mp_size_t total_limbs, flint_bitcnt_t bits, mp_size_t output_limbs)

Split an integer \((\text{limbs, total_limbs})\) into coefficients of the given number of \(\text{bits}\) and store as the coefficients of \(\text{poly}\) which are assumed to have space for \(\text{output_limbs} + 1\) limbs per coefficient. The coefficients of the polynomial do not need to be zeroed before calling this function, however the number of coefficients written is returned by the function and any coefficients beyond this point are not touched.

void fft_combine_limbs(mp_limb_t *res, mp_limb_t **poly, slong length, mp_size_t coeff_limbs, mp_size_t output_limbs, mp_size_t total_limbs)

Evaluate the polynomial \(\text{poly}\) of the given \(\text{length}\) at \(B^{\text{coeff_limbs}}\), where \(B = 2^\text{FLINT_BITS}\), and add the result to the integer \((\text{res, total_limbs})\) throwing away any bits that exceed the given number of limbs. The polynomial coefficients are assumed to have at least \(\text{output_limbs}\) limbs each, however any additional limbs are ignored.

If the integer is initially zero the result will just be the evaluation of the polynomial.

void fft_combine_bits(mp_limb_t *res, mp_limb_t **poly, slong length, flint_bitcnt_t bits, mp_size_t output_limbs, mp_size_t total_limbs)

Evaluate the polynomial \(\text{poly}\) of the given \(\text{length}\) at \(2^{\text{bits}}\) and add the result to the integer \((\text{res, total_limbs})\) throwing away any bits that exceed the given number of limbs. The polynomial coefficients are assumed to have at least \(\text{output_limbs}\) limbs each, however any additional limbs are ignored. If the integer is initially zero the result will just be the evaluation of the polynomial.
4.17.2 Test helper functions

```c
void fermat_to_mpz(mpz_t m, mp_limb_t *i, mp_size_t limbs)
```

Convert the Fermat number \((i, \text{limbs})\) modulo \(B^{\text{limbs}} + 1\) to an \(mpz_t\) \(m\). Assumes \(m\) has been initialised. This function is used only in test code.

4.17.3 Arithmetic modulo a generalised Fermat number

```c
void mpn_negmod_2exp11(mp_limb_t *z, const mp_limb_t *a, mp_size_t limbs)
```

Set \(z\) to the negation of the Fermat number \(a\) modulo \(B^{\text{limbs}} + 1\). The input \(a\) is expected to be fully reduced, and the output is fully reduced. Aliasing is permitted.

```c
void mpn_addmod_2exp11(mp_limb_t *r, mp_size_t limbs, mp_limb_signed_t c)
```

Adds the signed limb \(c\) to the generalised Fermat number \(r\) modulo \(B^{\text{limbs}} + 1\). The compiler should be able to inline this for the case that there is no overflow from the first limb.

```c
void mpn_normmod_2exp11(mp_limb_t *t, mp_size_t limbs)
```

Given \(t\) a signed integer of \(\text{limbs} + 1\) limbs in two’s complement format, reduce \(t\) to the corresponding value modulo the generalised Fermat number \(B^{\text{limbs}} + 1\), where \(B = 2^{\text{FLINT_BITS}}\).

```c
void mpn_mul_2expmod_2exp11(mp_limb_t *t, const mp_limb_t *i1, mp_size_t limbs, flint_bitcnt_t d)
```

Given \(i1\) a signed integer of \(\text{limbs} + 1\) limbs in two’s complement format reduced modulo \(B^{\text{limbs}} + 1\) up to some overflow, compute \(t = i1 \times 2^d\) modulo \(p\). The result will not necessarily be fully reduced. The number of bits \(d\) must be nonnegative and less than \(\text{FLINT_BITS}\). Aliasing is permitted.

4.17.4 Generic butterflies

```c
void fft_adjust(mp_limb_t *r, mp_limb_t *il, mp_size_t i, mp_size_t limbs, flint_bitcnt_t w)
```

Set \(r\) to \(i1\) times \(z^i\) modulo \(B^{\text{limbs}} + 1\) where \(z\) corresponds to multiplication by \(2^w\). This can be thought of as part of a butterfly operation. We require \(0 \leq i < n\) where \(nw = \text{limbs} \times \text{FLINT_BITS}\). Aliasing is not supported.

```c
void butterfly_lshB(mp_limb_t *t, mp_limb_t *u, mp_limb_t *il, mp_limb_t *i2, mp_size_t limbs, mp_size_t x, mp_size_t y)
```

We are given two integers \(i1\) and \(i2\) modulo \(B^{\text{limbs}} + 1\) which are not necessarily normalised. We compute \(t = (i1 + i2)B^x\) and \(u = (i1 - i2)B^y\) modulo \(p\). Aliasing between inputs and outputs is not permitted. We require \(x\) and \(y\) to be less than \(\text{limbs}\) and nonnegative.

```c
void butterfly_rshB(mp_limb_t *t, mp_limb_t *u, mp_limb_t *il, mp_limb_t *i2, mp_size_t limbs, mp_size_t x, mp_size_t y)
```

We are given two integers \(i1\) and \(i2\) modulo \(B^{\text{limbs}} + 1\) which are not necessarily normalised. We compute \(t = (i1 + i2)/B^x\) and \(u = (i1 - i2)/B^y\) modulo \(p\). Aliasing between inputs and outputs is not permitted. We require \(x\) and \(y\) to be less than \(\text{limbs}\) and nonnegative.
4.17.5 Radix 2 transforms

```c
void fft_butterfly(mp_limb_t *s, mp_limb_t *t, mp_limb_t *i1, mp_limb_t *i2, mp_size_t i, mp_size_t limbs, flint_bitcnt_t w)
```

Set $s = i1 + i2$, $t = z1^{-i}(i1 - i2)$ modulo $B^{\text{limbs}} + 1$ where $z1 = \exp(Pi*\text{I}/n)$ corresponds to multiplication by $2^w$. Requires $0 \leq i < n$ where $nw = \text{limbs*FLINT_BITS}$.

```c
void ifft_butterfly(mp_limb_t *s, mp_limb_t *t, mp_limb_t *i1, mp_limb_t *i2, mp_size_t i, mp_size_t limbs, flint_bitcnt_t w)
```

Set $s = i1 + z1^{i2}, t = i1 - z1^{i2}$ modulo $B^{\text{limbs}} + 1$ where $z1 = \exp(-Pi*\text{I}/n)$ corresponds to division by $2^w$. Requires $0 \leq i < 2n$ where $nw = \text{limbs*FLINT_BITS}$.

```c
void fft_radix2(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2)
```

The radix 2 DIF FFT works as follows:

Input: $[i0, i1, \ldots, i(m-1)]$ for $m = 2^n$ a power of 2.

Output: $[r0, r1, \ldots, r(m-1)] = \text{FFT}[i0, i1, \ldots, i(m-1)]$.

Algorithm:

- Recursively compute $[r0, r2, r4, \ldots, r(m-2)]$ = $\text{FFT}[i0+i(m/2), i1+i(m/2+1), \ldots, i(m/2-1)+i(m-1)]$

- Let $[t0, t1, \ldots, t(m/2-1)]$ = $[i0-i(m/2), i1-i(m/2+1), \ldots, i(m/2-1)-i(m-1)]$

- Let $[u0, u1, \ldots, u(m/2-1)]$ = $[z1^{0}*t0, z1^{1}*t1, \ldots, z1^{(m/2-1)}*t(m/2-1)]$
  where $z1 = \exp(2*Pi*\text{I}/m)$ corresponds to multiplication by $2^w$.

- Recursively compute $[r1, r3, \ldots, r(m-1)]$ = $\text{FFT}[u0, u1, \ldots, u(m/2-1)]$

The parameters are as follows:

- $2^n$ is the length of the input and output arrays
- $w$ is such that $2^w$ is an $2n$-th root of unity in the ring $\mathbb{Z}/p\mathbb{Z}$ that we are working in, i.e. $p = 2^{2n} + 1$ (here $n$ is divisible by GMP_LIMB_BITS)
- $ii$ is the array of inputs (each input is an array of limbs of length $wn/GMP\_LIMB\_BITS + 1$ (the extra limbs being a “carry limb”). Outputs are written in-place.

We require $nw$ to be at least 64 and the two temporary space pointers to point to blocks of size $nw + \text{FLINT_BITS}$ bits.

```c
void fft_truncate(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t trunc)
```

As for `fft_radix2` except that only the first `trunc` coefficients of the output are computed and the input is regarded as having (implied) zero coefficients from coefficient `trunc` onwards. The coefficients must exist as the algorithm needs to use this extra space, but their value is irrelevant. The value of `trunc` must be divisible by 2.

```c
void fft_truncatel(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t trunc)
```
As for `fft_radix2` except that only the first `trunc` coefficients of the output are computed. The transform still needs all `2n` input coefficients to be specified.

```c
void ifft_radix2(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2)
```

The radix 2 DIF IFFT works as follows:

Input: `[i0, i1, ..., i(m-1)]`, for \( m = 2^n \) a power of 2.

Output: `[r0, r1, ..., r(m-1)]`

\[ r_i = \text{IFFT}[i_0, i_1, ..., i(m-1)] \]

Algorithm:

- Recursively compute \([s0, s1, ..., s(m/2-1)]\)
  \[ r_i = \text{IFFT}[i_0, i_2, ..., i(m-2)] \]
- Recursively compute \([t(m/2), t(m/2+1), ..., t(m-1)]\)
  \[ r_i = \text{IFFT}[i_1, i_3, ..., i(m-1)] \]
- Let \([r0, r1, ..., r(m/2-1)]\)
  \[ r_i = [s0 + z1^i t0, s1 + z1^i t1, ..., s(m/2-1) + z1^i t(m/2-1)] \]
  where \( z1 = \exp(-2\pi i/m) \) corresponds to division by \( 2^w \).
- Let \([r(m/2), r(m/2+1), ..., r(m-1)]\)
  \[ r_i = [s0 - z1^i t0, s1 - z1^i t1, ..., s(m/2-1) - z1^i t(m/2-1)] \]

The parameters are as follows:

- \( 2n \) is the length of the input and output arrays
- \( w \) is such that \( 2^w \) is an \( 2^n \)-th root of unity in the ring \( \mathbb{Z}/p\mathbb{Z} \) that we are working in, i.e. \( p = 2^{wn} + 1 \) (here \( n \) is divisible by \( \text{GMP}_\text{LIMB}_\text{BITS} \))
- \( ii \) is the array of inputs (each input is an array of limbs of length \( wn/\text{GMP}_\text{LIMB}_\text{BITS} + 1 \) (the extra limbs being a “carry limb”). Outputs are written in-place.

We require \( nw \) to be at least 64 and the two temporary space pointers to point to blocks of size `n*w + FLINT_BITS` bits.

```c
void ifft_truncate(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t trunc)
```

As for `fft_radix2` except that the output is assumed to have zeros from coefficient `trunc` onwards and only the first `trunc` coefficients of the input are specified. The remaining coefficients need to exist as the extra space is needed, but their value is irrelevant. The value of `trunc` must be divisible by 2.

Although the implementation does not require it, we assume for simplicity that `trunc` is greater than \( n \). The algorithm begins by computing the inverse transform of the first \( n \) coefficients of the input array. The unspecified coefficients of the second half of the array are then written: coefficient \( trunc + i \) is computed as a twist of coefficient \( i \) by a root of unity. The values of these coefficients are then equal to what they would have been if the inverse transform of the right hand side of the input array had been computed with full data from the start. The function `ifft_truncate1` is then called on the entire right half of the input array with this auxiliary data filled in. Finally a single layer of the IFFT is completed on all the coefficients up to `trunc` being careful to note that this involves doubling the coefficients from `trunc` up to `n`.

```c
void ifft_truncate1(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t trunc)
```

Computes the first `trunc` coefficients of the radix 2 inverse transform assuming the first `trunc` coefficients are given and that the remaining coefficients have been set to the value they would have if an inverse transform had already been applied with full data.
The algorithm is the same as for \texttt{ifft_truncate} except that the coefficients from \texttt{trunc} onwards after the inverse transform are not inferred to be zero but the supplied values.

\begin{verbatim}
void \texttt{fft_butterfly_sqrt2}(mp_limb_t *s, mp_limb_t *t, mp_limb_t *i1, mp_limb_t *i2, mp_size_t i, mp_size_t limbs, flint_bitcnt_t w, mp_limb_t *temp)
\end{verbatim}

Let $w = 2k + 1$, $i = 2j + 1$. Set $s = i1 + i2$, $t = z1^{-i}(i1 - i2)$ modulo $B^{\text{limbs} + 1}$ where $z1^{-2} = \exp(Pi*I/n)$ corresponds to multiplication by $2^w$. Requires $0 \leq i < 2n$ where $nw = \text{limbs}\times\text{FLINT\_BITS}$.

Here $z1$ corresponds to multiplication by $2^k$ then multiplication by $(2^{-3nw/4} - 2^{-nw/4})$. We see $z1^{-i}$ corresponds to multiplication by $(2^{-3nw/4} - 2^{-nw/4})\times 2^{-j}(j+ik)$ which is the same as division by $(2^{-3nw/4} - 2^{-nw/4})$.

Of course, division by $2^{-j}(j+ik + 1)$ is the same as multiplication by $2^{-2\cdot n - j - ik - 1}$. The exponent is positive as $i \leq 2\cdot n$, $j < n$, $k < w/2$.

We first multiply by $2^{-2\cdot n - j - ik - 1 + wn/4}$ then multiply by an additional $2^{-nw/2}$ and subtract.

\begin{verbatim}
void \texttt{fft_butterfly_sqrt2}(mp_limb_t *s, mp_limb_t *t, mp_limb_t *i1, mp_limb_t *i2, mp_size_t i, mp_size_t limbs, flint_bitcnt_t w, mp_limb_t *temp)
\end{verbatim}

Let $w = 2k + 1$, $i = 2j + 1$. Set $s = i1 + z1^{-i}i2$, $t = i1 - z1^{-i}i2$ modulo $B^{\text{limbs} + 1}$ where $z1^{-2} = \exp(-Pi*I/n)$ corresponds to division by $2^w$. Requires $0 \leq i < 2n$ where $nw = \text{limbs}\times\text{FLINT\_BITS}$.

Here $z1$ corresponds to division by $2^k$ then by division by $(2^{-3nw/4} - 2^{-nw/4})$. We see $z1^{-i}$ corresponds to division by $(2^{-3nw/4} - 2^{-nw/4})\times 2^{-j}(j+ik)$ which is the same as division by $2^{-2\cdot n - j - ik - 1}$.

We first multiply by $2^{-2\cdot n - j - ik - 1 + wn/4}$ then multiply by an additional $2^{-nw/2}$ and subtract.

\begin{verbatim}
void \texttt{fft_truncate_sqrt2}(mp_limb_t **i1, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t trunc)
\end{verbatim}

As per \texttt{fft_truncate} except that the transform is twice the usual length, i.e. length $4n$ rather than $2n$. This is achieved by making use of twiddles by powers of a square root of 2, not powers of 2 in the first layer of the transform.

We require $nw$ to be at least 64 and the three temporary space pointers to point to blocks of size $n\cdot w + \text{FLINT\_BITS}$ bits.

\begin{verbatim}
void \texttt{ifft_truncate_sqrt2}(mp_limb_t **i1, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t trunc)
\end{verbatim}

As per \texttt{ifft_truncate} except that the transform is twice the usual length, i.e. length $4n$ instead of $2n$. This is achieved by making use of twiddles by powers of a square root of 2, not powers of 2 in the final layer of the transform.

We require $nw$ to be at least 64 and the three temporary space pointers to point to blocks of size $n\cdot w + \text{FLINT\_BITS}$ bits.

### 4.17.6 Matrix Fourier Transforms

\begin{verbatim}
void \texttt{fft_butterfly_twiddle}(mp_limb_t *u, mp_limb_t *v, mp_limb_t *s, mp_limb_t *t, mp_size_t limbs, flint_bitcnt_t b1, flint_bitcnt_t b2)
\end{verbatim}

Set $u = 2\cdot b1\cdot (s + t)$, $v = 2\cdot b2\cdot (s - t)$ modulo $B^{\text{limbs} + 1}$. This is used to compute $u = 2^{-w\cdot \text{tw1}}\cdot (s + t)$, $v = 2^{-w\cdot \text{tw1} + s\cdot \text{tw2}}\cdot (s - t)$ in the matrix Fourier transform. We effectively computing an ordinary butterfly with additional twiddles by $z1^{-rc}$ for row $r$ and column $c$ of the matrix of coefficients. Aliasing is not allowed.

\begin{verbatim}
void \texttt{ifft_butterfly_twiddle}(mp_limb_t *u, mp_limb_t *v, mp_limb_t *s, mp_limb_t *t, mp_size_t limbs, flint_bitcnt_t b1, flint_bitcnt_t b2)
\end{verbatim}

Set $u = s/2\cdot b1 + t/2\cdot b1$, $v = s/2\cdot b1 - t/2\cdot b1$ modulo $B^{\text{limbs} + 1}$. This is used to compute $u = 2^{-w\cdot \text{tw1}}\cdot s + 2^{-w\cdot \text{tw2}}\cdot t$, $v = 2^{-w\cdot \text{tw1} + \text{tw2}}\cdot s + 2^{-w\cdot \text{tw2}}\cdot t$ in the
matrix Fourier algorithm, i.e. effectively computing an ordinary butterfly with additional twiddles by $z^{-rc}$ for row $r$ and column $c$ of the matrix of coefficients. Aliasing is not allowed.

```c
void fft_radix2_twiddle(mp_limb_t **ii, mp_size_t is, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t w1, mp_size_t w2, mp_limb_t w, mp_size_t rs)
```

As for `fft_radix2` except that the coefficients are spaced by `is` in the array `ii` and an additional twist by $z^{-c*i}$ is applied to each coefficient where $i$ starts at $r$ and increases by `rs` as one moves from one coefficient to the next. Here $z$ corresponds to multiplication by $2^{\text{w2}}$.

```c
void ifft_radix2_twiddle(mp_limb_t **ii, mp_size_t is, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t w1, mp_size_t w2, mp_limb_t w, mp_size_t c, mp_size_t rs)
```

As for `ifft_radix2` except that the coefficients are spaced by `is` in the array `ii` and an additional twist by $z^{-(c*i)}$ is applied to each coefficient where $i$ starts at $r$ and increases by `rs` as one moves from one coefficient to the next. Here $z$ corresponds to multiplication by $2^{\text{w2}}$.

```c
void fft_truncate1_twiddle(mp_limb_t **ii, mp_size_t is, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t w1, mp_size_t w2, mp_size_t w, mp_size_t c, mp_size_t rs, mp_size_t trunc)
```

As per `fft_radix2_twiddle` except that the transform is truncated as per `fft_truncate1`.

```c
void ifft_truncate1_twiddle(mp_limb_t **ii, mp_size_t is, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t w1, mp_size_t w2, mp_size_t w, mp_size_t c, mp_size_t rs, mp_size_t trunc)
```

As per `ifft_radix2_twiddle` except that the transform is truncated as per `ifft_truncate1`.

```c
void fft_mfa_truncate_sqrt2(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1, mp_size_t trunc)
```

This is as per the `fft_truncate_sqrt2` function except that the matrix Fourier algorithm is used for the left and right FFTs. The total transform length is $4n$ where $n = 2^\text{depth}$ so that the left and right transforms are both length $2n$. We require $\text{trunc} > 2*n$ and that $\text{trunc}$ is divisible by $2n1$ (explained below). The coefficients are produced in an order different from `fft_truncate_sqrt2`.

The matrix Fourier algorithm, which is applied to each transform of length $2n$, works as follows. We set $n1$ to a power of 2 about the square root of $n$. The data is then thought of as a set of $n2$ rows each with $n1$ columns (so that $n1*n2 = 2n$).

The length $2n$ transform is then computed using a whole pile of short transforms. These comprise $n1$ column transforms of length $n2$ followed by some twiddles by roots of unity (namely $z^{rc}$ where $r$ is the row and $c$ the column within the data) followed by $n2$ row transforms of length $n1$. Along the way the data needs to be rearranged due to the fact that the short transforms output the data in binary reversed order compared with what is needed.

The matrix Fourier algorithm provides better cache locality by decomposing the long length $2n$ transforms into many transforms of about the square root of the original length.

For better cache locality the `sqrt2` layer of the full length $4n$ transform is folded in with the column FFTs performed as part of the first matrix Fourier algorithm on the left half of the data.

The second half of the data requires a truncated version of the matrix Fourier algorithm. This is achieved by truncating to an exact multiple of the row length so that the row transforms are full length. Moreover, the column transforms will then be truncated transforms and their truncated length needs to be a multiple of 2. This explains the condition on `trunc` given above.

To improve performance, the extra twiddles by roots of unity are combined with the butterflies performed at the last layer of the column transforms.

We require $n*w + \text{FLINT_BITS}$ bits.
void ifft_mfa_truncate_sqrt2(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1, mp_size_t trunc)

This is as per the ifft_truncate_sqrt2 function except that the matrix Fourier algorithm is used for the left and right IFFTs. The total transform length is $4n$ where $n = 2^\text{depth}$ so that the left and right transforms are both length $2n$. We require $\text{trunc} > 2*n$ and that $\text{trunc}$ is divisible by $2*n1$.

We set $n1$ to a power of 2 about the square root of $n$.

As per the matrix fourier FFT the sqrt2 layer is folded into the final column IFFTs for better cache locality and the extra twiddles that occur in the matrix Fourier algorithm are combined with the butterflied performed at the first layer of the final column transforms.

We require $nw$ to be at least 64 and the three temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

void fft_mfa_truncate_sqrt2_outer(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1, mp_size_t trunc)

Just the outer layers of fft_mfa_truncate_sqrt2.

void fft_mfa_truncate_sqrt2_inner(mp_limb_t **ii, mp_limb_t **jj, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1, mp_size_t trunc, mp_limb_t **tt)

The inner layers of fft_mfa_truncate_sqrt2 and ifft_mfa_truncate_sqrt2 combined with pointwise mults.

void ifft_mfa_truncate_sqrt2_outer(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1, mp_size_t trunc)

The outer layers of ifft_mfa_truncate_sqrt2 combined with normalisation.

### 4.17.7 Negacyclic multiplication

void fft_negacyclic(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp)

As per fft_radix2 except that it performs a sqrt2 negacyclic transform of length $2n$. This is the same as the radix 2 transform except that the $i$-th coefficient of the input is first multiplied by $\sqrt{2}i^w$.

We require $nw$ to be at least 64 and the two temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

void ifft_negacyclic(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp)

As per ifft_radix2 except that it performs a sqrt2 negacyclic inverse transform of length $2n$. This is the same as the radix 2 inverse transform except that the $i$-th coefficient of the output is finally divided by $\sqrt{2}i^w$.

We require $nw$ to be at least 64 and the two temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

void fft_naive_convolution_1(mp_limb_t *r, mp_limb_t *ii, mp_limb_t *jj, mp_size_t m)

Performs a naive negacyclic convolution of ii with jj, both of length $m$, and sets $r$ to the result. This is essentially multiplication of polynomials modulo $x^m + 1$.

void _fft_mulmod_2expp1(mp_limb_t *r1, mp_limb_t *i1, mp_limb_t *i2, mp_size_t r_limbs, flint_bitcnt_t depth, flint_bitcnt_t w)
Multiply \( i_1 \) by \( i_2 \) modulo \( B^{r_{\text{limbs}}} + 1 \) where \( r_{\text{limbs}} = \frac{n \times w}{\text{FLINT\_BITS}} \) with \( n = 2^{\text{depth}} \). Uses the negacyclic FFT convolution CRT’d with a 1 limb naive convolution. We require that \( \text{depth} \) and \( w \) have been selected as per the wrapper \_fft\_mul\_mod\_2\_expp1\below below.

\[ \text{slong} \text{ fft\_adjust\_limbs}(\text{mp\_size\_t} \text{ limbs}) \]

Given a number of limbs, returns a new number of limbs (no more than the next power of 2) which will work with the Nussbaumer code. It is only necessary to make this adjustment if \( \text{limbs} > \text{FFT\_MUL\_MOD\_2\_EXPP1\_CUTOFF} \).

\[ \text{void} \text{ fft\_mul\_mod\_2\_expp1}(\text{mp\_limb\_t} *r, \text{mp\_limb\_t} *i1, \text{mp\_limb\_t} *i2, \text{mp\_size\_t} n, \text{mp\_size\_t} w, \text{mp\_limb\_t} *tt) \]

As per \_fft\_mul\_mod\_2\_expp1 but with a tuned cutoff below which more classical methods are used for the convolution. The temporary space is required to fit \( n \times w + \text{FLINT\_BITS} \) bits. There are no restrictions on \( n \), but if \( \text{limbs} = \frac{n \times w}{\text{FLINT\_BITS}} \) then if \( \text{limbs} \) exceeds \( \text{FFT\_MUL\_MOD\_2\_EXPP1\_CUTOFF} \) the function \text{fft\_adjust\_limbs} must be called to increase the number of limbs to an appropriate value.

### 4.17.8 Integer multiplication

\[ \text{void} \text{ mul\_truncate\_sqrt2}(\text{mp\_ptr} r1, \text{mp\_srcptr} i1, \text{mp\_size\_t} n1, \text{mp\_srcptr} i2, \text{mp\_size\_t} n2, \text{flint\_bitcnt\_t} \text{ depth}, \text{flint\_bitcnt\_t} \text{ w}) \]

Integer multiplication using the radix 2 truncated sqrt2 transforms. Set \( (r_1, n_1 + n_2) \) to the product of \( (i_1, n_1) \) by \( (i_2, n_2) \). This is achieved through an FFT convolution of length at most \( 2^{(\text{depth} + 2)} \) with coefficients of size \( n \times w \) bits where \( n = 2^{\text{depth}} \).

We require \( \text{depth} \geq 6 \). The input data is broken into chunks of data not exceeding \( \frac{n \times (\text{depth} + 1)}{2} \) bits. If the breaking the first integer into chunks of this size results in \( j_1 \) coefficients and breaking the second integer results in \( j_2 \) chunks then \( j_1 + j_2 - 1 \leq 2^{(\text{depth} + 2)} \).

If \( n = 2^{\text{depth}} \) then we require \( n \times w \) to be at least 64.

\[ \text{void} \text{ mul\_mfa\_truncate\_sqrt2}(\text{mp\_ptr} r1, \text{mp\_srcptr} i1, \text{mp\_size\_t} n1, \text{mp\_srcptr} i2, \text{mp\_size\_t} n2, \text{flint\_bitcnt\_t} \text{ depth}, \text{flint\_bitcnt\_t} \text{ w}) \]

As for \text{mul\_truncate\_sqrt2} except that the cache friendly matrix Fourier algorithm is used.

If \( n = 2^{\text{depth}} \) then we require \( n \times w \) to be at least 64. Here we also require \( w \) to be \( 2^i \) for some \( i \geq 0 \).

\[ \text{void} \text{ flint\mpn\_mul\_fft\_main}(\text{mp\_ptr} r1, \text{mp\_srcptr} i1, \text{mp\_size\_t} n1, \text{mp\_srcptr} i2, \text{mp\_size\_t} n2) \]

The main integer multiplication routine. Sets \( (r_1, n_1 + n_2) \) to \( (i_1, n_1) \) times \( (i_2, n_2) \). We require \( n_1 \geq n_2 > 0 \).

### 4.17.9 Convolution

\[ \text{void} \text{ fft\_convolution}(\text{mp\_limb\_t} *i, \text{mp\_limb\_t} *j, \text{slong} \text{ depth}, \text{slong} \text{ limbs}, \text{slong} \text{ trunc}, \text{mp\_limb\_t} *t1, \text{mp\_limb\_t} *t2, \text{mp\_limb\_t} *s1, \text{mp\_limb\_t} *tt) \]

Perform an FFT convolution of \( i \) with \( j \), both of length \( 4 \times n \) where \( n = 2^{\text{depth}} \). Assume that all but the first \( \text{trunc} \) coefficients of the output (placed in \( i \)) are zero. Each coefficient is taken modulo \( B^{\text{limbs}} + 1 \). The temporary spaces \( t_1, t_2 \) and \( s_1 \) must have \( \text{limbs} + 1 \) limbs of space and \( tt \) must have \( 2 \times (\text{limbs} + 1) \) of free space.
4.17.10 FFT Precaching

void fft_precache(mp_limb_t **jj, slong depth, slong limbs, slong trunc, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **s1)

Precompute the FFT of jj for use with precache functions. The parameters are as for fft_convolution.

void fft_convolution_precache(mp_limb_t **ii, mp_limb_t **jj, slong depth, slong limbs, slong trunc, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **s1, mp_limb_t **tt)

As per fft_convolution except that it is assumed fft_precache has been called on jj with the same parameters. This will then run faster than if fft_convolution had been run with the original jj.

4.18 fft_small.h – FFT modulo word-size primes

This module currently requires building FLINT with support for AVX2 or NEON instructions.

4.18.1 Integer multiplication

type mpn_ctx_struct

type mpn_ctx_t

Context object for multiplications allowing non-FFT moduli. The structure contains FFT context objects for multiple FFT primes (currently 8) together with tables for Chinese remaindering.

void mpn_ctx_init(mpn_ctx_t R, ulong p)

Initialize multiplication context object with initial prime p.

void mpn_ctx_clear(mpn_ctx_t R)

Free memory allocated by the context object.

mpn_ctx_struct *get_default_mpn_ctx(void)

Return a pointer to a cached thread-local context object used by default for multiplications. Calling flint_cleanup() or flint_cleanup_master() frees the cache.

void mpn_ctx_mpn_mul(mpn_ctx_t R, ulong *r1, const ulong *i1, ulong n1, const ulong *i2, ulong n2)

void mpn_mul_default_mpn_ctx(mp_ptr r1, mp_srcptr i1, mp_size_t n1, mp_srcptr i2, mp_size_t n2)

Writes to r1 the product of the integers (i1, n1) and (i2, n2). Assumes that $n_1 \geq n_2 \geq 1$, respectively using a given context object R or the default thread-local object.

4.18.2 Polynomial arithmetic

void _nmod_poly_mul_mid_mpn_ctx(ulong *z, ulong zl, ulong zh, const ulong *a, ulong an, const ulong *b, ulong bn, nmod_t mod, mpn_ctx_t R)

void _nmod_poly_mul_mid_default_mpn_ctx(mp_ptr res, slong zl, slong zh, mp_srcptr a, slong an, mp_srcptr b, slong bn, nmod_t mod)

Writes to z the middle product containing coefficients in the range $[zl, zh]$ of the product of the polynomials (a, an) and (b, bn), respectively using a given context object R or the default thread-local object. Assumes that $an \geq bn \geq 1$.

int _fmpz_poly_mul_mid_mpn_ctx(fmpz *z, ulong zl, ulong zh, const fmpz *a, ulong an, const fmpz *b, ulong bn, mpn_ctx_t R)
int _fmpz_poly_mul_mid_default_mpn_ctx(fmpz *z, ulong zl, ulong zh, const fmpz *a, ulong an, const fmpz *b, ulong bn)

Like the nmod functions. Performs the multiplication and returns 1 if there are sufficiently many primes R to compute the result; otherwise returns 0 without touching the output.

void _nmod_poly_divrem_mpn_ctx(ulong *q, ulong *r, const ulong *a, ulong an, const ulong *b, ulong bn, nmod_t mod, mpn_ctx_t R)

Polynomial division with remainder.

4.18.3 Preconditioned polynomial arithmetic

type mul_precomp_struct

void _mul_precomp_init(mul_precomp_struct *M, const ulong *b, ulong bn, ulong btrunc, ulong depth, nmod_t mod, mpn_ctx_t R)

void _mul_precomp_clear(mul_precomp_struct *M)

Represents (b, bn) in transformed form for preconditioned multiplication.

int _nmod_poly_mul_mid_precomp(ulong *z, ulong zl, ulong zh, const ulong *a, ulong an, mul_precomp_struct *M, nmod_t mod, mpn_ctx_t R)

Polynomial multiplication given a precomputed transform M. Returns 1 if successful, 0 if the pre-computed transform is too short.

type nmod_poly_divrem_precomp_struct

void _nmod_poly_divrem_precomp_init(nmod_poly_divrem_precomp_struct *M, const ulong *b, ulong bn, Bn, nmod_t mod, mpn_ctx_t R)

void _nmod_poly_divrem_precomp_clear(nmod_poly_divrem_precomp_struct *M)

Represents (b, bn) and its inverse in transformed form for preconditioned multiplication.

int _nmod_poly_divrem_precomp(ulong *q, ulong *r, const ulong *a, ulong an, nmod_poly_divrem_precomp_struct *M, nmod_t mod, mpn_ctx_t R)

Polynomial multiplication given a precomputed transform M. Returns 1 if successful, 0 if the pre-computed transform is too short.

4.19 qsieve.h – Quadratic sieve

mp_limb_t qsieve_knuth_schroeppel(qs_t qs_inf)

Return the Knuth-Schroeppel multiplier for the n, integer to be factored based upon the Knuth-Schroeppel function.

mp_limb_t qsieve_primes_init(qs_t qs_inf)

Compute the factor base prime along with the inverse for kn, where k is Knuth-Schroeppel multiplier and n is the integer to be factored. It also computes the square root of kn modulo factor base primes.

mp_limb_t qsieve_primes_increment(qs_t qs_inf, mp_limb_t delta)

It increase the number of factor base primes by amount ‘delta’ and calculate inverse of those primes along with the square root of kn modulo those primes.

void qsieve_init_A0(qs_t qs_inf)

First it chooses the possible range of factor of A0, based on the number of bits in optimal value of A0. It tries to select range such that we have plenty of primes to choose from as well as number of factor in A0 are sufficient. For input of size less than 130 bit, this selection method doesn’t work therefore we randomly generate 2 or 3-subset of all the factor base prime as the factor of A0.
Otherwise, if we have to select \( s \) factor for \( A_0 \), we generate \( s-1 \)-subset from odd indices of the possible range of factor and then search last factor using binary search from the even indices of possible range of factor such that value of \( A_0 \) is close to it’s optimal value.

```c
void qsieve_next_A0(qs_t qs_inf)
{ // Find next candidate for \( A_0 \) as follows: generate next lexicographic \( s-1 \)-subset from the odd indices of possible range of factor base and choose the last factor from even indices using binary search so that value of \( A_0 \) is close to it’s optimal value.
}
```

```c
void qsieve_compute_pre_data(qs_t qs_inf)
{ // Precompute all the data associated with factor’s of \( A_0 \), since \( A_0 \) is going to be fixed for several \( A \).
}
```

```c
void qsieve_init_poly_first(qs_t qs_inf)
{ // Initializes the value of \( A = q_0 * A_0 \), where \( q_0 \) is non-factor base prime. precompute the data necessary for generating different \( B \) value using grey code formula. Combine the data calculated for the factor of \( A_0 \) along with the parameter \( q_0 \) to obtain data as for factor of \( A \). It also calculates the sieve offset for all the factor base prime, for first polynomial.
}
```

```c
void qsieve_init_poly_next(qs_t qs_inf, ulong i)
{ // Generate next polynomial or next \( B \) value for particular \( A \) and also updates the sieve offsets for all the factor base prime, for this \( B \) value.
}
```

```c
void qsieve_compute_C(fmpz_t C, qs_t qs_inf, qs_poly_t poly)
{ // Given \( A \) and \( B \), calculate \( C = (B^2 - A)/N \).
}
```

```c
void qsieve_do_sieving(qs_t qs_inf, unsigned char *sieve, qs_poly_t poly)
{ // First initialize the sieve array to zero, then for each \( p \in \text{factor base} \), add \( \log_2(p) \) to the locations \( \text{soln1}_p + i * p \) and \( \text{soln2}_p + i * p \) for \( i = 0, 1, 2, \ldots \), where \( \text{soln1}_p \) and \( \text{soln2}_p \) are the sieve offsets calculated for \( p \).
}
```

```c
void qsieve_do_sieving2(qs_t qs_inf, unsigned char *sieve, qs_poly_t poly)
{ // Perform the same task as above but instead of sieving over whole array at once divide the array in blocks and then sieve over each block for all the primes in factor base.
}
```

```c
ulong qsieve_evaluate_candidate(qs_t qs_inf, ulong i, unsigned char *sieve, qs_poly_t poly)
{ // For location \( i \) in sieve array value at which, is greater than sieve threshold, check the value of \( Q(x) \) at position \( i \) for smoothness. If value is found to be smooth then store it for later processing, else check the residue for the partial if it is found to be partial then store it for late processing.
}
```

```c
ulong qsieve_evaluate_sieve(qs_t qs_inf, unsigned char *sieve, qs_poly_t poly)
{ // Scan the sieve array for location at, which accumulated value is greater than sieve threshold.
}
```

```c
void qsieve_collect_relations(qs_t qs_inf, unsigned char *sieve)
{ // Call for initialization of polynomial, sieving, and scanning of sieve for all the possible polynomials for particular hypercube i.e. \( A \).
}
```

```c
void qsieve_write_to_file(qs_t qs_inf, mp_limb_t prime, const fmpz_t Y, const qs_poly_t poly)
{ // Write a relation to the file in a binary format as follows. First, write large prime of size \( \text{sizeof(mp_limb_t)} \), in case of full relation it is 1. After this, write the number of small primes with size \( \text{sizeof(ulong)} \). Then, write the small primes, with a total size of \( \text{number_of_small_primes} \times \text{sizeof(ulong)} \). Then, write the number of factors with a size of \( \text{sizeof(ulong)} \). After that, write the factors and their exponents in the format \( \text{factor}_1, \text{exponent}_1, \text{factor}_2, \ldots \), all with a total size of \( \text{2 \times number_of_factors} \times \text{sizeof(ulong)} \). Then write \( Y \) with the size of \( Y \) first (size \( \text{sizeof(ulong)} \)), that may be negative, and then its limbs (size \( \text{Y_size} \times \text{sizeof(mp_limb_t)} \)).
}
```

```c
hash_t *qsieve_get_table_entry(qs_t qs_inf, mp_limb_t prime)
{ // Return the pointer to the location of ‘prime’ hash table if it exist, else create and entry for it in hash table and return pointer to that.
}
```

```c
void qsieve_add_to_hashtable(qs_t qs_inf, mp_limb_t prime)
{ // Add ‘prime’ to the hast table.
}
```
relation_t q sieve_parse_relation(qs_t qs_inf, char *str)
    Given a string representation of relation from the file, parse it to obtain all the parameters of relation.

relation_t q sieve_merge_relation(qs_t qs_inf, relation_t a, relation_t b)
    Given two partial relation having same large prime, merge them to obtain a full relation.

int q sieve_compare_relation(const void *a, const void *b)
    Compare two relation based on, first large prime, then number of factor and then offsets of factor in factor base.

int q sieve_remove_duplicates(relation_t *rel_list, slong num_relations)
    Remove duplicate from given list of relations by sorting relations in the list.

void q sieve_insert_relation2(qs_t qs_inf, relation_t *rel_list, slong num_relations)
    Given a list of relations, insert each relation from the list into the matrix for further processing.

int q sieve_process_relation(qs_t qs_inf)
    After we have accumulated required number of relations, first process the file by reading all the relations, removes singleton. Then merge all the possible partial to obtain full relations.

void q sieve_factor(fmpz_factor_t factors, const fmpz_t n)
    Factor $n$ using the quadratic sieve method. It is required that $n$ is not a prime and not a perfect power. There is no guarantee that the factors found will be prime, or distinct.
5.1 fmpq.h – rational numbers

The fmpq_t data type represents rational numbers as fractions of multiprecision integers.

An fmpq_t is an array of length 1 of type fmpq, with fmpq being implemented as a pair of fmpz’s representing numerator and denominator.

This format is designed to allow rational numbers with small numerators or denominators to be stored and manipulated efficiently. When components no longer fit in single machine words, the cost of fmpq_t arithmetic is roughly the same as that of mpq_t arithmetic, plus a small amount of overhead.

A fraction is said to be in canonical form if the numerator and denominator have no common factor and the denominator is positive. Except where otherwise noted, all functions in the fmpq module assume that inputs are in canonical form, and produce outputs in canonical form. The user can manipulate the numerator and denominator of an fmpq_t as arbitrary integers, but then becomes responsible for canonicalising the number (for example by calling fmpq_canonicalise) before passing it to any library function.

For most operations, both a function operating on fmpq_t’s and an underscore version operating on fmpz_t components are provided. The underscore functions may perform less error checking, and may impose limitations on aliasing between the input and output variables, but generally assume that the components are in canonical form just like the non-underscore functions.

5.1.1 Types, macros and constants

**type fmpq**

An fmpq is implemented as a struct containing two fmpz’s, one for the numerator, and one for the denominator.

**type fmpq_t**

An array of length 1 of fmpq’s. This is used to pass fmpq’s around by reference without fuss, similar to the way mpq_t’s work.

```
fmpz *fmpq_numref(const fmpq_t x)
fmpz *fmpq_denref(const fmpq_t x)
```

Returns respectively a pointer to the numerator and denominator of x.
5.1.2 Memory management

void fmpq_init(fmpq_t x)
Initialises the fmpq_t variable x for use. Its value is set to 0.

void fmpq_clear(fmpq_t x)
Clears the fmpq_t variable x. To use the variable again, it must be re-initialised with fmpq_init.

5.1.3 Canonicalisation

void fmpq_canonicalise(fmpq_t res)
Puts res in canonical form: the numerator and denominator are reduced to lowest terms, and the denominator is made positive. If the numerator is zero, the denominator is set to one.
If the denominator is zero, the outcome of calling this function is undefined, regardless of the value of the numerator.

void _fmpq_canonicalise(fmpz_t num, fmpz_t den)
Does the same thing as fmpq_canonicalise, but for numerator and denominator given explicitly as fmpz_t variables. Aliasing of num and den is not allowed.

int fmpq_is_canonical(const fmpq_t x)
Returns nonzero if fmpq_t x is in canonical form (as produced by fmpq_canonicalise), and zero otherwise.

int _fmpq_is_canonical(const fmpz_t num, const fmpz_t den)
Does the same thing as fmpq_is_canonical, but for numerator and denominator given explicitly as fmpz_t variables.

5.1.4 Basic assignment

void fmpq_set(fmpq_t dest, const fmpq_t src)
Sets dest to a copy of src. No canonicalisation is performed.

void fmpq_swap(fmpq_t op1, fmpq_t op2)
Swaps the two rational numbers op1 and op2.

void fmpq_neg(fmpq_t dest, const fmpq_t src)
Sets dest to the additive inverse of src.

void fmpq_abs(fmpq_t dest, const fmpq_t src)
Sets dest to the absolute value of src.

void fmpq_zero(fmpq_t res)
Sets the value of res to 0.

void fmpq_one(fmpq_t res)
Sets the value of res to 1.
5.1.5 Comparison

```c
int fmpq_is_zero(const fmpq_t res)
    Returns nonzero if res has value 0, and returns zero otherwise.

int fmpq_is_one(const fmpq_t res)
    Returns nonzero if res has value 1, and returns zero otherwise.

int fmpq_is_pm1(const fmpq_t res)
    Returns nonzero if res has value \pm 1 and zero otherwise.

int fmpq_equal(const fmpq_t x, const fmpq_t y)
int fmpq_equal_fmpz(const fmpq_t x, const fmpz_t y)
int fmpq_equal_si(fmpq_t x, slong y)
int fmpq_equal_ui(const fmpq_t x, ulong y)
    Returns nonzero if x and y are equal, and zero otherwise.

int fmpq_sgn(const fmpq_t x)
    Returns the sign of the rational number x. That is, returns -1 if x < 0, 1 if x > 0 and 0 if x = 0.

int fmpq_cmp(const fmpq_t x, const fmpq_t y)
int fmpq_cmp_fmpz(const fmpq_t x, const fmpz_t y)
int fmpq_cmp_si(const fmpq_t x, slong y)
int fmpq_cmp_ui(const fmpq_t x, ulong y)
    Returns negative if x < y, zero if x = y, and positive if x > y.

void fmpq_height(fmpz_t height, const fmpq_t x)
    Sets height to the height of x, defined as the larger of the absolute values of the numerator and denominator of x.

void flint_bitcnt_t fmpq_height_bits(const fmpq_t x)
    Returns the number of bits in the height of x.
```

5.1.6 Conversion

```c
void fmpq_set_fmpz_frac(fmpq_t res, const fmpz_t p, const fmpz_t q)
    Sets res to the canonical form of the fraction p / q. This is equivalent to assigning the numerator and denominator separately and calling fmpq_canonicalise.

void fmpq_set_mpq(fmpq_t dest, const mpq_t src)
    Sets the value of dest to that of the mpq_t variable src.

void fmpq_set_si(fmpq_t res, slong p, ulong q)
    Sets res to the canonical form of the fraction p / q.

void _fmpq_set_si(fmpz_t rnum, fmpz_t rden, slong p, ulong q)
    Sets (rnum, rden) to the canonical form of the fraction p / q. rnum and rden may not be aliased.

void fmpq_set_ui(fmpq_t res, ulong p, ulong q)
    Sets res to the canonical form of the fraction p / q.

void _fmpq_set_ui(fmpz_t rnum, fmpz_t rden, ulong p, ulong q)
    Sets (rnum, rden) to the canonical form of the fraction p / q. rnum and rden may not be aliased.

void fmpq_set_mpq(fmpq_t dest, const mpq_t src)
    Sets the value of dest to that of the mpq_t variable src.
```
int \texttt{fmpq} \_\texttt{set} \_\texttt{str}(\texttt{fmpq} \_\texttt{t} \texttt{dest}, \texttt{const} \texttt{char} *\texttt{s}, \texttt{int} \texttt{base})

Sets the value of \texttt{dest} to the value represented in the string \texttt{s} in base \texttt{base}.

Returns 0 if no error occurs. Otherwise returns -1 and \texttt{dest} is set to zero.

double \texttt{fmpq} \_\texttt{get} \_\texttt{d}(\texttt{const} \texttt{fmpq} \_\texttt{t} \texttt{f})

Returns \texttt{f} as a double, rounding towards zero if \texttt{f} cannot be represented exactly. The return is system dependent if \texttt{f} is too large or too small to fit in a double.

void \texttt{fmpq} \_\texttt{get} \_\texttt{mpq}(\texttt{mpq} \_\texttt{t} \texttt{dest}, \texttt{const} \texttt{fmpq} \_\texttt{t} \texttt{src})

Sets the value of \texttt{dest}

int \texttt{fmpq} \_\texttt{get} \_\texttt{mpfr}(\texttt{mpfr} \_\texttt{t} \texttt{dest}, \texttt{const} \texttt{fmpq} \_\texttt{t} \texttt{src}, mpfr \_\texttt{rnd} \_\texttt{t} \texttt{rnd})

Sets the MPFR variable \texttt{dest} to the value of \texttt{src}, rounded to the nearest representable binary floating-point value in direction \texttt{rnd}. Returns the sign of the rounding, according to MPFR conventions.

\textbf{Note:} Requires that \texttt{mpfr} \_\texttt{h} has been included before any FLINT header is included.

\texttt{char} *\texttt{fmpq} \_\texttt{get} \_\texttt{str}(\texttt{char} *\texttt{str}, \texttt{int} \texttt{b}, \texttt{const} \texttt{fmpz} \_\texttt{t} \texttt{num}, \texttt{const} \texttt{fmpz} \_\texttt{t} \texttt{den})

\texttt{char} *\texttt{fmpq} \_\texttt{get} \_\texttt{str}(\texttt{char} *\texttt{str}, \texttt{int} \texttt{b}, \texttt{const} \texttt{fmpq} \_\texttt{t} \texttt{x})

Prints the string representation of \texttt{x} in base \texttt{b} \in \{2, 36\} to a suitable buffer.

If \texttt{str} is not \texttt{NULL}, this is used as the buffer and also the return value. If \texttt{str} is \texttt{NULL}, allocates sufficient space and returns a pointer to the string.

void \texttt{flint} \_\texttt{mpq} \_\texttt{init} \_\texttt{set} \_\texttt{readonly}(\texttt{mpq} \_\texttt{t} \texttt{z}, \texttt{const} \texttt{fmpq} \_\texttt{t} \texttt{f})

Sets the uninitialised \texttt{mpq} \_\texttt{t} \texttt{z} to the value of the readonly \texttt{fmpq} \_\texttt{t} \texttt{f}.

Note that it is assumed that \texttt{f} does not change during the lifetime of \texttt{z}.

The rational \texttt{z} has to be cleared by a call to \texttt{flint} \_\texttt{mpq} \_\texttt{clear} \_\texttt{readonly}().

The suggested use of the two functions is as follows:

```
\begin{verbatim}
mpq_t f;
...
{
    mpq_t z;
    flint_mpq_init_set_readonly(z, f);
    foo(..., z);
    flint_mpq_clear_readonly(z);
}
\end{verbatim}
```

This provides a convenient function for user code, only requiring to work with the types \texttt{fmpq} \_\texttt{t} and \texttt{mpq} \_\texttt{t}.

void \texttt{flint} \_\texttt{mpq} \_\texttt{clear} \_\texttt{readonly}(\texttt{mpq} \_\texttt{t} \texttt{z})

Clears the readonly \texttt{mpq} \_\texttt{t} \texttt{z}.

void \texttt{fmpq} \_\texttt{init} \_\texttt{set} \_\texttt{readonly}(\texttt{fmpq} \_\texttt{t} \texttt{f}, \texttt{const} \texttt{mpq} \_\texttt{t} \texttt{z})

Sets the uninitialised \texttt{fmpq} \_\texttt{t} \texttt{f} to a readonly version of the rational \texttt{z}.

Note that the value of \texttt{z} is assumed to remain constant throughout the lifetime of \texttt{f}.

The \texttt{fmpq} \_\texttt{t} \texttt{f} has to be cleared by calling the function \texttt{fmpq} \_\texttt{clear} \_\texttt{readonly}().

The suggested use of the two functions is as follows:

```
\begin{verbatim}
mpq_t z;
...
{
    fmpq_t f;
\end{verbatim}
```

(continues on next page)
void \texttt{fmpq\_clear\_readonly}(\texttt{fmpq\_t} \texttt{f})

Clears the readonly \texttt{fmpq\_t} \texttt{f}.

### 5.1.7 Input and output

\textbf{int} \texttt{fmpq\_fprint}(\texttt{FILE *file}, \texttt{const fmpq\_t x})

Prints \texttt{x} as a fraction to the stream \texttt{file}. The numerator and denominator are printed verbatim as integers, with a forward slash (/) printed in between.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

\textbf{int} \texttt{\_fmpq\_fprint}(\texttt{FILE *file}, \texttt{const fmpz\_t num}, \texttt{const fmpz\_t den})

Does the same thing as \texttt{fmpq\_fprint}, but for numerator and denominator given explicitly as \texttt{fmpz\_t} variables.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

\textbf{int} \texttt{fmpq\_print}(\texttt{const fmpq\_t x})

Prints \texttt{x} as a fraction. The numerator and denominator are printed verbatim as integers, with a forward slash (/) printed in between.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

\textbf{int} \texttt{\_fmpq\_print}(\texttt{const fmpz\_t num}, \texttt{const fmpz\_t den})

Does the same thing as \texttt{fmpq\_print}, but for numerator and denominator given explicitly as \texttt{fmpz\_t} variables.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

### 5.1.8 Random number generation

\textbf{void} \texttt{fmpq\_randtest}(\texttt{fmpq\_t res}, \texttt{flint\_rand\_t state}, \texttt{flint\_bitcnt\_t bits})

Sets \texttt{res} to a random value, with numerator and denominator having up to \texttt{bits} bits. The resulting fraction will be in canonical form. This function has an increased probability of generating special values which are likely to trigger corner cases.

\textbf{void} \texttt{\_fmpq\_randtest}(\texttt{fmpz\_t num}, \texttt{fmpz\_t den}, \texttt{flint\_rand\_t state}, \texttt{flint\_bitcnt\_t bits})

Does the same thing as \texttt{fmpq\_randtest}, but for numerator and denominator given explicitly as \texttt{fmpz\_t} variables. Aliasing of \texttt{num} and \texttt{den} is not allowed.

\textbf{void} \texttt{fmpq\_randtest\_not\_zero}(\texttt{fmpq\_t res}, \texttt{flint\_rand\_t state}, \texttt{flint\_bitcnt\_t bits})

As per \texttt{fmpq\_randtest}, but the result will not be 0. If \texttt{bits} is set to 0, an exception will result.

\textbf{void} \texttt{fmpq\_randbits}(\texttt{fmpq\_t res}, \texttt{flint\_rand\_t state}, \texttt{flint\_bitcnt\_t bits})

Sets \texttt{res} to a random value, with numerator and denominator both having exactly \texttt{bits} bits before canonicalisation, and then puts \texttt{res} in canonical form. Note that as a result of the canonicalisation, the resulting numerator and denominator can be slightly smaller than \texttt{bits} bits.

\textbf{void} \texttt{\_fmpq\_randbits}(\texttt{fmpz\_t num}, \texttt{fmpz\_t den}, \texttt{flint\_rand\_t state}, \texttt{flint\_bitcnt\_t bits})

Does the same thing as \texttt{fmpq\_randbits}, but for numerator and denominator given explicitly as \texttt{fmpz\_t} variables. Aliasing of \texttt{num} and \texttt{den} is not allowed.
5.1.9 Arithmetic

void fmpq_add(fmpz_t res, const fmpq_t op1, const fmpq_t op2)
void fmpq_sub(fmpz_t res, const fmpq_t op1, const fmpq_t op2)
void fmpq_mul(fmpz_t res, const fmpq_t op1, const fmpq_t op2)
void fmpq_div(fmpz_t res, const fmpq_t op1, const fmpq_t op2)

Sets res respectively to op1 + op2, op1 - op2, op1 * op2, or op1 / op2. Division by zero results in an error. Aliasing between any combination of the variables is allowed.

void _fmpq_add(fmpz_t rnum, fmpz_t rden, const fmpz_t op1num, const fmpz_t op1den, const fmpz_t op2num, const fmpz_t op2den)
void _fmpq_sub(fmpz_t rnum, fmpz_t rden, const fmpz_t op1num, const fmpz_t op1den, const fmpz_t op2num, const fmpz_t op2den)
void _fmpq_mul(fmpz_t rnum, fmpz_t rden, const fmpz_t op1num, const fmpz_t op1den, const fmpz_t op2num, const fmpz_t op2den)
void _fmpq_div(fmpz_t rnum, fmpz_t rden, const fmpz_t op1num, const fmpz_t op1den, const fmpz_t op2num, const fmpz_t op2den)

Sets (rnum, rden) to the canonical form of the sum, difference, product or quotient respectively of the fractions represented by (op1num, op1den) and (op2num, op2den). Aliasing between any combination of the variables is allowed, whilst no numerator is aliased with a denominator.

void _fmpq_add_si(fmpz_t rnum, fmpz_t rden, const fmpz_t p, const fmpz_t q, slong r)
void _fmpq_sub_si(fmpz_t rnum, fmpz_t rden, const fmpz_t p, const fmpz_t q, slong r)
void _fmpq_sub_ui(fmpz_t rnum, fmpz_t rden, const fmpz_t p, const fmpz_t q, ulong r)
void _fmpq_add_fmpz(fmpz_t rnum, fmpz_t rden, const fmpz_t p, const fmpz_t q, const fmpz_t r)
void _fmpq_sub_fmpz(fmpz_t rnum, fmpz_t rden, const fmpz_t p, const fmpz_t q, const fmpz_t r)

Sets (rnum, rden) to the canonical form of the sum or difference respectively of the fractions represented by (p, q) and (r, 1). Numerators may not be aliased with denominators.

void fmpq_add_si(fmpz_t res, const fmpq_t op1, slong c)
void fmpq_sub_si(fmpz_t res, const fmpq_t op1, slong c)
void fmpq_sub_ui(fmpz_t res, const fmpq_t op1, ulong c)
void fmpq_add_fmpz(fmpz_t res, const fmpq_t op1, const fmpz_t c)
void fmpq_sub_fmpz(fmpz_t res, const fmpq_t op1, const fmpz_t c)

Sets res to the sum or difference respectively of the fraction op1 and the integer c.

void fmpq_mul_si(fmpz_t res, const fmpq_t op1, const fmpz_t q, slong r)

Sets (rnum, rden) to the product of (p, q) and the integer r.

void fmpq_mul_si(fmpz_t res, const fmpq_t op1, slong c)

Sets res to the product of op1 and the integer c.

void _fmpq_mul_ui(fmpz_t rnum, fmpz_t rden, const fmpz_t p, const fmpz_t q, ulong r)

Sets (rnum, rden) to the product of (p, q) and the integer r.

void _fmpq_mul_ui(fmpz_t res, const fmpq_t op1, ulong c)

Sets res to the product of op1 and the integer c.

void *fmpq_addmul(fmpz_t res, const fmpq_t op1, const fmpq_t op2)
void *fmpq_submul(fmpz_t res, const fmpq_t op1, const fmpq_t op2)

Sets res to res + op1 * op2 or res - op1 * op2, respectively. Aliasing between any combination of the variables is allowed.
void _fmpz_submul(fmpz_t rnum, fmpz_t rden, const fmpz_t op1num, const fmpz_t op1den, const fmpz_t op2num, const fmpz_t op2den)

Sets \((r\text{num}, r\text{den})\) to the canonical form of the fraction \((r\text{num}, r\text{den}) + (o\text{p1num}, o\text{p1den})\) \(* (o\text{p2num}, o\text{p2den})\) or \((r\text{num}, r\text{den}) - (o\text{p1num}, o\text{p1den})\) \(* (o\text{p2num}, o\text{p2den})\) respectively. Aliasing between any combination of the variables is allowed, whilst no numerator is aliased with a denominator.

void fmpq_inv(fmpz_t dest, const fmpz_t src)
Sets dest to \(1 / \text{src}\).

void _fmpz_pow_si(fmpz_t rnum, fmpz_t rden, const fmpz_t opnum, const fmpz_t opden, slong e)
void fmpq_pow_si(fmpz_t res, const fmpz_t op, slong e)

Sets res to \(\text{op}^{e}\) raised to the power \(e\), where \(e\) is a slong. If \(e\) is 0 and \(\text{op}\) is 0, then res will be set to 1.

int fmpq_pow_fmpz(fmpz_t a, const fmpz_t b, const fmpz_t e)

Sets res to \(a^{e}\) raised to the power \(e\). Return 1 for success and 0 for failure.

void fmpq_mul_fmpz(fmpz_t res, const fmpz_t op, const fmpz_t x)

Sets res to the product of the rational number \(\text{op}\) and the integer \(x\).

void fmpq_div_fmpz(fmpz_t res, const fmpz_t op, const fmpz_t x)

Sets res to the quotient of the rational number \(\text{op}\) and the integer \(x\).

void fmpq_mul_2exp(fmpz_t res, const fmpz_t x, flint_bitcnt_t exp)

Sets res to \(x\) multiplied by \(2^{\text{exp}}\).

void fmpq_div_2exp(fmpz_t res, const fmpz_t x, flint_bitcnt_t exp)

Sets res to \(x\) divided by \(2^{\text{exp}}\).

void _fmpq_gcd(fmpz_t rnum, fmpz_t rden, const fmpz_t p, const fmpz_t q, const fmpz_t r, const fmpz_t s)

Set \((r\text{num}, r\text{den})\) to the gcd of \((p, q)\) and \((r, s)\) which we define to be the canonicalisation of \(\gcd(pq, rs)/(qs)\). Does not assume that \((r\text{num}, r\text{den})\), \((p, q)\) or \((r, s)\) are canonical. (This is apparently Euclid’s original definition and is stable under scaling of numerator and denominator. It also agrees with the gcd on the integers. Note that it does not agree with gcd as defined in \text{fmpq}._\text{poly}.) This definition agrees with the result as output by Sage and Pari/GP.

void fmpq_gcd(fmpz_t res, const fmpz_t op1, const fmpz_t op2)

Set res to the gcd of \(\text{op1}\) and \(\text{op2}\). See the low level function _fmpq_gcd for our definition of gcd.

void _fmpq_gcd_cofactors(fmpz_t gnum, fmpz_t gden, fmpz_t abar, fmpz_t bbar, const fmpz_t anum, const fmpz_t aden, const fmpz_t bnum, const fmpz_t bden)

void fmpq_gcd_cofactors(fmpz_t g, fmpz_t abar, fmpz_t bbar, const fmpz_t a, const fmpz_t b)

Set \(g\) to \(\gcd(a, b)\) as per \text{fmpq}._\text{gcd()} and also compute \(\overline{a} = a / g\) and \(\overline{b} = b / g\). Unlike _fmpq_gcd(), _fmpq_gcd_cofactors() requires canonical inputs.

void _fmpq_add_small(fmpz_t rnum, fmpz_t rden, slong p1, ulong q1, slong p2, ulong q2)

Sets \((r\text{num}, r\text{den})\) to the sum of \((p1, q1)\) and \((p2, q2)\). Assumes that \((p1, q1)\) and \((p2, q2)\) are in canonical form and that all inputs are between \text{COEFF_MIN} and \text{COEFF_MAX}.

void _fmpq_mul_small(fmpz_t rnum, fmpz_t rden, slong p1, ulong q1, slong p2, ulong q2)

Sets \((r\text{num}, r\text{den})\) to the product of \((p1, q1)\) and \((p2, q2)\). Assumes that \((p1, q1)\) and \((p2, q2)\) are in canonical form and that all inputs are between \text{COEFF_MIN} and \text{COEFF_MAX}.
5.1.10 Modular reduction and rational reconstruction

```c
int _fmpq_mod_fmpz(fmpq_t res, const fmpz_t num, const fmpz_t den, const fmpz_t mod)
int fmpq_mod_fmpz(fmpz_t x, const fmpz_t den, const fmpz_t mod)
```

Sets the integer res to the residue $a$ of $x = n/d = (\text{num}, \text{den})$ modulo the positive integer $m = \text{mod}$, defined as the 0 ≤ $a < m$ satisfying $n \equiv ad \pmod{m}$. If such an $a$ exists, 1 will be returned, otherwise 0 will be returned.

```c
int _fmpq_reconstruct_fmpz_2_naive(fmpq_t n, fmpz_t d, const fmpz_t a, const fmpz_t m, const fmpz_t N, const fmpz_t D)
int _fmpq_reconstruct_fmpz_2(fmpq_t n, fmpz_t d, const fmpz_t a, const fmpz_t m, const fmpz_t N, const fmpz_t D)
int fmpq_reconstruct_fmpz_2(fmpq_t res, const fmpz_t a, const fmpz_t m, const fmpz_t N, const fmpz_t D)
```

Reconstructs a rational number from its residue $a$ modulo $m$.

Given a modulus $m > 2$, a residue $0 \leq a < m$, and positive $N, D$ satisfying $2ND < m$, this function attempts to find a fraction $n/d$ with $0 \leq |n| \leq N$ and $0 < d \leq D$ such that $\gcd(n, d) = 1$ and $n \equiv ad \pmod{m}$. If a solution exists, then it is also unique. The function returns 1 if successful, and 0 to indicate that no solution exists.

```c
int _fmpq_reconstruct_fmpz(fmpq_t n, fmpz_t d, const fmpz_t a, const fmpz_t m)
int fmpq_reconstruct_fmpz(fmpq_t res, const fmpz_t a, const fmpz_t m)
```

Reconstructs a rational number from its residue $a$ modulo $m$, returning 1 if successful and 0 if no solution exists. Uses the balanced bounds $N = D = \lfloor \sqrt{m-1} \rfloor$.

5.1.11 Rational enumeration

```c
void _fmpq_next_minimal(fmpq_t rnum, fmpz_t rden, const fmpz_t num, const fmpz_t den)
void fmpq_next_minimal(fmpq_t res, const fmpz_t x)
```

Given $x = \text{num}/\text{den}$, assumed to be nonnegative and in canonical form, sets res to the next rational number in the sequence obtained by enumerating all positive denominators $q$, for each $q$ enumerating the numerators $1 \leq p < q$ in order and generating both $p/q$ and $q/p$, but skipping all $\gcd(p, q) \neq 1$. Starting with zero, this generates every nonnegative rational number once and only once, with the first few entries being:

0, 1, 1/2, 2, 1/3, 3, 2/3, 3/2, 1/4, 4, 3/4, 4/3, 1/5, 5, 2/5, ....

This enumeration produces the rational numbers in order of minimal height. It has the disadvantage of being somewhat slower to compute than the Calkin-Wilf enumeration.

```c
void _fmpq_next_signed_minimal(fmpq_t rnum, fmpz_t rden, const fmpz_t num, const fmpz_t den)
void fmpq_next_signed_minimal(fmpq_t res, const fmpz_t x)
```

Given a signed rational number $x = \text{num}/\text{den}$, assumed to be in canonical form, sets res to the next element in the minimal-height sequence generated by fmpq_next_minimal but with negative numbers interleaved:

0, 1, −1, 1/2, −1/2, 2, −2, 1/3, −1/3, ....

Starting with zero, this generates every rational number once and only once, in order of minimal height.

```c
void _fmpq_next_calkin_wilf(fmpq_t rnum, fmpz_t rden, const fmpz_t num, const fmpz_t den)
void fmpq_next_calkin_wilf(fmpq_t res, const fmpz_t x)
```

Given $x = \text{num}/\text{den}$, which is assumed to be nonnegative and in canonical form, sets res to the next number in the breadth-first traversal of the Calkin-Wilf tree. Starting with zero, this generates every nonnegative rational number once and only once, with the first few entries being:
0, 1/2, 2, 1/3, 3/2, 3, 1/4, 4/3, 3/5, 5/2, 2/5, ... 

Despite the appearance of the initial entries, the Calkin-Wilf enumeration does not produce the rational numbers in order of height: some small fractions will appear late in the sequence. This order has the advantage of being faster to produce than the minimal-height order.

```c
void _fmpq_next_signed_calkin_wilf(fmpz_t rnum, fmpz_t rden, const fmpq_t num, const fmpq_t den)
void fmpq_next_signed_calkin_wilf(fmpq_t res, const fmpq_t x)
```

Given a signed rational number \( x = \frac{\text{num}}{\text{den}} \), assumed to be in canonical form, sets \( \text{res} \) to the next element in the Calkin-Wilf sequence with negative numbers interleaved:

\[
0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, ...
\]

Starting with zero, this generates every rational number once and only once, but not in order of minimal height.

```c
void fmpq_farey_neighbors(fmpq_t l, fmpq_t r, const fmpq_t x, const fmpq_t Q)
void _fmpq_simplest_between(fmpz_t x_num, fmpz_t x_den, const fmpq_t l_num, const fmpq_t l_den, const fmpq_t r_num, const fmpq_t r_den)
void fmpq_simplest_between(fmpq_t x, const fmpq_t l, const fmpq_t r)
```

Set \( x \) to the simplest fraction in the closed interval \([l, r]\). The underscore version makes the additional assumption that \( l \leq r \). The endpoints \( l \) and \( r \) do not need to be canonical, but their denominators do need to be positive. \( x \) will always be returned in canonical form. A canonical fraction \( a/b \) is defined to be simpler than \( a_2/b_2 \) iff \( b_1 < b_2 \) or \( b_1 = b_2 \) and \( a_1 < a_2 \).

### 5.1.12 Continued fractions

```c
slong fmpq_get_cfrac(fmpz *c, fmpq_t rem, const fmpq_t x, slong n)
slong fmpq_get_cfrac_naive(fmpz *c, fmpq_t rem, const fmpq_t x, slong n)
```

Generates up to \( n \) terms of the (simple) continued fraction expansion of \( x \), writing the coefficients to the vector \( c \) and the remainder \( r \) to the \( \text{rem} \) variable. The return value is the number \( k \) of generated terms. The output satisfies

\[
x = c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{\ddots + \cfrac{1}{c_{k-1} + r}}}}
\]

If \( r \) is zero, the continued fraction expansion is complete. If \( r \) is nonzero, \( 1/r \) can be passed back as input to generate \( c_k, c_{k+1}, \ldots \). Calls to \texttt{fmpq_get_cfrac} can therefore be chained to generate the continued fraction incrementally, extracting any desired number of coefficients at a time.

In general, a rational number has exactly two continued fraction expansions. By convention, we generate the shorter one. The longer expansion can be obtained by replacing the last coefficient \( a_{k-1} \) by the pair of coefficients \( a_{k-1} - 1, 1 \).

The behavior of this function in corner cases is as follows:

- if \( x \) is infinite (anything over 0), \( \text{rem} \) will be zero and the return is \( k = 0 \) regardless of \( n \).
- else (if \( x \) is finite),
  - if \( n \leq 0 \), \( \text{rem} \) will be \( 1/x \) (allowing for infinite in the case \( x = 0 \)) and the return is \( k = 0 \)
  - else if \( n > 0 \), \( \text{rem} \) will finite and the return is \( 0 < k \leq n \).

### 5.1. \texttt{fmpz.}h – rational numbers
Essentially, if this function is called with canonical $x$ and $n > 0$, then $\text{rem}$ will be canonical. Therefore, applications relying on canonical $\text{fmpq_t}$’s should not call this function with $n \leq 0$.

void $\text{fmpq_set_cfrac}(\text{fmpq_t} \, x, \text{const fmpz *} \, c, \text{slong} \, n)$

Sets $x$ to the value of the continued fraction

$$ x = c_0 + \frac{1}{c_1 + \frac{1}{c_2 + \cdots + \frac{1}{c_{n-1}}}} $$

where all $c_i$ except $c_0$ should be nonnegative. It is assumed that $n > 0$.

For large $n$, this function implements a subquadratic algorithm. The convergents are given by a chain product of 2 by 2 matrices. This product is split in half recursively to balance the size of the coefficients.

$\text{slong fmpq_cfrac_bound(} \text{const fmpq_t} \, x \text{)}$

Returns an upper bound for the number of terms in the continued fraction expansion of $x$. The computed bound is not necessarily sharp.

We use the fact that the smallest denominator that can give a continued fraction of length $n$ is the Fibonacci number $F_{n+1}$.

5.1.13 Special functions

void $\_\text{fmpq_harmonic_ui}(\text{fmpz_t} \, \text{num}, \text{fmpz_t} \, \text{den}, \text{ulong} \, n)$
void $\text{fmpq_harmonic_ui}(\text{fmpq_t} \, x, \text{ulong} \, n)$

Computes the harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$. Table lookup is used for $H_n$ whose numerator and denominator fit in single limb. For larger $n$, a divide and conquer strategy is used.

5.1.14 Dedekind sums

Most of the definitions and relations used in the following section are given by Apostol [Apostol1997]. The Dedekind sum $s(h, k)$ is defined for all integers $h$ and $k$ as

$$ s(h, k) = \sum_{i=1}^{k-1} \left( \left( \frac{i}{k} \right) \left( \frac{hi}{k} \right) \right) $$

where

$$ ((x)) = \begin{cases} x - \lfloor x \rfloor - 1/2 & \text{if } x \in \mathbb{Q} \setminus \mathbb{Z} \\ 0 & \text{if } x \in \mathbb{Z}. \end{cases} $$

If $0 < h < k$ and $(h, k) = 1$, this reduces to

$$ s(h, k) = \sum_{i=1}^{k-1} \frac{h}{k} \left( \frac{hi}{k} - \left\lfloor \frac{hi}{k} \right\rfloor - \frac{1}{2} \right). $$

The main formula for evaluating the series above is the following. Letting $r_0 = k$, $r_1 = h$, $r_2, r_3, \ldots, r_n, r_{n+1} = 1$ be the remainder sequence in the Euclidean algorithm for computing GCD of $h$ and $k$,

$$ s(h, k) = \frac{1 - (-1)^n}{8} - \frac{1}{12} \sum_{i=1}^{n+1} (-1)^{i-1} \frac{1 + r_i^2 + r_{i-1}^2}{r_i r_{i-1}}. $$

Writing $s(h, k) = p/q$, some useful properties employed are $|s| < k/12$, $q \mid 6k$ and $2|p| < k^2$. 

Chapter 5. Rational numbers
void fmpq_dedekind_sum(fmpq_t s, const fmpz_t h, const fmpz_t k)
void fmpq_dedekind_sum_naive(fmpq_t s, const fmpz_t h, const fmpz_t k)

Computes \( s(h, k) \) for arbitrary \( h \) and \( k \). The naive version uses a straightforward implementation of the defining sum using fmpz arithmetic and is slow for large \( k \).

5.2 fmpq_vec.h – vectors over rational numbers

5.2.1 Memory management

fmpq * fmpq_vec_init(slong n)

Initialises a vector of fmpq values of length \( n \) and sets all values to 0. This is equivalent to generating a fmpz vector of length \( 2n \) with _fmpz_vec_init and setting all denominators to 1.

void _fmpq_vec_clear(fmpq *vec, slong n)

Frees an fmpq vector.

5.2.2 Randomisation

void _fmpq_vec_randtest(fmpq *f, flint_rand_t state, slong len, flint_bitcnt_t bits)

Sets the entries of a vector of the given length to random rationals with numerator and denominator having up to the given number of bits per entry.

void _fmpq_vec_randtest_uniq_sorted(fmpq *vec, flint_rand_t state, slong len, flint_bitcnt_t bits)

Sets the entries of a vector of the given length to random distinct rationals with numerator and denominator having up to the given number of bits per entry. The entries in the vector are sorted.

5.2.3 Sorting

void _fmpq_vec_sort(fmpq *vec, slong len)

Sorts the entries of (vec, len).

5.2.4 Conversions

void _fmpq_vec_set_fmpz_vec(fmpq *res, const fmpz *vec, slong len)

Sets (res, len) to (vec, len).

void _fmpq_vec_get_fmpz_vec_fmpz(fmpz *num, fmpz_t den, const fmpq *a, slong len)

Find a common denominator den of the entries of a and set (num, len) to the corresponding numerators.

5.2.5 Dot product

void _fmpq_vec_dot(fmpq_t res, const fmpq *vec1, const fmpq *vec2, slong len)

Sets res to the dot product of the vectors (vec1, len) and (vec2, len).
5.2.6 Input and output

int _fmpq_vec_fprint(FILE *file, const fmpq *vec, slong len)
    Prints the vector of given length to the stream file. The format is the length followed by two
    spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fmpq_vec_print(const fmpq *vec, slong len)
    Prints the vector of given length to stdout.
    For further details, see _fmpq_vec_fprint() .

5.3 fmpq_mat.h – matrices over the rational numbers

The fmpq_mat_t data type represents matrices over \( \mathbb{Q} \).

A rational matrix is stored as an array of fmpq elements in order to allow convenient and efficient
manipulation of individual entries. In general, fmpq_mat functions assume that input entries are in
canonical form, and produce output with entries in canonical form.

Since rational arithmetic is expensive, computations are typically performed by clearing denominators,
performing the heavy work over the integers, and converting the final result back to a rational matrix.
The fmpq_mat functions take care of such conversions transparently. For users who need fine-grained
control, various functions for conversion between rational and integer matrices are provided.

5.3.1 Types, macros and constants

type fmpq_mat_struct

type fmpq_mat_t

5.3.2 Memory management

void fmpq_mat_init(fmpq_mat_t mat, slong rows, slong cols)
    Initialises a matrix with the given number of rows and columns for use.

void fmpq_mat_init_set(fmpq_mat_t mat1, const fmpq_mat_t mat2)
    Initialises mat1 and sets it equal to mat2.

void fmpq_mat_clear(fmpq_mat_t mat)
    Frees all memory associated with the matrix. The matrix must be reinitialised if it is to be used
    again.

void fmpq_mat_swap(fmpq_mat_t mat1, fmpq_mat_t mat2)
    Swaps two matrices. The dimensions of mat1 and mat2 are allowed to be different.

void fmpq_mat_swap_entrywise(fmpq_mat_t mat1, fmpq_mat_t mat2)
    Swaps two matrices by swapping the individual entries rather than swapping the contents of the
    structs.
5.3.3 Entry access

```c
fmpq *fmpq_mat_entry(const fmpq_mat_t mat, slong i, slong j)
```
Gives a reference to the entry at row \( i \) and column \( j \). The reference can be passed as an input or output variable to any `fmpq` function for direct manipulation of the matrix element. No bounds checking is performed.

```c
fmpz *fmpq_mat_entry_num(const fmpq_mat_t mat, slong i, slong j)
```
Gives a reference to the numerator of the entry at row \( i \) and column \( j \). The reference can be passed as an input or output variable to any `fmpz` function for direct manipulation of the matrix element. No bounds checking is performed.

```c
fmpz *fmpq_mat_entry_den(const fmpq_mat_t mat, slong i, slong j)
```
Gives a reference to the denominator of the entry at row \( i \) and column \( j \). The reference can be passed as an input or output variable to any `fmpz` function for direct manipulation of the matrix element. No bounds checking is performed.

```c
slong fmpq_mat_nrows(const fmpq_mat_t mat)
```
Return the number of rows of the matrix \( \text{mat} \).

```c
slong fmpq_mat_ncols(const fmpq_mat_t mat)
```
Return the number of columns of the matrix \( \text{mat} \).

5.3.4 Basic assignment

```c
void fmpq_mat_set(fmpq_mat_t dest, const fmpq_mat_t src)
```
Sets the entries in \( \text{dest} \) to the same values as in \( \text{src} \), assuming the two matrices have the same dimensions.

```c
void fmpq_mat_zero(fmpq_mat_t mat)
```
Sets \( \text{mat} \) to the zero matrix.

```c
void fmpq_mat_one(fmpq_mat_t mat)
```
Let \( m \) be the minimum of the number of rows and columns in the matrix \( \text{mat} \). This function sets the first \( m \times m \) block to the identity matrix, and the remaining block to zero.

```c
void fmpq_mat_transpose(fmpq_mat_t rop, const fmpq_mat_t op)
```
Sets the matrix \( \text{rop} \) to the transpose of the matrix \( \text{op} \), assuming that their dimensions are compatible.

```c
void fmpq_mat_swap_rows(fmpq_mat_t mat, slong *perm, slong r, slong s)
```
Swaps rows \( r \) and \( s \) of \( \text{mat} \). If \( \text{perm} \) is non-NULL, the permutation of the rows will also be applied to \( \text{perm} \).

```c
void fmpq_mat_swap_cols(fmpq_mat_t mat, slong *perm, slong r, slong s)
```
Swaps columns \( r \) and \( s \) of \( \text{mat} \). If \( \text{perm} \) is non-NULL, the permutation of the columns will also be applied to \( \text{perm} \).

```c
void fmpq_mat_invert_rows(fmpq_mat_t mat, slong *perm)
```
Swaps rows \( i \) and \( r - i \) of \( \text{mat} \) for \( 0 \leq i < r/2 \), where \( r \) is the number of rows of \( \text{mat} \). If \( \text{perm} \) is non-NULL, the permutation of the rows will also be applied to \( \text{perm} \).

```c
void fmpq_mat_invert_cols(fmpq_mat_t mat, slong *perm)
```
Swaps columns \( i \) and \( c - i \) of \( \text{mat} \) for \( 0 \leq i < c/2 \), where \( c \) is the number of columns of \( \text{mat} \). If \( \text{perm} \) is non-NULL, the permutation of the columns will also be applied to \( \text{perm} \).
5.3.5 Addition, scalar multiplication

```c
void fmpq_mat_add(fmpq_mat_t mat, const fmpq_mat_t mat1, const fmpq_mat_t mat2)
Sets mat to the sum of mat1 and mat2, assuming that all three matrices have the same dimensions.
```

```c
void fmpq_mat_sub(fmpq_mat_t mat, const fmpq_mat_t mat1, const fmpq_mat_t mat2)
Sets mat to the difference of mat1 and mat2, assuming that all three matrices have the same dimensions.
```

```c
void fmpq_mat_neg(fmpq_mat_t rop, const fmpq_mat_t op)
Sets rop to the negative of op, assuming that the two matrices have the same dimensions.
```

```c
void fmpq_mat_scalar_mul_fmpq(fmpq_mat_t rop, const fmpq_mat_t op, const fmpq_t x)
Sets rop to op multiplied by the rational x, assuming that the two matrices have the same dimensions.
Note that the rational x may not be aliased with any part of the entries of rop.
```

```c
void fmpq_mat_scalar_mul_fmpz(fmpq_mat_t rop, const fmpq_mat_t op, const fmpz_t x)
Sets rop to op multiplied by the integer x, assuming that the two matrices have the same dimensions.
Note that the integer x may not be aliased with any part of the entries of rop.
```

```c
void fmpq_mat_scalar_div_fmpz(fmpq_mat_t rop, const fmpq_mat_t op, const fmpz_t x)
Sets rop to op divided by the integer x, assuming that the two matrices have the same dimensions and that x is non-zero.
Note that the integer x may not be aliased with any part of the entries of rop.
```

5.3.6 Input and output

```c
void fmpq_mat_print(const fmpq_mat_t mat)
Prints the matrix mat to standard output.
```

5.3.7 Random matrix generation

```c
void fmpq_mat_randbits(fmpq_mat_t mat, flint_rand_t state, flint_bitcnt_t bits)
This is equivalent to applying fmpq_randbits to all entries in the matrix.
```

```c
void fmpq_mat_randtest(fmpq_mat_t mat, flint_rand_t state, flint_bitcnt_t bits)
This is equivalent to applying fmpq_randtest to all entries in the matrix.
```

5.3.8 Window

```c
void fmpq_mat_window_init(fmpq_mat_t window, const fmpq_mat_t mat, slong r1, slong c1, slong r2, slong c2)
Initializes the matrix window to be an r2 - r1 by c2 - c1 submatrix of mat whose (0,0) entry is the (r1, c1) entry of mat. The memory for the elements of window is shared with mat.
```

```c
void fmpq_mat_window_clear(fmpq_mat_t window)
Clears the matrix window and releases any memory that it uses. Note that the memory to the underlying matrix that window points to is not freed.
```
5.3.9 Concatenate

void \texttt{fmpq\_mat\_concat\_vertical}(\texttt{fmpq\_mat\_t res, const fmpq\_mat\_t mat1, const fmpq\_mat\_t mat2})

Sets \texttt{res} to vertical concatenation of \texttt{(mat1, mat2)} in that order. Matrix dimensions: \texttt{mat1}: \(m \times n\), \texttt{mat2}: \(k \times n\), \texttt{res}: \((m + k) \times n\).

void \texttt{fmpq\_mat\_concat\_horizontal}(\texttt{fmpq\_mat\_t res, const fmpq\_mat\_t mat1, const fmpq\_mat\_t mat2})

Sets \texttt{res} to horizontal concatenation of \texttt{(mat1, mat2)} in that order. Matrix dimensions: \texttt{mat1}: \(m \times n\), \texttt{mat2}: \(m \times k\), \texttt{res}: \(m \times (n + k)\).

5.3.10 Special matrices

void \texttt{fmpq\_mat\_hilbert\_matrix}(\texttt{fmpq\_mat\_t mat})

Sets \texttt{mat} to a Hilbert matrix of the given size. That is, the entry at row \(i\) and column \(j\) is set to \(1/(i + j + 1)\).

5.3.11 Basic comparison and properties

int \texttt{fmpq\_mat\_equal}(\texttt{const fmpq\_mat\_t mat1, const fmpq\_mat\_t mat2})

Returns nonzero if \texttt{mat1} and \texttt{mat2} have the same shape and all their entries agree, and returns zero otherwise. Assumes the entries in both \texttt{mat1} and \texttt{mat2} are in canonical form.

int \texttt{fmpq\_mat\_is\_integral}(\texttt{const fmpq\_mat\_t mat})

Returns nonzero if all entries in \texttt{mat} are integer-valued, and returns zero otherwise. Assumes that the entries in \texttt{mat} are in canonical form.

int \texttt{fmpq\_mat\_is\_zero}(\texttt{const fmpq\_mat\_t mat})

Returns nonzero if all entries in \texttt{mat} are zero, and returns zero otherwise.

int \texttt{fmpq\_mat\_is\_one}(\texttt{const fmpq\_mat\_t mat})

Returns nonzero if \texttt{mat} ones along the diagonal and zeros elsewhere, and returns zero otherwise.

int \texttt{fmpq\_mat\_is\_empty}(\texttt{const fmpq\_mat\_t mat})

Returns a non-zero value if the number of rows or the number of columns in \texttt{mat} is zero, and otherwise returns zero.

int \texttt{fmpq\_mat\_is\_square}(\texttt{const fmpq\_mat\_t mat})

Returns a non-zero value if the number of rows is equal to the number of columns in \texttt{mat}, and otherwise returns zero.

5.3.12 Integer matrix conversion

int \texttt{fmpq\_mat\_get\_fmpz\_mat}(\texttt{fmpz\_mat\_t dest, const fmpq\_mat\_t mat})

Sets \texttt{dest} to \texttt{mat} and returns nonzero if all entries in \texttt{mat} are integer-valued. If not all entries in \texttt{mat} are integer-valued, sets \texttt{dest} to an undefined matrix and returns zero. Assumes that the entries in \texttt{mat} are in canonical form.

void \texttt{fmpq\_mat\_get\_fmpz\_mat\_entrywise}(\texttt{fmpz\_mat\_t num, fmpz\_mat\_t den, const fmpq\_mat\_t mat})

Sets the integer matrices \texttt{num} and \texttt{den} respectively to the numerators and denominators of the entries in \texttt{mat}. 

5.3. \texttt{fmpq\_mat.h} – matrices over the rational numbers 285
void fmpq_mat_get_fmpz_mat_matwise(fmpz_mat_t num, fmpz_t den, const fmpq_mat_t mat)

Converts all entries in mat to a common denominator, storing the rescaled numerators in num and the denominator in den. The denominator will be minimal if the entries in mat are in canonical form.

void fmpq_mat_get_fmpz_mat_rowwise(fmpz_mat_t num, fmpz *den, const fmpq_mat_t mat)

Clears denominators in mat row by row. The rescaled numerators are written to num, and the denominator of row i is written to position i in den which can be a preinitialised fmpz vector. Alternatively, NULL can be passed as the den variable, in which case the denominators will not be stored.

void fmpq_mat_get_fmpz_mat_rowwise_2(fmpz_mat_t num, fmpz_mat_t num2, fmpz *den, const fmpq_mat_t mat, const fmpq_mat_t mat2)

Clears denominators row by row of both mat and mat2, writing the respective numerators to num and num2. This is equivalent to concatenating mat and mat2 horizontally, calling fmpq_mat_get_fmpz_mat_rowwise, and extracting the two submatrices in the result.

void fmpq_mat_get_fmpz_mat_colwise(fmpz_mat_t num, fmpz *den, const fmpq_mat_t mat)

Clears denominators in mat column by column. The rescaled numerators are written to num, and the denominator of column i is written to position i in den which can be a preinitialised fmpz vector. Alternatively, NULL can be passed as the den variable, in which case the denominators will not be stored.

void fmpq_mat_set_fmpz_mat(fmpq_mat_t dest, const fmpz_mat_t src)

Sets dest to src.

void fmpq_mat_set_fmpz_mat_div_fmpz(fmpq_mat_t mat, const fmpz_mat_t num, const fmpz_t den)

Sets mat to the integer matrix num divided by the common denominator den.

5.3.13 Modular reduction and rational reconstruction

void fmpq_mat_get_fmpz_mat_mod_fmpz(fmpz_mat_t dest, const fmpq_mat_t mat, const fmpz_t mod)

Sets each entry in dest to the corresponding entry in mat, reduced modulo mod.

int fmpq_mat_set_fmpz_mat_mod_fmpz(fmpq_mat_t X, const fmpz_mat_t Xmod, const fmpz_t mod)

Sets X to the entrywise rational reconstruction integer matrix Xmod modulo mod, and returns nonzero if the reconstruction is successful. If rational reconstruction fails for any element, returns zero and sets the entries in X to undefined values.

5.3.14 Matrix multiplication

void fmpq_mat_mul_direct(fmpq_mat_t C, const fmpq_mat_t A, const fmpq_mat_t B)

Sets C to the matrix product AB, computed naively using rational arithmetic. This is typically very slow and should only be used in circumstances where clearing denominators would consume too much memory.

void fmpq_mat_mul_cleared(fmpq_mat_t C, const fmpq_mat_t A, const fmpq_mat_t B)

Sets C to the matrix product AB, computed by clearing denominators and multiplying over the integers.

void fmpq_mat_mul(fmpq_mat_t C, const fmpq_mat_t A, const fmpq_mat_t B)

Sets C to the matrix product AB. This simply calls fmpq_mat_mul_cleared.
void fmpq_mat_mul_fmpz_mat(fmpq_mat_t C, const fmpq_mat_t A, const fmpz_mat_t B)

Sets C to the matrix product AB, with B an integer matrix. This function works efficiently by clearing denominators of A.

void fmpq_mat_mul_r_fmpz_mat(fmpq_mat_t C, const fmpz_mat_t A, const fmpq_mat_t B)

Sets C to the matrix product AB, with A an integer matrix. This function works efficiently by clearing denominators of B.

void fmpq_mat_mul_fmpq_vec(fmpq *c, const fmpq_mat_t A, const fmpq *b, slong blen)
void fmpq_mat_mul_fmpz_vec(fmpq *c, const fmpz_mat_t A, const fmpz *b, slong blen)
void fmpq_mat_mul_fmpq_vec_ptr(fmpq *const *c, const fmpq_mat_t A, const fmpq *const *b, slong blen)
void fmpq_mat_mul_fmpz_vec_ptr(fmpq *const *c, const fmpz_mat_t A, const fmpz *const *b, slong blen)

Compute a matrix-vector product of A and (b, blen) and store the result in c. The vector (b, blen) is either truncated or zero-extended to the number of columns of A. The number entries written to c is always equal to the number of rows of A.

void fmpq_mat_fmpq_vec_mul(fmpq *c, const fmpq *a, slong alen, const fmpq_mat_t B)
void fmpq_mat_fmpz_vec_mul(fmpq *c, const fmpz *a, slong alen, const fmpq_mat_t B)
void fmpq_mat_fmpq_vec_mul_ptr(fmpq *const *c, const fmpq *const *a, slong alen, const fmpq_mat_t B)
void fmpq_mat_fmpz_vec_mul_ptr(fmpq *const *c, const fmpz *const *a, slong alen, const fmpq_mat_t B)

Compute a vector-matrix product of (a, alen) and B and and store the result in c. The vector (a, alen) is either truncated or zero-extended to the number of rows of B. The number entries written to c is always equal to the number of columns of B.

5.3.15 Kronecker product

void fmpq_mat_kronecker_product(fmpq_mat_t C, const fmpq_mat_t A, const fmpq_mat_t B)

Sets C to the Kronecker product of A and B.

5.3.16 Trace

void fmpq_mat_trace(fmpq_t trace, const fmpq_mat_t mat)

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

5.3.17 Determinant

void fmpq_mat_det(fmpq_t det, const fmpq_mat_t mat)

Sets det to the determinant of mat. In the general case, the determinant is computed by clearing denominators and computing a determinant over the integers. Matrices of size 0, 1 or 2 are handled directly.
5.3.18 Nonsingular solving

```c
int fmpq_mat_solve_fraction_free(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
int fmpq_mat_solve_dixon(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
int fmpq_mat_solve_multi_mod(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
int fmpq_mat_solve(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
```

Solves $AX = B$ for nonsingular $A$. Returns nonzero if $A$ is nonsingular or if the right hand side is empty, and zero otherwise.

All algorithms clear denominators to obtain a rescaled system over the integers. The \texttt{fraction\_free} algorithm uses FFLU solving over the integers. The \texttt{dixon} and \texttt{multi\_mod} algorithms use Dixon $p$-adic lifting or multimodular solving, followed by rational reconstruction with an adaptive stopping test. The \texttt{dixon} and \texttt{multi\_mod} algorithms are generally the best choice for large systems.

The default method chooses an algorithm automatically.

```c
int fmpq_mat_solve_fmpz_mat_fraction_free(fmpq_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
int fmpq_mat_solve_fmpz_mat_dixon(fmpq_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
int fmpq_mat_solve_fmpz_mat_multi_mod(fmpq_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
int fmpq_mat_solve_fmpz_mat(fmpq_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
```

Solves $AX = B$ for nonsingular $A$, where $A$ and $B$ are integer matrices. Returns nonzero if $A$ is nonsingular or if the right hand side is empty, and zero otherwise.

```c
int fmpq_mat_can_solve_multi_mod(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
int fmpq_mat_can_solve_fraction_free(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
int fmpq_mat_can_solve_fmpz_mat_dixon(fmpq_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
int fmpq_mat_can_solve_dixon(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
int fmpq_mat_can_solve(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
```

Returns 1 if $AX = B$ has a solution and if so, sets $X$ to one such solution. The matrices can have any shape but must have the same number of rows.

```c
int fmpq_mat_can_solve_fmpz_mat(fmpq_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
```

Returns 1 if $AX = B$ has a solution and if so, sets $X$ to one such solution. The matrices can have any shape but must have the same number of rows.

```c
int fmpq_mat_can_solve_fmpz_mat_dixon(fmpq_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
```

Returns 1 if $AX = B$ has a solution and if so, sets $X$ to one such solution. The matrices can have any shape but must have the same number of rows.

```c
int fmpq_mat_can_solve_dixon(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
```

Returns 1 if $AX = B$ has a solution and if so, sets $X$ to one such solution. The matrices can have any shape but must have the same number of rows.

```c
int fmpq_mat_can_solve(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
```

Returns 1 if $AX = B$ has a solution and if so, sets $X$ to one such solution. The matrices can have any shape but must have the same number of rows.
5.3.19 Inverse

```c
int fmpq_mat_inv(fmpq_mat_t B, const fmpq_mat_t A)
```
Sets \( B \) to the inverse matrix of \( A \) and returns nonzero. Returns zero if \( A \) is singular. \( A \) must be a square matrix.

5.3.20 Echelon form

```c
int fmpq_mat_pivot(slong *perm, fmpq_mat_t mat, slong r, slong c)
```
Helper function for row reduction. Returns 1 if the entry of \( mat \) at row \( r \) and column \( c \) is nonzero. Otherwise searches for a nonzero entry in the same column among rows \( r+1, r+2, \ldots \). If a nonzero entry is found at row \( s \), swaps rows \( r \) and \( s \) and the corresponding entries in \( perm \) (unless \( NULL \)) and returns -1. If no nonzero pivot entry is found, leaves the inputs unchanged and returns 0.

```c
slong fmpq_mat_rref_classical(fmpq_mat_t B, const fmpq_mat_t A)
```
Sets \( B \) to the reduced row echelon form of \( A \) and returns the rank. Performs Gauss-Jordan elimination directly over the rational numbers. This algorithm is usually inefficient and is mainly intended to be used for testing purposes.

```c
slong fmpq_mat_rref_fraction_free(fmpq_mat_t B, const fmpq_mat_t A)
```
Sets \( B \) to the reduced row echelon form of \( A \) and returns the rank. Clears denominators and performs fraction-free Gauss-Jordan elimination using \( fmpz_mat \) functions.

```c
slong fmpq_mat_rref(fmpq_mat_t B, const fmpq_mat_t A)
```
Sets \( B \) to the reduced row echelon form of \( A \) and returns the rank. This function automatically chooses between the classical and fraction-free algorithms depending on the size of the matrix.

5.3.21 Gram-Schmidt Orthogonalisation

```c
void fmpq_mat_gso(fmpq_mat_t B, const fmpq_mat_t A)
```
Takes a subset of \( \mathbb{Q}^m = \{a_1, a_2, \ldots, a_n\} \) (as the columns of a \( m \times n \) matrix \( A \)) and generates an orthogonal set \( S' = \{b_1, b_2, \ldots, b_n\} \) (as the columns of the \( m \times n \) matrix \( B \)) that spans the same subspace of \( \mathbb{Q}^m \) as \( S \).

5.3.22 Transforms

```c
void fmpq_mat_similarity(fmpq_mat_t A, slong r, fmpq_t d)
```
Applies a similarity transform to the \( n \times n \) matrix \( M \) in-place.

If \( P \) is the \( n \times n \) identity matrix the zero entries of whose row \( r \) (0-indexed) have been replaced by \( d \), this transform is equivalent to \( M = P^{-1}MP \).

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

5.3.23 Characteristic polynomial

```c
void _fmpq_mat_charpoly(fmpz *coeffs, fmpz_t den, const fmpq_mat_t mat)
```
Set \( (coeffs, den) \) to the characteristic polynomial of the given \( n \times n \) matrix.

```c
void fmpq_mat_charpoly(fmpq_poly_t pol, const fmpq_mat_t mat)
```
Set \( pol \) to the characteristic polynomial of the given \( n \times n \) matrix. If \( mat \) is not square, an exception is raised.
5.3.24 Minimal polynomial

```c
slong _fmpq_mat_minpoly(fmpz *coeffs, fmpz_t den, const fmpq_mat_t mat)
```

Set \((\text{coeffs}, \text{den})\) to the minimal polynomial of the given \(n \times n\) matrix and return the length of the polynomial.

```c
void fmpq_mat_minpoly(fmpq_poly_t pol, const fmpq_mat_t mat)
```

Set \(\text{pol}\) to the minimal polynomial of the given \(n \times n\) matrix. If \(\text{mat}\) is not square, an exception is raised.

5.4 `fmpq_poly.h` – univariate polynomials over the rational numbers

The `fmpq_poly_t` data type represents elements of \(\mathbb{Q}[x]\). The `fmpq_poly` module provides routines for memory management, basic arithmetic, and conversions from or to other types.

A rational polynomial is stored as the quotient of an integer polynomial and an integer denominator. To be more precise, the coefficient vector of the numerator can be accessed with the function `fmpq_poly_numref()` and the denominator with `fmpq_poly_denref()`. Although one can construct use cases in which a representation as a list of rational coefficients would be beneficial, the choice made here is typically more efficient.

We can obtain a unique representation based on this choice by enforcing, for non-zero polynomials, that the numerator and denominator are coprime and that the denominator is positive. The unique representation of the zero polynomial is chosen as \(0/1\).

Similar to the situation in the `fmpz_poly_t` case, an `fmpq_poly_t` object also has a `length` parameter, which denotes the length of the vector of coefficients of the numerator. We say a polynomial is `normalised` either if this length is zero or if the leading coefficient is non-zero.

We say a polynomial is in `canonical` form if it is given in the unique representation discussed above and normalised.

The functions provided in this module roughly fall into two categories:

On the one hand, there are functions mainly provided for the user, whose names do not begin with an underscore. These typically operate on polynomials of type `fmpq_poly_t` in canonical form and, unless specified otherwise, permit aliasing between their input arguments and between their output arguments.

On the other hand, there are versions of these functions whose names are prefixed with a single underscore. These typically operate on polynomials given in the form of a triple of object of types `fmpz *`, `fmpz_t`, and `slong`, containing the numerator, denominator and length, respectively. In general, these functions expect their input to be normalised, i.e. they do not allow zero padding, and to be in lowest terms, and they do not allow their input and output arguments to be aliased.

5.4.1 Types, macros and constants

```c
type fmpq_poly_struct

type fmpq_poly_t
```
5.4.2 Memory management

void fmpq_poly_init(fmpq_poly_t poly)
    Initialises the polynomial for use. The length is set to zero.

void fmpq_poly_init2(fmpq_poly_t poly, slong alloc)
    Initialises the polynomial with space for at least alloc coefficients and sets the length to zero. The alloc coefficients are all set to zero.

void fmpq_poly_realloc(fmpq_poly_t poly, slong alloc)
    Reallocates the given polynomial to have space for alloc coefficients. If alloc is zero then the polynomial is cleared and then reinitialised. If the current length is greater than alloc then poly is first truncated to length alloc. Note that this might leave the rational polynomial in non-canonical form.

void fmpq_poly_fit_length(fmpq_poly_t poly, slong len)
    If len is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least len coefficients. No data is lost when calling this function. The function efficiently deals with the case where fit_length() is called many times in small increments by at least doubling the number of allocated coefficients when len is larger than the number of coefficients currently allocated.

void _fmpq_poly_set_length(fmpq_poly_t poly, slong len)
    Sets the length of the numerator polynomial to len, demoting coefficients beyond the new length. Note that this method does not guarantee that the rational polynomial is in canonical form.

void fmpq_poly_clear(fmpq_poly_t poly)
    Clears the given polynomial, releasing any memory used. The polynomial must be reinitialised in order to be used again.

void _fmpq_poly_normalise(fmpq_poly_t poly)
    Sets the length of poly so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. Note that this function does not guarantee the coprimality of the numerator polynomial and the integer denominator.

void _fmpq_poly_canonicalise(fmpz *poly, fmpz_t den, slong len)
    Puts (poly, den) of length len into canonical form. It is assumed that the array poly contains a non-zero entry in position len - 1 whenever len > 0. Assumes that den is non-zero.

void fmpq_poly_canonicalise(fmpq_poly_t poly)
    Puts the polynomial poly into canonical form. Firstly, the length is set to the actual length of the numerator polynomial. For non-zero polynomials, it is then ensured that the numerator and denominator are coprime and that the denominator is positive. The canonical form of the zero polynomial is a zero numerator polynomial and a one denominator.

int _fmpq_poly_is_canonical(const fmpz *poly, const fmpz_t den, slong len)
    Returns whether the polynomial is in canonical form.

int fmpq_poly_is_canonical(const fmpq_poly_t poly)
    Returns whether the polynomial is in canonical form.
5.4.3 Polynomial parameters

\[\text{slong } \text{fmpq\_poly\_degree}(\text{const } \text{fmpq\_poly\_t poly})\]

Returns the degree of \( \text{poly} \), which is one less than its length, as a \text{slong}.

\[\text{slong } \text{fmpq\_poly\_length}(\text{const } \text{fmpq\_poly\_t poly})\]

Returns the length of \( \text{poly} \).

5.4.4 Accessing the numerator and denominator

\[\text{fmpz } * \text{fmpq\_poly\_numref}(\text{fmpq\_poly\_t poly})\]

Returns a reference to the numerator polynomial as an array.

Note that, because of a delayed initialisation approach, this might be NULL for zero polynomials. This situation can be salvaged by calling either \text{fmpq\_poly\_fit\_length}() or \text{fmpq\_poly\_realloc}().

This function is implemented as a macro returning \((\text{poly})\rightarrow\text{coeffs}\).

\[\text{fmpz\_t } \text{fmpq\_poly\_denref}(\text{fmpq\_poly\_t poly})\]

Returns a reference to the denominator as a \text{fmpz\_t}. The integer is guaranteed to be properly initialised.

This function is implemented as a macro returning \((\text{poly})\rightarrow\text{den}\).

\[\text{void } \text{fmpq\_poly\_get\_numerator}(\text{fmpz\_poly\_t res, const } \text{fmpq\_poly\_t poly})\]

Sets \( \text{res} \) to the numerator of \( \text{poly} \), e.g. the primitive part as an \text{fmpz\_poly\_t} if it is in canonical form.

\[\text{void } \text{fmpq\_poly\_get\_denominator}(\text{fmpz\_t den, const } \text{fmpq\_poly\_t poly})\]

Sets \( \text{res} \) to the denominator of \( \text{poly} \).

5.4.5 Random testing

The functions \text{fmpq\_poly\_randtest\_foo()} provide random polynomials suitable for testing. On an integer level, this means that long strings of zeros and ones in the binary representation are favoured as well as the special absolute values 0, 1, \text{COEFF\_MAX}, and \text{WORD\_MAX}. On a polynomial level, the integer numerator has a reasonable chance to have a non-trivial content.

\[\text{void } \text{fmpq\_poly\_randtest}(\text{fmpq\_poly\_t f, flint\_rand\_t state, slong len, flint\_bitcnt\_t bits})\]

Sets \( f \) to a random polynomial with coefficients up to the given length and where each coefficient has up to the given number of bits. The coefficients are signed randomly. One must call \text{flint\_randinit()} before calling this function.

\[\text{void } \text{fmpq\_poly\_randtest\_unsigned}(\text{fmpq\_poly\_t f, flint\_rand\_t state, slong len, flint\_bitcnt\_t bits})\]

Sets \( f \) to a random polynomial with coefficients up to the given length and where each coefficient has up to the given number of bits. One must call \text{flint\_randinit()} before calling this function.

\[\text{void } \text{fmpq\_poly\_randtest\_not\_zero}(\text{fmpq\_poly\_t f, flint\_rand\_t state, slong len, flint\_bitcnt\_t bits})\]

As for \text{fmpq\_poly\_randtest()} except that \text{len} and \text{bits} may not be zero and the polynomial generated is guaranteed not to be the zero polynomial. One must call \text{flint\_randinit()} before calling this function.
### 5.4.6 Assignment, swap, negation

```c
void fmpq_poly_set(fmpq_poly_t poly1, const fmpq_poly_t poly2)
    Sets poly1 to equal poly2.

void fmpq_poly_set_si(fmpq_poly_t poly, slong x)
    Sets poly to the integer x.

void fmpq_poly_set_ui(fmpq_poly_t poly, ulong x)
    Sets poly to the integer x.

void fmpq_poly_set_fmpz(fmpq_poly_t poly, const fmpz_t x)
    Sets poly to the integer x.

void fmpq_poly_set_fmpq(fmpq_poly_t poly, const fmpq_t x)
    Sets poly to the rational \( x \), which is assumed to be given in lowest terms.

void fmpq_poly_set_fmpz_poly(fmpq_poly_t rop, const fmpz_poly_t op)
    Sets the rational polynomial rop to the same value as the integer polynomial op.

void fmpq_poly_set_nmod_poly(fmpq_poly_t rop, const nmod_poly_t op)
    Sets the coefficients of rop to the residues in op, normalised to the interval \(-m/2 \leq r < m/2\) where \( m \) is the modulus.

void fmpq_poly_get_nmod_poly(nmod_poly_t rop, const fmpq_poly_t op, int den)
    Sets the coefficients of rop to the coefficients in the denominator of op, reduced by the modulus of rop. The result is multiplied by the inverse of the denominator of op. It is assumed that the reduction of the denominator of op is invertible.

int _fmpq_poly_set_str(fmpz *poly, fmpz_t den, const char *str, slong len)
    Sets (poly, den) to the polynomial specified by the null-terminated string str of len coefficients. The input format is a sequence of coefficients separated by one space.

    The result is only guaranteed to be in lowest terms if all coefficients in the input string are in lowest terms.

    Returns 0 if no error occurred. Otherwise, returns -1 in which case the resulting value of (poly, den) is undefined. If str is not null-terminated, calling this method might result in a segmentation fault.

int fmpq_poly_set_str(fmpq_poly_t poly, const char *str)
    Sets poly to the polynomial specified by the null-terminated string str. The input format is the same as the output format of fmpq_poly_get_str: the length given as a decimal integer, then two spaces, then the list of coefficients separated by one space.

    The result is only guaranteed to be in canonical form if all coefficients in the input string are in lowest terms.

    Returns 0 if no error occurred. Otherwise, returns -1 in which case the resulting value of poly is set to zero. If str is not null-terminated, calling this method might result in a segmentation fault.

char *fmpq_poly_get_str(const fmpq_poly_t poly)
    Returns the string representation of poly.

char *fmpq_poly_get_str_pretty(const fmpq_poly_t poly, const char *var)
    Returns the pretty representation of poly, using the null-terminated string var not equal to "\0" as the variable name.
```
void \texttt{fmpq\_poly\_zero}(\texttt{fmpq\_poly\_t} \texttt{poly})
Sets \texttt{poly} to zero.

void \texttt{fmpq\_poly\_one}(\texttt{fmpq\_poly\_t} \texttt{poly})
Sets \texttt{poly} to the constant polynomial 1.

void \texttt{fmpq\_poly\_neg}(\texttt{fmpq\_poly\_t} \texttt{poly1}, \texttt{const fmpq\_poly\_t} \texttt{poly2})
Sets \texttt{poly1} to the additive inverse of \texttt{poly2}.

void \texttt{fmpq\_poly\_inv}(\texttt{fmpq\_poly\_t} \texttt{poly1}, \texttt{const fmpq\_poly\_t} \texttt{poly2})
Sets \texttt{poly1} to the multiplicative inverse of \texttt{poly2} if possible. Otherwise, if \texttt{poly2} is not a unit, leaves \texttt{poly1} unmodified and calls \texttt{abort()}.

void \texttt{fmpq\_poly\_swap}(\texttt{fmpq\_poly\_t} \texttt{poly1}, \texttt{fmpq\_poly\_t} \texttt{poly2})
Efficiently swaps the polynomials \texttt{poly1} and \texttt{poly2}.

void \texttt{fmpq\_poly\_truncate}(\texttt{fmpq\_poly\_t} \texttt{poly}, \texttt{slong} \texttt{n})
If the current length of \texttt{poly} is greater than \texttt{n}, it is truncated to the given length. Discarded coefficients are demoted, but they are not necessarily set to zero.

void \texttt{fmpq\_poly\_set\_trunc}(\texttt{fmpq\_poly\_t} \texttt{res}, \texttt{const fmpq\_poly\_t} \texttt{poly}, \texttt{slong} \texttt{n})
Sets \texttt{res} to a copy of \texttt{poly}, truncated to length \texttt{n}.

void \texttt{fmpq\_poly\_get\_slice}(\texttt{fmpq\_poly\_t} \texttt{rop}, \texttt{const fmpq\_poly\_t} \texttt{op}, \texttt{slong} \texttt{i}, \texttt{slong} \texttt{j})
Returns the slice with coefficients from \texttt{x}^\texttt{i} (including) to \texttt{x}^\texttt{j} (excluding).

void \texttt{fmpq\_poly\_reverse}(\texttt{fmpq\_poly\_t} \texttt{res}, \texttt{const fmpq\_poly\_t} \texttt{poly}, \texttt{slong} \texttt{n})
This function considers the polynomial \texttt{poly} to be of length \texttt{n}, notionally truncating and zero padding if required, and reverses the result. Since the function normalises its result \texttt{res} may be of length less than \texttt{n}.

\textbf{5.4.7 Getting and setting coefficients}

void \texttt{fmpq\_poly\_get\_coeff\_fmpz}(\texttt{fmpz\_t} \texttt{x}, \texttt{const fmpq\_poly\_t} \texttt{poly}, \texttt{slong} \texttt{n})
Retrieves the \texttt{n}th coefficient of the numerator of \texttt{poly}.

void \texttt{fmpq\_poly\_get\_coeff\_fmpq}(\texttt{fmpq\_t} \texttt{x}, \texttt{const fmpq\_poly\_t} \texttt{poly}, \texttt{slong} \texttt{n})
Retrieves the \texttt{n}th coefficient of \texttt{poly}, in lowest terms.

void \texttt{fmpq\_poly\_set\_coeff\_si}(\texttt{fmpq\_poly\_t} \texttt{poly}, \texttt{slong} \texttt{n}, \texttt{slong} \texttt{x})
Sets the \texttt{n}th coefficient in \texttt{poly} to the integer \texttt{x}.

void \texttt{fmpq\_poly\_set\_coeff\_ui}(\texttt{fmpq\_poly\_t} \texttt{poly}, \texttt{slong} \texttt{n}, \texttt{ulong} \texttt{x})
Sets the \texttt{n}th coefficient in \texttt{poly} to the integer \texttt{x}.

void \texttt{fmpq\_poly\_set\_coeff\_fmpz}(\texttt{fmpq\_poly\_t} \texttt{poly}, \texttt{slong} \texttt{n}, \texttt{const fmpz\_t} \texttt{x})
Sets the \texttt{n}th coefficient in \texttt{poly} to the integer \texttt{x}.

void \texttt{fmpq\_poly\_set\_coeff\_fmpq}(\texttt{fmpq\_poly\_t} \texttt{poly}, \texttt{slong} \texttt{n}, \texttt{const fmpq\_t} \texttt{x})
Sets the \texttt{n}th coefficient in \texttt{poly} to the rational \texttt{x}.
5.4.8 Comparison

int _fmpq_poly_equal_trunc(const fmpz_t poly1, const fmpz_t den1, slong len1, const fmpz_t poly2, const fmpz_t den2, slong len2, slong n)

Returns 1 if poly1 and poly2 notionally truncated to length 𝑛 are equal, otherwise returns 0.

int _fmpq_poly_cmp(const fmpz_t lpoly, const fmpz_t lden, const fmpz_t rpoly, const fmpz_t rden, slong len)

Compares two non-zero polynomials, assuming they have the same length 𝑙𝑒𝑛 > 0.

The polynomials are expected to be provided in canonical form.

int fmpq_poly_is_one(const fmpq_poly_t poly)

Returns 1 if poly is the constant polynomial 1, otherwise returns 0.

int fmpq_poly_is_zero(const fmpq_poly_t poly)

Returns 1 if poly is the zero polynomial, otherwise returns 0.

int fmpq_poly_is_gen(const fmpq_poly_t poly)

Returns 1 if poly is the degree 1 polynomial 𝑥, otherwise returns 0.

5.4.9 Addition and subtraction

void _fmpq_poly_add(fmpz *rpoly, fmpz_t rden, const fmpz_t den1, const fmpz *poly1, const fmpz_t den2, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2)

Forms the sum (rpoly, rden) of (poly1, den1, len1) and (poly2, den2, len2), placing the result into canonical form.

Assumes that rpoly is an array of length the maximum of len1 and len2. The input operands are assumed to be in canonical form and are also allowed to be of length 0.

(rpoly, rden) and (poly1, den1) may be aliased, but (rpoly, rden) and (poly2, den2) may not be aliased.

void _fmpq_poly_add_can(fmpz *rpoly, fmpz_t rden, const fmpz_t den1, const fmpz *poly1, const fmpz_t den2, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, int can)

As per _fmpq_poly_add except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

void fmpq_poly_add(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2)

Sets res to the sum of poly1 and poly2, using Henrici’s algorithm.

void fmpq_poly_add_can(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, int can)

As per fmpq_poly_add except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.
void _fmpq_poly_add_series(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong n)

As per _fmpq_poly_add but the inputs are first notionally truncated to length n. If n is less than len1 or len2 then the output only needs space for n coefficients. We require n ≥ 0.

void _fmpq_poly_add_series_can(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong n, int can)

As per _fmpq_poly_add_can but the inputs are first notionally truncated to length n. If n is less than len1 or len2 then the output only needs space for n coefficients. We require n ≥ 0.

void fmpq_poly_add_series(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, slong n)

As per fmpq_poly_add but the inputs are first notionally truncated to length n.

void fmpq_poly_add_series_can(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, slong n, int can)

As per fmpq_poly_add_can but the inputs are first notionally truncated to length n.

void _fmpq_poly_sub(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2)

Forms the difference (rpoly, rden) of (poly1, den1, len1) and (poly2, den2, len2), placing the result into canonical form.

Assumes that rpoly is an array of length the maximum of len1 and len2. The input operands are assumed to be in canonical form and are also allowed to be of length 0.

(rpoly, rden) and (poly1, den1, len1) may be aliased, but (rpoly, rden) and (poly2, den2, len2) may not be aliased.

void _fmpq_poly_sub_can(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, int can)

As per _fmpq_poly_sub except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

void fmpq_poly_sub(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2)

Sets res to the difference of poly1 and poly2, using Henrici’s algorithm.

void fmpq_poly_sub_can(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, int can)

As per _fmpq_poly_sub except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

void _fmpq_poly_sub_series(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong n)

As per _fmpq_poly_sub but the inputs are first notionally truncated to length n. If n is less than len1 or len2 then the output only needs space for n coefficients. We require n ≥ 0.

void _fmpq_poly_sub_series_can(fmpz *poly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong n, int can)

As per _fmpq_poly_sub_can but the inputs are first notionally truncated to length n. If n is less than len1 or len2 then the output only needs space for n coefficients. We require n ≥ 0.

void fmpq_poly_sub_series(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, slong n)

As per fmpq_poly_sub but the inputs are first notionally truncated to length n.
void \texttt{fmpq\_poly\_sub\_series\_can}(\texttt{fmpq\_poly\_t} res, const \texttt{fmpq\_poly\_t} poly1, const \texttt{fmpq\_poly\_t} poly2, \texttt{slong} n, int can)

As per \texttt{fmpq\_poly\_sub\_can} but the inputs are first notionally truncated to length \( n \).

### 5.4.10 Scalar multiplication and division

void \_\texttt{fmpq\_poly\_scalar\_mul\_si}(\texttt{fmpz *}rpoly, \texttt{fmpz\_t} rden, const \texttt{fmpz *}poly, const \texttt{fmpz\_t} den, \texttt{slong} len, \texttt{slong} c)

Sets \((\texttt{rpoly}, \texttt{rden}, \texttt{len})\) to the product of \( c \) of \((\texttt{poly}, \texttt{den}, \texttt{len})\).

If the input is normalised, then so is the output, provided it is non-zero. If the input is in lowest terms, then so is the output. However, even if neither of these conditions are met, the result will be (mathematically) correct.

Supports exact aliasing between \((\texttt{rpoly}, \texttt{den})\) and \((\texttt{poly}, \texttt{den})\).

void \_\texttt{fmpq\_poly\_scalar\_mul\_ui}(\texttt{fmpz *}rpoly, \texttt{fmpz\_t} rden, const \texttt{fmpz *}poly, const \texttt{fmpz\_t} den, \texttt{slong} len, \texttt{ulong} c)

Sets \((\texttt{rpoly}, \texttt{rden}, \texttt{len})\) to the product of \( c \) of \((\texttt{poly}, \texttt{den}, \texttt{len})\).

If the input is normalised, then so is the output, provided it is non-zero. If the input is in lowest terms, then so is the output. However, even if neither of these conditions are met, the result will be (mathematically) correct.

Supports exact aliasing between \((\texttt{rpoly}, \texttt{den})\) and \((\texttt{poly}, \texttt{den})\).

void \_\texttt{fmpq\_poly\_scalar\_mul\_fmpz}(\texttt{fmpz *}rpoly, \texttt{fmpz\_t} rden, const \texttt{fmpz *}poly, const \texttt{fmpz\_t} den, \texttt{slong} len, const \texttt{fmpz\_t} r)

Sets \((\texttt{rpoly}, \texttt{rden})\) to the product of \( r/s \) and \((\texttt{poly}, \texttt{den}, \texttt{len})\), in lowest terms.

Assumes that \((\texttt{poly}, \texttt{den}, \texttt{len})\) and \( r/s \) are provided in lowest terms. Assumes that \texttt{rpoly} is an array of length \texttt{len}. Supports aliasing of \((\texttt{rpoly}, \texttt{den})\) and \((\texttt{poly}, \texttt{den})\). The \texttt{fmpz\_t}'s \( r \) and \( s \) may not be part of \((\texttt{rpoly}, \texttt{rden})\).

void \texttt{fmpq\_poly\_scalar\_mul\_fmpq}(\texttt{fmpq\_poly\_t} rop, const \texttt{fmpq\_poly\_t} op, const \texttt{fmpq\_t} c)

void \texttt{fmpq\_poly\_scalar\_mul\_si}(\texttt{fmpq\_poly\_t} rop, const \texttt{fmpq\_poly\_t} op, \texttt{fmpq\_t} c)

void \texttt{fmpq\_poly\_scalar\_mul\_ui}(\texttt{fmpq\_poly\_t} rop, const \texttt{fmpq\_poly\_t} op, \texttt{slong} c)

Sets \( \texttt{rop} \) to \( c \) times \( \texttt{op} \).

void \texttt{fmpq\_poly\_scalar\_div\_fmpz}(\texttt{fmpq\_poly\_t} rop, const \texttt{fmpq\_poly\_t} op, const \texttt{fmpq\_t} c)

Sets \( \texttt{rop} \) to \( c \) times \( \texttt{op} \). Assumes that the \texttt{fmpq\_t} \( c \) is not part of \( \texttt{rop} \).

void \_\texttt{fmpq\_poly\_scalar\_div\_fmpq}(\texttt{fmpq\_poly\_t} rop, const \texttt{fmpq\_poly\_t} op, const \texttt{fmpq\_t} c)

Sets \((\texttt{rop}, \texttt{rden})\) to \((\texttt{op}, \texttt{den}, \texttt{len})\) divided by \( c \), in lowest terms.

Assumes that \( \texttt{len} \) is positive. Assumes that \( c \) is non-zero. Supports aliasing between \((\texttt{rop}, \texttt{rden})\) and \((\texttt{op}, \texttt{den})\). Assumes that \( c \) is not part of \((\texttt{rop}, \texttt{rden})\).
void _fmpq_poly_scalar_div_si(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len, slong c)

Sets (rpoly, rden, len) to (poly, den, len) divided by c, in lowest terms.
Assumes that len is positive. Assumes that c is non-zero. Supports aliasing between (rpoly, rden) and (poly, den).

void _fmpq_poly_scalar_div_ui(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len, ulong c)

Sets (rpoly, rden, len) to (poly, den, len) divided by c, in lowest terms.
Assumes that len is positive. Assumes that c is non-zero. Supports aliasing between (rpoly, rden) and (poly, den).

void _fmpq_poly_scalar_div_fmpq(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len, const fmpz_t r, const fmpz_t s)

Sets (rpoly, rden, len) to (poly, den, len) divided by r/s, in lowest terms.
Assumes that len is positive. Assumes that r/s is non-zero and in lowest terms. Supports aliasing between (rpoly, rden) and (poly, den). The fmpz_t's r and s may not be part of (rpoly, poly).

void fmpq_poly_scalar_div_si(fmpq_poly_t rop, const fmpq_poly_t op, slong c)

void fmpq_poly_scalar_div_ui(fmpq_poly_t rop, const fmpq_poly_t op, ulong c)

void fmpq_poly_scalar_div_fmpq(fmpq_poly_t rop, const fmpq_poly_t op, const fmpz_t c)

Sets rop to op divided by the scalar c.

5.4.11 Multiplication

void _fmpq_poly_mul(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2)

Sets (rpoly, rden, len1 + len2 - 1) to the product of (poly1, den1, len1) and (poly2, den2, len2). If the input is provided in canonical form, then so is the output.
Assumes len1 >= len2 > 0. Allows zero-padding in the input. Does not allow aliasing between the inputs and outputs.

void fmpq_poly_mul(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2)

Sets res to the product of poly1 and poly2.

void _fmpq_poly_mullow(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong n)

Sets (rpoly, rden, n) to the low n coefficients of (poly1, den1) and (poly2, den2). The output is not guaranteed to be in canonical form.
Assumes len1 >= len2 > 0 and 0 < n <= len1 + len2 - 1. Allows for zero-padding in the inputs. Does not allow aliasing between the inputs and outputs.

void fmpq_poly_mullow(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, slong n)

Sets res to the product of poly1 and poly2, truncated to length n.

void fmpq_poly_addmul(fmpq_poly_t rop, const fmpq_poly_t op1, const fmpq_poly_t op2)

Adds the product of op1 and op2 to rop.

void fmpq_poly_submul(fmpq_poly_t rop, const fmpq_poly_t op1, const fmpq_poly_t op2)

Subtracts the product of op1 and op2 from rop.
5.4.12 Powering

```c
void _fmpq_poly_pow(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len, ulong e)
```

Sets \((rpoly, rden)\) to \((poly, den)^e\), assuming \(e, len > 0\). Assumes that \(rpoly\) is an array of length at least \(e * (len - 1) + 1\). Supports aliasing of \((rpoly, den)\) and \((poly, den)\).

```c
void fmpq_poly_pow(fmpq_poly_t res, const fmpq_poly_t poly, ulong e)
```

Sets \(res\) to \(poly^e\), where the only special case \(0^0\) is defined as 1.

```c
void _fmpq_poly_pow_trunc(fmpz *res, fmpz_t rden, const fmpz *f, const fmpz_t fden, slong flen, ulong exp, slong len)
```

Sets \((rpoly, rden, len)\) to \((poly, den)^e\) truncated to length \(len\), where \(len\) is at most \(e * (flen - 1) + 1\).

```c
void fmpq_poly_pow_trunc(fmpq_poly_t res, const fmpq_poly_t poly, ulong e, slong n)
```

Sets \(res\) to \(poly^e\) truncated to length \(n\).

5.4.13 Shifting

```c
void fmpq_poly_shift_left(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
```

Set \(res\) to \(poly\) shifted left by \(n\) coefficients. Zero coefficients are inserted.

```c
void fmpq_poly_shift_right(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
```

Set \(res\) to \(poly\) shifted right by \(n\) coefficients. If \(n\) is equal to or greater than the current length of \(poly\), \(res\) is set to the zero polynomial.

5.4.14 Euclidean division

```c
void _fmpq_poly_divrem(fmpz *Q, fmpz_t q, fmpz *R, fmpz_t r, const fmpz *A, const fmpz_t a, slong lenA, const fmpz *B, const fmpz_t b, slong lenB, const fmpz_preinvn_t inv)
```

Finds the quotient \((Q, q)\) and remainder \((R, r)\) of the Euclidean division of \((A, a)\) by \((B, b)\).

Assumes that \(lenA >= lenB > 0\). Assumes that \(R\) has space for \(lenA\) coefficients, although only the bottom \(lenB - 1\) will carry meaningful data on exit. Supports no aliasing between the two outputs, or between the inputs and the outputs.

An optional precomputed inverse of the leading coefficient of \(B\) from \texttt{fmpz\_preinvn\_init} can be supplied. Otherwise \(inv\) should be \texttt{NULL}.

Note: \texttt{fmpz.h} has to be included before \texttt{fmpq\_poly.h} in order for the latter to declare this function.

```c
void fmpq_poly_divrem(fmpq_poly_t Q, fmpq_poly_t R, const fmpq_poly_t poly1, const fmpq_poly_t poly2)
```

Finds the quotient \(Q\) and remainder \(R\) of the Euclidean division of \(poly1\) by \(poly2\).

```c
void _fmpq_poly_div(fmpz *Q, fmpz_t q, const fmpz *A, const fmpz_t a, slong lenA, const fmpz *B, const fmpz_t b, slong lenB, const fmpz_preinvn_t inv)
```

Finds the quotient \((Q, q)\) of the Euclidean division of \((A, a)\) by \((B, b)\).

Assumes that \(lenA >= lenB > 0\). Supports no aliasing between the inputs and the outputs.

An optional precomputed inverse of the leading coefficient of \(B\) from \texttt{fmpz\_preinvn\_init} can be supplied. Otherwise \(inv\) should be \texttt{NULL}.

Note: \texttt{fmpz.h} has to be included before \texttt{fmpq\_poly.h} in order for the latter to declare this function.
void \texttt{fmpq\_poly\_div}(\texttt{fmpq\_poly\_t} Q, const \texttt{fmpq\_poly\_t} poly1, const \texttt{fmpq\_poly\_t} poly2)  
Finds the quotient \(Q\) and remainder \(R\) of the Euclidean division of \(poly1\) by \(poly2\).

void \_fmpq\_poly\_rem(\texttt{fmpz \*R, fmpz\_t r}, const \texttt{fmpz\_t A}, const \texttt{fmpz\_t a}, const \texttt{fmpz\_t B}, const \texttt{fmpz\_t b}, \texttt{slong} lenA, \texttt{slong} lenB, const \texttt{fmpz\_preinvn\_t inv})  
Finds the remainder \((R, r)\) of the Euclidean division of \((A, a)\) by \((B, b)\).

Assumes that \(\text{lenA} >\geq \text{lenB} > 0\). Supports no aliasing between the inputs and the outputs.

An optional precomputed inverse of the leading coefficient of \(B\) from \texttt{fmpz\_preinvn\_init} can be supplied. Otherwise \texttt{inv} should be \texttt{NULL}.

Note: \texttt{fmpz.h} has to be included before \texttt{fmpq\_poly.h} in order for the latter to declare this function.

void \texttt{fmpq\_poly\_rem}(\texttt{fmpq\_poly\_t R, const fmpq\_poly\_t poly1, const fmpq\_poly\_t poly2})  
Finds the remainder \(R\) of the Euclidean division of \(poly1\) by \(poly2\).

\section{5.4.15 Powering}

\texttt{fmpq\_poly\_struct \_fmpq\_poly\_powers\_precompute}(const \texttt{fmpz \*B}, const \texttt{fmpz\_t denB}, \texttt{slong} len)  
Computes \(2\text{en} - 1\) powers of \(x\) modulo the polynomial \(B\) of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

void \texttt{fmpq\_poly\_powers\_precompute}(\texttt{fmpq\_poly\_powers\_precomp\_t pinv, fmpq\_poly\_t poly})  
Computes \(2\text{en} - 1\) powers of \(x\) modulo the polynomial \(B\) of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

void \_fmpq\_poly\_powers\_clear(\texttt{fmpq\_poly\_struct \*powers, slong len})  
Clean up resources used by precomputed powers which have been computed by \_fmpq\_poly\_powers\_precompute.

void \texttt{fmpq\_poly\_powers\_clear}(\texttt{fmpq\_poly\_powers\_precomp\_t pinv})  
Clean up resources used by precomputed powers which have been computed by \texttt{fmpq\_poly\_powers\_precompute}.

void \_fmpq\_poly\_rem\_powers\_precomp(\texttt{fmpz \*A, fmpz\_t denA, slong m, const fmpz \*B, const fmpz\_t denB, slong n, fmpq\_poly\_struct \*const powers})  
Set \(A\) to the remainder of \(A\) divide \(B\) given precomputed powers mod \(B\) provided by \_fmpq\_poly\_powers\_precompute. No aliasing is allowed.

This function is only faster if \(m \leq 2 \cdot n - 1\).

The output of this function is \textit{not} canonicalised.

void \texttt{fmpq\_poly\_rem\_powers\_precomp}(\texttt{fmpq\_poly\_t R, const fmpq\_poly\_t A, const fmpq\_poly\_t B, const fmpq\_poly\_powers\_precomp\_t B\_inv})  
Set \(R\) to the remainder of \(A\) divide \(B\) given precomputed powers mod \(B\) provided by \texttt{fmpq\_poly\_powers\_precompute}.

This function is only faster if \(A->\text{length} \leq 2\text{B->}\text{length} - 1\).  
The output of this function is \textit{not} canonicalised.
5.4.16 Divisibility testing

```c
int _fmpq_poly_divides(fmpz *qpoly, fmpz_t qden, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2)
```

Return 1 if (poly2, den2, len2) divides (poly1, den1, len1) and set (qpoly, qden, len1 - len2 + 1) to the quotient. Otherwise return 0. Requires that qpoly has space for len1 - len2 + 1 coefficients and that len1 >= len2 > 0.

```c
int fmpq_poly_divides(fmpq_poly_t q, const fmpq_poly_t poly1, const fmpq_poly_t poly2)
```

Return 1 if poly2 divides poly1 and set q to the quotient. Otherwise return 0.

```c
slong fmpq_poly_remove(fmpq_poly_t q, const fmpq_poly_t poly1, const fmpq_poly_t poly2)
```

Sets q to the quotient of poly1 by the highest power of poly2 which divides it, and returns the power. The divisor poly2 must not be constant or an exception is raised.

5.4.17 Power series division

```c
void _fmpq_poly_inv_series_newton(fmpz *rpol, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len, slong n)
```

Computes the first n terms of the inverse power series of (poly, den, len) using Newton iteration. The result is produced in canonical form.

Assumes that n ≥ 1 and that poly has non-zero constant term. Does not support aliasing.

```c
void fmpq_poly_inv_series_newton(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
```

Computes the first n terms of the inverse power series of poly using Newton iteration, assuming that poly has non-zero constant term and n ≥ 1.

```c
void _fmpq_poly_inv_series(fmpz *rpol, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong den_len, slong n)
```

Computes the first n terms of the inverse power series of (poly, den, len).

The result is produced in canonical form.

Assumes that n ≥ 1 and that poly has non-zero constant term. Does not support aliasing.

```c
void fmpq_poly_inv_series(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
```

Computes the first n terms of the inverse power series of poly, assuming that poly has non-zero constant term and n ≥ 1.

```c
void _fmpq_poly_div_series(fmpz *Q, fmpz_t denQ, const fmpz *A, const fmpz_t denA, slong lenA, const fmpz *B, const fmpz_t denB, slong lenB, slong n)
```

Divides (A, denA, lenA) by (B, denB, lenB) as power series over Q, assuming B has non-zero constant term and that all lengths are positive.

Aliasing is not supported.

This function ensures that the numerator and denominator are coprime on exit.

```c
void fmpq_poly_div_series(fmpq_poly_t Q, const fmpq_poly_t A, const fmpq_poly_t B, slong n)
```

Performs power series division in $\mathbb{Q}[x]/(x^n)$. The function considers the polynomials A and B as power series of length n starting with the constant terms. The function assumes that B has non-zero constant term and n ≥ 1.
5.4.18 Greatest common divisor

void \_fmpq\_poly\_gcd(fmpz \*G, fmpz\_t denG, const fmpz \*A, slong lenA, const fmpz \*B, slong lenB)
Computes the monic greatest common divisor \( G \) of \( A \) and \( B \).
Assumes that \( G \) has space for \( \text{len}(B) \) coefficients, where \( \text{len}(A) \geq \text{len}(B) > 0 \).
Aliasing between the output and input arguments is not supported.
Does not support zero-padding.

void fmpq\_poly\_gcd(fmpq\_poly\_t G, const fmpq\_poly\_t A, const fmpq\_poly\_t B)
Computes the monic greatest common divisor \( G \) of \( A \) and \( B \).
In the special case when \( A = B = 0 \), sets \( G = 0 \).

void \_fmpq\_poly\_xgcd(fmpz \*G, fmpz\_t denG, fmpz \*S, fmpz\_t denS, fmpz \*T, fmpz\_t denT, const fmpz \*A, const fmpq\_poly\_t denA, slong lenA, const fmpz \*B, const fmpq\_poly\_t denB, slong lenB)
Computes polynomials \( G, S, \) and \( T \) such that \( G = \gcd(A, B) = SA + TB \), where \( G \) is the monic greatest common divisor of \( A \) and \( B \).
Assumes that \( G, S, \) and \( T \) have space for \( \text{len}(B) \), \( \text{len}(B) \), and \( \text{len}(A) \) coefficients, respectively, where it is also assumed that \( \text{len}(A) \geq \text{len}(B) > 0 \).
Does not support zero padding of the input arguments.

void fmpq\_poly\_xgcd(fmpq\_poly\_t G, fmpq\_poly\_t S, fmpq\_poly\_t T, const fmpq\_poly\_t A, const fmpq\_poly\_t B)
Computes polynomials \( G, S, \) and \( T \) such that \( G = \gcd(A, B) = SA + TB \), where \( G \) is the monic greatest common divisor of \( A \) and \( B \).
Corner cases are handled as follows. If \( A = B = 0 \), returns \( G = S = T = 0 \). If \( A \neq 0, B = 0 \), returns the suitable scalar multiple of \( G = A, S = 1, \) and \( T = 0 \). The case when \( A = 0, B \neq 0 \) is handled similarly.

void \_fmpq\_poly\_lcm(fmpz \*L, fmpz\_t denL, const fmpz \*A, slong lenA, const fmpz \*B, slong lenB)
Computes the monic least common multiple \( L \) of \( A \) and \( B \).
Assumes that \( L \) has space for \( \text{len}(A) + \text{len}(B) - 1 \) coefficients, where \( \text{len}(A) \geq \text{len}(B) > 0 \).
Aliasing between the output and input arguments is not supported.
Does not support zero-padding.

void fmpq\_poly\_lcm(fmpq\_poly\_t L, const fmpq\_poly\_t A, const fmpq\_poly\_t B)
Computes the monic least common multiple \( L \) of \( A \) and \( B \).
In the special case when \( A = B = 0 \), sets \( L = 0 \).

void \_fmpq\_poly\_resultant(fmpq\_t rnum, fmpq\_t rden, const fmpz \*poly1, const fmpq\_poly\_t den1, slong len1, const fmpz \*poly2, const fmpq\_poly\_t den2, slong len2)
Sets \( (\text{rnum}, \text{rden}) \) to the resultant of the two input polynomials.
Assumes that \( \text{len1} \geq \text{len2} > 0 \). Does not support zero-padding of the input polynomials. Does not support aliasing of the input and output arguments.

void fmpq\_poly\_resultant(fmpq\_t r, const fmpq\_poly\_t f, const fmpq\_poly\_t g)
Returns the resultant of \( f \) and \( g \).
Enumerating the roots of \( f \) and \( g \over \mathbb{Q} \) as \( r_1, \ldots, r_m \) and \( s_1, \ldots, s_n \), respectively, and letting \( x \) and \( y \) denote the leading coefficients, the resultant is defined as
\[
x^{\deg(f)} y^{\deg(g)} \prod_{1 \leq i, j \leq n} (r_i - s_j).
\]
We handle special cases as follows: if one of the polynomials is zero, the resultant is zero. Note that otherwise if one of the polynomials is constant, the last term in the above expression is the empty product.

```c
void fmpq_poly_resultant_div(fmpq_t r, const fmpq_poly_t f, const fmpq_poly_t g, const fmpz_t div, slong nbits)
```

Returns the resultant of \( f \) and \( g \) divided by \( \text{div} \) under the assumption that the result has at most \( n\text{bits} \) bits. The result must be an integer.

### 5.4.19 Derivative and integral

```c
void _fmpq_poly_derivative(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len)
Sets (rpoly, rden, len - 1) to the derivative of (poly, den, len). Does nothing if len <= 1. Supports aliasing between the two polynomials.

void fmpq_poly_derivative(fmpq_poly_t res, const fmpq_poly_t poly)
Sets res to the derivative of poly.

void _fmpq_poly_nth_derivative(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den, ulong n, slong len)
Sets (rpoly, rden, len - n) to the nth derivative of (poly, den, len). Does nothing if len <= n. Supports aliasing between the two polynomials.

void fmpq_poly_nth_derivative(fmpq_poly_t res, const fmpq_poly_t poly, ulong n)
Sets res to the nth derivative of poly.

void _fmpq_poly_integral(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len)
Sets (rpoly, rden, len) to the integral of (poly, den, len - 1). Assumes len >= 0. Supports aliasing between the two polynomials. The output will be in canonical form if the input is in canonical form.

void fmpq_poly_integral(fmpq_poly_t res, const fmpq_poly_t poly)
Sets res to the integral of poly. The constant term is set to zero. In particular, the integral of the zero polynomial is the zero polynomial.

### 5.4.20 Square roots

```c
void _fmpq_poly_sqrt_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the square root of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 1. Does not support aliasing between the input and output polynomials.

void fmpq_poly_sqrt_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the square root of f to order n > 1. Requires f to have constant term 1.

void _fmpq_poly_invsqrt_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the inverse square root of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 1. Does not support aliasing between the input and output polynomials.

void fmpq_poly_invsqrt_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the inverse square root of f to order n > 0. Requires f to have constant term 1.
```
5.4.21 Power sums

void _fmpq_poly_power_sums(fmpz *res, fmpz_t rden, const fmpz *poly, slong len, slong n)
Compute the (truncated) power sums series of the polynomial (poly,len) up to length n using
Newton identities.

void fmpq_poly_power_sums(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
Compute the (truncated) power sum series of the monic polynomial poly up to length n using
Newton identities. That is the power series whose coefficient of degree i is the sum of the i-th
power of all (complex) roots of the polynomial poly.

void _fmpq_poly_power_sums_to_poly(fmpz *res, const fmpz *poly, const fmpz_t den, slong len)
Compute an integer polynomial given by its power sums series (poly,den,len).

void fmpq_poly_power_sums_to_fmpz_poly(fmpz_poly_t res, const fmpq_poly_t Q)
Compute the integer polynomial with content one and positive leading coefficient given by its power
sums series Q.

void fmpq_poly_power_sums_to_poly(fmpq_poly_t res, const fmpq_poly_t Q)
Compute the monic polynomial from its power sums series Q.

5.4.22 Transcendental functions

void _fmpq_poly_log_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the logarithm of (f, fden, flen). Assumes n > 0
and that (f, fden, flen) has constant term 1. Supports aliasing between the input and output
polynomials.

void fmpq_poly_log_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the logarithm of f to order n > 0. Requires f to have constant
term 1.

void _fmpq_poly_exp_series(fmpz *g, fmpz_t gden, const fmpz *h, const fmpz_t hden, slong hlen, slong n)
Sets (g, gden, n) to the series expansion of the exponential function of (h, hden, hlen). Assu-
mes n > 0, hlen > 0 and that (h, hden, hlen) has constant term 0. Supports aliasing be-
tween the input and output polynomials.

void fmpq_poly_exp_series(fmpq_poly_t res, const fmpq_poly_t h, slong n)
Sets res to the series expansion of the exponential function of h to order n > 0. Requires f to have constant
term 0.

void _fmpq_poly_expexpinv_series(fmpz *res1, fmpz_t res1den, fmpz *res2, fmpz_t res2den, const
fmpz *h, const fmpz_t hden, slong hlen, slong n)
The same as fmpq_poly_exp_series, but simultaneously computes the exponential (in res1,
res1den) and its multiplicative inverse (in res2, res2den). Supports aliasing between the input
and output polynomials.

void fmpq_poly_expexpinv_series(fmpq_poly_t res1, fmpq_poly_t res2, const fmpq_poly_t h, slong n)
The same as fmpq_poly_exp_series, but simultaneously computes the exponential (in res1) and
its multiplicative inverse (in res2).

void _fmpq_poly_atan_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the inverse tangent of (f, fden, flen). Assumes
n > 0 and that (f, fden, flen) has constant term 0. Supports aliasing between the input and output
polynomials.
void fmpq_poly_atan_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the inverse tangent of f to order n > 0. Requires f to have constant term 0.

void _fmpq_poly_atanh_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the inverse hyperbolic tangent of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 0. Supports aliasing between the input and output polynomials.

void fmpq_poly_atanh_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the inverse hyperbolic tangent of f to order n > 0. Requires f to have constant term 0.

void _fmpq_poly_asin_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the inverse sine of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 0. Supports aliasing between the input and output polynomials.

void fmpq_poly_asin_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the inverse sine of f to order n > 0. Requires f to have constant term 0.

void _fmpq_poly_asinh_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the inverse hyperbolic sine of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 0. Supports aliasing between the input and output polynomials.

void fmpq_poly_asinh_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the inverse hyperbolic sine of f to order n > 0. Requires f to have constant term 0.

void _fmpq_poly_tan_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the tangent function of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 0. Does not support aliasing between the input and output polynomials.

void fmpq_poly_tan_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the tangent function of f to order n > 0. Requires f to have constant term 0.

void _fmpq_poly_sin_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the sine of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 0. Supports aliasing between the input and output polynomials.

void fmpq_poly_sin_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the sine of f to order n > 0. Requires f to have constant term 0.

void _fmpq_poly_cos_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the cosine of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 0. Supports aliasing between the input and output polynomials.
void fmpq_poly_cos_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the cosine of f to order n > 0. Requires f to have constant term 0.

void _fmpq_poly_sin_cos_series(fmpz *s, fmpz_t sden, fmpz *c, fmpz_t cden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (s, sden, n) to the series expansion of the sine of (f, fden, flen), and (c, cden, n) to the series expansion of the cosine. Assumes n > 0 and that (f, fden, flen) has constant term 0. Supports aliasing between the input and output polynomials.

void fmpq_poly_sin_cos_series(fmpq_poly_t res1, fmpq_poly_t res2, const fmpq_poly_t f, slong n)
Sets res1 to the series expansion of the sine of f to order n > 0, and res2 to the series expansion of the cosine. Requires f to have constant term 0.

void _fmpq_poly_sinh_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the hyperbolic sine of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 0. Does not support aliasing between the input and output polynomials.

void fmpq_poly_sinh_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the hyperbolic sine of f to order n > 0. Requires f to have constant term 0.

void _fmpq_poly_cosh_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the hyperbolic cosine of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 0. Does not support aliasing between the input and output polynomials.

void fmpq_poly_cosh_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the hyperbolic cosine of f to order n > 0. Requires f to have constant term 0.

void _fmpq_poly_sinh_cosh_series(fmpz *s, fmpz_t sden, fmpz *c, fmpz_t cden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (s, sden, n) to the series expansion of the hyperbolic sine of (f, fden, flen), and (c, cden, n) to the series expansion of the hyperbolic cosine. Assumes n > 0 and that (f, fden, flen) has constant term 0. Supports aliasing between the input and output polynomials.

void fmpq_poly_sinh_cosh_series(fmpq_poly_t res1, fmpq_poly_t res2, const fmpq_poly_t f, slong n)
Sets res1 to the series expansion of the hyperbolic sine of f to order n > 0, and res2 to the series expansion of the hyperbolic cosine. Requires f to have constant term 0.

void _fmpq_poly_tanh_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen, slong n)
Sets (g, gden, n) to the series expansion of the hyperbolic tangent of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 0. Does not support aliasing between the input and output polynomials.

void fmpq_poly_tanh_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
Sets res to the series expansion of the hyperbolic tangent of f to order n > 0. Requires f to have constant term 0.
5.4.23 Orthogonal polynomials

void _fmpq_poly_legendre_p(fmpz *coeffs, fmpz_t den, ulong n)
Sets coeffs to the coefficient array of the Legendre polynomial \( P_n(x) \), defined by \((n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)\), for \( n \geq 0 \). Sets den to the overall denominator. The coefficients are calculated using a hypergeometric recurrence. The length of the array will be \( n+1 \). To improve performance, the common denominator is computed in one step and the coefficients are evaluated using integer arithmetic. The denominator is given by \( \gcd((n!)^2 + \lfloor n/4 \rfloor + \ldots) \). See fmpz_poly for the shifted Legendre polynomials.

void fmpq_poly_legendre_p(fmpz_poly_t poly, ulong n)
Sets poly to the Legendre polynomial \( P_n(x) \), defined by \((n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)\), for \( n \geq 0 \). The coefficients are calculated using a hypergeometric recurrence. To improve performance, the common denominator is computed in one step and the coefficients are evaluated using integer arithmetic. The denominator is given by \( \gcd(n!, 2^n) = 2^{\lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \ldots} \). See fmpz_poly for the shifted Legendre polynomials.

void _fmpq_poly_laguerre_l(fmpz *coeffs, fmpz_t den, ulong n)
Sets coeffs to the coefficient array of the Laguerre polynomial \( L_n(x) \), defined by \((n+1)L_{n+1}(x) = (2n+1-x)L_n(x) - nL_{n-1}(x)\), for \( n \geq 0 \). Sets den to the overall denominator. The coefficients are calculated using a hypergeometric recurrence. The length of the array will be \( n+1 \).

void fmpq_poly_laguerre_l(fmpz_poly_t poly, ulong n)
Sets poly to the Laguerre polynomial \( L_n(x) \), defined by \((n+1)L_{n+1}(x) = (2n+1-x)L_n(x) - nL_{n-1}(x)\), for \( n \geq 0 \). The coefficients are calculated using a hypergeometric recurrence.

void _fmpq_poly_gegenbauer_c(fmpz *coeffs, fmpz_t den, ulong n, const fmpq_t a)
Sets coeffs to the coefficient array of the Gegenbauer (ultraspherical) polynomial \( C_n^{(\alpha)}(x) = \frac{(2\alpha)^n}{n!} {}_2F_1(-n,2\alpha+n;\alpha+\frac{1}{2}; \frac{1-x}{2}) \), for integer \( n \geq 0 \) and rational \( \alpha > 0 \). Sets den to the overall denominator. The coefficients are calculated using a hypergeometric recurrence.

void fmpq_poly_gegenbauer_c(fmpz_poly_t poly, ulong n, const fmpq_t a)
Sets poly to the Gegenbauer (ultraspherical) polynomial \( C_n^{(\alpha)}(x) = \frac{(2\alpha)^n}{n!} {}_2F_1(-n,2\alpha+n;\alpha+\frac{1}{2}; \frac{1-x}{2}) \), for integer \( n \geq 0 \) and rational \( \alpha > 0 \). The coefficients are calculated using a hypergeometric recurrence.

5.4.24 Evaluation

void _fmpq_poly_evaluate_fmpz(fmpz_t rnum, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len, const fmpz_t a)
Evaluates the polynomial \( (poly, den, len) \) at the integer \( a \) and sets \( (rnum, rden) \) to the result in lowest terms.

void fmpq_poly_evaluate_fmpz(fmpz_t res, const fmpz_poly_t poly, const fmpz_t a)
Evaluates the polynomial poly at the integer \( a \) and sets res to the result.

void _fmpq_poly_evaluate_fmpq(fmpz_t rnum, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len, const fmpz_t anum, const fmpz_t aden)
Evaluates the polynomial \( (poly, den, len) \) at the rational \( (anum, aden) \) and sets \( (rnum, rden) \) to the result in lowest terms. Aliasing between \( (rnum, rden) \) and \( (anum, aden) \) is not supported.

void fmpq_poly_evaluate_fmpq(fmpz_t res, const fmpz_poly_t poly, const fmpz_t a)
Evaluates the polynomial poly at the rational \( a \) and sets res to the result.
5.4.25 Interpolation

void _fmpq_poly_interpolate_fmpz_vec(fmpz *poly, fmpz *den, const fmpz *xs, const fmpz *ys, slong n)
  
Sets poly / den to the unique interpolating polynomial of degree at most \( n - 1 \) satisfying \( f(x_i) = y_i \) for every pair \( x_i, y_i \) in xs and ys.

The vector poly must have room for \( n+1 \) coefficients, even if the interpolating polynomial is shorter.

It is assumed that the \( x \) values are distinct.

This function uses a simple \( O(n^2) \) implementation of Lagrange interpolation, clearing denominators to avoid working with fractions. It is currently not designed to be efficient for large \( n \).

void fmpq_poly_interpolate_fmpz_vec(fmpq_poly_t poly, const fmpz *xs, const fmpz *ys, slong n)
  
Sets poly to the unique interpolating polynomial of degree at most \( n - 1 \) satisfying \( f(x_i) = y_i \) for every pair \( x_i, y_i \) in xs and ys. It is assumed that the \( x \) values are distinct.

5.4.26 Composition

void _fmpq_poly_compose(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2)
  
Sets (res, den) to the composition of \( (poly1, den1, len1) \) and \( (poly2, den2, len2) \), assuming \( len1, len2 > 0 \).

Assumes that res has space for \( (len1 - 1) * (len2 - 1) + 1 \) coefficients. Does not support aliasing.

void fmpq_poly_compose(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2)
  
Sets res to the composition of poly1 and poly2.

void _fmpq_poly_rescale(fmpz *res, fmpz_t denr, const fmpz *poly, const fmpz_t den, slong len, const fmpz_t anum, const fmpz_t aden)
  
Sets (res, denr, len) to \( (poly, den, len) \) with the indeterminate rescaled by \( (anum, aden) \).

Assumes that len > 0 and that \( (anum, aden) \) is non-zero and in lowest terms. Supports aliasing between (res, denr, len) and \( (poly, den, len) \).

void fmpq_poly_rescale(fmpq_poly_t res, const fmpq_poly_t poly, const fmpq_t a)
  
Sets res to poly with the indeterminate rescaled by a.

5.4.27 Power series composition

void _fmpq_poly_compose_series_horner(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong n)
  
Sets (res, den, n) to the composition of \( (poly1, den1, len1) \) and \( (poly2, den2, len2) \) modulo \( x^n \), where the constant term of poly2 is required to be zero.

Assumes that \( len1, len2, n > 0 \), that \( len1, len2 <= n \), that \( (len1-1) * (len2-1) + 1 <= n \), and that res has space for \( n \) coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses the Horner scheme. The default fmpq_poly composition algorithm is automatically used when the composition can be performed over the integers.
void _fmpq_poly_compose_series_brent_kung(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t den1, slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong n)

Sets (res, den, n) to the composition of (poly1, den1, len1) and (poly2, den2, len2) modulo $x^n$, where the constant term of poly2 is required to be zero.

Assumes that len1, len2, n > 0, that len1, len2 <= n, that (len1-1) * (len2-1) + 1 <= n, and that res has space for n coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses Brent-Kung algorithm 2.1 [BrentKung1978]. The default fmpz_poly composition algorithm is automatically used when the composition can be performed over the integers.

void fmpq_poly_compose_series_brent_kung(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, slong n)

Sets res to the composition of poly1 and poly2 modulo $x^n$, where the constant term of poly2 is required to be zero.

This implementation uses Brent-Kung algorithm 2.1 [BrentKung1978]. The default fmpz_poly composition algorithm is automatically used when the composition can be performed over the integers.

void _fmpq_poly_compose_series_horner(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, slong n)

Sets res to the composition of poly1 and poly2 modulo $x^n$, where the constant term of poly2 is required to be zero.

This implementation uses the Horner scheme. The default fmpz_poly composition algorithm is automatically used when the composition can be performed over the integers.
5.4.28 Power series reversion

```c
void _fmpq_poly_revert_series_lagrange(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t den1, slong len1, slong n)
```

Sets \((\text{res}, \text{den})\) to the power series reversion of \((\text{poly1}, \text{den1}, \text{len1})\) modulo \(x^n\).

The constant term of \(\text{poly2}\) is required to be zero and the linear term is required to be nonzero. Assumes that \(n > 0\). Does not support aliasing between any of the inputs and the output.

This implementation uses the Lagrange inversion formula. The default \texttt{fmpz\_poly} reversion algorithm is automatically used when the reversion can be performed over the integers.

```c
void fmpq_poly_revert_series_lagrange(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
```

Sets \(\text{res}\) to the power series reversion of \(\text{poly1} \mod x^n\). The constant term of \(\text{poly2}\) is required to be zero and the linear term is required to be nonzero.

This implementation uses the Lagrange inversion formula. The default \texttt{fmpz\_poly} reversion algorithm is automatically used when the reversion can be performed over the integers.

```c
void _fmpq_poly_revert_series_lagrange_fast(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t den1, slong len1, slong n)
```

Sets \((\text{res}, \text{den})\) to the power series reversion of \((\text{poly1}, \text{den1}, \text{len1})\) modulo \(x^n\).

The constant term of \(\text{poly2}\) is required to be zero and the linear term is required to be nonzero.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula. The default \texttt{fmpz\_poly} reversion algorithm is automatically used when the reversion can be performed over the integers.

```c
void fmpq_poly_revert_series_lagrange_fast(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
```

Sets \(\text{res}\) to the power series reversion of \(\text{poly1} \mod x^n\). The constant term of \(\text{poly2}\) is required to be zero and the linear term is required to be nonzero.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula. The default \texttt{fmpz\_poly} reversion algorithm is automatically used when the reversion can be performed over the integers.

```c
void _fmpq_poly_revert_series_newton(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t den1, slong len1, slong n)
```

Sets \((\text{res}, \text{den})\) to the power series reversion of \((\text{poly1}, \text{den1}, \text{len1})\) modulo \(x^n\).

The constant term of \(\text{poly2}\) is required to be zero and the linear term is required to be nonzero.

This implementation uses Newton iteration. The default \texttt{fmpz\_poly} reversion algorithm is automatically used when the reversion can be performed over the integers.

```c
void fmpq_poly_revert_series_newton(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
```

Sets \(\text{res}\) to the power series reversion of \(\text{poly1} \mod x^n\). The constant term of \(\text{poly2}\) is required to be zero and the linear term is required to be nonzero.

This implementation uses Newton iteration. The default \texttt{fmpz\_poly} reversion algorithm is automatically used when the reversion can be performed over the integers.

```c
void _fmpq_poly_revert_series(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t den1, slong len1, slong n)
```

Sets \((\text{res}, \text{den})\) to the power series reversion of \((\text{poly1}, \text{den1}, \text{len1})\) modulo \(x^n\).

The constant term of \(\text{poly2}\) is required to be zero and the linear term is required to be nonzero.

This implementation defaults to using Newton iteration. The default \texttt{fmpz\_poly} reversion algorithm is automatically used when the reversion can be performed over the integers.
void fmpq_poly_revert_series(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
    Sets res to the power series reversion of poly1 modulo $x^n$. The constant term of poly2 is required to be zero and the linear term is required to be nonzero.

This implementation defaults to using Newton iteration. The default fmpz_poly reversion algorithm is automatically used when the reversion can be performed over the integers.

5.4.29 Gaussian content

void _fmpq_poly_content(fmpz_t res, const fmpz *poly, const fmpz_t den, slong len)
    Sets res to the content of (poly, den, len). If len == 0, sets res to zero.

void fmpq_poly_content(fmpq_t res, const fmpq_poly_t poly)
    Sets res to the content of poly. The content of the zero polynomial is defined to be zero.

void _fmpq_poly_primitive_part(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len)
    Sets (rpoly, rden, len) to the primitive part, with non-negative leading coefficient, of (poly, den, len). Assumes that len > 0. Supports aliasing between the two polynomials.

void fmpq_poly_primitive_part(fmpq_poly_t res, const fmpq_poly_t poly)
    Sets res to the primitive part, with non-negative leading coefficient, of poly.

int _fmpq_poly_is_monic(const fmpz *poly, const fmpz_t den, slong len)
    Returns whether the polynomial (poly, den, len) is monic. The zero polynomial is not monic by definition.

int fmpq_poly_is_monic(const fmpq_poly_t poly)
    Returns whether the polynomial poly is monic. The zero polynomial is not monic by definition.

void _fmpq_poly_make_monic(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len)
    Sets (rpoly, rden, len) to the monic scalar multiple of (poly, den, len). Assumes that len > 0. Supports aliasing between the two polynomials.

void fmpq_poly_make_monic(fmpq_poly_t res, const fmpq_poly_t poly)
    Sets res to the monic scalar multiple of poly whenever poly is non-zero. If poly is the zero polynomial, sets res to zero.

5.4.30 Square-free

int fmpq_poly_is_squarefree(const fmpq_poly_t poly)
    Returns whether the polynomial poly is square-free. A non-zero polynomial is defined to be square-free if it has no non-unit square factors. We also define the zero polynomial to be square-free.

5.4.31 Input and output

int _fmpq_poly_print(const fmpz *poly, const fmpz_t den, slong len)
    Prints the polynomial (poly, den, len) to stdout. In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fmpq_poly_print(const fmpq_poly_t poly)
    Prints the polynomial to stdout. In case of success, returns a positive value. In case of failure, returns a non-positive value.
int `_fmpq_poly_print_pretty`(const `fmpz` *poly, const `fmpz_t` den, `slong` len, const char *x)

int _fmpq_poly_print_pretty(const `fmpq_poly_t` poly, const char *var)

    Prints the pretty representation of `poly` to `stdout`, using the null-terminated string `var` not equal to "\0" as the variable name.

    In the current implementation always returns 1.

int _fmpq_poly_fprint(FILE *file, const `fmpz` *poly, const `fmpz_t` den, `slong` len)

    Prints the polynomial `(poly, den, len)` to the stream `file`.

    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fmpq_poly_fprint_pretty(FILE *file, const `fmpz` *poly, const `fmpz_t` den, `slong` len, const char *x)

int fmpq_poly_print_pretty(FILE *file, const `fmpq_poly_t` poly, const char *var)

    Prints the pretty representation of `poly` to `stdout`, using the null-terminated string `var` not equal to "\0" as the variable name.

    In the current implementation, always returns 1.

int fmpq_poly_read(`fmpq_poly_t` poly)

    Reads a polynomial from `stdin`, storing the result in `poly`.

    In case of success, returns a positive number. In case of failure, returns a non-positive value.

int fmpq_poly_fread(FILE *file, `fmpq_poly_t` poly)

    Reads a polynomial from the stream `file`, storing the result in `poly`.

    In case of success, returns a positive number. In case of failure, returns a non-positive value.

5.5 `fmpq_mpoly_factor.h` – factorisation of multivariate polynomials over the rational numbers

5.5.1 Types, macros and constants

type `fmpq_mpoly_factor_struct`

    A struct for holding a factored rational polynomial. There is a single constant and a product of bases to corresponding exponents.

type `fmpq_mpoly_factor_t`

    An array of length 1 of `fmpq_mpoly_factor_struct`.

5.5.2 Memory management

void `fmpq_mpoly_factor_init`(`fmpq_mpoly_factor_t` f, const `fmpq_mpoly_ctx_t` ctx)

    Initialise `f`.

void `fmpq_mpoly_factor_clear`(`fmpq_mpoly_factor_t` f, const `fmpq_mpoly_ctx_t` ctx)

    Clear `f`.
5.5.3 Basic manipulation

```c
slong fmpz_mpoly_factor_length(const fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)
   Return the length of the product in f.

void fmpz_mpoly_factor_get_constant_fmpz(fmpz_t c, const fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)
   Set c to the constant of f.

void fmpz_mpoly_factor_get_base(fmpz_mpoly_t B, const fmpz_mpoly_factor_t f, slong i, const fmpz_mpoly_ctx_t ctx)
   Set (resp. swap) B to (resp. with) the base of the term of index i in A.

slong fmpz_mpoly_factor_get_exp_si(fmpz_mpoly_factor_t f, slong i, const fmpz_mpoly_ctx_t ctx)
   Return the exponent of the term of index i in A. It is assumed to fit an slong.

void fmpz_mpoly_factor_sort(fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)
   Sort the product of f first by exponent and then by base.

int fmpz_mpoly_factor_make_monic(fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)
int fmpz_mpoly_factor_make_integral(fmpz_mpoly_factor_t f, const fmpz_mpoly_ctx_t ctx)
   Make the bases in f monic (resp. integral and primitive with positive leading coefficient). Return 1 for success, 0 for failure.
```

5.5.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and f is undefined. None of these functions multiply f by A; f is simply set to a factorisation of A, and thus these functions should not depend on the initial value of the output f. The normalization of the factors is not yet specified: use `fmpz_mpoly_factor_make_monic()` or `fmpz_mpoly_factor_make_integral()` for common normalizations.

```c
int fmpz_mpoly_factor_squarefree(fmpz_mpoly_factor_t f, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
   Set f to a factorization of A where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call `fmpz_mpoly_factor_sort()` and then multiply the bases with the desired exponent.

int fmpz_mpoly_factor(fmpz_mpoly_factor_t f, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
   Set f to a factorization of A where the bases are irreducible.
```

5.6 fmpz_mpoly.h – multivariate polynomials over the rational numbers

The exponents follow the `mpoly` interface. No references to the coefficients are available.
5.6.1 Types, macros and constants

define enum fmpq_mpoly_struct

A structure holding a multivariate rational polynomial. It is implemented as a fmpq_t holding the
content of the polynomial and a primitive integer polynomial.

define enum fmpq_mpoly_t

An array of length 1 of fmpq_mpoly_struct.

define enum fmpq_mpoly_ctx_struct

Context structure representing the parent ring of an fmpq_mpoly.

define enum fmpq_mpoly_ctx_t

An array of length 1 of fmpq_mpoly_ctx_struct.

5.6.2 Context object

define void fmpq_mpoly_ctx_init

Initialise a context object for a polynomial ring with the given number of variables and the given
ordering. The possibilities for the ordering are ORD_LEX, ORD_DEGLEX and ORD_DEGREVLEX.

define slong fmpq_mpoly_ctx_nvars

Return the number of variables used to initialize the context.

define ordering_t fmpq_mpoly_ctx_ord

Return the ordering used to initialize the context.

define void fmpq_mpoly_ctx_clear

Release up any space allocated by ctx.

5.6.3 Memory management

define void fmpq_mpoly_init

Initialise A for use with the given and initialised context object. Its value is set to zero.

define void fmpq_mpoly_init2

Initialise A for use with the given and initialised context object. Its value is set to zero. It is
allocated with space for alloc terms and at least MPOLY_MIN_BITS bits for the exponents.

define void fmpq_mpoly_init3

Initialise A for use with the given and initialised context object. Its value is set to zero. It is
allocated with space for alloc terms and bits bits for the exponents.

define void fmpq_mpoly_fit_length

Ensure that A has space for at least len terms.

define void fmpq_mpoly_fit_bits

Ensure that the exponent fields of A have at least bits bits.

define void fmpq_mpoly_realloc

Reallocate A to have space for alloc terms. Assumes the current length of the polynomial is not
greater than alloc.

define void fmpq_mpoly_clear

Release any space allocated for A.
5.6.4 Input/Output

The variable strings in x start with the variable of most significance at index 0. If x is NULL, the variables are named x1, x2, etc.

```c
char *fmpq_mpoly_get_str_pretty(const fmpq_mpoly_t A, const char **x, const fmpq_mpoly_ctx_t ctx)
```

Return a string, which the user is responsible for cleaning up, representing A, given an array of variable strings x.

```c
int fmpq_mpoly_fprint_pretty(FILE *file, const fmpq_mpoly_t A, const char **x, const fmpq_mpoly_ctx_t ctx)
```

Print a string representing A to file.

```c
int fmpq_mpoly_print_pretty(const fmpq_mpoly_t A, const char **x, const fmpq_mpoly_ctx_t ctx)
```

Print a string representing A to stdout.

```c
int fmpq_mpoly_set_str_pretty(fmpq_mpoly_t A, const char *str, const char **x, const fmpq_mpoly_ctx_t ctx)
```

Set A to the polynomial in the null-terminates string str given an array x of variable strings. If parsing str fails, A is set to zero, and -1 is returned. Otherwise, 0 is returned. The operations +, -, *, and / are permitted along with integers and the variables in x. The character ^ must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

5.6.5 Basic manipulation

```c
void fmpq_mpoly_gen(fmpq_mpoly_t A, slong var, const fmpq_mpoly_ctx_t ctx)
```

Set A to the variable of index var, where var = 0 corresponds to the variable with the most significance with respect to the ordering.

```c
int fmpq_mpoly_is_gen(const fmpq_mpoly_t A, slong var, const fmpq_mpoly_ctx_t ctx)
```

If var ≥ 0, return 1 if A is equal to the var-th generator, otherwise return 0. If var < 0, return 1 if the polynomial is equal to any generator, otherwise return 0.

```c
void fmpq_mpoly_set(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)
```

Set A to B.

```c
int fmpq_mpoly_equal(const fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)
```

Return 1 if A is equal to B, else return 0.

```c
void fmpq_mpoly_swap(fmpq_mpoly_t A, fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)
```

Efficiently swap A and B.

5.6.6 Constants

```c
int fmpq_mpoly_is_fmpq(const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Return 1 if A is a constant, else return 0.

```c
void fmpq_mpoly_get_fmpq(fmpq_t c, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Assuming that A is a constant, set c to this constant. This function throws if A is not a constant.

```c
void fmpq_mpoly_set_fmpq(fmpq_mpoly_t A, const fmpq_t c, const fmpq_mpoly_ctx_t ctx)
```

```c
void fmpq_mpoly_set_fmpz(fmpq_mpoly_t A, const fmpz_t c, const fmpq_mpoly_ctx_t ctx)
```

```c
void fmpq_mpoly_set_ui(fmpq_mpoly_t A, ulong c, const fmpq_mpoly_ctx_t ctx)
```

```c
void fmpq_mpoly_set_si(fmpq_mpoly_t A, slong c, const fmpq_mpoly_ctx_t ctx)
```

Set A to the constant c.
void fmpq_mpoly_zero(fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    Set A to the constant 0.

void fmpq_mpoly_one(fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    Set A to the constant 1.

int fmpq_mpoly_equal_fmpq(const fmpq_mpoly_t A, const fmpq_t c, const fmpq_mpoly_ctx_t ctx)
    Return 1 if A is equal to the constant c, else return 0.

int fmpq_mpoly_equal_fmpz(const fmpq_mpoly_t A, const fmpz_t c, const fmpq_mpoly_ctx_t ctx)
    Return 1 if A is equal to the constant 0, else return 0.

int fmpq_mpoly_equal_ui(const fmpq_mpoly_t A, ulong c, const fmpq_mpoly_ctx_t ctx)
    Return 1 if A is equal to the constant 1, else return 0.

int fmpq_mpoly_equal_si(const fmpq_mpoly_t A, slong c, const fmpq_mpoly_ctx_t ctx)
    Return 1 if A is equal to the constant 0, else return 0.

5.6.7 Degrees

int fmpq_mpoly_degrees_fit_si(const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    Return 1 if the degrees of A with respect to each variable fit into an slong, otherwise return 0.

void fmpq_mpoly_degrees_fmpz(fmpz **degs, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
void fmpq_mpoly_degrees_si(slong *degs, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    Set degs to the degrees of A with respect to each variable. If A is zero, all degrees are set to −1.

void fmpq_mpoly_degree_fmpz(fmpz_t deg, const fmpq_mpoly_t A, slong var, const fmpq_mpoly_ctx_t ctx)

slong fmpq_mpoly_degree_si(const fmpq_mpoly_t A, slong var, const fmpq_mpoly_ctx_t ctx)
    Either return or set deg to the degree of A with respect to the variable of index var. If A is zero, the degree is defined to be −1.

int fmpq_mpoly_total_degree_fits_si(const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    Return 1 if the total degree of A fits into an slong, otherwise return 0.

void fmpq_mpoly_total_degree_fmpz(fmpz_t tdeg, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)

slong fmpq_mpoly_total_degree_si(const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    Either return or set tdeg to the total degree of A. If A is zero, the total degree is defined to be −1.

void fmpq_mpoly_used_vars(int *used, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    For each variable index i, set used[i] to nonzero if the variable of index i appears in A and to zero otherwise.

5.6.8 Coefficients

void fmpq_mpoly_get_denominator(fmpz_t d, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    Set d to the denominator of A, the smallest positive integer d such that d×A has integer coefficients.

void fmpq_mpoly_get_coeff_fmpq_monomial(fmpq_t c, const fmpq_mpoly_t A, const fmpq_mpoly_t M, const fmpq_mpoly_ctx_t ctx)
    Assuming that M is a monomial, set c to the coefficient of the corresponding monomial in A. This function throws if M is not a monomial.
void fmpq_mpoly_set_coeff_fmpq_monomial(fmpq_mpoly_t A, const fmpq_t c, const fmpq_mpoly_t M, const fmpq_mpoly_ctx_t ctx)

Assuming that \( M \) is a monomial, set the coefficient of the corresponding monomial in \( A \) to \( c \). This function throws if \( M \) is not a monomial.

void fmpq_mpoly_get_coeff_fmpq_fmpz(fmpq_t c, const fmpq_mpoly_t A, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_get_coeff_fmpq_ui(fmpq_t c, const fmpq_mpoly_t A, const ulong *exp, const fmpq_mpoly_ctx_t ctx)

Set \( c \) to the coefficient of the monomial with exponent \( \text{exp} \).

void fmpq_mpoly_set_coeff_fmpq_fmpz(fmpq_mpoly_t A, const fmpq_t c, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_set_coeff_fmpq_ui(fmpq_mpoly_t A, const fmpq_t c, const ulong *exp, const fmpq_mpoly_ctx_t ctx)

Set the coefficient of the monomial with exponent \( \text{exp} \) to \( c \).

void fmpq_mpoly_get_coeff_vars_ui(fmpq_mpoly_t C, const fmpq_mpoly_t A, const slong *vars, const ulong *exps, slong length, const fmpq_mpoly_ctx_t ctx)

Set \( C \) to the coefficient of \( A \) with respect to the variables in \( \text{vars} \) with powers in the corresponding array \( \text{exps} \). Both \( \text{vars} \) and \( \text{exps} \) point to array of length \( \text{length} \). It is assumed that \( 0 < \text{length} \leq nvars(A) \) and that the variables in \( \text{vars} \) are distinct.

5.6.9 Comparison

int fmpq_mpoly_cmp(const fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)

Return 1 (resp. -1, or 0) if \( A \) is after (resp. before, same as) \( B \) in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when \( A \) and \( B \) are both monomials.

5.6.10 Container operations

These functions try to deal efficiently with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw. The mutating functions here are not guaranteed to leave the polynomial in reduced form (see fmpq_mpoly_is_canonical() for a definition of reduced). This means that even if nonzero terms with distinct exponents have been constructed in the correct order, a call to fmpq_mpoly_reduce() is necessary to ensure that the polynomial is in canonical form. As with the fmpz_mpoly module, a call to fmpq_mpoly_sort_terms() followed by a call to fmpq_mpoly_combine_like_terms() should leave the polynomial in canonical form.

fmpq *fmpq_mpoly_content_ref(fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)

Return a reference to the content of \( A \).

fmpz_mpoly_struct *fmpq_mpoly_zpoly_ref(fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)

Return a reference to the integer polynomial of \( A \).

fmpz *fmpq_mpoly_zpoly_term_coeff_ref(fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)

Return a reference to the coefficient of index \( i \) of the integer polynomial of \( A \).

int fmpq_mpoly_is_canonical(const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)

Return 1 if \( A \) is in canonical form. Otherwise, return 0. An fmpq_mpoly_t is represented as the product of an fmpq_t content and an fmpz_mpoly_t zpoly. The representation is considered canonical when either (1) both content and zpoly are zero, or (2) both content and zpoly are nonzero and canonical and zpoly is reduced. A nonzero zpoly is considered reduced when the coefficients have GCD one and the leading coefficient is positive.

5.6. fmpq_mpoly.h – multivariate polynomials over the rational numbers
slong fmpq_mpoly_length(const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)

Return the number of terms stored in A. If the polynomial is in canonical form, this will be the number of nonzero coefficients.

void fmpq_mpoly_resize(fmpq_mpoly_t A, slong new_length, const fmpq_mpoly_ctx_t ctx)

Set the length of A to new_length. Terms are either deleted from the end, or new zero terms are appended.

void fmpq_mpoly_get_term_coeff_fmpq(fmpq_t c, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)

Set c to coefficient of index i

void fmpq_mpoly_set_term_coeff_fmpq(fmpq_mpoly_t A, slong i, const fmpq_t c, const fmpq_mpoly_ctx_t ctx)

Set the coefficient of index i to c.

int fmpq_mpoly_term_exp_fits_si(const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)

int fmpq_mpoly_term_exp_fits_ui(const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)

Return 1 if all entries of the exponent vector of the term of index i fit into an slong (resp. a ulong). Otherwise, return 0.

void fmpq_mpoly_get_term_exp_fmpz(fmpz **exps, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_get_term_exp_ui(ulong *exps, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_get_term_exp_si(slong *exps, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)

Set exp to the exponent vector of the term of index i. The _ui (resp. _si) version throws if any entry does not fit into a ulong (resp. slong).

ulong fmpq_mpoly_get_term_var_exp_ui(const fmpq_mpoly_t A, slong i, slong var, const fmpq_mpoly_ctx_t ctx)

slong fmpq_mpoly_get_term_var_exp_si(const fmpq_mpoly_t A, slong i, slong var, const fmpq_mpoly_ctx_t ctx)

Return the exponent of the variable var of the term of index i. This function throws if the exponent does not fit into a ulong (resp. slong).

void fmpq_mpoly_set_term_exp_fmpz(fmpq_mpoly_t A, slong i, fmpz *const *exps, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_set_term_exp_ui(fmpq_mpoly_t A, slong i, const ulong *exps, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_set_term_exp_si(fmpq_mpoly_t A, slong i, const slong *exps, const fmpq_mpoly_ctx_t ctx)

Set the exponent vector of the term of index i to exp.

void fmpq_mpoly_get_term(fmpq_mpoly_t M, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)

Set M to the term of index i in A.

void fmpq_mpoly_get_term_monomial(fmpq_mpoly_t M, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)

Set M to the monomial of the term of index i in A. The coefficient of M will be one.

void fmpq_mpoly_push_term_fmpq_fmpz(fmpq_mpoly_t A, const fmpq_t c, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_push_term_fmpq_ffmpz(fmpq_mpoly_t A, const fmpq_t c, const fmpz *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_push_term_fmpz_fmpz(fmpq_mpoly_t A, const fmpz_t c, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)
void fmpq_mpoly_push_term_fmpz_ffmpz(fmpq_mpoly_t A, const fmpz_t c, const fmpz *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_push_term_ui_fmpz(fmpq_mpoly_t A, ulong c, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_push_term_ui_ffmpz(fmpq_mpoly_t A, ulong c, const fmpz *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_push_term_si_fmpz(fmpq_mpoly_t A, slong c, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_push_term_si_ffmpz(fmpq_mpoly_t A, slong c, const fmpz *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_push_term_fmpq_ui(fmpq_mpoly_t A, const fmpq_t c, const ulong *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_push_term_fmpz_ui(fmpq_mpoly_t A, const fmpz_t c, const ulong *exp, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_push_term_ui_ui(fmpq_mpoly_t A, ulong c, const ulong *exp, const fmpq_mpoly_ctx_t ctx)

5.6.11 Random generation

void fmpq_mpoly_randtest_bound(fmpq_mpoly_t A, flint_rand_t state, slong length, mp_limb_t coeff_bits, ulong exp_bound, const fmpq_mpoly_ctx_t ctx)

Generate a random polynomial with length up to length and exponents in the range $[0, \text{exp\_bound} - 1]$. The exponents of each variable are generated by calls to n_randint(state, exp\_bound).

void fmpq_mpoly_randtest_bounds(fmpq_mpoly_t A, flint_rand_t state, slong length, mp_limb_t coeff_bits, ulong *exp\_bounds, const fmpq_mpoly_ctx_t ctx)

Generate a random polynomial with length up to length and exponents whose packed form does not exceed the given bit count. The parameter coeff_bits to the three functions fmpq_mpoly_randtest_{bound|bounds|bits} is merely a suggestion for the approximate bit count of the resulting coefficients.

void fmpq_mpoly_randtest_bits(fmpq_mpoly_t A, flint_rand_t state, slong length, mp_limb_t coeff_bits, mp_limb_t *exp\_bits, const fmpq_mpoly_ctx_t ctx)

Generate a random polynomial with length up to length and exponents whose packed form does not exceed the given bit count. The parameter coeff_bits to the three functions fmpq_mpoly_randtest_{bound|bounds|bits} is merely a suggestion for the approximate bit count of the resulting coefficients.
5.6.12 Addition/Subtraction

void fmpq_mpoly_add_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_t c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_add_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_add_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_add_si(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong c, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B + c$.

void fmpq_mpoly_sub_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_t c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_sub_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_sub_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_sub_si(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong c, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B - c$.

void fmpq_mpoly_add(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_t C, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B + C$.

void fmpq_mpoly_sub(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_t C, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B - C$.

5.6.13 Scalar operations

void fmpq_mpoly_neg(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $-B$.

void fmpq_mpoly_scalar_mul_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_t c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_scalar_mul_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_scalar_mul_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_scalar_mul_si(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong c, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B \times c$.

void fmpq_mpoly_scalar_div_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_t c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_scalar_div_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_scalar_div_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong c, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_scalar_div_si(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong c, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B/c$. 

320 Chapter 5. Rational numbers
void fmpq_mpoly_make_monic(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B$ divided by the leading coefficient of $B$. This throws if $B$ is zero.

All of these functions run quickly if $A$ and $B$ are aliased.

### 5.6.14 Differentiation/Integration

void fmpq_mpoly_derivative(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong var, const fmpq_mpoly_ctx_t ctx)

Set $A$ to the derivative of $B$ with respect to the variable of index $var$.

void fmpq_mpoly_integral(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong var, const fmpq_mpoly_ctx_t ctx)

Set $A$ to the integral with the fewest number of terms of $B$ with respect to the variable of index $var$.

### 5.6.15 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

int fmpq_mpoly_evaluate_all_fmpq(fmpq_t ev, const fmpq_mpoly_t A, fmpq *const *vals, const fmpq_mpoly_ctx_t ctx)

Set $ev$ to the evaluation of $A$ where the variables are replaced by the corresponding elements of the array $vals$. Return 1 for success and 0 for failure.

int fmpq_mpoly_evaluate_one_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong var, const fmpq_t val, const fmpq_mpoly_ctx_t ctx)

Set $A$ to the evaluation of $B$ where the variable of index $var$ is replaced by $val$. Return 1 for success and 0 for failure.

int fmpq_mpoly_compose_fmpq_poly(fmpq_poly_t A, const fmpq_mpoly_t B, fmpq_poly_struct *const *C, const fmpq_mpoly_ctx_t ctxB)

Set $A$ to the evaluation of $B$ where the variables are replaced by the corresponding elements of the array $C$. The context object of $B$ is $ctxB$. Return 1 for success and 0 for failure.

int fmpq_mpoly_compose_fmpq_mpoly(fmpq_mpoly_t A, const fmpq_mpoly_t B, fmpq_mpoly_struct *const *C, const fmpq_mpoly_ctx_t ctxB, const fmpq_mpoly_ctx_t ctxAC)

Set $A$ to the evaluation of $B$ where the variables are replaced by the corresponding elements of the array $C$. Both $A$ and the elements of $C$ have context object $ctxAC$, while $B$ has context object $ctxB$. Neither $A$ nor $B$ is allowed to alias any other polynomial. Return 1 for success and 0 for failure.

void fmpq_mpoly_compose_fmpq_mpoly_gen(fmpq_mpoly_t A, const fmpq_mpoly_t B, const slong *c, const fmpq_mpoly_ctx_t ctxB, const fmpq_mpoly_ctx_t ctxAC)

Set $A$ to the evaluation of $B$ where the variable of index $i$ in $ctxB$ is replaced by the variable of index $c[i]$ in $ctxAC$. The length of the array $C$ is the number of variables in $ctxB$. If any $c[i]$ is negative, the corresponding variable of $B$ is replaced by zero. Otherwise, it is expected that $c[i]$ is less than the number of variables in $ctxAC$.
5.6.16 Multiplication

void fmpq_mpoly_mul(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_t C, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B \times C$.

5.6.17 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

int fmpq_mpoly_pow_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t k, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B$ raised to the $k$-th power. Return 1 for success and 0 for failure.

int fmpq_mpoly_pow_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong k, const fmpq_mpoly_ctx_t ctx)

Set $A$ to $B$ raised to the $k$-th power. Return 1 for success and 0 for failure.

5.6.18 Division

int fmpq_mpoly_divides(fmpq_mpoly_t Q, const fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)

If $A$ is divisible by $B$, set $Q$ to the exact quotient and return 1. Otherwise, set $Q$ to zero and return 0. Note that the function fmpq_mpoly_div() may be faster if the quotient is known to be exact.

void fmpq_mpoly_div(fmpq_mpoly_t Q, const fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)

Set $Q$ to the quotient of $A$ by $B$, discarding the remainder.

void fmpq_mpoly_divrem(fmpq_mpoly_t Q, fmpq_mpoly_t R, const fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)

Set $Q$ and $R$ to the quotient and remainder of $A$ divided by $B$.

void fmpq_mpoly_divrem_ideal(fmpq_mpoly_struct **Q, fmpq_mpoly_t R, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)

This function is as per fmpq_mpoly_divrem() except that it takes an array of divisor polynomials $B$ and it returns an array of quotient polynomials $Q$. The number of divisor (and hence quotient) polynomials is given by len.

5.6.19 Greatest Common Divisor

void fmpq_mpoly_content(fmpq_t g, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)

Set $g$ to the (nonnegative) gcd of the coefficients of $A$.

void fmpq_mpoly_term_content(fmpq_mpoly_t M, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)

Set $M$ to the GCD of the terms of $A$. If $A$ is zero, $M$ will be zero. Otherwise, $M$ will be a monomial with coefficient one.

int fmpq_mpoly_content_vars(fmpq_mpoly_t g, const fmpq_mpoly_t A, slong *vars, slong vars_length, const fmpq_mpoly_ctx_t ctx)

Set $g$ to the GCD of the coefficients of $A$ when viewed as a polynomial in the variables $vars$. Return 1 for success and 0 for failure. Upon success, $g$ will be independent of the variables $vars$. 
Try to set $G$ to the monic GCD of $A$ and $B$. The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and $G$ is left untouched.

Do the operation of $fmpq_mpoly_gcd()$ and also compute $A\bar{b} = A/G$ and $B\bar{b} = B/G$ if successful.

Try to set $G$ to the GCD of $A$ and $B$ using various algorithms.

Try to set $R$ to the resultant of $A$ and $B$ with respect to the variable of index $var$.

Try to set $D$ to the discriminant of $A$ with respect to the variable of index $var$.

If $A$ is a perfect square return 1 and set $Q$ to the square root with positive leading coefficient. Otherwise return 0 and set $Q$ to zero.

Return 1 if $A$ is a perfect square, otherwise return 0.

An $fmpq_mpoly_univar_t$ holds a univariate polynomial in some main variable with $fmpq_mpoly_t$ coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of $\mathbb{Q}[x_1,\ldots,x_m]$ as an element of $(\mathbb{Q}[x_1,\ldots,x_{v-1},x_{v+1},\ldots,x_m])[x_v]$ and vice versa.

Initialize $A$.

Clear $A$.

Swap $A$ and $B$. 

5.6.21 Univariate Functions

5.6. fmpq_mpoly.h – multivariate polynomials over the rational numbers
Flint Documentation, Release 3.1.0-dev

void fmpq_mpoly_to_univar(fmpq_mpoly_univar_t A, const fmpq_mpoly_t B, slong var, const fmpq_mpoly_ctx_t ctx)

Set A to a univariate form of B by pulling out the variable of index var. The coefficients of A will still belong to the content ctx but will not depend on the variable of index var.

void fmpq_mpoly_from_univar(fmpq_mpoly_t A, const fmpq_mpoly_univar_t B, slong var, const fmpq_mpoly_ctx_t ctx)

Set A to the normal form of B by putting in the variable of index var. This function is undefined if the coefficients of B depend on the variable of index var.

int fmpq_mpoly_univar_degree_fits_si(const fmpq_mpoly_univar_t A, const fmpq_mpoly_ctx_t ctx)

Return 1 if the degree of A with respect to the main variable fits an slong. Otherwise, return 0.

slong fmpq_mpoly_univar_length(const fmpq_mpoly_univar_t A, const fmpq_mpoly_ctx_t ctx)

Return the number of terms in A with respect to the main variable.

slong fmpq_mpoly_univar_get_term_exp_si(fmpq_mpoly_univar_t A, slong i, const fmpq_mpoly_ctx_t ctx)

Return the exponent of the term of index i of A.

void fmpq_mpoly_univar_get_term_coeff(fmpq_mpoly_t c, const fmpq_mpoly_univar_t A, slong i, const fmpq_mpoly_ctx_t ctx)

void fmpq_mpoly_univar_swap_term_coeff(fmpq_mpoly_t c, fmpq_mpoly_univar_t A, slong i, const fmpq_mpoly_ctx_t ctx)

Set (resp. swap) c to (resp. with) the coefficient of the term of index i of A.

5.7 fmpz_poly_q.h – rational functions over the rational numbers

The module fmpz_poly_q provides functions for performing arithmetic on rational functions in \( \mathbb{Q}(t) \), represented as quotients of integer polynomials of type fmpz_poly_t. These functions start with the prefix fmpz_poly_q_.

Rational functions are stored in objects of type fmpz_poly_q_t, which is an array of fmpz_poly_q_struct’s of length one. This permits passing parameters of type fmpz_poly_q_t by reference.

The representation of a rational function as the quotient of two integer polynomials can be made canonical by demanding the numerator and denominator to be coprime (as integer polynomials) and the denominator to have positive leading coefficient. As the only special case, we represent the zero function as \( 0/1 \). All arithmetic functions assume that the operands are in this canonical form, and canonicalize their result. If the numerator or denominator is modified individually, for example using the macros fmpz_poly_q_numref() and fmpz_poly_q_denref(), it is the user’s responsibility to canonicalize the rational function using the function fmpz_poly_qcanonicalise() if necessary.

All methods support aliasing of their inputs and outputs unless explicitly stated otherwise, subject to the following caveat. If different rational functions (as objects in memory, not necessarily in the mathematical sense) share some of the underlying integer polynomial objects, the behaviour is undefined.

The basic arithmetic operations, addition, subtraction and multiplication, are all implemented using adapted versions of Henrici’s algorithms, see [Hen1956]. Differentiation is implemented in a way slightly improving on the algorithm described in [Hor1972].
5.7.1 Simple example

The following example computes the product of two rational functions and prints the result:

```c
#include "fmpz_poly_q.h"
int main()
{
    char * str, * strf, * strg;
    fmpz_poly_q_t f, g;
    fmpz_poly_q_init(f);
    fmpz_poly_q_init(g);
    fmpz_poly_q_set_str(f, "2 1 3/1 2");
    fmpz_poly_q_set_str(g, "1 3/2 2 7");
    strf = fmpz_poly_q_get_str_pretty(f, "t");
    strg = fmpz_poly_q_get_str_pretty(g, "t");
    fmpz_poly_q_mul(f, f, g);
    str = fmpz_poly_q_get_str_pretty(f, "t");
    flint_printf("%s * %s = %s\n", strf, strg, str);
    free(str);
    free(strf);
    free(strg);
    fmpz_poly_q_clear(f);
    fmpz_poly_q_clear(g);
}
```

The output is:

\[(3t+1)/2 \times 3/(7t+2) = (9t+3)/(14t+4)\]

5.7.2 Types, macros and constants

- **type** fmpz_poly_q_struct
- **type** fmpz_poly_q_t

5.7.3 Memory management

- **void** fmpz_poly_q_init(fmpz_poly_q_t rop)
  - Initialises rop.
- **void** fmpz_poly_q_clear(fmpz_poly_q_t rop)
  - Clears the object rop.
- **fmpz_poly_struct** *fmpz_poly_q_numref(const fmpz_poly_q_t op)
  - Returns a reference to the numerator of \(op\).
- **fmpz_poly_struct** *fmpz_poly_q_denref(const fmpz_poly_q_t op)
  - Returns a reference to the denominator of \(op\).
- **void** fmpz_poly_q_canonicalise(fmpz_poly_q_t rop)
  - Brings \(rop\) into canonical form, only assuming that the denominator is non-zero.
- **int** fmpz_poly_q_is_canonical(const fmpz_poly_q_t op)
  - Checks whether the rational function \(op\) is in canonical form.
5.7.4 Randomisation

void fmpz_poly_q_randtest(fmpz_poly_q_t poly, flint_rand_t state, slong len1, flint_bitcnt_t bits1, slong len2, flint_bitcnt_t bits2)

Sets poly to a random rational function.

void fmpz_poly_q_randtest_not_zero(fmpz_poly_q_t poly, flint_rand_t state, slong len1, flint_bitcnt_t bits1, slong len2, flint_bitcnt_t bits2)

Sets poly to a random non-zero rational function.

5.7.5 Assignment

void fmpz_poly_q_set(fmpz_poly_q_t rop, const fmpz_poly_q_t op)

Sets the element rop to the same value as the element op.

void fmpz_poly_q_set_si(fmpz_poly_q_t rop, slong op)

Sets the element rop to the value given by the slong op.

void fmpz_poly_q_swap(fmpz_poly_q_t op1, fmpz_poly_q_t op2)

Swaps the elements op1 and op2.

   This is done efficiently by swapping pointers.

void fmpz_poly_q_zero(fmpz_poly_q_t rop)

Sets rop to zero.

void fmpz_poly_q_one(fmpz_poly_q_t rop)

Sets rop to one.

void fmpz_poly_q_neg(fmpz_poly_q_t rop, const fmpz_poly_q_t op)

Sets the element rop to the additive inverse of op.

void fmpz_poly_q_inv(fmpz_poly_q_t rop, const fmpz_poly_q_t op)

Sets the element rop to the multiplicative inverse of op.

   Assumes that the element op is non-zero.

5.7.6 Comparison

int fmpz_poly_q_is_zero(const fmpz_poly_q_t op)

   Returns whether the element op is zero.

int fmpz_poly_q_is_one(const fmpz_poly_q_t op)

   Returns whether the element rop is equal to the constant polynomial 1.

int fmpz_poly_q_equal(const fmpz_poly_q_t op1, const fmpz_poly_q_t op2)

   Returns whether the two elements op1 and op2 are equal.
5.7.7 Addition and subtraction

void fmpz_poly_q_add(fmpz_poly_q_t rop, const fmpz_poly_q_t op1, const fmpz_poly_q_t op2)
        Sets rop to the sum of op1 and op2.

void fmpz_poly_q_sub(fmpz_poly_q_t rop, const fmpz_poly_q_t op1, const fmpz_poly_q_t op2)
        Sets rop to the difference of op1 and op2.

void fmpz_poly_q_addmul(fmpz_poly_q_t rop, const fmpz_poly_q_t op1, const fmpz_poly_q_t op2)
        Adds the product of op1 and op2 to rop.

void fmpz_poly_q_submul(fmpz_poly_q_t rop, const fmpz_poly_q_t op1, const fmpz_poly_q_t op2)
        Subtracts the product of op1 and op2 from rop.

5.7.8 Scalar multiplication and division

void fmpz_poly_q_scalar_mul_si(fmpz_poly_q_t rop, const fmpz_poly_q_t op, slong x)
        Sets rop to the product of the rational function op and the slong integer x.

void fmpz_poly_q_scalar_mul_fmpz(fmpz_poly_q_t rop, const fmpz_poly_q_t op, const fmpz_t x)
        Sets rop to the product of the rational function op and the fmpz_t integer x.

void fmpz_poly_q_scalar_mul_fmpq(fmpz_poly_q_t rop, const fmpz_poly_q_t op, const fmpq_t x)
        Sets rop to the product of the rational function op and the fmpq_t rational x.

void fmpz_poly_q_scalar_div_si(fmpz_poly_q_t rop, const fmpz_poly_q_t op, slong x)
        Sets rop to the quotient of the rational function op and the slong integer x.

void fmpz_poly_q_scalar_div_fmpz(fmpz_poly_q_t rop, const fmpz_poly_q_t op, const fmpz_t x)
        Sets rop to the quotient of the rational function op and the fmpz_t integer x.

void fmpz_poly_q_scalar_div_fmpq(fmpz_poly_q_t rop, const fmpz_poly_q_t op, const fmpq_t x)
        Sets rop to the quotient of the rational function op and the fmpq_t rational x.

5.7.9 Multiplication and division

void fmpz_poly_q_mul(fmpz_poly_q_t rop, const fmpz_poly_q_t op1, const fmpz_poly_q_t op2)
        Sets rop to the product of op1 and op2.

void fmpz_poly_q_div(fmpz_poly_q_t rop, const fmpz_poly_q_t op1, const fmpz_poly_q_t op2)
        Sets rop to the quotient of op1 and op2.

5.7.10 Powering

void fmpz_poly_q_pow(fmpz_poly_q_t rop, const fmpz_poly_q_t op, ulong exp)
        Sets rop to the exp-th power of op.

        The corner case of exp == 0 is handled by setting rop to the constant function 1. Note that this
        includes the case 0^0 = 1.
5.7.11 Derivative

```c
void fmpz_poly_q_derivative(fmpz_poly_q_t rop, const fmpz_poly_q_t op)
    Sets rop to the derivative of op.
```

5.7.12 Evaluation

```c
int fmpz_poly_q_evaluate_fmpq(fmpq_t rop, const fmpz_poly_q_t f, const fmpq_t a)
    Sets rop to \( f \) evaluated at the rational \( a \).
    If the denominator evaluates to zero at \( a \), returns non-zero and does not modify any of the variables.
    Otherwise, returns 0 and sets rop to the rational \( f(a) \).
```

5.7.13 Input and output

The following three methods enable users to construct elements of type `fmpz_poly_q_t` from strings or to obtain string representations of such elements. The format used is based on the FLINT format for integer polynomials of type `fmpz_poly_t`, which we recall first: A non-zero polynomial \( a_0 + a_1 X + \cdots + a_n X^n \) of length \( n + 1 \) is represented by the string "\( n+1 \ a_0 \ a_1 \ldots \ a_n \)" where there are two space characters following the length and single space characters separating the individual coefficients. There is no leading or trailing white-space. The zero polynomial is simply represented by "0". We adapt this notation for rational functions as follows. We denote the zero function by "0". Given a non-zero function with numerator and denominator string representations `num` and `den`, respectively, we use the string `num/den` to represent the rational function, unless the denominator is equal to one, in which case we simply use `num`. There is also a `_pretty` variant available, which bases the string parts for the numerator and denominator on the output of the function `fmpz_poly_get_str_pretty` and introduces parentheses where necessary. Note that currently these functions are not optimised for performance and are intended to be used only for debugging purposes or one-off input and output, rather than as a low-level parser.

```c
int fmpz_poly_q_set_str(fmpz_poly_q_t rop, const char *s)
    Sets rop to the rational function given by the string s.

char *fmpz_poly_q_get_str(const fmpz_poly_q_t op)
    Returns the string representation of the rational function op.

char *fmpz_poly_q_get_str_pretty(const fmpz_poly_q_t op, const char *x)
    Returns the pretty string representation of the rational function op.

int fmpz_poly_q_print(const fmpz_poly_q_t op)
    Prints the representation of the rational function op to stdout.

int fmpz_poly_q_print_pretty(const fmpz_poly_q_t op, const char *x)
    Prints the pretty representation of the rational function op to stdout.
```

5.8 `fmpz_mpoly_q.h` – multivariate rational functions over \( \mathbb{Q} \)

An `fmpz_mpoly_q_t` represents an element of \( \mathbb{Q}(x_1, \ldots, x_n) \) for fixed \( n \) as a pair of Flint multivariate polynomials (`fmpz_mpoly_t`). Instances are always kept in canonical form by ensuring that the GCD of numerator and denominator is 1 and that the coefficient of the leading term of the denominator is positive.

The user must create a multivariate polynomial context (`fmpz_mpoly_ctx_t`) specifying the number of variables \( n \) and the monomial ordering.
5.8.1 Types and macros

type fmpz_mpoly_q_struct

type fmpz_mpoly_q_t
    An fmpz_mpoly_q_struct consists of a pair of fmpz_mpoly_struct:s. An fmpz_mpoly_q_t is defined as an array of length one of type fmpz_mpoly_q_struct, permitting an fmpz_mpoly_q_t to be passed by reference.

fmpz_mpoly_q_numref(x)
    Macro returning a pointer to the numerator of x which can be used as an fmpz_mpoly_t.

fmpz_mpoly_q_denref(x)
    Macro returning a pointer to the denominator of x which can be used as an fmpz_mpoly_t.

5.8.2 Memory management

void fmpz_mpoly_q_init(fmpz_mpoly_q_t res, const fmpz_mpoly_ctx_t ctx)
    Initializes res for use, and sets its value to zero.

void fmpz_mpoly_q_clear(fmpz_mpoly_q_t res, const fmpz_mpoly_ctx_t ctx)
    Clears res, freeing or recycling its allocated memory.

5.8.3 Assignment

void fmpz_mpoly_q_swap(fmpz_mpoly_q_t x, fmpz_mpoly_q_t y, const fmpz_mpoly_ctx_t ctx)
    Swaps the values of x and y efficiently.

void fmpz_mpoly_q_set(fmpz_mpoly_q_t res, const fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_q_set_fmpq(fmpz_mpoly_q_t res, const fmpq_t x, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_q_set_fmpz(fmpz_mpoly_q_t res, const fmpz_t x, const fmpz_mpoly_ctx_t ctx)

void fmpz_mpoly_q_set_si(fmpz_mpoly_q_t res, slong x, const fmpz_mpoly_ctx_t ctx)
    Sets res to the value x.

5.8.4 Canonicalisation

void fmpz_mpoly_q_canonicalise(fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)
    Puts the numerator and denominator of x in canonical form by removing common content and making the leading term of the denominator positive.

int fmpz_mpoly_q_is_canonical(const fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)
    Returns whether x is in canonical form.

    In addition to verifying that the numerator and denominator have no common content and that the leading term of the denominator is positive, this function checks that the denominator is nonzero and that the numerator and denominator have correctly sorted terms (these properties should normally hold; verifying them provides an extra consistency check for test code).
5.8.5 Properties

int fmpz_mpoly_q_is_zero(const fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)

Returns whether \( x \) is the constant 0.

int fmpz_mpoly_q_is_one(const fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)

Returns whether \( x \) is the constant 1.

void fmpz_mpoly_q_used_vars(int *used, const fmpz_mpoly_q_t f, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_q_used_vars_num(int *used, const fmpz_mpoly_q_t f, const fmpz_mpoly_ctx_t ctx)
void fmpz_mpoly_q_used_vars_den(int *used, const fmpz_mpoly_q_t f, const fmpz_mpoly_ctx_t ctx)

For each variable, sets the corresponding entry in \( \text{used} \) to the boolean flag indicating whether that variable appears in the rational function (respectively its numerator or denominator).

5.8.6 Special values

void fmpz_mpoly_q_zero(fmpz_mpoly_q_t res, const fmpz_mpoly_ctx_t ctx)

Sets \( \text{res} \) to the constant 0.

void fmpz_mpoly_q_one(fmpz_mpoly_q_t res, const fmpz_mpoly_ctx_t ctx)

Sets \( \text{res} \) to the constant 1.

void fmpz_mpoly_q_gen(fmpz_mpoly_q_t res, slong i, const fmpz_mpoly_ctx_t ctx)

Sets \( \text{res} \) to the generator \( x_i+1 \). Requires \( 0 \leq i < n \) where \( n \) is the number of variables of \( \text{ctx} \).

5.8.7 Input and output

The variable strings in \( x \) start with the variable of most significance at index 0. If \( x \) is \text{NULL}, the variables are named \( x_1, x_2, \) etc.

void fmpz_mpoly_q_print_pretty(const fmpz_mpoly_q_t f, const char **x, const fmpz_mpoly_ctx_t ctx)

Prints \( \text{res} \) to standard output. If \( x \) is not \text{NULL}, the strings in \( x \) are used as the symbols for the variables.

cchar *fmpz_mpoly_q_get_str_pretty(const fmpz_mpoly_q_t f, const char **x, const fmpz_mpoly_ctx_t ctx)

Return a string, which the user is responsible for cleaning up, representing \( f \), given an array of variable strings \( x \).

int fmpz_mpoly_q_set_str_pretty(fmpz_mpoly_q_t res, const char *s, const char **x, fmpz_mpoly_ctx_t ctx)

Set \( \text{res} \) to the fraction in the null-terminated string \( \text{str} \) given an array \( x \) of variable strings. If parsing \( \text{str} \) fails, \( \text{res} \) is set to zero, and \(-1\) is returned. Otherwise, \( 0 \) is returned. The operations \(+, -, \ast, \) and \(/\) are permitted along with integers and the variables in \( x \). The character “\(^\text{^}\)” must be immediately followed by the (integer) exponent. If division by zero occurs, parsing fails.
5.8.8 Random generation

```c
void fmpz_mpoly_q_randtest(fmpz_mpoly_q_t res, flint_rand_t state, slong length, mp_limb_t coeff_bits, slong exp_bound, const fmpz_mpoly_ctx_t ctx)
```

Sets res to a random rational function where both numerator and denominator have up to length terms, coefficients up to size coeff_bits, and exponents strictly smaller than exp_bound.

5.8.9 Comparisons

```c
int fmpz_mpoly_q_equal(const fmpz_mpoly_q_t x, const fmpz_mpoly_q_t y, const fmpz_mpoly_ctx_t ctx)
```

Returns whether x and y are equal.

5.8.10 Arithmetic

```c
void fmpz_mpoly_q_neg(fmpz_mpoly_q_t res, const fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)
```

Sets res to the negation of x.

```c
void fmpz_mpoly_q_add(fmpz_mpoly_q_t res, const fmpz_mpoly_q_t x, const fmpz_mpoly_q_t y, const fmpz_mpoly_ctx_t ctx)
```

Sets res to the sum of x and y.

```c
void fmpz_mpoly_q_sub(fmpz_mpoly_q_t res, const fmpz_mpoly_q_t x, const fmpz_mpoly_q_t y, const fmpz_mpoly_ctx_t ctx)
```

Sets res to the difference of x and y.

```c
void fmpz_mpoly_q_mul(fmpz_mpoly_q_t res, const fmpz_mpoly_q_t x, const fmpz_mpoly_q_t y, const fmpz_mpoly_ctx_t ctx)
```

Sets res to the product of x and y.
void \texttt{fmpz\_mpoly\_q\_div\_fmpz}(\texttt{fmpz\_mpoly\_q\_t} res, const \texttt{fmpz\_mpoly\_q\_t} x, const \texttt{fmpz\_t} y, const \texttt{fmpz\_mpoly\_ctx\_t} ctx)

void \texttt{fmpz\_mpoly\_q\_div\_si}(\texttt{fmpz\_mpoly\_q\_t} res, const \texttt{fmpz\_mpoly\_q\_t} x, \texttt{slong} y, const \texttt{fmpz\_mpoly\_ctx\_t} ctx)

Sets \textit{res} to the quotient of \textit{x} and \textit{y}. Division by zero calls \texttt{flint\_abort}.

void \texttt{fmpz\_mpoly\_q\_inv}(\texttt{fmpz\_mpoly\_q\_t} res, const \texttt{fmpz\_mpoly\_q\_t} x, const \texttt{fmpz\_mpoly\_ctx\_t} ctx)

Sets \textit{res} to the inverse of \textit{x}. Division by zero calls \texttt{flint\_abort}.

\textbf{5.8.11 Content}

void \texttt{\_fmpz\_mpoly\_q\_content}(\texttt{fmpz\_t} num, \texttt{fmpz\_t} den, const \texttt{fmpz\_mpoly\_t} xnum, const \texttt{fmpz\_mpoly\_t} xden, const \texttt{fmpz\_mpoly\_ctx\_t} ctx)

void \texttt{fmpz\_mpoly\_q\_content}(\texttt{fmpq\_t} res, const \texttt{fmpz\_mpoly\_q\_t} x, const \texttt{fmpz\_mpoly\_ctx\_t} ctx)

Sets \textit{res} to the content of the coefficients of \textit{x}.
6.1 nmod.h – integers mod n (word-size n)

6.1.1 Modular reduction and arithmetic

```c
void nmod_init(nmod_t *mod, mp_limb_t n)
```
Initialises the given nmod_t structure for reduction modulo $n$ with a precomputed inverse.

```c
NMOD_BITS(mod)
```
Macro giving the number of bits in $mod.n$.

```c
NMOD_CAN_USE_SHOUP(mod)
```
Macro returning whether Shoup’s algorithm can be used for preconditioned multiplication mod $mod.n$.

```c
NMOD_RED2(r, a_hi, a_lo, mod)
```
Macro to set $r$ to $a$ reduced modulo $mod.n$, where $a$ consists of two limbs ($a_hi$, $a_lo$). The mod parameter must be a valid nmod_t structure. It is assumed that $a_hi$ is already reduced modulo $mod.n$.

```c
NMOD_RED(r, a, mod)
```
Macro to set $r$ to $a$ reduced modulo $mod.n$. The mod parameter must be a valid nmod_t structure.

```c
NMOD2_RED2(r, a_hi, a_lo, mod)
```
Macro to set $r$ to $a$ reduced modulo $mod.n$, where $a$ consists of two limbs ($a_hi$, $a_lo$). The mod parameter must be a valid nmod_t structure. No assumptions are made about $a_hi$.

```c
NMOD_RED3(r, a_hi, a_me, a_lo, mod)
```
Macro to set $r$ to $a$ reduced modulo $mod.n$, where $a$ consists of three limbs ($a_hi$, $a_me$, $a_lo$). The mod parameter must be a valid nmod_t structure. It is assumed that $a_hi$ is already reduced modulo $mod.n$.

```c
NMOD_MUL_PRENORMAL(res, a, b, mod)
```
Macro to set $r$ to $ab$ modulo $mod.n$. The mod parameter must be a valid nmod_t structure. It is assumed that $a$, $b$ are already reduced modulo $mod.n$ and that either $a$ or $b$ is prenormalised by left-shifting by $mod.norm$.

```c
NMOD_MUL_FULLWORD(res, a, b, mod)
```
Macro to set $r$ to $ab$ modulo $mod.n$. The mod parameter must be a valid nmod_t structure. It is assumed that $a$, $b$ are already reduced modulo $mod.n$ and that $mod.n$ is exactly FLINT_BITS bits large.

```c
NMOD_ADDMUL(r, a, b, mod)
```
Macro to set $r$ to $r + ab$ reduced modulo $mod.n$. The mod parameter must be a valid nmod_t structure. It is assumed that $r$, $a$, $b$ are already reduced modulo $mod.n$. 


mp_limb_t _nmod_add(mp_limb_t a, mp_limb_t b, nmod_t mod)

Returns \(a + b\) modulo mod.n. It is assumed that mod is no more than FLINT_BITS - 1 bits. It is assumed that a and b are already reduced modulo mod.n.

mp_limb_t nmod_add(mp_limb_t a, mp_limb_t b, nmod_t mod)

Returns \(a + b\) modulo mod.n. No assumptions are made about mod.n. It is assumed that a and b are already reduced modulo mod.n.

mp_limb_t _nmod_sub(mp_limb_t a, mp_limb_t b, nmod_t mod)

Returns \(a - b\) modulo mod.n. It is assumed that mod is no more than FLINT_BITS - 1 bits. It is assumed that a and b are already reduced modulo mod.n.

mp_limb_t nmod_sub(mp_limb_t a, mp_limb_t b, nmod_t mod)

Returns \(a - b\) modulo mod.n. No assumptions are made about mod.n. It is assumed that a and b are already reduced modulo mod.n.

mp_limb_t nmod_neg(mp_limb_t a, nmod_t mod)

Returns \(-a\) modulo mod.n. It is assumed that a is already reduced modulo mod.n, but no assumptions are made about the latter.

mp_limb_t nmod_mul(mp_limb_t a, mp_limb_t b, nmod_t mod)

Returns \(ab\) modulo mod.n. No assumptions are made about mod.n. It is assumed that a and b are already reduced modulo mod.n.

mp_limb_t _nmod_mul_fullword(mp_limb_t a, mp_limb_t b, nmod_t mod)

Returns \(ab\) modulo mod.n. Requires that mod.n is exactly FLINT_BITS large. It is assumed that a and b are already reduced modulo mod.n.

mp_limb_t nmod_inv(mp_limb_t a, nmod_t mod)

Returns \(a^{-1}\) modulo mod.n. The inverse is assumed to exist.

mp_limb_t nmod_div(mp_limb_t a, mp_limb_t b, nmod_t mod)

Returns \(ab^{-1}\) modulo mod.n. The inverse of b is assumed to exist. It is assumed that a is already reduced modulo mod.n.

int nmod_divides(mp_limb_t *a, mp_limb_t b, mp_limb_t c, nmod_t mod)

If \(a \cdot c = b \mod n\) has a solution for a return 1 and set a to such a solution. Otherwise return 0 and leave a undefined.

mp_limb_t nmod_pow_ui(mp_limb_t a, ulong e, nmod_t mod)

Returns \(a^e\) modulo mod.n. No assumptions are made about mod.n. It is assumed that a is already reduced modulo mod.n.

mp_limb_t nmod_pow_fmpz(mp_limb_t a, const fmpz_t e, nmod_t mod)

Returns \(a^e\) modulo mod.n. No assumptions are made about mod.n. It is assumed that a is already reduced modulo mod.n and that e is not negative.

### 6.1.2 Discrete Logarithms via Pohlig-Hellman

void nmod_discrete_log_pohlig_hellman_init(nmod_discrete_log_pohlig_hellman_t L)

Initialize L. Upon initialization L is not ready for computation.

void nmod_discrete_log_pohlig_hellman_clear(nmod_discrete_log_pohlig_hellman_t L)

Free any space used by L.

double nmod_discrete_log_pohlig_hellman_precompute_prime(nmod_discrete_log_pohlig_hellman_t L, mp_limb_t p)

Configure L for discrete logarithms modulo p to an internally chosen base. It is assumed that p is prime. The return is an estimate on the number of multiplications needed for one run.
mp_limb_t nmod_discrete_log_pohlig_hellman_primitive_root(const
nmod_discrete_log_pohlig_hellman_t L)

Return the internally stored base.

ulong nmod_discrete_log_pohlig_hellman_run(const nmod_discrete_log_pohlig_hellman_t L,
mp_limb_t y)

Return the logarithm of y with respect to the internally stored base. y is expected to be reduced
modulo the p. The function is undefined if the logarithm does not exist.

6.2 nmod_vec.h – vectors over integers mod n (word-size n)

6.2.1 Memory management

mp_ptr _nmod_vec_init(slong len)
Returns a vector of the given length. The entries are not necessarily zero.

void _nmod_vec_clear(mp_ptr vec)
Frees the memory used by the given vector.

6.2.2 Random functions

void _nmod_vec_randtest(mp_ptr vec, flint_rand_t state, slong len, nmod_t mod)
Sets vec to a random vector of the given length with entries reduced modulo mod.n.

6.2.3 Basic manipulation and comparison

void _nmod_vec_set(mp_ptr res, mp_srcptr vec, slong len)
Copies len entries from the vector vec to res.

void _nmod_vec_zero(mp_ptr vec, slong len)
Zeros the given vector of the given length.

void _nmod_vec_swap(mp_ptr a, mp_ptr b, slong length)
Swaps the vectors a and b of length n by actually swapping the entries.

void _nmod_vec_reduce(mp_ptr res, mp_srcptr vec, slong len, nmod_t mod)
Reduces the entries of (vec, len) modulo mod.n and set res to the result.

flint_bitcnt_t _nmod_vec_max_bits(mp_srcptr vec, slong len)
Returns the maximum number of bits of any entry in the vector.

int _nmod_vec_equal(mp_srcptr vec, mp_srcptr vec2, slong len)
Returns ~1` if (vec, len) is equal to (vec2, len), otherwise returns ~0`. 
6.2.4 Printing

```c
void _nmod_vec_print_pretty(mp_srcptr vec, slong len, nmod_t mod)
Pretty-prints vec to stdout. A header is printed followed by the vector enclosed in brackets. Each
entry is right-aligned to the width of the modulus written in decimal, and the entries are separated
by spaces. For example:

[length-12 integer vector mod 197>
[ 33 181 107 61 32 11 80 138 34 171 86 156]
```

```c
int _nmod_vec_fprint_pretty(FILE *file, mp_srcptr vec, slong len, nmod_t mod)
Same as _nmod_vec_print_pretty but printing to file.
```

```c
int _nmod_vec_print(mp_srcptr vec, slong len, nmod_t mod)
Currently, same as _nmod_vec_print_pretty.
```

```c
int _nmod_vec_fprint(FILE *f, mp_srcptr vec, slong len, nmod_t mod)
Currently, same as _nmod_vec_fprint_pretty.
```

6.2.5 Arithmetic operations

```c
void _nmod_vec_add(mp_ptr res, mp_srcptr vec1, mp_srcptr vec2, slong len, nmod_t mod)
Sets (res, len) to the sum of (vec1, len) and (vec2, len).
```

```c
void _nmod_vec_sub(mp_ptr res, mp_srcptr vec1, mp_srcptr vec2, slong len, nmod_t mod)
Sets (res, len) to the difference of (vec1, len) and (vec2, len).
```

```c
void _nmod_vec_neg(mp_ptr res, mp_srcptr vec, slong len, nmod_t mod)
Sets (res, len) to the negation of (vec, len).
```

```c
void _nmod_vec_scalar_mul_nmod(mp_ptr res, mp_srcptr vec, slong len, mp_limb_t c, nmod_t mod)
Sets (res, len) to (vec, len) multiplied by c. The element c and all elements of vec are assumed
to be less than mod.n.
```

```c
void _nmod_vec_scalar_mul_nmod_shoup(mp_ptr res, mp_srcptr vec, slong len, mp_limb_t c, nmod_t mod)
Sets (res, len) to (vec, len) multiplied by c using n_mulmod_shoup(). mod.n should be less
than 2^FLINT_BITS-1. c and all elements of vec should be less than mod.n.
```

```c
void _nmod_vec_scalar_addmul_nmod(mp_ptr res, mp_srcptr vec, slong len, mp_limb_t c, nmod_t mod)
Adds (vec, len) times c to the vector (res, len). The element c and all elements of vec are
assumed to be less than mod.n.
```

6.2.6 Dot products

```c
int _nmod_vec_dot_bound_lims(slong len, nmod_t mod)
Returns the number of limbs (0, 1, 2 or 3) needed to represent the unreduced dot product of two
vectors of length len having entries modulo mod.n, assuming that len is nonnegative and that
mod.n is nonzero. The computed bound is tight. In other words, this function returns the precise
limb size of len times (mod.n - 1) \^ 2.
```

```c
NMOD_VEC_DOT(res, i, len, expr1, expr2, mod, nlimbs)
Effectively performs the computation:
```
res = 0;
for (i = 0; i < len; i++)
    res += (expr1) * (expr2);

but with the arithmetic performed modulo $\text{mod}$. The $\text{nlimbs}$ parameter should be 0, 1, 2 or 3, specifying the number of limbs needed to represent the unreduced result.

$\text{nmod.h}$ has to be included in order for this macro to work (order of inclusions does not matter).

\[\text{mp}_\text{limb}_\text{t} \_\text{nmod}_\text{vec}_\text{dot}(\text{mp}_\text{srcptr} \text{vec1}, \text{mp}_\text{srcptr} \text{vec2}, \text{slong} \text{len}, \text{nmod}_\text{t} \text{mod}, \text{int} \text{nlimbs})\]

Returns the dot product of (vec1, len) and (vec2, len). The nlimbs parameter should be 0, 1, 2 or 3, specifying the number of limbs needed to represent the unreduced result.

\[\text{mp}_\text{limb}_\text{t} \_\text{nmod}_\text{vec}_\text{dot}_\text{rev}(\text{mp}_\text{srcptr} \text{vec1}, \text{mp}_\text{srcptr} \text{vec2}, \text{slong} \text{len}, \text{nmod}_\text{t} \text{mod}, \text{int} \text{nlimbs})\]

The same as _nmod_vec_dot, but reverses vec2.

\[\text{mp}_\text{limb}_\text{t} \_\text{nmod}_\text{vec}_\text{dot}_\text{ptr}(\text{mp}_\text{srcptr} \text{vec1}, \text{const} \text{mp}_\text{ptr} \*\text{vec2}, \text{slong} \text{offset}, \text{slong} \text{len}, \text{nmod}_\text{t} \text{mod}, \text{int} \text{nlimbs})\]

Returns the dot product of (vec1, len) and the values at vec2[i][offset]. The nlimbs parameter should be 0, 1, 2 or 3, specifying the number of limbs needed to represent the unreduced result.

### 6.3 nmod_mat.h – matrices over integers mod n (word-size n)

An nmod_mat_t represents a matrix of integers modulo $n$, for any non-zero modulus $n$ that fits in a single limb, up to $2^{32} - 1$ or $2^{64} - 1$.

The nmod_mat_t type is defined as an array of nmod_mat_struct’s of length one. This permits passing parameters of type nmod_mat_t by reference.

An nmod_mat_t internally consists of a single array of mp_limb_t’s, representing a dense matrix in row-major order. This array is only directly indexed during memory allocation and deallocation. A separate array holds pointers to the start of each row, and is used for all indexing. This allows the rows of a matrix to be permuted quickly by swapping pointers.

Matrices having zero rows or columns are allowed.

The shape of a matrix is fixed upon initialisation. The user is assumed to provide input and output variables whose dimensions are compatible with the given operation.

It is assumed that all matrices passed to a function have the same modulus. The modulus is assumed to be a prime number in functions that perform some kind of division, solving, or Gaussian elimination (including computation of rank and determinant), but can be composite in functions that only perform basic manipulation and ring operations (e.g. transpose and matrix multiplication).

The user can manipulate matrix entries directly, but must assume responsibility for normalising all values to the range $[0, n)$.

#### 6.3.1 Types, macros and constants

**type nmod_mat_struct**

**type nmod_mat_t**
6.3.2 Memory management

void nmod_mat_init(nmod_mat_t mat, slong rows, slong cols, mp_limb_t n)
    Initialises mat to a rows-by-cols matrix with coefficients modulo n, where n can be any nonzero integer that fits in a limb. All elements are set to zero.

void nmod_mat_init_set(nmod_mat_t mat, const nmod_mat_t src)
    Initialises mat and sets its dimensions, modulus and elements to those of src.

void nmod_mat_clear(nmod_mat_t mat)
    Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an nmod_mat_t object.

void nmod_mat_set(nmod_mat_t mat, const nmod_mat_t src)
    Sets mat to a copy of src. It is assumed that mat and src have identical dimensions.

void nmod_mat_swap(nmod_mat_t mat1, nmod_mat_t mat2)
    Exchanges mat1 and mat2.

void nmod_mat_swap_entrywise(nmod_mat_t mat1, nmod_mat_t mat2)
    Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

6.3.3 Basic properties and manipulation

nmod_mat_entry(mat, i, j)
    Directly accesses the entry in mat in row i and column j, indexed from zero. No bounds checking is performed. This macro can be used both for reading and writing coefficients.

mp_limb_t nmod_mat_get_entry(const nmod_mat_t mat, slong i, slong j)
    Get the entry at row i and column j of the matrix mat.

mp_limb_t *nmod_mat_entry_ptr(const nmod_mat_t mat, slong i, slong j)
    Return a pointer to the entry at row i and column j of the matrix mat.

void nmod_mat_set_entry(nmod_mat_t mat, slong i, slong j, mp_limb_t x)
    Set the entry at row i and column j of the matrix mat to x.

slong nmod_mat_nrows(const nmod_mat_t mat)
    Returns the number of rows in mat.

slong nmod_mat_ncols(const nmod_mat_t mat)
    Returns the number of columns in mat.

void nmod_mat_zero(nmod_mat_t mat)
    Sets all entries of the matrix mat to zero.

int nmod_mat_is_zero(const nmod_mat_t mat)
    Returns 1 if all entries of the matrix mat are zero.
6.3.4 Window

void nmod_mat_window_init(nmod_mat_t window, const nmod_mat_t mat, slong r1, slong c1, slong r2, slong c2)

Initializes the matrix window to be an \( r2 - r1 \) by \( c2 - c1 \) submatrix of mat whose \((0,0)\) entry is the \((r1, c1)\) entry of mat. The memory for the elements of window is shared with mat.

void nmod_mat_window_clear(nmod_mat_t window)

Clears the matrix window and releases any memory that it uses. Note that the memory to the underlying matrix that window points to is not freed.

6.3.5 Concatenate

void nmod_mat_concat_vertical(nmod_mat_t res, const nmod_mat_t mat1, const nmod_mat_t mat2)

Sets res to vertical concatenation of \((mat1, mat2)\) in that order. Matrix dimensions : \( mat1 : m \times n \), \( mat2 : k \times n \), res : \((m + k) \times n\).

void nmod_mat_concat_horizontal(nmod_mat_t res, const nmod_mat_t mat1, const nmod_mat_t mat2)

Sets res to horizontal concatenation of \((mat1, mat2)\) in that order. Matrix dimensions : \( mat1 : m \times n \), \( mat2 : m \times k \), res : \(m \times (n + k)\).

6.3.6 Printing

void nmod_mat_print_pretty(const nmod_mat_t mat)

Pretty-prints mat to stdout. A header is printed followed by the rows enclosed in brackets. Each column is right-aligned to the width of the modulus written in decimal, and the columns are separated by spaces. For example:

\[
\begin{bmatrix}
<2 \times 3 \text{ integer matrix mod } 2903> \\
[ 0 & 0 & 2607] \\
[ 622 & 0 & 0]
\end{bmatrix}
\]

int nmod_mat_fprint_pretty(FILE *file, const nmod_mat_t mat)

Same as nmod_mat_print_pretty but printing to file.

int nmod_mat_print(const nmod_mat_t mat)

Currently, same as nmod_mat_print_pretty.

int nmod_mat_fprint(FILE *f, const nmod_mat_t mat)

Currently, same as nmod_mat_fprint_pretty.

6.3.7 Random matrix generation

void nmod_mat_randtest(nmod_mat_t mat, flint_rand_t state)

Sets the elements to a random matrix with entries between 0 and \( m - 1 \) inclusive, where \( m \) is the modulus of mat. A sparse matrix is generated with increased probability.

void nmod_mat_randfull(nmod_mat_t mat, flint_rand_t state)

Sets the element to random numbers likely to be close to the modulus of the matrix. This is used to test potential overflow-related bugs.
int nmod_mat_randpermdiag(nmod_mat_t mat, flint_rand_t state, mp_srcptr diag, slong n)

Sets mat to a random permutation of the diagonal matrix with \( n \) leading entries given by the vector diag. It is assumed that the main diagonal of mat has room for at least \( n \) entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

void nmod_mat_randrank(nmod_mat_t mat, flint_rand_t state, slong rank)

Sets mat to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random integers between 0 and \( m - 1 \) inclusive, where \( m \) is the modulus of mat.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling nmod_mat_randops() .

void nmod_mat_randops(nmod_mat_t mat, slong count, flint_rand_t state)

Randomises mat by performing elementary row or column operations. More precisely, at most count random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

void nmod_mat_randtril(nmod_mat_t mat, flint_rand_t state, int unit)

Sets mat to a random lower triangular matrix. If unit is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

void nmod_mat_randtriu(nmod_mat_t mat, flint_rand_t state, int unit)

Sets mat to a random upper triangular matrix. If unit is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

6.3.8 Comparison

int nmod_mat_equal(const nmod_mat_t mat1, const nmod_mat_t mat2)

Returns nonzero if mat1 and mat2 have the same dimensions and elements, and zero otherwise. The moduli are ignored.

int nmod_mat_is_zero_row(const nmod_mat_t mat, slong i)

Returns a non-zero value if row \( i \) of mat is zero.

6.3.9 Transposition and permutations

void nmod_mat_transpose(nmod_mat_t B, const nmod_mat_t A)

Sets B to the transpose of A. Dimensions must be compatible. B and A may be the same object if and only if the matrix is square.

void nmod_mat_swap_rows(nmod_mat_t mat, slong *perm, slong r, slong s)

Swaps rows \( r \) and \( s \) of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

void nmod_mat_swap_cols(nmod_mat_t mat, slong *perm, slong r, slong s)

Swaps columns \( r \) and \( s \) of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.

void nmod_mat_invert_rows(nmod_mat_t mat, slong *perm)

Swaps rows \( i \) and \( r - i \) of mat for \( 0 \leq i < \lfloor r/2 \rfloor \), where \( r \) is the number of rows of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

void nmod_mat_invert_cols(nmod_mat_t mat, slong *perm)

Swaps columns \( i \) and \( c - i \) of mat for \( 0 \leq i < \lfloor c/2 \rfloor \), where \( c \) is the number of columns of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.
void nmod_mat_permute_rows(nmod_mat_t mat, const slong *perm_act, slong *perm_store)

Permutes rows of the matrix mat according to permutation perm_act and, if perm_store is not NULL, apply the same permutation to it.

6.3.10 Addition and subtraction

void nmod_mat_add(nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

Computes $C = A + B$. Dimensions must be identical.

void nmod_mat_sub(nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

Computes $C = A - B$. Dimensions must be identical.

void nmod_mat_neg(nmod_mat_t A, const nmod_mat_t B)

Sets $B = -A$. Dimensions must be identical.

6.3.11 Matrix-scalar arithmetic

void nmod_mat_scalar_mul(nmod_mat_t B, const nmod_mat_t A, mp_limb_t c)

Sets $B = cA$, where the scalar $c$ is assumed to be reduced modulo the modulus. Dimensions of $A$ and $B$ must be identical.

void nmod_mat_scalar_addmul_ui(nmod_mat_t dest, const nmod_mat_t X, const nmod_mat_t Y, const mp_limb_t b)

Sets $dest = X + bY$, where the scalar $b$ is assumed to be reduced modulo the modulus. Dimensions of dest, X and Y must be identical. dest can be aliased with X or Y.

void nmod_mat_scalar_mul_fmpz(nmod_mat_t res, const nmod_mat_t M, const fmpz_t c)

Sets $B = cA$, where the scalar $c$ is of type fmpz_t. Dimensions of $A$ and $B$ must be identical.

6.3.12 Matrix multiplication

void nmod_mat_mul(nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and Strassen multiplication.

void _nmod_mat_mul_classical_op(nmod_mat_t D, const nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B, int op)

Sets $D = A \cdot B \text{ op } C$ where op is +1 for addition, -1 for subtraction and 0 to ignore C.

void nmod_mat_mul_classical(nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. $C$ is not allowed to be aliased with $A$ or $B$. Uses classical matrix multiplication, creating a temporary transposed copy of $B$ to improve memory locality if the matrices are large enough, and packing several entries of $B$ into each word if the modulus is very small.

void _nmod_mat_mul_classical_threaded_pool_op(nmod_mat_t D, const nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B, int op, thread_pool_handle *threads, slong num_threads)

Multithreaded version of _nmod_mat_mul_classical.

void _nmod_mat_mul_classical_threaded_op(nmod_mat_t D, const nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B, int op)

Multithreaded version of _nmod_mat_mul_classical.
void nmod_mat_mul_classical_threaded(nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

Multithreaded version of nmod_mat_mul_classical.

void nmod_mat_mul_strassen(nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. $C$ is not allowed to be aliased with $A$ or $B$. Uses Strassen multiplication (the Strassen-Winograd variant).

int nmod_mat_mul_blas(nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

Tries to set $C = AB$ using BLAS and returns 1 for success and 0 for failure. Dimensions must be compatible for matrix multiplication.

void nmod_mat_addmul(nmod_mat_t D, const nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

Sets $D = C + AB$. $C$ and $D$ may be aliased with each other but not with $A$ or $B$. Automatically selects between classical and Strassen multiplication.

void nmod_mat_submul(nmod_mat_t D, const nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

Sets $D = C - AB$. $C$ and $D$ may be aliased with each other but not with $A$ or $B$.

void nmod_mat_mul_nmod_vec(mp_limb_t* c, const nmod_mat_t A, const mp_limb_t* b, slong blen)

void nmod_mat_mul_nmod_vec_ptr(mp_limb_t* const *c, const nmod_mat_t A, const mp_limb_t *const *b, slong blen)

Compute a matrix-vector product of $A$ and $(b, \text{blen})$ and store the result in $c$. The vector $(b, \text{blen})$ is either truncated or zero-extended to the number of columns of $A$. The number of entries written to $c$ is always equal to the number of rows of $A$.

void nmod_mat_nmod_vec_mul(mp_limb_t* c, const mp_limb_t* a, slong alen, const nmod_mat_t B)

void nmod_mat_nmod_vec_mul_ptr(mp_limb_t* const *c, const mp_limb_t* const *a, slong alen, const nmod_mat_t B)

Compute a vector-matrix product of $(a, \text{alen})$ and $B$ and store the result in $c$. The vector $(a, \text{alen})$ is either truncated or zero-extended to the number of rows of $B$. The number of entries written to $c$ is always equal to the number of columns of $B$.

### 6.3.13 Matrix Exponentiation

void _nmod_mat_pow(nmod_mat_t dest, const nmod_mat_t mat, ulong pow)

Sets $\text{dest} = \text{mat}^\text{pow}$. $\text{dest}$ and $\text{mat}$ cannot be aliased. Implements exponentiation by squaring.

void nmod_mat_pow(nmod_mat_t dest, const nmod_mat_t mat, ulong pow)

Sets $\text{dest} = \text{mat}^\text{pow}$. $\text{dest}$ and $\text{mat}$ may be aliased. Implements exponentiation by squaring.

### 6.3.14 Trace

mp_limb_t nmod_mat_trace(const nmod_mat_t mat)

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.
6.3.15 Determinant and rank

\textit{mp\_limb\_t nmod\_mat\_det\_howell}(\textit{const nmod\_mat\_t A})

Returns the determinant of \(A\).

\textit{mp\_limb\_t nmod\_mat\_det}(\textit{const nmod\_mat\_t A})

Returns the determinant of \(A\).

\textit{slong nmod\_mat\_rank}(\textit{const nmod\_mat\_t A})

Returns the rank of \(A\). The modulus of \(A\) must be a prime number.

6.3.16 Inverse

\textit{int nmod\_mat\_inv}(\textit{nmod\_mat\_t B, const nmod\_mat\_t A})

Sets \(B = A^{-1}\) and returns 1 if \(A\) is invertible. If \(A\) is singular, returns 0 and sets the elements of \(B\) to undefined values.

\(A\) and \(B\) must be square matrices with the same dimensions and modulus. The modulus must be prime.

6.3.17 Triangular solving

\textit{void nmod\_mat\_solve\_tril}(\textit{nmod\_mat\_t X, const nmod\_mat\_t L, const nmod\_mat\_t B, int unit})

Sets \(X = L^{-1}B\) where \(L\) is a full rank lower triangular square matrix. If \(\text{unit} = 1\), \(L\) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \(X\) and \(B\) are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

\textit{void nmod\_mat\_solve\_tril\_classical}(\textit{nmod\_mat\_t X, const nmod\_mat\_t L, const nmod\_mat\_t B, int unit})

Sets \(X = L^{-1}B\) where \(L\) is a full rank lower triangular square matrix. If \(\text{unit} = 1\), \(L\) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \(X\) and \(B\) are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

\textit{void nmod\_mat\_solve\_tril\_recursive}(\textit{nmod\_mat\_t X, const nmod\_mat\_t L, const nmod\_mat\_t B, int unit})

Sets \(X = L^{-1}B\) where \(L\) is a full rank lower triangular square matrix. If \(\text{unit} = 1\), \(L\) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \(X\) and \(B\) are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

\[
\begin{pmatrix} A & 0 \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}X \\ D^{-1}(Y - CA^{-1}X) \end{pmatrix}
\]

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

\textit{void nmod\_mat\_solve\_triu}(\textit{nmod\_mat\_t X, const nmod\_mat\_t U, const nmod\_mat\_t B, int unit})

Sets \(X = U^{-1}B\) where \(U\) is a full rank upper triangular square matrix. If \(\text{unit} = 1\), \(U\) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \(X\) and \(B\) are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

\textit{void nmod\_mat\_solve\_triu\_classical}(\textit{nmod\_mat\_t X, const nmod\_mat\_t U, const nmod\_mat\_t B, int unit})

Sets \(X = U^{-1}B\) where \(U\) is a full rank upper triangular square matrix. If \(\text{unit} = 1\), \(U\) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \(X\) and \(B\) are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.
void nmod_mat_solve_triu_recursive(nmod_mat_t X, const nmod_mat_t U, const nmod_mat_t B, int unit)

Sets \( X = U^{-1}B \) where \( U \) is a full rank upper triangular square matrix. If \( \text{unit} = 1 \), \( U \) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \( X \) and \( B \) are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

\[
\begin{pmatrix} A & B \\ 0 & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}(X - BD^{-1}Y) \\ D^{-1}Y \end{pmatrix}
\]

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

6.3.18 Nonsingular square solving

int nmod_mat_solve(nmod_mat_t X, const nmod_mat_t A, const nmod_mat_t B)

Solves the matrix-matrix equation \( AX = B \) over \( \mathbb{Z}/p\mathbb{Z} \) where \( p \) is the modulus of \( X \) which must be a prime number. \( X \), \( A \), and \( B \) should have the same moduli.

Returns 1 if \( A \) has full rank; otherwise returns 0 and sets the elements of \( X \) to undefined values.

The matrix \( A \) must be square.

int nmod_mat_can_solve_inner(slong *rank, slong *perm, slong *pivots, nmod_mat_t X, const nmod_mat_t A, const nmod_mat_t B)

As for nmod_mat_can_solve() except that if rank is not NULL the value it points to will be set to the rank of \( A \). If perm is not NULL then it must be a valid initialised permutation whose length is the number of rows of \( A \). After the function call it will be set to the row permutation given by LU decomposition of \( A \). If pivots is not NULL then it must an initialised vector. Only the first *rank of these will be set by the function call. They are set to the columns of the pivots chosen by the LU decomposition of \( A \).

int nmod_mat_can_solve(nmod_mat_t X, const nmod_mat_t A, const nmod_mat_t B)

Solves the matrix-matrix equation \( AX = B \) over \( \mathbb{Z}/p\mathbb{Z} \) where \( p \) is the modulus of \( X \) which must be a prime number. \( X \), \( A \), and \( B \) should have the same moduli.

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of \( X \) to zero. If more than one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of \( A \) and it may be singular.

int nmod_mat_solve_vec(mp_ptr x, const nmod_mat_t A, mp_srcptr b)

Solves the matrix-vector equation \( Ax = b \) over \( \mathbb{Z}/p\mathbb{Z} \) where \( p \) is the modulus of \( A \) which must be a prime number.

Returns 1 if \( A \) has full rank; otherwise returns 0 and sets the elements of \( x \) to undefined values.

6.3.19 LU decomposition

slong nmod_mat_lu(slong *P, nmod_mat_t A, int rank_check)

slong nmod_mat_lu_classical(slong *P, nmod_mat_t A, int rank_check)

slong nmod_mat_lu_classical_delayed(slong *P, nmod_mat_t A, int rank_check)

slong nmod_mat_lu_recursive(slong *P, nmod_mat_t A, int rank_check)

Computes a generalised LU decomposition \( LU = PA \) of a given matrix \( A \), returning the rank of \( A \).

If \( A \) is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix \( L \) and an upper triangular matrix \( U \) (the diagonal of \( L \) will not be stored explicitly).
If $A$ is an arbitrary matrix of rank $r$, $U$ will be in row echelon form having $r$ nonzero rows, and $L$ will be lower triangular but truncated to $r$ columns, having implicit ones on the $r$ first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if $A$ is detected to be rank-deficient.

The `classical` version uses direct Gaussian elimination. The `classical_delayed` version also uses Gaussian elimination, but performs delayed modular reductions. The `recursive` version uses block recursive decomposition. The default function chooses an algorithm automatically.

### 6.3.20 Reduced row echelon form

```c
slong nmod_mat_rref(nmod_mat_t A)
```

Puts $A$ in reduced row echelon form and returns the rank of $A$.

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

```c
slong nmod_mat_reduce_row(nmod_mat_t A, slong *P, slong *L, slong n)
```

Reduce row $n$ of the matrix $A$, assuming the prior rows are in Gauss form. However those rows may not be in order. The entry $i$ of the array $P$ is the row of $A$ which has a pivot in the $i$-th column. If no such row exists, the entry of $P$ will be $-1$. The function returns the column in which the $n$-th row has a pivot after reduction. This will always be chosen to be the first available column for a pivot from the left. This information is also updated in $P$. Entry $i$ of the array $L$ contains the number of possibly nonzero columns of $A$ row $i$. This speeds up reduction in the case that $A$ is chambered on the right. Otherwise the entries of $L$ can all be set to the number of columns of $A$. We require the entries of $L$ to be monotonic increasing.

### 6.3.21 Nullspace

```c
slong nmod_mat_nullspace(nmod_mat_t X, const nmod_mat_t A)
```

Computes the nullspace of $A$ and returns the nullity.

More precisely, this function sets $X$ to a maximum rank matrix such that $AX = 0$ and returns the rank of $X$. The columns of $X$ will form a basis for the nullspace of $A$.

$X$ must have sufficient space to store all basis vectors in the nullspace.

This function computes the reduced row echelon form and then reads off the basis vectors.

### 6.3.22 Transforms

```c
void nmod_mat_similarity(nmod_mat_t M, slong r, ulong d)
```

Applies a similarity transform to the $n \times n$ matrix $M$ in-place.

If $P$ is the $n \times n$ identity matrix the zero entries of whose row $r$ (0-indexed) have been replaced by $d$, this transform is equivalent to $M = P^{-1} MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial. The value $d$ is required to be reduced modulo the modulus of the entries in the matrix.

---

6.3. nmod_mat.h – matrices over integers mod n (word-size n) 345
6.3.23 Characteristic polynomial

```c
void nmod_mat_charpoly_berkowitz(nmod_poly_t p, const nmod_mat_t M)
void nmod_mat_charpoly_danilevsky(nmod_poly_t p, const nmod_mat_t M)
void nmod_mat_charpoly(nmod_poly_t p, const nmod_mat_t M)
```

Compute the characteristic polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised. The `danilevsky` algorithm assumes that the modulus is prime.

6.3.24 Minimal polynomial

```c
void nmod_mat_minpoly(nmod_poly_t p, const nmod_mat_t M)
```

Compute the minimal polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised.

6.3.25 Strong echelon form and Howell form

```c
void nmod_mat_strong_echelon_form(nmod_mat_t A)
```

Puts $A$ into strong echelon form. The Howell form and the strong echelon form are equal up to permutation of the rows, see [FieHof2014] for a definition of the strong echelon form and the algorithm used here. Note that [FieHof2014] defines strong echelon form as a lower left normal form, while the implemented version returns an upper right normal form, agreeing with the definition of Howell form in [StoMul1998].

$A$ must have at least as many rows as columns.

```c
slong nmod_mat_howell_form(nmod_mat_t A)
```

Puts $A$ into Howell form and returns the number of non-zero rows. For a definition of the Howell form see [StoMul1998]. The Howell form is computed by first putting $A$ into strong echelon form and then ordering the rows.

$A$ must have at least as many rows as columns.

6.4 nmod_poly.h – univariate polynomials over integers mod n (word-size n)

The `nmod_poly_t` data type represents elements of $\mathbb{Z}/n\mathbb{Z}[x]$ for a fixed modulus $n$. The `nmod_poly` module provides routines for memory management, basic arithmetic and some higher level functions such as GCD, etc.

Each coefficient of an `nmod_poly_t` is of type `mp_limb_t` and represents an integer reduced modulo the fixed modulus $n$.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

The `nmod_poly_t` type is a typedef for an array of length 1 of `nmod_poly_struct`'s. This permits passing parameters of type `nmod_poly_t` by reference.

In reality one never deals directly with the `struct` and simply deals with objects of type `nmod_poly_t`. For simplicity we will think of an `nmod_poly_t` as a `struct`, though in practice to access fields of this `struct`, one needs to dereference first, e.g. to access the `length` field of an `nmod_poly_t` called `poly1` one writes `poly1->length`.

An `nmod_poly_t` is said to be normalised if either `length` is zero, or if the leading coefficient of the polynomial is non-zero. All `nmod_poly` functions expect their inputs to be normalised and for all coefficients to be reduced modulo $n$ and unless otherwise specified they produce output that is normalised with coefficients reduced modulo $n$. 

346 Chapter 6. Integers mod n
It is recommended that users do not access the fields of an \texttt{nmod\_poly\_t} or its coefficient data directly, but make use of the functions designed for this purpose, detailed below.

Functions in \texttt{nmod\_poly} do all the memory management for the user. One does not need to specify the maximum length in advance before using a polynomial object. FLINT reallocates space automatically as the computation proceeds, if more space is required.

### 6.4.1 Simple example

The following example computes the square of the polynomial $5x^3 + 6$ in $\mathbb{Z}/7\mathbb{Z}[x]$.

```c
#include "nmod_poly.h"

int main()
{
    nmod_poly_t x, y;
    nmod_poly_init(x, 7);
    nmod_poly_init(y, 7);
    nmod_poly_set_coeff_ui(x, 3, 5);
    nmod_poly_set_coeff_ui(x, 0, 6);
    nmod_poly_mul(y, x, x);
    nmod_poly_print(x); flint_printf("\n");
    nmod_poly_print(y); flint_printf("\n");
    nmod_poly_clear(x);
    nmod_poly_clear(y);
}
```

The output is:

```
4 7 6 0 0 5
7 7 1 0 0 4 0 0 4
```

### 6.4.2 Types, macros and constants

- \texttt{type nmod\_poly\_struct}
- \texttt{type nmod\_poly\_t}

### 6.4.3 Helper functions

- \texttt{int signed_mpn_sub_n(mp\_ptr res, mp\_srcptr op1, mp\_srcptr op2, slong n)}
  
  If $\text{op1} \geq \text{op2}$ return 0 and set $\text{res}$ to $\text{op1} - \text{op2}$ else return 1 and set $\text{res}$ to $\text{op2} - \text{op1}$.

### 6.4.4 Memory management

- \texttt{void nmod\_poly\_init(nmod\_poly\_t poly, mp\_limb\_t n)}
  
  Initialises \texttt{poly}. It will have coefficients modulo $n$.

- \texttt{void nmod\_poly\_init\_preinv(nmod\_poly\_t poly, mp\_limb\_t n, mp\_limb\_t ninv)}
  
  Initialises \texttt{poly}. It will have coefficients modulo $n$. The caller supplies a precomputed inverse limb generated by \texttt{n\_preinvert\_limb()}.

- \texttt{void nmod\_poly\_init\_mod(nmod\_poly\_t poly, const nmod\_t mod)}
  
  Initialises \texttt{poly} using an already initialised modulus \texttt{mod}.

6.4. \texttt{nmod\_poly.h – univariate polynomials over integers mod n (word-size n)}
void \texttt{nmod\_poly\_init2(nmod\_poly\_t poly, mp\_limb\_t n, slong alloc)}

Initialises \texttt{poly}. It will have coefficients modulo \texttt{n}. Up to \texttt{alloc} coefficients may be stored in \texttt{poly}.

void \texttt{nmod\_poly\_init2\_preinv(nmod\_poly\_t poly, mp\_limb\_t n, mp\_limb\_t ninv, slong alloc)}

Initialises \texttt{poly}. It will have coefficients modulo \texttt{n}. The caller supplies a precomputed inverse limb generated by \texttt{n\_preinvert\_limb()}. Up to \texttt{alloc} coefficients may be stored in \texttt{poly}.

void \texttt{nmod\_poly\_realloc(nmod\_poly\_t poly, slong alloc)}

Reallocates \texttt{poly} to the given length. If the current length is less than \texttt{alloc}, the polynomial is truncated and normalised. If \texttt{alloc} is zero, the polynomial is cleared.

void \texttt{nmod\_poly\_clear(nmod\_poly\_t poly)}

Clears the polynomial and releases any memory it used. The polynomial cannot be used again until it is initialised.

void \texttt{nmod\_poly\_fit\_length(nmod\_poly\_t poly, slong alloc)}

Ensures \texttt{poly} has space for at least \texttt{alloc} coefficients. This function only ever grows the allocated space, so no data loss can occur.

void \texttt{\_nmod\_poly\_normalise(nmod\_poly\_t poly)}

Internal function for normalising a polynomial so that the top coefficient, if there is one at all, is not zero.

### 6.4.5 Polynomial properties

\texttt{slong nmod\_poly\_length(const nmod\_poly\_t poly)}

Returns the length of the polynomial \texttt{poly}. The zero polynomial has length zero.

\texttt{slong nmod\_poly\_degree(const nmod\_poly\_t poly)}

Returns the degree of the polynomial \texttt{poly}. The zero polynomial is deemed to have degree \(-1\).

\texttt{mp\_limb\_t nmod\_poly\_modulus(const nmod\_poly\_t poly)}

Returns the modulus of the polynomial \texttt{poly}. This will be a positive integer.

\texttt{flint\_bitcnt\_t nmod\_poly\_max\_bits(const nmod\_poly\_t poly)}

Returns the maximum number of bits of any coefficient of \texttt{poly}.

\texttt{int nmod\_poly\_is\_unit(const nmod\_poly\_t poly)}

Returns 1 if the polynomial is a nonzero constant (in the case of prime modulus, this is equivalent to being a unit), otherwise 0.

\texttt{int nmod\_poly\_is\_monic(const nmod\_poly\_t poly)}

Returns 1 if the polynomial is monic, i.e. nonzero with leading coefficient 1, otherwise 0.

### 6.4.6 Assignment and basic manipulation

void \texttt{nmod\_poly\_set(nmod\_poly\_t a, const nmod\_poly\_t b)}

Sets \texttt{a} to a copy of \texttt{b}.

void \texttt{nmod\_poly\_swap(nmod\_poly\_t poly1, nmod\_poly\_t poly2)}

Efficiently swaps \texttt{poly1} and \texttt{poly2} by swapping pointers internally.

void \texttt{nmod\_poly\_zero(nmod\_poly\_t res)}

Sets \texttt{res} to the zero polynomial.

void \texttt{nmod\_poly\_truncate(nmod\_poly\_t poly, slong len)}

Truncates \texttt{poly} to the given length and normalises it. If \texttt{len} is greater than the current length of \texttt{poly}, then nothing happens.
void \texttt{nmod\_poly\_set\_trunc}(nmod\_poly\_t res, \texttt{const} nmod\_poly\_t poly, \texttt{slong} len)
  Notionally truncate \texttt{poly} to length \texttt{len} and set \texttt{res} to the result. The result is normalised.

void \_\texttt{nmod\_poly\_reverse}(mp\_ptr output, mp\_srcptr input, \texttt{slong} len, \texttt{slong} m)
  Sets \texttt{output} to the reverse of \texttt{input}, which is of length \texttt{len}, but thinking of it as a polynomial of length \texttt{m}, notionally zero-padded if necessary. The length \texttt{m} must be non-negative, but there are no other restrictions. The polynomial \texttt{output} must have space for \texttt{m} coefficients. Supports aliasing of \texttt{output} and \texttt{input}, but the behaviour is undefined in case of partial overlap.

void \texttt{nmod\_poly\_reverse}(nmod\_poly\_t output, \texttt{const} nmod\_poly\_t input, \texttt{slong} m)
  Sets \texttt{output} to the reverse of \texttt{input}, thinking of it as a polynomial of length \texttt{m}, notionally zero-padded if necessary. The length \texttt{m} must be non-negative, but there are no other restrictions. The output polynomial will be set to length \texttt{m} and then normalised.

\section*{Randomization}

void \texttt{nmod\_poly\_randtest}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len)
  Generates a random polynomial with length up to \texttt{len}.

void \texttt{nmod\_poly\_randtest\_irreducible}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len)
  Generates a random irreducible polynomial with length up to \texttt{len}.

void \texttt{nmod\_poly\_randtest\_monic}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len)
  Generates a random monic polynomial with length \texttt{len}.

void \texttt{nmod\_poly\_randtest\_monic\_irreducible}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len)
  Generates a random monic irreducible polynomial with length \texttt{len}.

void \texttt{nmod\_poly\_randtest\_monic\_primitive}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len)
  Generates a random monic irreducible primitive polynomial with length \texttt{len}.

void \texttt{nmod\_poly\_randtest\_trinomial}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len)
  Generates a random monic trinomial of length \texttt{len}.

int \texttt{nmod\_poly\_randtest\_trinomial\_irreducible}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len, \texttt{slong} max\_attempts)
  Attempts to set \texttt{poly} to a monic irreducible trinomial of length \texttt{len}. It will generate up to \texttt{max\_attempts} trinomials in attempt to find an irreducible one. If \texttt{max\_attempts} is 0, then it will keep generating trinomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

void \texttt{nmod\_poly\_randtest\_pentomial}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len)
  Generates a random monic pentomial of length \texttt{len}.

int \texttt{nmod\_poly\_randtest\_pentomial\_irreducible}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len, \texttt{slong} max\_attempts)
  Attempts to set \texttt{poly} to a monic irreducible pentomial of length \texttt{len}. It will generate up to \texttt{max\_attempts} pentomials in attempt to find an irreducible one. If \texttt{max\_attempts} is 0, then it will keep generating pentomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

void \texttt{nmod\_poly\_randtest\_sparse\_irreducible}(nmod\_poly\_t poly, \texttt{flint\_rand\_t} state, \texttt{slong} len)
  Attempts to set \texttt{poly} to a sparse, monic irreducible polynomial with length \texttt{len}. It attempts to find an irreducible trinomial. If that does not succeed, it attempts to find a irreducible pentomial. If that fails, then \texttt{poly} is just set to a random monic irreducible polynomial.
6.4.8 Getting and setting coefficients

`ulong nmod_poly_get_coeff_ui(const nmod_poly_t poly, slong j)`
Returns the coefficient of `poly` at index `j`, where coefficients are numbered with zero being the constant coefficient, and returns it as an `ulong`. If `j` refers to a coefficient beyond the end of `poly`, zero is returned.

`void nmod_poly_set_coeff_ui(nmod_poly_t poly, slong j, ulong c)`
Sets the coefficient of `poly` at index `j`, where coefficients are numbered with zero being the constant coefficient, to the value `c` reduced modulo the modulus of `poly`. If `j` refers to a coefficient beyond the current end of `poly`, the polynomial is first resized, with intervening coefficients being set to zero.

6.4.9 Input and output

`char *nmod_poly_get_str(const nmod_poly_t poly)`
Writes `poly` to a string representation. The format is as described for `nmod_poly_print()`. The string must be freed by the user when finished. For this it is sufficient to call `flint_free()`.

`char *nmod_poly_get_str_pretty(const nmod_poly_t poly, const char *x)`
Writes `poly` to a pretty string representation. The format is as described for `nmod_poly_print_pretty()`. The string must be freed by the user when finished. For this it is sufficient to call `flint_free()`.

It is assumed that the top coefficient is non-zero.

`int nmod_poly_set_str(nmod_poly_t poly, const char *s)`
Reads `poly` from a string `s`. The format is as described for `nmod_poly_print()`. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned.

`int nmod_poly_print(const nmod_poly_t a)`
Prints the polynomial to `stdout`. The length is printed, followed by a space, then the modulus. If the length is zero this is all that is printed, otherwise two spaces followed by a space separated list of coefficients is printed, beginning with the constant coefficient.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

`int nmod_poly_print_pretty(const nmod_poly_t a, const char *x)`
Prints the polynomial to `stdout` using the string `x` to represent the indeterminate.

It is assumed that the top coefficient is non-zero.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

`int nmod_poly_fread(FILE *f, nmod_poly_t poly)`
Reads `poly` from the file stream `f`. If this is a file that has just been written, the file should be closed then opened again. The format is as described for `nmod_poly_print()`. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned.

`int nmod_poly_fprint(FILE *f, const nmod_poly_t poly)`
Writes a polynomial to the file stream `f`. If this is a file then the file should be closed and reopened before being read. The format is as described for `nmod_poly_print()`. If the polynomial is written correctly, a positive value is returned, otherwise a non-positive value is returned.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

`int nmod_poly_fprint_pretty(FILE *f, const nmod_poly_t poly, const char *x)`
Writes a polynomial to the file stream `f`. If this is a file then the file should be closed and reopened before being read. The format is as described for `nmod_poly_print_pretty()`. If the polynomial is written correctly, a positive value is returned, otherwise a non-positive value is returned.

It is assumed that the top coefficient is non-zero.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

```c
int nmod_poly_read(nmod_poly_t poly)
```

Read `poly` from stdin. The format is as described for `nmod_poly_print()`. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned.

### 6.4.10 Comparison

```c
int nmod_poly_equal(const nmod_poly_t a, const nmod_poly_t b)
```

Returns 1 if the polynomials are equal, otherwise 0.

```c
int nmod_poly_equal_nmod(const nmod_poly_t poly, ulong cst)
```

Returns 1 if the polynomial `poly` is constant, equal to `cst`, otherwise 0. `cst` is assumed to be already reduced, i.e. less than the modulus of `poly`.

```c
int nmod_poly_equal_ui(const nmod_poly_t poly, ulong cst)
```

Returns 1 if the polynomial `poly` is constant and equal to `cst` up to reduction modulo the modulus of `poly`, otherwise returns 0.

```c
int nmod_poly_equal_trunc(const nmod_poly_t poly1, const nmod_poly_t poly2, slong n)
```

Notionally truncate `poly1` and `poly2` to length `n` and return 1 if the truncations are equal, otherwise return 0.

```c
int nmod_poly_is_zero(const nmod_poly_t poly)
```

Returns 1 if the polynomial `poly` is the zero polynomial, otherwise returns 0.

```c
int nmod_poly_is_one(const nmod_poly_t poly)
```

Returns 1 if the polynomial `poly` is the constant polynomial 1, otherwise returns 0.

```c
int nmod_poly_is_gen(const nmod_poly_t poly)
```

Returns 1 if the polynomial is the generating indeterminate (i.e. has degree 1, constant coefficient 0, and leading coefficient 1), otherwise returns 0.

### 6.4.11 Shifting

```c
void _nmod_poly_shift_left(mp_ptr res, mp_srcptr poly, slong len, slong k)
```

Sets (res, len + k) to (poly, len) shifted left by `k` coefficients. Assumes that `res` has space for `len + k` coefficients.

```c
void nmod_poly_shift_left(nmod_poly_t res, const nmod_poly_t poly, slong k)
```

Sets `res` to `poly` shifted left by `k` coefficients, i.e. multiplied by $x^k$.

```c
void _nmod_poly_shift_right(mp_ptr res, mp_srcptr poly, slong len, slong k)
```

Sets (res, len - k) to (poly, len) shifted left by `k` coefficients. It is assumed that `k ≤ len` and that `res` has space for at least `len - k` coefficients.

```c
void nmod_poly_shift_right(nmod_poly_t res, const nmod_poly_t poly, slong k)
```

Sets `res` to `poly` shifted right by `k` coefficients, i.e. divide by $x^k$ and throw away the remainder. If `k` is greater than or equal to the length of `poly`, the result is the zero polynomial.
6.4.12 Addition and subtraction

```c
void _nmod_poly_add(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t mod)
        Sets res to the sum of (poly1, len1) and (poly2, len2). There are no restrictions on the lengths.
```

```c
void nmod_poly_add(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)
        Sets res to the sum of poly1 and poly2.
```

```c
void nmod_poly_add_series(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, slong n)
        Notionally truncate poly1 and poly2 to length n and set res to the sum.
```

```c
void _nmod_poly_sub(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t mod)
        Sets res to the difference of (poly1, len1) and (poly2, len2). There are no restrictions on the lengths.
```

```c
void nmod_poly_sub(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)
        Sets res to the difference of poly1 and poly2.
```

```c
void nmod_poly_sub_series(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, slong n)
        Notionally truncate poly1 and poly2 to length n and set res to the difference.
```

```c
void _nmod_poly_make_monic(mp_ptr output, mp_srcptr input, slong len, nmod_t mod)
        Sets output to be the scalar multiple of input of length len > 0 that has leading coefficient one, if such a polynomial exists. If the leading coefficient of input is not invertible, output is set to the multiple of input whose leading coefficient is the greatest common divisor of the leading coefficient and the modulus of input.
```

```c
void nmod_poly_make_monic(nmod_poly_t output, const nmod_poly_t input)
        Sets output to be the scalar multiple of input with leading coefficient one, if such a polynomial exists. If input is zero an exception is raised. If the leading coefficient of input is not invertible, output is set to the multiple of input whose leading coefficient is the greatest common divisor of the leading coefficient and the modulus of input.
```

6.4.13 Scalar multiplication and division

```c
void nmod_poly_scalar_mul_nmod(nmod_poly_t res, const nmod_poly_t poly, ulong c)
        Sets res to poly multiplied by c. The element c is assumed to be less than the modulus of poly.
```

```c
void nmod_poly_scalar_addmul_nmod(nmod_poly_t res, const nmod_poly_t poly, ulong c)
        Adds poly multiplied by c to res. The element c is assumed to be less than the modulus of poly.
```

```c
void _nmod_poly_make_monic(mp_ptr output, mp_srcptr input, slong len, nmod_t mod)
        Sets output to be the scalar multiple of input of length len > 0 that has leading coefficient one, if such a polynomial exists. If the leading coefficient of input is not invertible, output is set to the multiple of input whose leading coefficient is the greatest common divisor of the leading coefficient and the modulus of input.
```

```c
void nmod_poly_make_monic(nmod_poly_t output, const nmod_poly_t input)
        Sets output to be the scalar multiple of input with leading coefficient one, if such a polynomial exists. If input is zero an exception is raised. If the leading coefficient of input is not invertible, output is set to the multiple of input whose leading coefficient is the greatest common divisor of the leading coefficient and the modulus of input.
```
6.4.14 Bit packing and unpacking

```c
void _nmod_poly_bit_pack(mp_ptr res, mp_srcptr poly, slong len, flint_bitcnt_t bits)
```

Packs `len` coefficients of `poly` into fields of the given number of bits in the large integer `res`, i.e. evaluates `poly` at $2^\text{bits}$ and store the result in `res`. Assumes `len > 0` and `bits > 0`. Also assumes that no coefficient of `poly` is bigger than `bits/2` bits. We also assume `bits < 3 * FLINT_BITS`.

```c
void _nmod_poly_bit_unpack(mp_ptr res, slong len, mp_srcptr mpn, ulong bits, nmod_t mod)
```

Unpacks `len` coefficients stored in the big integer `mpn` in bit fields of the given number of bits, reduces them modulo the given modulus, then stores them in the polynomial `res`. We assume `len > 0` and `3 * FLINT_BITS > bits > 0`. There are no restrictions on the size of the actual coefficients as stored within the bitfields.

```c
void nmod_poly_bit_pack(fmpz_t f, const nmod_poly_t poly, flint_bitcnt_t bit_size)
```

Packs `poly` into bitfields of size `bit_size`, writing the result to `f`.

```c
void nmod_poly_bit_unpack(nmod_poly_t poly, const fmpz_t f, flint_bitcnt_t bit_size)
```

Unpacks the polynomial from fields of size `bit_size` as represented by the integer `f`.

```c
void _nmod_poly_KS2_pack1(mp_ptr res, mp_srcptr op, slong n, slong s, ulong b, ulong k, slong r)
```

Same as `_nmod_poly_KS2_pack`, but requires `b <= FLINT_BITS`.

```c
void _nmod_poly_KS2_pack(mp_ptr res, mp_srcptr op, slong n, slong s, ulong b, ulong k, slong r)
```

Bit packing routine used by KS2 and KS4 multiplication.

```c
void _nmod_poly_KS2_unpack1(mp_ptr res, mp_srcptr op, slong n, ulong b, ulong k)
```

Same as `_nmod_poly_KS2_unpack`, but requires `0 < 2 * b <= FLINT_BITS` (i.e. writes one word per coefficient).

```c
void _nmod_poly_KS2_unpack2(mp_ptr res, mp_srcptr op, slong n, ulong b, ulong k)
```

Same as `_nmod_poly_KS2_unpack`, but requires `FLINT_BITS < 2 * b < 2*FLINT_BITS` (i.e. writes two words per coefficient).

```c
void _nmod_poly_KS2_unpack3(mp_ptr res, mp_srcptr op, slong n, ulong b, ulong k)
```

Same as `_nmod_poly_KS2_unpack`, but requires `2 * FLINT_BITS < b < 3 * FLINT_BITS` (i.e. writes three words per coefficient).

```c
void _nmod_poly_KS2_unpack(mp_ptr res, mp_srcptr op, slong n, ulong b, ulong k)
```

Bit unpacking code used by KS2 and KS4 multiplication.

6.4.15 KS2/KS4 Reduction

```c
void _nmod_poly_KS2_reduce(mp_ptr res, slong s, mp_srcptr op, slong n, ulong w, nmod_t mod)
```

Reduction code used by KS2 and KS4 multiplication.

```c
void _nmod_poly_KS2_recover_reduce1(mp_ptr res, slong s, mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t mod)
```

Same as `_nmod_poly_KS2_recover_reduce`, but requires `0 < 2 * b <= FLINT_BITS`.

```c
void _nmod_poly_KS2_recover_reduce2(mp_ptr res, slong s, mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t mod)
```

Same as `_nmod_poly_KS2_recover_reduce`, but requires `FLINT_BITS < 2 * b < 2*FLINT_BITS`.

```c
void _nmod_poly_KS2_recover_reduce2b(mp_ptr res, slong s, mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t mod)
```

Same as `_nmod_poly_KS2_recover_reduce`, but requires `b == FLINT_BITS`.

6.4. nmod_poly.h – univariate polynomials over integers mod n (word-size n)
void _nmod_poly_KS2_recover_reduce3(mp_ptr res, slong s, mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t mod)

Same as _nmod_poly_KS2_recover_reduce, but requires $2 \times \text{FLINT\_BITS} < 2 \times b \leq 3 \times \text{FLINT\_BITS}$. 

void _nmod_poly_KS2_recover_reduce(mp_ptr res, slong s, mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t mod)

Reduction code used by KS4 multiplication.

### 6.4.16 Multiplication

void _nmod_poly_mul_classical(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t mod)

Sets $(res, len1 + len2 - 1)$ to the product of $(poly1, len1)$ and $(poly2, len2)$. Assumes $len1 \geq len2 > 0$. Aliasing of inputs and output is not permitted.

void nmod_poly_mul_classical(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)

Sets $res$ to the product of $poly1$ and $poly2$.

void _nmod_poly_mullow_classical(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, slong trunc, nmod_t mod)

Sets $res$ to the lower $\text{trunc}$ coefficients of the product of $(poly1, len1)$ and $(poly2, len2)$. Assumes that $len1 \geq len2 > 0$ and $\text{trunc} > 0$. Aliasing of inputs and output is not permitted.

void nmod_poly_mullow_classical(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, slong trunc)

Sets $res$ to the lower $\text{trunc}$ coefficients of the product of $poly1$ and $poly2$.

void _nmod_poly_mulhigh_classical(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, slong start, nmod_t mod)

Computes the product of $(poly1, len1)$ and $(poly2, len2)$ and writes the coefficients from $\text{start}$ onwards into the high coefficients of $res$, the remaining coefficients being arbitrary but reduced. Assumes that $len1 \geq len2 > 0$. Aliasing of inputs and output is not permitted.

void nmod_poly_mulhigh_classical(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, slong start)

Computes the product of $poly1$ and $poly2$ and writes the coefficients from $\text{start}$ onwards into the high coefficients of $res$, the remaining coefficients being arbitrary but reduced.

void _nmod_poly_mul_KS(mp_ptr out, mp_srcptr in1, slong len1, mp_srcptr in2, slong len2, flint_bitcnt_t bits, nmod_t mod)

Sets $res$ to the product of $in1$ and $in2$ assuming the output coefficients are at most the given number of bits wide. If $\text{bits}$ is set to 0 an appropriate value is computed automatically. Assumes that $len1 \geq len2 > 0$.

void nmod_poly_mul_KS(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, flint_bitcnt_t bits)

Sets $res$ to the product of $poly1$ and $poly2$ assuming the output coefficients are at most the given number of bits wide. If $\text{bits}$ is set to 0 an appropriate value is computed automatically.

void _nmod_poly_mul_KS2(mp_ptr res, mp_srcptr op1, slong n1, mp_srcptr op2, slong n2, nmod_t mod)

Sets $res$ to the product of $op1$ and $op2$. Assumes that $len1 \geq len2 > 0$.

void nmod_poly_mul_KS2(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)

Sets $res$ to the product of $poly1$ and $poly2$. 

---

Chapter 6. Integers mod n
void _nmod_poly_mul_high(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, flint_bitcnt_t f, slong lenf, nmod_t mod)

Sets all but the low n coefficients of res to the corresponding coefficients of the product of poly1 and poly2, the remaining coefficients being arbitrary.

void _nmod_poly_mulmod(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, mp_srcptr f, slong lenf, nmod_t mod)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. It is required that len1 + len2 - lenf > 0, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use _nmod_poly_mul instead. Aliasing of f and res is not permitted.

void nmod_poly_mulmod(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, const nmod_poly_t f)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f.
void _nmod_poly_mulmod_preinv(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, mp_srcptr f, slong lenf, mp_srcptr finv, slong lenfinv, nmod_t mod)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f.

It is required that finv is the inverse of the reverse of f mod x^lenf. It is required that len1 + len2 − lenf > 0, which is equivalent to requiring that the result will actually be reduced. It is required that len1 < lenf and len2 < lenf. Otherwise, simply use _nmod_poly_mul instead.

Aliasing of *res with any of the inputs is not permitted.

void nmod_poly_mulmod_preinv(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, const nmod_poly_t f, const nmod_poly_t finv)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. finv is the inverse of the reverse of f. It is required that poly1 and poly2 are reduced modulo f.

### 6.4.17 Powering

void _nmod_poly_pow_binexp(mp_ptr res, mp_srcptr poly, slong len, ulong e, nmod_t mod)

Raises poly of length len to the power e and sets res to the result. We require that res has enough space for (len - 1)*e + 1 coefficients. Assumes that len > 0, e > 1. Aliasing is not permitted. Uses the binary exponentiation method.

void _nmod_poly_pow_trunc_binexp(mp_ptr res, mp_srcptr poly, slong len, ulong e, slong trunc, nmod_t mod)

Sets res to the low trunc coefficients of poly (assumed to be zero padded if necessary to length trunc) to the power e. This is equivalent to doing a powering followed by a truncation. We require that res has enough space for trunc coefficients, that trunc > 0 and that e > 1. Aliasing is not permitted. Uses the binary exponentiation method.

void _nmod_poly_pow_uibinexp(mp_ptr res, mp_srcptr poly, ulong e, mp_srcptr f, slong lenf, nmod_t mod)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf − 1. The output res must have room for lenf − 1 coefficients.
void \texttt{nmod\_poly\_powmod\_ui\_binexp}(nmod\_poly\_t res, const nmod\_poly\_t poly, ulong e, const nmod\_poly\_t f) 
Sets \texttt{res} to \texttt{poly} raised to the power \texttt{e} modulo \texttt{f}, using binary exponentiation. We require \texttt{e} >= 0.

void \_\texttt{nmod\_poly\_powmod\_fmpz\_binexp}(mp\_ptr res, mp\_srcptr poly, fmpz\_t e, mp\_srcptr f, slong lenf, nmod\_t mod) 
Sets \texttt{res} to \texttt{poly} raised to the power \texttt{e} modulo \texttt{f}, using binary exponentiation. We require \texttt{e} > 0. We require \texttt{lenf} > 1. It is assumed that \texttt{poly} is already reduced modulo \texttt{f} and zero-padded as necessary to have length exactly \texttt{lenf} - 1. The output \texttt{res} must have room for \texttt{lenf} - 1 coefficients.

void \texttt{nmod\_poly\_powmod\_fmpz\_binexp\_preinv}(nmod\_poly\_t res, const nmod\_poly\_t poly, fmpz\_t e, const nmod\_poly\_t f) 
Sets \texttt{res} to \texttt{poly} raised to the power \texttt{e} modulo \texttt{f}, using binary exponentiation. We require \texttt{e} >= 0.

void \_\texttt{nmod\_poly\_powmod\_ui\_binexp\_preinv}(mp\_ptr res, mp\_srcptr poly, ulong e, mp\_srcptr f, 
slong lenf, mp\_srcptr finv, slong lenfinv, nmod\_t mod) 
Sets \texttt{res} to \texttt{poly} raised to the power \texttt{e} modulo \texttt{f}, using binary exponentiation. We require \texttt{e} > 0. We require \texttt{lenf} > 1. It is assumed that \texttt{poly} is already reduced modulo \texttt{f} and zero-padded as necessary to have length exactly \texttt{lenf} - 1. The output \texttt{res} must have room for \texttt{lenf} - 1 coefficients.

void \texttt{nmod\_poly\_powmod\_ui\_binexp\_preinv}(nmod\_poly\_t res, const nmod\_poly\_t poly, ulong e, 
const nmod\_poly\_t f, const nmod\_poly\_t finv) 
Sets \texttt{res} to \texttt{poly} raised to the power \texttt{e} modulo \texttt{f}, using binary exponentiation. We require \texttt{e} >= 0. We require \texttt{finv} to be the inverse of the reverse of \texttt{f}.

void \_\texttt{nmod\_poly\_powmod\_fmpz\_binexp\_preinv}(mp\_ptr res, mp\_srcptr poly, fmpz\_t e, mp\_srcptr f, 
slong lenf, mp\_srcptr finv, slong lenfinv, nmod\_t mod) 
Sets \texttt{res} to \texttt{poly} raised to the power \texttt{e} modulo \texttt{f}, using binary exponentiation. We require \texttt{e} > 0. We require \texttt{lenf} > 1. It is assumed that \texttt{poly} is already reduced modulo \texttt{f} and zero-padded as necessary to have length exactly \texttt{lenf} - 1. The output \texttt{res} must have room for \texttt{lenf} - 1 coefficients.

void \texttt{nmod\_poly\_powmod\_fmpz\_binexp\_preinv}(nmod\_poly\_t res, const nmod\_poly\_t poly, fmpz\_t e, 
const nmod\_poly\_t f, const nmod\_poly\_t finv) 
Sets \texttt{res} to \texttt{poly} raised to the power \texttt{e} modulo \texttt{f}, using binary exponentiation. We require \texttt{e} >= 0. We require \texttt{finv} to be the inverse of the reverse of \texttt{f}.

void \_\texttt{nmod\_poly\_powmod\_x\_ui\_preinv}(mp\_ptr res, ulong e, mp\_srcptr f, slong lenf, mp\_srcptr finv, 
slong lenfinv, nmod\_t mod) 
Sets \texttt{res} to \texttt{x} raised to the power \texttt{e} modulo \texttt{f}, using sliding window exponentiation. We require \texttt{e} > 0. We require \texttt{finv} to be the inverse of the reverse of \texttt{f}.

void \_\texttt{nmod\_poly\_powmod\_x\_ui\_preinv}(nmod\_poly\_t res, ulong e, const nmod\_poly\_t f, const 
nmod\_poly\_t finv) 
Sets \texttt{res} to \texttt{x} raised to the power \texttt{e} modulo \texttt{f}, using sliding window exponentiation. We require \texttt{e} >= 0. We require \texttt{finv} to be the inverse of the reverse of \texttt{f}.

void \_\texttt{nmod\_poly\_powmod\_x\_fmpz\_preinv}(mp\_ptr res, fmpz\_t e, mp\_srcptr f, slong lenf, mp\_srcptr 
finv, slong lenfinv, nmod\_t mod)
Sets $\text{res}$ to $x$ raised to the power $e$ modulo $f$, using sliding window exponentiation. We require $e > 0$. We require $\text{finv}$ to be the inverse of the reverse of $f$.

We require $\text{flen} > 2$. The output $\text{res}$ must have room for $\text{flen} - 1$ coefficients.

```c
void nmod_poly_powmod_x_fmpz_preinv(nmod_poly_t res, fmpz_t e, const nmod_poly_t f, const nmod_poly_t finv)
```

Sets $\text{res}$ to $x$ raised to the power $e$ modulo $f$, using sliding window exponentiation. We require $e \geq 0$. We require $\text{finv}$ to be the inverse of the reverse of $f$.

```c
void _nmod_poly_powers_mod_preinv_naive(mp_ptr *res, mp_srcptr f, slong flen, slong n, mp_srcptr g, slong glen, mp_srcptr ginv, slong ginvlen, const nmod_t mod)
```

Compute $f^0, f^1, \ldots, f^{(n-1)} \mod g$, where $g$ has length $\text{glen}$ and $f$ is reduced mod $g$ and has length $\text{flen}$ (possibly zero spaced). Assumes $\text{res}$ is an array of $n$ arrays each with space for at least $\text{glen} - 1$ coefficients and that $\text{flen} > 0$. We require that $\text{ginv}$ of length $\text{ginvlen}$ is set to the power series inverse of the reverse of $g$.

```c
void nmod_poly_powers_mod_bsgs(nmod_poly_struct *res, const nmod_poly_t f, slong n, const nmod_poly_t g)
```

Sets the entries of the array $\text{res}$ to $f^0, f^1, \ldots, f^{(n-1)} \mod g$. No aliasing is permitted between the entries of $\text{res}$ and either of the inputs.

```c
void _nmod_poly_powers_mod_preinv_threaded_pool(mp_ptr *res, mp_srcptr f, slong flen, slong n, mp_srcptr g, slong glen, mp_srcptr ginv, slong ginvlen, const nmod_t mod, thread_pool_handle *threads, slong num_threads)
```

Compute $f^0, f^1, \ldots, f^{(n-1)} \mod g$, where $g$ has length $\text{glen}$ and $f$ is reduced mod $g$ and has length $\text{flen}$ (possibly zero spaced). Assumes $\text{res}$ is an array of $n$ arrays each with space for at least $\text{glen} - 1$ coefficients and that $\text{flen} > 0$. We require that $\text{ginv}$ of length $\text{ginvlen}$ is set to the power series inverse of the reverse of $g$.

```c
void _nmod_poly_powers_mod_preinv_threaded(mp_ptr *res, mp_srcptr f, slong flen, slong n, mp_srcptr g, slong glen, mp_srcptr ginv, slong ginvlen, const nmod_t mod)
```

Compute $f^0, f^1, \ldots, f^{(n-1)} \mod g$, where $g$ has length $\text{glen}$ and $f$ is reduced mod $g$ and has length $\text{flen}$ (possibly zero spaced). Assumes $\text{res}$ is an array of $n$ arrays each with space for at least $\text{glen} - 1$ coefficients and that $\text{flen} > 0$. We require that $\text{ginv}$ of length $\text{ginvlen}$ is set to the power series inverse of the reverse of $g$.

```c
void nmod_poly_powers_mod_bsgs(nmod_poly_struct *res, const nmod_poly_t f, slong n, const nmod_poly_t g)
```

Sets the entries of the array $\text{res}$ to $f^0, f^1, \ldots, f^{(n-1)} \mod g$. No aliasing is permitted between the entries of $\text{res}$ and either of the inputs.

### 6.4.18 Division

```c
void _nmod_poly_divrem_basecase(mp_ptr Q, mp_ptr R, mp_srcptr A, slong A_len, mp_srcptr B, slong B_len, nmod_t mod)
```

Finds $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. If $\text{len}(B) = 0$ an exception is raised. We require that $W$ is temporary space of $\text{NMOD\_DIVREM\_BC\_ITCH}(A\_len, B\_len, \text{mod})$ coefficients.

```c
void nmod_poly_divrem_basecase(nmod_poly_t Q, nmod_poly_t R, const nmod_poly_t A, const nmod_poly_t B)
```

Finds $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. If $\text{len}(B) = 0$ an exception is raised.
void _nmod_poly_divrem(mp_ptr Q, mp_ptr R, mp_srcptr A, slong lenA, mp_srcptr B, slong lenB, nmod_t mod)

Computes $Q$ and $R$ such that $A = BQ + R$ with \( \text{len}(R) \) less than \( \text{len}B \), where $A$ is of length \( \text{len}A \) and $B$ is of length $\text{len}B$. We require that $Q$ have space for $\text{len}A - \text{len}B + 1$ coefficients.

void nmod_poly_divrem(nmod_poly_t Q, nmod_poly_t R, const nmod_poly_t A, const nmod_poly_t B)

Computes $Q$ and $R$ such that $A = BQ + R$ with \( \text{len}(R) < \text{len}(B) \).

void _nmod_poly_div(mp_ptr Q, mp_ptr R, mp_srcptr A, slong lenA, mp_srcptr B, slong lenB, nmod_t mod)

Notionally computes polynomials $Q$ and $R$ such that $A = BQ + R$ with \( \text{len}(R) \) less than \( \text{len}B \), where $A$ is of length \( \text{len}A \) and $B$ is of length \( \text{len}B \), but returns only $Q$. We require that $Q$ have space for \( \text{len}A - \text{len}B + 1 \) coefficients.

void nmod_poly_div(nmod_poly_t Q, const nmod_poly_t A, const nmod_poly_t B)

Computes the quotient $Q$ on polynomial division of $A$ and $B$.

void _nmod_poly_rem_q1(mp_ptr R, mp_ptr A, slong lenA, mp_ptr B, slong lenB, nmod_t mod)

void _nmod_poly_rem(mp_ptr R, mp_ptr A, slong lenA, mp_ptr B, slong lenB, nmod_t mod)

Computes the remainder $R$ on polynomial division of $A$ by $B$.

void nmod_poly_rem(nmod_poly_t R, const nmod_poly_t A, const nmod_poly_t B)

Computes the remainder $R$ on polynomial division of $A$ by $B$.

void _nmod_poly_inv_series_basecase(mp_ptr Qinv, mp_ptr Q, slong Qlen, slong n, nmod_t mod)

Given $Q$ of length $Qlen$ whose leading coefficient is invertible modulo the given modulus, finds a polynomial $Qinv$ of length $n$ such that the top $n$ coefficients of the product $Q \times Qinv$ is $x^{n-1}$. Requires that $n > 0$. This function can be viewed as inverting a power series.

void nmod_poly_inv_series_basecase(nmod_poly_t Qinv, const nmod_poly_t Q, slong n)

Given $Q$ of length at least $n$ find $Qinv$ of length $n$ such that the top $n$ coefficients of the product $Q \times Qinv$ is $x^{n-1}$. An exception is raised if $n = 0$ or if the length of $Q$ is less than $n$. The leading coefficient of $Q$ must be invertible modulo the modulus of $Q$. This function can be viewed as inverting a power series.

void _nmod_poly_inv_series_newton(mp_ptr Qinv, mp_ptr Q, slong Qlen, slong n, nmod_t mod)

Given $Q$ of length $Qlen$ whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length $n$ such that $Q \times Qinv$ is 1 modulo $x^n$. Requires $n > 0$. This function can be viewed as inverting a power series via Newton iteration.

void nmod_poly_inv_series_newton(nmod_poly_t Qinv, const nmod_poly_t Q, slong n)

Given $Q$ find $Qinv$ such that $Q \times Qinv$ is 1 modulo $x^n$. The constant coefficient of $Q$ must be invertible modulo the modulus of $Q$. An exception is raised if this is not the case or if $n = 0$. This function can be viewed as inverting a power series via Newton iteration.

void _nmod_poly_inv_series(mp_ptr Qinv, mp_ptr Q, slong Qlen, slong n, nmod_t mod)

Given $Q$ of length $Qlen$ whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length $n$ such that $Q \times Qinv$ is 1 modulo $x^n$. Requires $n > 0$. This function can be viewed as inverting a power series.

void nmod_poly_inv_series(nmod_poly_t Qinv, const nmod_poly_t Q, slong n)

Given $Q$ find $Qinv$ such that $Q \times Qinv$ is 1 modulo $x^n$. The constant coefficient of $Q$ must be invertible modulo the modulus of $Q$. An exception is raised if this is not the case or if $n = 0$. This function can be viewed as inverting a power series.
Given polynomials $A$ and $B$ of length $\text{Alen}$ and $\text{Blen}$, finds the polynomial $Q$ of length $n$ such that $Q \cdot B = A$ modulo $x^n$. We assume $n > 0$ and that the constant coefficient of $B$ is invertible modulo the given modulus. The polynomial $Q$ must have space for $n$ coefficients.

The algorithm used is to call $\text{nmod_poly_divrem_newton_n_preinv()}$ and then multiply out and compute the remainder.

Notionally computes polynomials $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only $Q$. We require that $Q$ have space for $\text{len}(A) - \text{len}(B) + 1$ coefficients and assume that the leading coefficient of $B$ is a unit. Furthermore, we assume that $\text{Binv}$ is the inverse of the reverse of $B$ modulo $x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

Computes $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, where $A$ is of length $\text{len}(A)$ and $B$ is of length $\text{len}(B)$. We require that $Q$ have space for $\text{len}(A) - \text{len}(B) + 1$ coefficients. Furthermore, we assume that $\text{Binv}$ is the inverse of the reverse of $B$ modulo $x^{\text{len}(B)}$. The algorithm used is to call $\text{div_newton_n_preinv()}$ and then multiply out and compute the remainder.

Computes $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume $\text{Binv}$ is the inverse of the reverse of $B$ modulo $x^{\text{len}(B)}$.

It is required that the length of $A$ is less than or equal to $2^4$ the length of $B - 2$. The algorithm used is to call $\text{div_n()}$ and then multiply out and compute the remainder.
mp_limb_t nmod_poly_div_root(mp_ptr Q, mp_srcptr A, slong len, mp_limb_t c, nmod_t mod)

Sets \((Q, \text{len}-1)\) to the quotient of \((A, \text{len})\) on division by \((x - c)\), and returns the remainder, equal to the value of \(A\) evaluated at \(c\). \(A\) and \(Q\) are allowed to be the same, but may not overlap partially in any other way.

mp_limb_t nmod_poly_div_root(nmod_poly_t Q, const nmod_poly_t A, mp_limb_t c)

Sets \(Q\) to the quotient of \(A\) on division by \((x - c)\), and returns the remainder, equal to the value of \(A\) evaluated at \(c\).

6.4.19 Divisibility testing

int _nmod_poly_divides_classical(mp_ptr Q, mp_srcptr A, slong lenA, mp_srcptr B, slong lenB, nmod_t mod)

Returns 1 if \((B, \text{lenB})\) divides \((A, \text{lenA})\) and sets \((Q, \text{lenA} - \text{lenB} + 1)\) to the quotient. Otherwise, returns 0 and sets \((Q, \text{lenA} - \text{lenB} + 1)\) to zero. We require that \(\text{lenA} >= \text{lenB} > 0\).

int nmod_poly_divides_classical(nmod_poly_t Q, const nmod_poly_t A, const nmod_poly_t B)

Returns 1 if \(B\) divides \(A\) and sets \(Q\) to the quotient. Otherwise returns 0 and sets \(Q\) to zero.

int _nmod_poly_divides(mp_ptr Q, mp_srcptr A, slong lenA, mp_srcptr B, slong lenB, nmod_t mod)

Returns 1 if \((B, \text{lenB})\) divides \((A, \text{lenA})\) and sets \((Q, \text{lenA} - \text{lenB} + 1)\) to the quotient. Otherwise, returns 0 and sets \((Q, \text{lenA} - \text{lenB} + 1)\) to zero. We require that \(\text{lenA} >= \text{lenB} > 0\).

int nmod_poly_divides(nmod_poly_t Q, const nmod_poly_t A, const nmod_poly_t B)

Returns 1 if \(B\) divides \(A\) and sets \(Q\) to the quotient. Otherwise returns 0 and sets \(Q\) to zero.

ulong nmod_poly_remove(nmod_poly_t f, const nmod_poly_t p)

Removes the highest possible power of \(p\) from \(f\) and returns the exponent.

6.4.20 Derivative and integral

void _nmod_poly_derivative(mp_ptr x_prime, mp_srcptr x, slong len, nmod_t mod)

Sets the first \(\text{len} - 1\) coefficients of \(x\_prime\) to the derivative of \(x\) which is assumed to be of length \(\text{len}\). It is assumed that \(\text{len} > 0\).

void nmod_poly_derivative(nmod_poly_t x_prime, const nmod_poly_t x)

Sets \(x\_\text{prime}\) to the derivative of \(x\).

void _nmod_poly_integral(mp_ptr x_int, mp_srcptr x, slong len, nmod_t mod)

Set the first \(\text{len}\) coefficients of \(x\_\text{int}\) to the integral of \(x\) which is assumed to be of length \(\text{len} - 1\). The constant term of \(x\_\text{int}\) is set to zero. It is assumed that \(\text{len} > 0\). The result is only well-defined if the modulus is a prime number strictly larger than the degree of \(x\). Supports aliasing between the two polynomials.

void nmod_poly_integral(nmod_poly_t x_int, const nmod_poly_t x)

Set \(x\_\text{int}\) to the indefinite integral of \(x\) with constant term zero. The result is only well-defined if the modulus is a prime number strictly larger than the degree of \(x\).
6.4.21 Evaluation

\texttt{mp\_limb\_t nmod\_poly\_evaluate\_nmmod(mp\_srcptr poly, slong len, mp\_limb\_t c, nmod\_t mod)}

Evaluates \texttt{poly} at the value \texttt{c} and reduces modulo the given modulus of \texttt{poly}. The value \texttt{c} should be reduced modulo the modulus. The algorithm used is Horner’s method.

\texttt{mp\_limb\_t nmod\_poly\_evaluate\_nmmod(const nmod\_poly\_t poly, mp\_limb\_t c)}

Evaluates \texttt{poly} at the value \texttt{c} and reduces modulo the modulus of \texttt{poly}. The value \texttt{c} should be reduced modulo the modulus. The algorithm used is Horner’s method.

\texttt{void nmod\_poly\_evaluate\_mat\_horner(nmod\_mat\_t dest, const nmod\_poly\_t poly, const nmod\_mat\_t c)}

Evaluates \texttt{poly} with matrix as an argument at the value \texttt{c} and stores the result in \texttt{dest}. The dimension and modulus of \texttt{dest} is assumed to be same as that of \texttt{c}. \texttt{dest} and \texttt{c} may be aliased. Horner’s Method is used to compute the result.

\texttt{void nmod\_poly\_evaluate\_mat\_paterson\_stockmeyer(nmod\_mat\_t dest, const nmod\_poly\_t poly, const nmod\_mat\_t c)}

Evaluates \texttt{poly} with matrix as an argument at the value \texttt{c} and stores the result in \texttt{dest}. The dimension and modulus of \texttt{dest} is assumed to be same as that of \texttt{c}. \texttt{dest} and \texttt{c} may be aliased. Paterson-Stockmeyer algorithm is used to compute the result. The algorithm is described in [Paterson1973].

\texttt{void nmod\_poly\_evaluate\_mat(nmod\_mat\_t dest, const nmod\_poly\_t poly, const nmod\_mat\_t c)}

Evaluates \texttt{poly} with matrix as an argument at the value \texttt{c} and stores the result in \texttt{dest}. The dimension and modulus of \texttt{dest} is assumed to be same as that of \texttt{c}. \texttt{dest} and \texttt{c} may be aliased. This function automatically switches between Horner’s method and the Paterson-Stockmeyer algorithm.

6.4.22 Multipoint evaluation

\texttt{void \_nmod\_poly\_evaluate\_nmmod\_vec\_iter(mp\_ptr ys, mp\_srcptr poly, slong len, mp\_srcptr xs, slong n, nmod\_t mod)}

Evaluates (coeffs, len) at the \texttt{n} values given in the vector \texttt{xs}, writing the output values to \texttt{ys}. The values in \texttt{xs} should be reduced modulo the modulus. Uses Horner’s method iteratively.

\texttt{void nmod\_poly\_evaluate\_nmmod\_vec\_iter(mp\_ptr ys, const nmod\_poly\_t poly, mp\_srcptr xs, slong n)}

Evaluates \texttt{poly} at the \texttt{n} values given in the vector \texttt{xs}, writing the output values to \texttt{ys}. The values in \texttt{xs} should be reduced modulo the modulus. Uses Horner’s method iteratively.

\texttt{void \_nmod\_poly\_evaluate\_nmmod\_vec\_fast\_precomp(mp\_ptr vs, mp\_srcptr poly, slong plen, const mp\_ptr \*tree, slong len, nmod\_t mod)}

Evaluates (poly, plen) at the \texttt{len} values given by the precomputed subproduct tree \texttt{tree}.

\texttt{void \_nmod\_poly\_evaluate\_nmmod\_vec\_fast(mp\_ptr ys, mp\_srcptr poly, slong len, mp\_srcptr xs, slong n, nmod\_t mod)}

Evaluates (coeffs, len) at the \texttt{n} values given in the vector \texttt{xs}, writing the output values to \texttt{ys}. The values in \texttt{xs} should be reduced modulo the modulus. Uses fast multipoint evaluation, building a temporary subproduct tree.

\texttt{void nmod\_poly\_evaluate\_nmmod\_vec\_fast(mp\_ptr ys, const nmod\_poly\_t poly, mp\_srcptr xs, slong n)}

Evaluates \texttt{poly} at the \texttt{n} values given in the vector \texttt{xs}, writing the output values to \texttt{ys}. The values in \texttt{xs} should be reduced modulo the modulus. Uses fast multipoint evaluation, building a temporary subproduct tree.
void _nmod_poly_evaluate_nmod_vec(mp_ptr ys, mp_srcptr poly, slong len, mp_srcptr xs, slong n, nmod_t mod)

Evaluates (poly, len) at the n values given in the vector xs, writing the output values to ys. The values in xs should be reduced modulo the modulus.

void nmod_poly_evaluate_nmod_vec(mp_ptr ys, const nmod_poly_t poly, mp_srcptr xs, slong n)

Evaluates poly at the n values given in the vector xs, writing the output values to ys. The values in xs should be reduced modulo the modulus.

6.4.23 Interpolation

void _nmod_poly_interpolate_nmod_vec(mp_ptr poly, mp_srcptr xs, mp_srcptr ys, slong n, nmod_t mod)

Sets poly to the unique polynomial of length at most n that interpolates the n given evaluation points xs and values ys. If the interpolating polynomial is shorter than length n, the leading coefficients are set to zero.

The values in xs and ys should be reduced modulo the modulus, and all xs must be distinct. Aliasing between poly and xs or ys is not allowed.

void nmod_poly_interpolate_nmod_vec(nmod_poly_t poly, mp_srcptr xs, mp_srcptr ys, slong n)

Sets poly to the unique polynomial of length n that interpolates the n given evaluation points xs and values ys. The values in xs and ys should be reduced modulo the modulus, and all xs must be distinct.

void _nmod_poly_interpolation_weights(mp_ptr w, const mp_ptr *tree, slong len, nmod_t mod)

Sets w to the barycentric interpolation weights for fast Lagrange interpolation with respect to a given subproduct tree.

void _nmod_poly_interpolate_nmod_vec_fast_precomp(mp_ptr poly, mp_srcptr ys, const mp_ptr *tree, mp_srcptr weights, slong len, nmod_t mod)

Performs interpolation using the fast Lagrange interpolation algorithm, generating a temporary subproduct tree.

The function values are given as ys. The function takes a precomputed subproduct tree tree and barycentric interpolation weights weights corresponding to the roots.

void _nmod_poly_interpolate_nmod_vec_fast(mp_ptr poly, mp_srcptr xs, mp_srcptr ys, slong n, nmod_t mod)

Performs interpolation using the fast Lagrange interpolation algorithm, generating a temporary subproduct tree.

void nmod_poly_interpolate_nmod_vec_fast(nmod_poly_t poly, mp_srcptr xs, mp_srcptr ys, slong n)

Performs interpolation using the fast Lagrange interpolation algorithm, generating a temporary subproduct tree.

void _nmod_poly_interpolate_nmod_vec_newton(mp_ptr poly, mp_srcptr xs, mp_srcptr ys, slong n, nmod_t mod)

Forms the interpolating polynomial in the Newton basis using the method of divided differences and then converts it to monomial form.

void nmod_poly_interpolate_nmod_vec_newton(nmod_poly_t poly, mp_srcptr xs, mp_srcptr ys, slong n)

Forms the interpolating polynomial in the Newton basis using the method of divided differences and then converts it to monomial form.
void _nmod_poly_interpolate_nmod_vec_barycentric(mp_ptr poly, mp_srcptr xs, mp_srcptr ys, slong n, nmod_t mod)

Forms the interpolating polynomial using a naive implementation of the barycentric form of Lagrange interpolation.

void nmod_poly_interpolate_nmod_vec_barycentric(nmod_poly_t poly, mp_srcptr xs, mp_srcptr ys, slong n)

Forms the interpolating polynomial using a naive implementation of the barycentric form of Lagrange interpolation.

6.4.24 Composition

void _nmod_poly_compose_horner(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t mod)

Composes poly1 of length len1 with poly2 of length len2 and sets res to the result, i.e. evaluates poly1 at poly2. The algorithm used is Horner’s algorithm. We require that res have space for (len1 - 1)*(len2 - 1) + 1 coefficients. It is assumed that len1 > 0 and len2 > 0.

void nmod_poly_compose_horner(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)

Composes poly1 with poly2 and sets res to the result, i.e. evaluates poly1 at poly2. The algorithm used is Horner’s algorithm.

void _nmod_poly_compose_divconquer(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t mod)

Composes poly1 of length len1 with poly2 of length len2 and sets res to the result, i.e. evaluates poly1 at poly2. The algorithm used is the divide and conquer algorithm. We require that res have space for (len1 - 1)*(len2 - 1) + 1 coefficients. It is assumed that len1 > 0 and len2 > 0.

void nmod_poly_compose_divconquer(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)

Composes poly1 with poly2 and sets res to the result, i.e. evaluates poly1 at poly2. The algorithm used is the divide and conquer algorithm.

void _nmod_poly_compose(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t mod)

Composes poly1 of length len1 with poly2 of length len2 and sets res to the result, i.e. evaluates poly1 at poly2. We require that res have space for (len1 - 1)*(len2 - 1) + 1 coefficients. It is assumed that len1 > 0 and len2 > 0.

void nmod_poly_compose(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)

Composes poly1 with poly2 and sets res to the result, that is, evaluates poly1 at poly2.

6.4.25 Taylor shift

void _nmod_poly_taylor_shift_horner(mp_ptr poly, mp_limb_t c, slong len, nmod_t mod)

Performs the Taylor shift composing poly by \( x + c \) in-place. Uses an efficient version Horner’s rule.

void nmod_poly_taylor_shift_horner(nmod_poly_t g, const nmod_poly_t f, mp_limb_t c)

Performs the Taylor shift composing f by \( x + c \).

void _nmod_poly_taylor_shift_convolution(mp_ptr poly, mp_limb_t c, slong len, nmod_t mod)

Performs the Taylor shift composing poly by \( x + c \) in-place. Writes the composition as a single convolution with cost \( O(M(n)) \). We require that the modulus is a prime at least as large as the length.
void \texttt{nmod\_poly\_taylor\_shift\_convolution}(\texttt{nmod\_poly\_t} g, \texttt{const nmod\_poly\_t} f, \texttt{mp\_limb\_t} c)

Performs the Taylor shift composing \( f \) by \( x + c \). Writes the composition as a single convolution with cost \( O(M(n)) \). We require that the modulus is a prime at least as large as the length.

void \texttt{nmod\_poly\_taylor\_shift}(\texttt{mp\_ptr} poly, \texttt{mp\_limb\_t} c, \texttt{slong} len, \texttt{nmod\_t} mod)

Performs the Taylor shift composing \( poly \) by \( x + c \) in-place. We require that the modulus is a prime.

void \texttt{nmod\_poly\_taylor\_shift}(\texttt{nmod\_poly\_t} g, \texttt{const nmod\_poly\_t} f, \texttt{mp\_limb\_t} c)

Performs the Taylor shift composing \( f \) by \( x + c \). We require that the modulus is a prime.

\subsection{6.4.26 Modular composition}

void \texttt{nmod\_poly\_compose\_mod\_horner}(\texttt{mp\_ptr} res, \texttt{mp\_srcptr} f, \texttt{slong} lenf, \texttt{mp\_srcptr} g, \texttt{mp\_srcptr} h, \texttt{slong} lenh, \texttt{nmod\_t} mod)

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner’s rule.

void \texttt{nmod\_poly\_compose\_mod\_horner}(\texttt{nmod\_poly\_t} res, \texttt{const nmod\_poly\_t} f, \texttt{const nmod\_poly\_t} g, \texttt{const nmod\_poly\_t} h)

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero. The algorithm used is Horner’s rule.

void \texttt{nmod\_poly\_compose\_mod\_brent\_kung}(\texttt{mp\_ptr} res, \texttt{mp\_srcptr} f, \texttt{slong} lenf, \texttt{mp\_srcptr} g, \texttt{mp\_srcptr} h, \texttt{slong} lenh, \texttt{nmod\_t} mod)

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). We also require that the length of \( f \) is less than the length of \( h \). The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

void \texttt{nmod\_poly\_compose\_mod\_brent\_kung}(\texttt{nmod\_poly\_t} res, \texttt{const nmod\_poly\_t} f, \texttt{const nmod\_poly\_t} g, \texttt{const nmod\_poly\_t} h)

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). The algorithm used is the Brent-Kung matrix algorithm.

void \texttt{nmod\_poly\_compose\_mod\_brent\_kung\_preinv}(\texttt{mp\_ptr} res, \texttt{mp\_srcptr} f, \texttt{slong} lenf, \texttt{mp\_srcptr} g, \texttt{mp\_srcptr} h, \texttt{slong} lenh, \texttt{mp\_srcptr} hinv, \texttt{slong} lenhinv, \texttt{nmod\_t} mod)

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). We also require that the length of \( f \) is less than the length of \( h \). Furthermore, we require \( \texttt{hinv} \) to be the inverse of the reverse of \( h \). The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

void \texttt{nmod\_poly\_compose\_mod\_brent\_kung\_preinv}(\texttt{nmod\_poly\_t} res, \texttt{const nmod\_poly\_t} f, \texttt{const nmod\_poly\_t} g, \texttt{const nmod\_poly\_t} h, \texttt{const nmod\_poly\_t} hinv)

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). Furthermore, we require \( \texttt{hinv} \) to be the inverse of the reverse of \( h \). The algorithm used is the Brent-Kung matrix algorithm.

void \texttt{nmod\_poly\_reduce\_matrix\_mod\_poly}(\texttt{nmod\_mat\_t} A, \texttt{const nmod\_mat\_t} B, \texttt{const nmod\_poly\_t} f)

Sets the \( i \)th row of \( A \) to the reduction of the \( i \)th row of \( B \) modulo \( f \) for \( i = 1, \ldots, \sqrt{\text{deg}(f)} \). We require \( B \) to be at least a \( \sqrt{\text{deg}(f)} \times \text{deg}(f) \) matrix and \( f \) to be nonzero.

\section{6.4. \texttt{nmod\_poly\_h} – univariate polynomials over integers \texttt{mod n} (word-size \texttt{n})}

\subsection{6.4.26 Modular composition}
void _nmod_poly_precompute_matrix_worker(void *arg_ptr)
  Worker function version of _nmod_poly_precompute_matrix. Input/output is stored
  in nmod_poly_matrix_precompute_arg_t.

void _nmod_poly_precompute_matrix(nmod_mat_t A, mp_srcptr f, mp_srcptr g, slong leng,
  mp_srcptr ginv, slong lenginv, nmod_t mod)
  Sets the ith row of A to $f^i$ modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$
  matrix. We require ginv to be the inverse of the reverse of g and g to be nonzero. f has to be
  reduced modulo g and of length one less than leng (possibly with zero padding).

void nmod_poly_precompute_matrix(nmod_mat_t A, const nmod_poly_t f, const nmod_poly_t g,
  const nmod_poly_t ginv)
  Sets the ith row of A to $f^i$ modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$
  matrix. We require ginv to be the inverse of the reverse of g.

void _nmod_poly_compose_mod_brent_kung_precomp_preinv_worker(void *arg_ptr)
  Worker function version of _nmod_poly_compose_mod_brent_kung_precomp_preinv. Input/output is stored in
  nmod_poly_compose_mod_precomp_preinv_arg_t.

void _nmod_poly_compose_mod_brent_kung_precomp_preinv(mp_ptr res, mp_ptr f, slong lenf,
  const nmod_mat_t A, mp_ptr h, slong lenh, mp_ptr hinv, slong lenhinv, nmod_t mod)
  Sets res to the composition $f(h)$ modulo h. We require that h is nonzero. We require that the ith
  row of A contains $g^i$ for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require
  that the length of f is less than the length of h. Furthermore, we require hinv to be the inverse of
  the reverse of h. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

void nmod_poly_compose_mod_brent_kung_precomp_preinv(nmod_poly_t res, const nmod_poly_t f,
  const nmod_mat_t A, const nmod_poly_t h, const nmod_poly_t hinv)
  Sets res to the composition $f(h)$ modulo h. We require that the ith row of A contains $g^i$ for
  $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that
  f has smaller degree than h. Furthermore, we require hinv to be the inverse of the reverse of h.
  This version of Brent-Kung modular composition is particularly useful if one has to perform several
  modular composition of the form $f(g)$ modulo h for fixed g and h.

void _nmod_poly_compose_mod_brent_kung_vec_preinv(nmod_poly_struct *res, const
  nmod_poly_struct *polys, slong len1, slong l, mp_ptr g, slong leng, mp_ptr h,
  slong lenh, mp_ptr hinv, slong lenhinv, nmod_t mod)
  Sets res to the composition $f_i(g)$ modulo h for $1 \leq i \leq l$, where $f_i$ are the first l elements
  of polys. We require that h is nonzero and that the length of g is less than the length of h. We also
  require that the length of $f_i$ is less than the length of h. We require res to have enough memory
  allocated to hold l nmod_poly_struct’s. The entries of res need to be initialised and l needs to be
  less than len1. Furthermore, we require hinv to be the inverse of the reverse of h. The output
  is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

void nmod_poly_compose_mod_brent_kung_vec_preinv(nmod_poly_struct *res, const
  nmod_poly_struct *polys, slong len1, slong n, const nmod_poly_t g, const nmod_poly_t h,
  const nmod_poly_t hinv)
  Sets res to the composition $f_i(g)$ modulo h for $1 \leq i \leq n$ where $f_i$ are the first n elements
  of polys. We require res to have enough memory allocated to hold n nmod_poly_struct. The entries of res

366 Chapter 6. Integers mod n
need to be initialised and n needs to be less than len1. We require that h is nonzero and that \( f_i \) and \( g \) have smaller degree than \( h \). Furthermore, we require \( \text{hinv} \) to be the inverse of the reverse of \( h \). No aliasing of res and polys is allowed. The algorithm used is the Brent-Kung matrix algorithm.

```c
void _nmod_poly_compose_mod_brent_kung_vec_preinv_threaded_pool(nmod_poly_struct *res,
const nmod_poly_struct *polys, slong lempolys,
slong l, mp_srcptr g, slong glen,
mp_srcptr poly, slong len,
mp_srcptr polyinv, slong leninv, nmod_t mod,
thread_pool_handle *threads,
slong num_threads)
```

Multithreaded version of \( \text{nmod\_poly\_compose\_mod\_brent\_kung\_vec\_preinv}() \). Distributing the Horner evaluations across \( \text{flint\_get\_num\_threads}() \) threads.

```c
void nmod_poly_compose_mod_brent_kung_vec_preinv_threaded_pool(nmod_poly_struct *res,
const nmod_poly_struct *polys, slong lempolys,
const nmod_poly_t g, const nmod_poly_t poly,
const nmod_poly_t polyinv,
thread_pool_handle *threads, slong num_threads)
```

Multithreaded version of \( \text{nmod\_poly\_compose\_mod\_brent\_kung\_vec\_preinv}() \). Distributing the Horner evaluations across \( \text{flint\_get\_num\_threads}() \) threads.

```c
void nmod_poly_compose_mod_brent_kung_vec_preinv_threaded(nmod_poly_struct *res, const nmod_poly_struct *polys, slong len1, slong n, const nmod_poly_t g, const nmod_poly_t poly, const nmod_poly_t polyinv)
```

Multithreaded version of \( \text{nmod\_poly\_compose\_mod\_brent\_kung\_vec\_preinv}() \). Distributing the Horner evaluations across \( \text{flint\_get\_num\_threads}() \) threads.

```c
void _nmod_poly_compose_mod(mp_ptr res, mp_srcptr f, slong lenf, mp_srcptr g, mp_srcptr h, slong lenh, nmod_t mod)
```

Sets res to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```c
void nmod_poly_compose_mod(nmod_poly_t res, const nmod_poly_t f, const nmod_poly_t g, const nmod_poly_t h)
```

Sets res to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero.
6.4.27 Greatest common divisor

\texttt{slong \_nmod\_poly\_gcd\_euclidean(mp\_ptr G, mp\_srcptr A, slong lenA, mp\_srcptr B, slong lenB, nmod\_t mod)}

Computes the GCD of \(A\) of length \(\text{lenA}\) and \(B\) of length \(\text{lenB}\), where \(\text{lenA} \gg \text{lenB} > 0\). The length of the GCD \(G\) is returned by the function. No attempt is made to make the GCD monic. It is required that \(G\) have space for \(\text{lenB}\) coefficients.

\texttt{void nmod\_poly\_gcd\_euclidean(nmod\_poly\_t G, const nmod\_poly\_t A, const nmod\_poly\_t B)}

Computes the GCD of \(A\) and \(B\). The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \(P\) is defined to be \(P\). Except in the case where the GCD is zero, the GCD \(G\) is made monic.

\texttt{slong \_nmod\_poly\_hgcd(mp\_ptr *M, slong *lenM, mp\_ptr A, slong *lenA, mp\_ptr B, slong *lenB, mp\_srcptr a, slong lena, mp\_srcptr b, slong lenb, nmod\_t mod)}

Computes the HGCD of \(a\) and \(b\), that is, a matrix \(M\), a sign \(\sigma\) and two polynomials \(A\) and \(B\) such that \((A, B)^t = M^{-1}(a, b)^t, \sigma = \det(M), \)

\text{and \(A\) and \(B\) are consecutive remainders in the Euclidean remainder sequence for the division of \(a\) by \(b\) satisfying \(\deg(A) \geq \frac{\deg(a)}{2} > \deg(B)\). Furthermore, \(M\) will be the product of \([[q 1] [1 0]]\) for the quotients \(q\) generated by such a remainder sequence. Assumes that \(\text{len}(a) > \text{len}(b) > 0\), i.e. \(\deg(a) > \deg(b) > 1\).}

Assumes that \(A\) and \(B\) have space of size at least \(\text{len}(a)\) and \(\text{len}(b)\), respectively. On exit, \(*\text{lenA}\) and \(*\text{lenB}\) will contain the correct lengths of \(A\) and \(B\).

Assumes that \(\text{M}[0], \text{M}[1], \text{M}[2]\), and \(\text{M}[3]\) each point to a vector of size at least \(\text{len}(a)\).

\texttt{slong \_nmod\_poly\_gcd\_hgcd(mp\_ptr G, mp\_srcptr A, slong lenA, mp\_srcptr B, slong lenB, nmod\_t mod)}

Computes the monic GCD of \(A\) and \(B\), assuming that \(\text{len}(A) \geq \text{len}(B) > 0\).

Assumes that \(G\) has space for \(\text{len}(B)\) coefficients and returns the length of \(G\) on output.

\texttt{void nmod\_poly\_gcd\_hgcd(nmod\_poly\_t G, const nmod\_poly\_t A, const nmod\_poly\_t B)}

Computes the monic GCD of \(A\) and \(B\) using the HGCD algorithm.

As a special case, the GCD of two zero polynomials is defined to be the zero polynomial.

The time complexity of the algorithm is \(O(n \log^2 n)\). For further details, see \cite{ThullYap1990}.

\texttt{slong \_nmod\_poly\_gcd(mp\_ptr G, mp\_srcptr A, slong lenA, mp\_srcptr B, slong lenB, nmod\_t mod)}

Computes the GCD of \(A\) of length \(\text{lenA}\) and \(B\) of length \(\text{lenB}\), where \(\text{lenA} \gg \text{lenB} > 0\). The length of the GCD \(G\) is returned by the function. No attempt is made to make the GCD monic. It is required that \(G\) have space for \(\text{lenB}\) coefficients.

\texttt{void nmod\_poly\_gcd(nmod\_poly\_t G, const nmod\_poly\_t A, const nmod\_poly\_t B)}

Computes the GCD of \(A\) and \(B\). The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \(P\) is defined to be \(P\). Except in the case where the GCD is zero, the GCD \(G\) is made monic.

\texttt{slong \_nmod\_poly\_xgcd\_euclidean(mp\_ptr G, mp\_ptr S, mp\_ptr T, mp\_srcptr A, slong A\_len, mp\_srcptr B, slong B\_len, nmod\_t mod)}

Computes the GCD of \(A\) and \(B\) together with cofactors \(S\) and \(T\) such that \(SA + TB = G\). Returns the length of \(G\). Assumes that \(\text{len}(A) \geq \text{len}(B) \geq 1\) and \((\text{len}(A), \text{len}(B)) \neq (1, 1)\).

No attempt is made to make the GCD monic.
Requires that \( G \) have space for \( \text{len}(B) \) coefficients. Writes \( \text{len}(B) - 1 \) and \( \text{len}(A) - 1 \) coefficients to \( S \) and \( T \), respectively. Note that, in fact, \( \text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1) \) and \( \text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1) \).

No aliasing of input and output operands is permitted.

```c
void nmod_poly_xgcd_euclidean(nmod_poly_t G, nmod_poly_t S, nmod_poly_t T, const nmod_poly_t A, const nmod_poly_t B)
```
Computes the GCD of \( A \) and \( B \). The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \( P \) is defined to be \( P \). Except in the case where the GCD is zero, the GCD \( G \) is made monic.

Polynomials \( S \) and \( T \) are computed such that \( S*A + T*B = G \). The length of \( S \) will be at most \( \text{len}B \) and the length of \( T \) will be at most \( \text{len}A \).

```c
slong _nmod_poly_xgcd_hgcd(mp_ptr G, mp_ptr S, mp_ptr T, mp_srcptr A, slong A_len, mp_srcptr B, slong B_len, nmod_t mod)
```
Computes the GCD of \( A \) and \( B \), where \( \text{len}(A) \geq \text{len}(B) > 0 \), together with cofactors \( S \) and \( T \) such that \( SA + TB = G \). Returns the length of \( G \).

No attempt is made to make the GCD monic.

Requires that \( G \) have space for \( \text{len}(B) \) coefficients. Writes \( \text{len}(B) - 1 \) and \( \text{len}(A) - 1 \) coefficients to \( S \) and \( T \), respectively. Note that, in fact, \( \text{len}(S) \leq \text{len}(B) - \text{len}(G) \) and \( \text{len}(T) \leq \text{len}(A) - \text{len}(G) \).

Both \( S \) and \( T \) must have space for at least 2 coefficients.

No aliasing of input and output operands is permitted.

```c
void nmod_poly_xgcd_hgcd(nmod_poly_t G, nmod_poly_t S, nmod_poly_t T, const nmod_poly_t A, const nmod_poly_t B)
```
Computes the GCD of \( A \) and \( B \). The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \( P \) is defined to be \( P \). Except in the case where the GCD is zero, the GCD \( G \) is made monic.

Polynomials \( S \) and \( T \) are computed such that \( S*A + T*B = G \). The length of \( S \) will be at most \( \text{len}B \) and the length of \( T \) will be at most \( \text{len}A \).

```c
slong _nmod_poly_xgcd(mp_ptr G, mp_ptr S, mp_ptr T, mp_srcptr A, slong lenA, mp_srcptr B, slong lenB, nmod_t mod)
```
Computes the GCD of \( A \) and \( B \), where \( \text{len}(A) \geq \text{len}(B) > 0 \), together with cofactors \( S \) and \( T \) such that \( SA + TB = G \). Returns the length of \( G \).

No attempt is made to make the GCD monic.

Requires that \( G \) have space for \( \text{len}(B) \) coefficients. Writes \( \text{len}(B) - 1 \) and \( \text{len}(A) - 1 \) coefficients to \( S \) and \( T \), respectively. Note that, in fact, \( \text{len}(S) \leq \text{len}(B) - \text{len}(G) \) and \( \text{len}(T) \leq \text{len}(A) - \text{len}(G) \).

No aliasing of input and output operands is permitted.

```c
void nmod_poly_xgcd(nmod_poly_t G, nmod_poly_t S, nmod_poly_t T, const nmod_poly_t A, const nmod_poly_t B)
```
Computes the GCD of \( A \) and \( B \). The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \( P \) is defined to be \( P \). Except in the case where the GCD is zero, the GCD \( G \) is made monic.

The polynomials \( S \) and \( T \) are set such that \( S*A + T*B = G \). The length of \( S \) will be at most \( \text{len}B \) and the length of \( T \) will be at most \( \text{len}A \).

```c
mp_limb_t nmod_poly_resultant_euclidean(mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t mod)
```
Returns the resultant of \((\text{poly}1, \text{len}1)\) and \((\text{poly}2, \text{len}2)\) using the Euclidean algorithm.

Assumes that \( \text{len}1 \geq \text{len}2 > 0 \).

Assumes that the modulus is prime.
**Flint Documentation, Release 3.1.0-dev**

`mp_limb_t nmod_poly_resultant_euclidean(const nmod_poly_t f, const nmod_poly_t g)`

Computes the resultant of \( f \) and \( g \) using the Euclidean algorithm.

For two non-zero polynomials \( f(x) = a_m x^m + \cdots + a_0 \) and \( g(x) = b_n x^n + \cdots + b_0 \) of degrees \( m \) and \( n \), the resultant is defined to be

\[
a_m b_n^m \prod_{(x,y):f(x)=g(y)=0} (x-y).
\]

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

`mp_limb_t nmod_poly_resultant_hgcd(mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t mod)`

Returns the resultant of \( (poly1, \text{len1}) \) and \( (poly2, \text{len2}) \) using the half-gcd algorithm.

This algorithm computes the half-gcd as per \( \_nmod_poly_gcd_hgcd() \) but additionally updates the resultant every time a division occurs. The half-gcd algorithm computes the GCD recursively. Given inputs \( a \) and \( b \) it lets \( m = \text{len}(a)/2 \) and (recursively) performs all quotients in the Euclidean algorithm which do not require the low \( m \) coefficients of \( a \) and \( b \).

This performs quotients in exactly the same order as the ordinary Euclidean algorithm except that the low \( m \) coefficients of the polynomials in the remainder sequence are not computed. A correction step after hgcd has been called computes these low \( m \) coefficients (by matrix multiplication by a transformation matrix also computed by hgcd).

This means that from the point of view of the resultant, all but the last quotient performed by a recursive call to hgcd is an ordinary quotient as per the usual Euclidean algorithm. However, the final quotient may give a remainder of less than \( m+1 \) coefficients, which won’t be corrected until the hgcd correction step is performed afterwards.

To compute the adjustments to the resultant coming from this corrected quotient, we save the relevant information in an \_nmod_poly_res_t struct at the time the quotient is performed so that when the correction step is performed later, the adjustments to the resultant can be computed at that time also.

The only time an adjustment to the resultant is not required after a call to hgcd is if hgcd does nothing (the remainder may already have had less than \( m+1 \) coefficients when hgcd was called).

Assumes that \( \text{len1} \geq \text{len2} > 0 \).

Assumes that the modulus is prime.

`mp_limb_t nmod_poly_resultant_hgcd(const nmod_poly_t f, const nmod_poly_t g)`

Computes the resultant of \( f \) and \( g \) using the half-gcd algorithm.

For two non-zero polynomials \( f(x) = a_m x^m + \cdots + a_0 \) and \( g(x) = b_n x^n + \cdots + b_0 \) of degrees \( m \) and \( n \), the resultant is defined to be

\[
a_m b_n^m \prod_{(x,y):f(x)=g(y)=0} (x-y).
\]

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

`mp_limb_t nmod_poly_resultant(mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t mod)`

Returns the resultant of \( (poly1, \text{len1}) \) and \( (poly2, \text{len2}) \).

Assumes that \( \text{len1} \geq \text{len2} > 0 \).

Assumes that the modulus is prime.
For two non-zero polynomials \( f(x) = a_m x^m + \cdots + a_0 \) and \( g(x) = b_n x^n + \cdots + b_0 \) of degrees \( m \) and \( n \), the resultant is defined to be

\[
a_n b_m \prod_{(x,y):f(x)=g(y)=0} (x - y).
\]

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

\[
\text{slong}_n \text{mod}_n \text{poly}_n \text{gcdinv}(mp \_limb_t \*G, mp \_limb_t \*S, \text{const} mp \_limb_t \*A, \text{slong lenA}, \text{const} mp \_limb_t \*B, \text{slong lenB}, \text{const nmod}_n \_t \text{mod})
\]

Computes \((G, \text{lenA}), (S, \text{lenB}-1)\) such that \(G \sim S \cdot A \mod B\), returning the actual length of \(G\).

Assumes that \(0 < \text{len}(A) < \text{len}(B)\).

\[
\text{void nmod}_n \text{poly}_n \text{gcdinv}(nmod \_poly_t \*G, nmod \_poly_t \*S, \text{const} nmod \_poly_t \*A, \text{const} nmod \_poly_t \*B)
\]

Computes polynomials \(G\) and \(S\), both reduced modulo \(B\), such that \(G \equiv SA \mod B\), where \(B\) is assumed to have \(\text{len}(B) \geq 2\).

In the case that \(A = 0 \mod B\), returns \(G = S = 0\).

\[
\text{int nmod}_n \text{poly}_n \text{invmod}(mp \_limb_t \*A, \text{const} mp \_limb_t \*B, \text{slong lenB}, \text{const} mp \_limb_t \*P, \text{slong lenP}, \text{const nmod}_n \_t \text{mod})
\]

Attempts to set \((A, \text{lenP}-1)\) to the inverse of \((B, \text{lenB})\) modulo the polynomial \((P, \text{lenP})\). Returns 1 if \((B, \text{lenB})\) is invertible and 0 otherwise.

Assumes that \(0 < \text{len}(B) < \text{len}(P)\), and hence also \(\text{len}(P) \geq 2\), but supports zero-padding in \((B, \text{lenB})\).

Does not support aliasing.

Assumes that \(\text{mod}\) is a prime number.

\[
\text{int nmod}_n \text{poly}_n \text{invmod}(nmod \_poly_t \*A, \text{const} nmod \_poly_t \*B, \text{const} nmod \_poly_t \*P)
\]

Attempts to set \(A\) to the inverse of \(B\) modulo \(P\) in the polynomial ring \((\mathbb{Z}/p\mathbb{Z})[X]\), where we assume that \(p\) is a prime number.

If \(\text{len}(P) < 2\), raises an exception.

If the greatest common divisor of \(B\) and \(P\) is 1, returns 1 and sets \(A\) to the inverse of \(B\). Otherwise, returns 0 and the value of \(A\) on exit is undefined.

### 6.4.28 Discriminant

\[
mp \_limb_t \_nmod \_poly \_discriminant(mp \_srcptr poly, \text{slong len}, \text{nmod}_n \_t \text{mod})
\]

Return the discriminant of \((poly, \text{len})\). Assumes \(\text{len} > 1\).

\[
mp \_limb_t \_nmod \_poly \_discriminant(\text{const} nmod \_poly_t \*f)
\]

Return the discriminant of \(f\). We normalise the discriminant so that \(\text{disc}(f) = (-1)^{n(n-1)/2} \text{res}(f, f') / \text{lc}(f)^{n-m-2}\), where \(n = \text{len}(f)\) and \(m = \text{len}(f')\). Thus \(\text{disc}(f) = \text{lc}(f)^{2n-2} \prod_{i<j} (r_i - r_j)^2\), where \(\text{lc}(f)\) is the leading coefficient of \(f\) and \(r_i\) are the roots of \(f\).
6.4.29 Power series composition

void _nmod_poly_compose_series(mp_ptr res, mp_srcptr poly1, slong len1, mp_srcptr poly2, slong len2, slong n, nmod_t mod)

Sets res to the composition of poly1 and poly2 modulo \( x^n \), where the constant term of poly2 is required to be zero.

Assumes that len1, len2, n > 0, that len1, len2 <= n, and that \((\text{len1}-1) \cdot (\text{len2}-1) + 1 = n\), and that res has space for n coefficients. Does not support aliasing between any of the inputs and the output.

Wraps _gr_poly_compose_series() which chooses automatically between various algorithms.

void nmod_poly_compose_series(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, slong n)

Sets res to the composition of poly1 and poly2 modulo \( x^n \), where the constant term of poly2 is required to be zero.

6.4.30 Power series reversion

void _nmod_poly_revert_series(mp_ptr Qinv, mp_srcptr Q, slong Qlen, slong n, nmod_t mod)

void nmod_poly_revert_series(nmod_poly_t Qinv, const nmod_poly_t Q, slong n)

Sets Qinv to the compositional inverse or reversion of Q as a power series, i.e. computes \( Q^{-1} \) such that \( Q^{-1}(Q(x)) = x \mod x^n \).

It is required that \( Q_0 = 0 \) and that \( Q_1 \) as well as the integers 1, 2, ..., \( n - 1 \) are invertible modulo the modulus.

Wraps _gr_poly_revert_series() which chooses automatically between various algorithms.

6.4.31 Square roots

The series expansions for \( \sqrt{h} \) and \( 1/\sqrt{h} \) are defined by means of the generalised binomial theorem \( h^{r} = (1+y)^{r} = \sum_{k=0}^{\infty} \binom{r}{k} y^{k} \). It is assumed that h has constant term 1 and that the coefficients \( 2^{-k} \) exist in the coefficient ring (i.e. 2 must be invertible).

void _nmod_poly_invsqrt_series(mp_ptr g, mp_srcptr h, slong hlen, slong n, nmod_t mod)

Set the first n terms of g to the series expansion of \( 1/\sqrt{h} \). It is assumed that \( n > 0 \), that h has constant term 1. Aliasing is not permitted.

void nmod_poly_invsqrt_series(nmod_poly_t g, const nmod_poly_t h, slong n)

Set g to the series expansion of \( 1/\sqrt{h} \) to order \( O(x^n) \). It is assumed that h has constant term 1.

void _nmod_poly_sqrt_series(mp_ptr g, mp_srcptr h, slong hlen, slong n, nmod_t mod)

Set the first n terms of g to the series expansion of \( \sqrt{h} \). It is assumed that \( n > 0 \), that h has constant term 1. Aliasing is not permitted.

void nmod_poly_sqrt_series(nmod_poly_t g, const nmod_poly_t h, slong n)

Set g to the series expansion of \( \sqrt{h} \) to order \( O(x^n) \). It is assumed that h has constant term 1.

int _nmod_poly_sqrt(mp_ptr s, mp_srcptr p, slong n, nmod_t mod)

If \( p, n \) is a perfect square, sets \( s, n / 2 + 1 \) to a square root of p and returns 1. Otherwise returns 0.

int nmod_poly_sqrt(nmod_poly_t s, const nmod_poly_t p)

If p is a perfect square, sets s to a square root of p and returns 1. Otherwise returns 0.
6.4.32 Power sums

void _nmod_poly_power_sums_naive(mp_ptr res, mp_srcptr poly, slong len, slong n, nmod_t mod)
Compute the (truncated) power sums series of the polynomial (poly,len) up to length n using
Newton identities.

void nmod_poly_power_sums_naive(nmod_poly_t res, const nmod_poly_t poly, slong n)
Compute the (truncated) power sum series of the polynomial poly up to length n using Newton
identities.

void _nmod_poly_power_sums_schoenhage(mp_ptr res, mp_srcptr poly, slong len, slong n, nmod_t mod)
Compute the (truncated) power sums series of the polynomial (poly,len) up to length n using a
series expansion (a formula due to Schoenhage).

void nmod_poly_power_sums_schoenhage(nmod_poly_t res, const nmod_poly_t poly, slong n)
Compute the (truncated) power sums series of the polynomial poly up to length n using a series
expansion (a formula due to Schoenhage).

void _nmod_poly_power_sums(mp_ptr res, mp_srcptr poly, slong len, slong n, nmod_t mod)
Compute the (truncated) power sums series of the polynomial (poly,len) up to length n.

void nmod_poly_power_sums(nmod_poly_t res, const nmod_poly_t poly, slong n)
Compute the (truncated) power sums series of the polynomial poly up to length n.

void _nmod_poly_power_sums_to_poly_naive(mp_ptr res, mp_srcptr poly, slong len, nmod_t mod)
Compute the (monic) polynomial given by its power sums series (poly,len) using Newton iden-
tities.

void nmod_poly_power_sums_to_poly_naive(nmod_poly_t res, const nmod_poly_t Q)
Compute the (monic) polynomial given by its power sums series Q using Newton identities.

void _nmod_poly_power_sums_to_poly_schoenhage(mp_ptr res, mp_srcptr poly, slong len, nmod_t mod)
Compute the (monic) polynomial given by its power sums series (poly,len) using series expansion
(a formula due to Schoenhage).

void nmod_poly_power_sums_to_poly_schoenhage(nmod_poly_t res, const nmod_poly_t Q)
Compute the (monic) polynomial given by its power sums series Q using series expansion (a formula
due to Schoenhage).

void _nmod_poly_power_sums_to_poly(mp_ptr res, mp_srcptr poly, slong len, nmod_t mod)
Compute the (monic) polynomial given by its power sums series (poly,len).

void nmod_poly_power_sums_to_poly(nmod_poly_t res, const nmod_poly_t Q)
Compute the (monic) polynomial given by its power sums series Q.

6.4.33 Transcendental functions

The elementary transcendental functions of a formal power series $h$ are defined as

$$
\exp(h(x)) = \sum_{k=0}^{\infty} \frac{(h(x))^k}{k!} \\
\log(h(x)) = \int_0^x \frac{h'(t)}{h(t)} dt \\
\arctan(h(x)) = \int_0^x \frac{h'(t)}{1+(h(t))^2} dt \\
\operatorname{artanh}(h(x)) = \int_0^x \frac{h'(t)}{1-(h(t))^2} dt \\
\arcsin(h(x)) = \int_0^x \frac{h'(t)}{\sqrt{1-(h(t))^2}} dt
$$
asinh(h(x)) = \int_0^x \frac{h'(t)}{\sqrt{1 + h(t)^2}} dt

The functions sin, cos, tan, etc. are defined using standard inverse or functional relations. The logarithm function assumes that h has constant term 1. All other functions assume that h has constant term 0. All functions assume that the coefficient 1/k or 1/k! exists for all indices k. When computing to order \(O(x^n)\), the modulus p must therefore be a prime satisfying \(p \geq n\). Further, we always require that \(p > 2\) in order to be able to multiply by 1/2 for internal purposes. If the input does not satisfy all these conditions, results are undefined. Except where otherwise noted, functions are implemented with optimal (up to constants) complexity \(O(M(n))\), where \(M(n)\) is the cost of polynomial multiplication.

void _nmod_poly_log_series(mp_ptr g, mp_srcptr h, slong hlen, slong n, nmod_t mod)
Set g = log(h) + O(x^n). Assumes \(n > 0\) and \(hlen > 0\). Aliasing of g and h is allowed.

void _nmod_poly_log_series(nmod_poly_t g, const nmod_poly_t h, slong n)
Set g = log(h) + O(x^n). The case h = 1 + cx^r is automatically detected and handled efficiently.

void _nmod_poly_exp_series(mp_ptr f, mp_srcptr h, slong hlen, slong n, nmod_t mod)
Set f = exp(h) + O(x^n) where h is a polynomial. Assume \(n > 0\). Aliasing of g and h is not allowed.
Uses Newton iteration (an improved version of the algorithm in [HanZim2004]). For small n, falls back to the basecase algorithm.

void _nmod_poly_expexpinv_series(mp_ptr f, mp_ptr g, mp_srcptr h, slong hlen, slong n, nmod_t mod)
Set f = exp(h) + O(x^n) and g = exp(-h) + O(x^n), more efficiently for large n than performing a separate inversion to obtain g. Assumes \(n > 0\) and that h is zero-padded as necessary to length n. Aliasing is not allowed.
Uses Newton iteration (the version given in [HanZim2004]). For small n, falls back to the basecase algorithm.

void _nmod_poly_exp_series(nmod_poly_t g, const nmod_poly_t h, slong n)
Set g = exp(h) + O(x^n). The case h = cx^r is automatically detected and handled efficiently. Otherwise this function automatically uses the basecase algorithm for small n and Newton iteration otherwise.

void _nmod_poly_atan_series(mp_ptr f, mp_srcptr h, slong hlen, slong n, nmod_t mod)
Set g = atan(h) + O(x^n). Assumes \(n > 0\). Aliasing of g and h is allowed.

void _nmod_poly_atan_series(nmod_poly_t g, const nmod_poly_t h, slong n)
Set g = atan(h) + O(x^n).

void _nmod_poly_atanh_series(mp_ptr f, mp_srcptr h, slong hlen, slong n, nmod_t mod)
Set g = atanh(h) + O(x^n). Assumes \(n > 0\). Aliasing of g and h is allowed.

void _nmod_poly_atanh_series(nmod_poly_t g, const nmod_poly_t h, slong n)
Set g = atanh(h) + O(x^n).

void _nmod_poly_asin_series(mp_ptr f, mp_srcptr h, slong hlen, slong n, nmod_t mod)
Set g = asin(h) + O(x^n). Assumes \(n > 0\). Aliasing of g and h is allowed.

void _nmod_poly_asin_series(nmod_poly_t g, const nmod_poly_t h, slong n)
Set g = asin(h) + O(x^n).

void _nmod_poly_asinh_series(mp_ptr f, mp_srcptr h, slong hlen, slong n, nmod_t mod)
Set g = asinh(h) + O(x^n). Assumes \(n > 0\) and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed. The value is computed using the identity \(\sin(x) = 2\tan(x/2)/(1+\tan^2(x/2))\).
void nmod_poly_sin_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set $g = \sin(h) + O(x^n)$.  

void _nmod_poly_cos_series(mp_ptr g, mp_srcptr h, slong n, nmod_t mod)
    Set $g = \cos(h) + O(x^n)$. Assumes $n > 0$ and that $h$ is zero-padded as necessary to length $n$. Aliasing of $g$ and $h$ is allowed. The value is computed using the identity $\cos(x) = (1 - \tan^2(x/2))/(1 + \tan^2(x/2))$.

void nmod_poly_cos_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set $g = \cos(h) + O(x^n)$.  

void _nmod_poly_tan_series(mp_ptr g, mp_srcptr h, slong n, nmod_t mod)
    Set $g = \tan(h) + O(x^n)$. Assumes $n > 0$ and that $h$ is zero-padded as necessary to length $n$. Aliasing of $g$ and $h$ is allowed. The value is computed using the identity $\tan(h) = (e^{x} - e^{-x})/2$.

void nmod_poly_tan_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set $g = \tan(h) + O(x^n)$.  

void _nmod_poly_sin_series(mp_ptr g, mp_srcptr h, slong n, nmod_t mod)
    Set $g = \sinh(h) + O(x^n)$. Assumes $n > 0$ and that $h$ is zero-padded as necessary to length $n$. Aliasing of $g$ and $h$ is not allowed. Uses the identity $\sinh(x) = (e^x - e^{-x})/2$.

void nmod_poly_sinh_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set $g = \sinh(h) + O(x^n)$.  

void _nmod_poly_cosh_series(mp_ptr g, mp_srcptr h, slong n, nmod_t mod)
    Set $g = \cosh(h) + O(x^n)$. Assumes $n > 0$ and that $h$ is zero-padded as necessary to length $n$. Aliasing of $g$ and $h$ is not allowed. Uses the identity $\cosh(x) = (e^x + e^{-x})/2$.

void nmod_poly_cosh_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set $g = \cosh(h) + O(x^n)$.  

void _nmod_poly_tanh_series(mp_ptr g, mp_srcptr h, slong n, nmod_t mod)
    Set $g = \tanh(h) + O(x^n)$. Assumes $n > 0$ and that $h$ is zero-padded as necessary to length $n$. Uses the identity $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$.

void nmod_poly_tanh_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set $g = \tanh(h) + O(x^n)$.  

### 6.4.34 Special polynomials

int _nmod_poly_conway(mp_ptr op, ulong prime, slong deg)
    Sets $op$ to the coefficients to the Conway polynomial $C_{p,d}$, where $p$ is $prime$ and $d$ is $deg$. This is done by checking against Frank Lübeck’s database [Lüb2004], which has been compressed in FLINT. Returns 1 in case of success and returns 0 in case of failure.

ulong _nmod_poly_conway_rand(slong *degree, flint_rand_t state, int type)
    Returns a pseudorandom prime and sets $degree$ that when put into `_nmod_poly_conway()` will always succeed.

Here, $type$ can be the following values:

- 0 for which there is a bijection between the image of this function and the database of Conway polynomials,
- 1 returns a random prime found in the database and sets $degree$ to some degree less than 15 along with some prime found in the database,
- 2 returns a random prime less than $2^{10}$ and sets $degree$ to some random degree found in the database,
- 3 returns a random prime less than $2^{10}$ and sets $degree$ to some random degree less than 15.
6.4.35 Products

void _nmod_poly_product_roots_nmod_vec(mp_ptr poly, mp_srcptr xs, slong n, nmod_t mod)

Sets (poly, n + 1) to the monic polynomial which is the product of \((x - x_0)(x - x_1)\cdots(x - x_{n-1})\), the roots \(x_i\) being given by \(xs\).

Aliasing of the input and output is not allowed.

void nmod_poly_product_roots_nmod_vec(nmod_poly_t poly, mp_srcptr xs, slong n)

Sets poly to the monic polynomial which is the product of \((x - x_0)(x - x_1)\cdots(x - x_{n-1})\), the roots \(x_i\) being given by \(xs\).

int nmod_poly_find_distinct_nonzero_roots(mp_limb_t *roots, const nmod_poly_t A)

If \(A\) has \(\deg(A)\) distinct nonzero roots in \(\mathbb{F}_p\), write these roots out to \(\text{roots}[0]\) to \(\text{roots}[\deg(A) - 1]\) and return 1. Otherwise, return 0. It is assumed that \(A\) is nonzero and that the modulus of \(A\) is prime. This function uses Rabin’s probabilistic method via \(\text{gcd}'s\) with \((x + \delta)p^{\frac{n}{2} - 1}\).

6.4.36 Subproduct trees

\(mp_ptr *\_nmod\_poly\_tree\_alloc(slong\ \text{len})\)

Allocates space for a subproduct tree of the given length, having linear factors at the lowest level.

Entry \(i\) in the tree is a pointer to a single array of limbs, capable of storing \(\lfloor n/2^i \rfloor\) subproducts of degree \(2^i\) adjacently, plus a trailing entry if \(n/2^i\) is not an integer.

For example, a tree of length 7 built from monic linear factors has the following structure, where spaces have been inserted for illustrative purposes:

\[
\begin{align*}
X1 & \quad X1 \quad X1 \quad X1 \quad X1 \quad X1 \\
XX1 & \quad XX1 \quad XX1 \quad X1 \\
XXXX1 & \quad XX1 \quad X1 \\
XXXXXXX1 & \\
\end{align*}
\]

void _nmod_poly_tree_free(mp_ptr *tree, slong len)

Free the allocated space for the subproduct.

void _nmod_poly_tree_build(mp_ptr *tree, mp_srcptr roots, slong len, nmod_t mod)

Builds a subproduct tree in the preallocated space from the len monic linear factors \((x - r_i)\). The top level product is not computed.

6.4.37 Inflation and deflation

void nmod_poly Inflate(nmod_poly_t result, const nmod_poly_t input, ulong inflation)

Sets result to the inflated polynomial \(p(x^n)\) where \(p\) is given by input and \(n\) is given by deflation.

void nmod_poly_deflate(nmod_poly_t result, const nmod_poly_t input, ulong deflation)

Sets result to the deflated polynomial \(p(x^{1/n})\) where \(p\) is given by input and \(n\) is given by deflation. Requires \(n > 0\).

ulong nmod_poly_deflation(const nmod_poly_t input)

Returns the largest integer by which input can be deflated. As special cases, returns 0 if input is the zero polynomial and 1 if input is a constant polynomial.
6.4.38 Chinese Remaindering

In all of these functions the moduli (mod.n) of all of the nmod_poly's involved is assumed to match and be prime.

void nmod_poly_multi_crt_init(nmod_poly_multi_crt_t CRT)
Initialize CRT for Chinese remaindering.

int nmod_poly_multi_crt_precompute(nmod_poly_multi_crt_t CRT, const nmod_poly_struct *moduli, slong len)

int nmod_poly_multi_crt_precompute_p(nmod_poly_multi_crt_t CRT, const nmod_poly_struct *const *moduli, slong len)
Configure CRT for repeated Chinese remaindering of moduli. The number of moduli, len, should be positive. A return of 0 indicates that the compilation failed and future calls to nmod_poly_multi_crt_precomp() will leave the output undefined. A return of 1 indicates that the compilation was successful, which occurs if and only if either (1) len == 1 and modulus + 0 is nonzero, or (2) all of the moduli have positive degree and are pairwise relatively prime.

void nmod_poly_multi_crt_precomp(nmod_poly_t output, const nmod_poly_multi_crt_t CRT, const nmod_poly_struct *values)
void nmod_poly_multi_crt_precomp_p(nmod_poly_t output, const nmod_poly_multi_crt_t CRT, const nmod_poly_struct *const *values)
Set output to the polynomial of lowest possible degree that is congruent to values + i modulo the moduli + i in nmod_poly_multi_crt_precompute(). The inputs values + 0, ..., values + len - 1 where len was used in nmod_poly_multi_crt_precompute() are expected to be valid and have modulus matching the modulus of the moduli used in nmod_poly_multi_crt_precompute().

int nmod_poly_multi_crt(nmod_poly_t output, const nmod_poly_struct *moduli, const nmod_poly_struct *values, slong len)
Perform the same operation as nmod_poly_multi_crt_precomp() while internally constructing and destroying the precomputed data. All of the remarks in nmod_poly_multi_crt_precompute() apply.

void nmod_poly_multi_crt_clear(nmod_poly_multi_crt_t CRT)
Free all space used by CRT.

slong _nmod_poly_multi_crt_local_size(const nmod_poly_multi_crt_t CRT)
Return the required length of the output for _nmod_poly_multi_crt_run().

void _nmod_poly_multi_crt_run(nmod_poly_struct *outputs, const nmod_poly_multi_crt_t CRT, const nmod_poly_struct *inputs)
void _nmod_poly_multi_crt_run_p(nmod_poly_struct *outputs, const nmod_poly_multi_crt_t CRT, const nmod_poly_struct *const *inputs)
Perform the same operation as nmod_poly_multi_crt_precomp() using supplied temporary space. The actual output is placed in outputs + 0, and outputs should contain space for all temporaries and should be at least as long as _nmod_poly_multi_crt_local_size(CRT). Of course the moduli of these temporaries should match the modulus of the inputs.
6.4.39 Berlekamp-Massey Algorithm

The `nmod_berlekamp_massey_t` manages an unlimited stream of points \(a_1, a_2, \ldots\). At any point in time, after, say, \(n\) points have been added, a call to `nmod_berlekamp_massey_reduce()` will calculate the polynomials \(U, V\) and \(R\) in the extended Euclidean remainder sequence with

\[ U x^n + V(a_1 x^{n-1} + a_{n-1} x + \cdots + a_n) = R, \quad \deg(U) < \deg(V) \leq n/2, \quad \deg(R) < n/2. \]

The polynomials \(V\) and \(R\) may be obtained with `nmod_berlekamp_massey_V_poly()` and `nmod_berlekamp_massey_R_poly()`. This class differs from `fmpz_mod_poly_minpoly()` in the following respect. Let \(v_i\) denote the coefficient of \(x^i\) in \(V\). `fmpz_mod_poly_minpoly()` will return a polynomial \(V\) of lowest degree that annihilates the whole sequence \(a_1, \ldots, a_n\) as

\[ \sum_i v_i a_j x^i = 0, \quad 1 \leq j < n - \deg(V). \]

The cost is that a polynomial of degree \(n - 1\) might be returned and the return is not generally uniquely determined by the input sequence. For the `nmod_berlekamp_massey_t` we have

\[ \sum_{i,j} v_i a_{j+i} x^{-j} = -U + \frac{R}{x^n}, \]

and it can be seen that \(\sum_{i,j} v_i a_{j+i}\) is zero for \(1 \leq j < n - \deg(R)\). Thus whether or not \(V\) has annihilated the whole sequence may be checked by comparing the degrees of \(V\) and \(R\).

```c
void nmod_berlekamp_massey_init(nmod_berlekamp_massey_t B, mp_limb_t p)
    Initialize B in characteristic p with an empty stream.

void nmod_berlekamp_massey_clear(nmod_berlekamp_massey_t B)
    Free any space used by B.

void nmod_berlekamp_massey_start_over(nmod_berlekamp_massey_t B)
    Empty the stream of points in B.

void nmod_berlekamp_massey_set_prime(nmod_berlekamp_massey_t B, mp_limb_t p)
    Set the characteristic of the field and empty the stream of points in B.

void nmod_berlekamp_massey_add_points(nmod_berlekamp_massey_t B, const mp_limb_t *a, slong count)
    Add point(s) to the stream processed by B. The addition of any number of points will not update the V and R polynomial.

int nmod_berlekamp_massey_reduce(nmod_berlekamp_massey_t B)
    Ensure that the polynomials V and R are up to date. The return value is 1 if this function changed V and 0 otherwise. For example, if this function is called twice in a row without adding any points in between, the return of the second call should be 0. As another example, suppose the object is emptied, the points 1, 1, 2, 3 are added, then reduce is called. This reduce should return 1 with \(\deg(R) < \deg(V) = 2\) because the Fibonacci sequence has been recognized. The further addition of the two points 5, 8 and a reduce will result in a return value of 0.

slong nmod_berlekamp_massey_point_count(const nmod_berlekamp_massey_t B)
    Return the number of points stored in B.

const mp_limb_t *nmod_berlekamp_massey_points(const nmod_berlekamp_massey_t B)
    Return a pointer to the array of points stored in B. This may be NULL if nmod_berlekamp_massey_point_count() returns 0.
```
const nmod_poly_struct *nmod_berlekamp_massey_V_poly(const nmod_berlekamp_massey_t B)
    Return the polynomial $V$ in $B$.

const nmod_poly_struct *nmod_berlekamp_massey_R_poly(const nmod_berlekamp_massey_t B)
    Return the polynomial $R$ in $B$.

6.5 nmod_poly_mat.h – matrices of univariate polynomials over integers mod n (word-size n)

The nmod_poly_mat_t data type represents matrices whose entries are polynomials having coefficients in $\mathbb{Z}/n\mathbb{Z}$. We generally assume that $n$ is a prime number.

The nmod_poly_mat_t type is defined as an array of nmod_poly_mat_struct’s of length one. This permits passing parameters of type nmod_poly_mat_t by reference.

A matrix internally consists of a single array of nmod_poly_struct’s, representing a dense matrix in row-major order. This array is only directly indexed during memory allocation and deallocation. A separate array holds pointers to the start of each row, and is used for all indexing. This allows the rows of a matrix to be permuted quickly by swapping pointers.

Matrices having zero rows or columns are allowed.

The shape of a matrix is fixed upon initialisation. The user is assumed to provide input and output variables whose dimensions are compatible with the given operation.

6.5.1 Types, macros and constants

type nmod_poly_mat_struct

type nmod_poly_mat_t

6.5.2 Memory management

void nmod_poly_mat_init(nmod_poly_mat_t mat, slong rows, slong cols, mp_limb_t n)
    Initialises a matrix with the given number of rows and columns for use. The modulus is set to $n$.

void nmod_poly_mat_init_set(nmod_poly_mat_t mat, const nmod_poly_mat_t src)
    Initialises a matrix $mat$ of the same dimensions and modulus as $src$, and sets it to a copy of $src$.

void nmod_poly_mat_clear(nmod_poly_mat_t mat)
    Frees all memory associated with the matrix. The matrix must be reinitialised if it is to be used again.

6.5.3 Truncate, shift

void nmod_poly_mat_set_trunc(nmod_poly_mat_t res, const nmod_poly_mat_t pmat, long len)
    Set $res$ to the truncation of $pmat$ to length $len$. Entries of $res$ are normalized.

void nmod_poly_mat_truncate(nmod_poly_mat_t pmat, long len)
    Truncates $pmat$ to the given length $len$, and normalize its entries. If $len$ is greater than the maximum length of the entries of $pmat$, then nothing happens.

void nmod_poly_mat_shift_left(nmod_poly_mat_t res, const nmod_poly_mat_t pmat, slong k)
    Sets $res$ to $pmat$ shifted left by $k$ coefficients, that is, multiplied by $x^k$.  

6.5. nmod_poly_mat.h – matrices of univariate polynomials over integers mod n (word-size n)
void nmod_poly_mat_shift_right(nmod_poly_mat_t res, const nmod_poly_mat_t pmat, slong k)
Sets res to pmat shifted right by k coefficients, that is, divide by $x^k$ and throw away the remainder. If k is greater than or equal to the length of pmat, the result is the zero polynomial matrix.

6.5.4 Basic properties

slong nmod_poly_mat_nrows(const nmod_poly_mat_t mat)
Returns the number of rows in mat.

slong nmod_poly_mat_ncols(const nmod_poly_mat_t mat)
Returns the number of columns in mat.

mp_limb_t nmod_poly_mat_modulus(const nmod_poly_mat_t mat)
Returns the modulus of mat.

6.5.5 Basic assignment and manipulation

nmod_poly_struct *nmod_poly_mat_entry(const nmod_poly_mat_t mat, slong i, slong j)
 Gives a reference to the entry at row i and column j. The reference can be passed as an input or output variable to any nmod_poly function for direct manipulation of the matrix element. No bounds checking is performed.

void nmod_poly_mat_set(nmod_poly_mat_t mat1, const nmod_poly_mat_t mat2)
Sets mat1 to a copy of mat2.

void nmod_poly_mat_set_nmod_mat(nmod_poly_mat_t pmat, const nmod_mat_t cmat)
Sets the already-initialized polynomial matrix pmat to a constant matrix with the same entries as cmat. Both input matrices must have the same dimensions and modulus.

void nmod_poly_mat_swap(nmod_poly_mat_t mat1, nmod_poly_mat_t mat2)
Swaps mat1 and mat2 efficiently.

void nmod_poly_mat_swap_entrywise(nmod_poly_mat_t mat1, nmod_poly_mat_t mat2)
Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

6.5.6 Input and output

void nmod_poly_mat_print(const nmod_poly_mat_t mat, const char *x)
Prints the matrix mat to standard output, using the variable x.

6.5.7 Random matrix generation

void nmod_poly_mat_randtest(nmod_poly_mat_t mat, flint_rand_t state, slong len)
This is equivalent to applying nmod_poly_randtest to all entries in the matrix.

void nmod_poly_mat_randtest_sparse(nmod_poly_mat_t A, flint_rand_t state, slong len, float density)
Creates a random matrix with the amount of nonzero entries given approximately by the density variable, which should be a fraction between 0 (most sparse) and 1 (most dense).
The nonzero entries will have random lengths between 1 and len.
6.5.8 Special matrices

```c
void nmod_poly_mat_zero(nmod_poly_mat_t mat)
    Sets mat to the zero matrix.

void nmod_poly_mat_one(nmod_poly_mat_t mat)
    Sets mat to the unit or identity matrix of given shape, having the element 1 on the main diagonal and zeros elsewhere. If mat is nonsquare, it is set to the truncation of a unit matrix.
```

6.5.9 Basic comparison and properties

```c
int nmod_poly_mat_equal(const nmod_poly_mat_t mat1, const nmod_poly_mat_t mat2)
    Returns nonzero if mat1 and mat2 have the same shape and all their entries agree, and returns zero otherwise.

int nmod_poly_mat_equal_nmod_mat(const nmod_poly_mat_t pmat, const nmod_mat_t cmat)
    Returns nonzero if pmat is a constant matrix with the same dimensions and entries as cmat; returns zero otherwise.

int nmod_poly_mat_is_zero(const nmod_poly_mat_t mat)
    Returns nonzero if all entries in mat are zero, and returns zero otherwise.

int nmod_poly_mat_is_one(const nmod_poly_mat_t mat)
    Returns nonzero if all entry of mat on the main diagonal are the constant polynomial 1 and all remaining entries are zero, and returns zero otherwise. The matrix need not be square.

int nmod_poly_mat_is_empty(const nmod_poly_mat_t mat)
    Returns a non-zero value if the number of rows or the number of columns in mat is zero, and otherwise returns zero.

int nmod_poly_mat_is_square(const nmod_poly_mat_t mat)
    Returns a non-zero value if the number of rows is equal to the number of columns in mat, and otherwise returns zero.

void nmod_poly_mat_get_coeff_mat(nmod_mat_t coeff, const nmod_poly_mat_t pmat, slong deg)
    Sets coeff to be the coefficient of pmat of degree deg, where pmat is seen as a polynomial with matrix coefficients and coefficients are numbered from zero. coeff must be already initialized with the right dimensions and modulus. For entries of pmat of degree less than deg, the corresponding entry of coeff is zero.

void nmod_poly_mat_set_coeff_mat(nmod_poly_mat_t pmat, const nmod_mat_t coeff, slong deg)
    Sets the coefficient of pmat of degree deg to coeff, where pmat is seen as a polynomial with matrix coefficients and coefficients are numbered from zero. For each entry of pmat, if deg is larger than its degree, this entry is first resized to the appropriate length, with intervening coefficients being set to zero.
```

6.5.10 Norms

```c
slong nmod_poly_mat_max_length(const nmod_poly_mat_t A)
    Returns the maximum polynomial length among all the entries in A.

slong nmod_poly_mat_degree(const nmod_poly_mat_t pmat)
    Returns the degree of the polynomial matrix pmat. The zero matrix is deemed to have degree −1.
```
6.5.11 Evaluation

```c
void nmod_poly_mat_evaluate_nmod(nmod_mat_t B, const nmod_poly_mat_t A, mp_limb_t x)
```

Sets the `nmod_mat_t B` to `A` evaluated entrywise at the point `x`.

6.5.12 Arithmetic

```c
void nmod_poly_mat_scalar_mul_nmod_poly(nmod_poly_mat_t B, const nmod_poly_mat_t A, const nmod_poly_t c)
```

Sets `B` to `A` multiplied entrywise by the polynomial `c`.

```c
void nmod_poly_mat_scalar_mul_nmod(nmod_poly_mat_t B, const nmod_poly_mat_t A, mp_limb_t c)
```

Sets `B` to `A` multiplied entrywise by the coefficient `c`, which is assumed to be reduced modulo the modulus.

```c
void nmod_poly_mat_add(nmod_poly_mat_t C, const nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the sum of `A` and `B`. All matrices must have the same shape. Aliasing is allowed.

```c
void nmod_poly_mat_sub(nmod_poly_mat_t C, const nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the sum of `A` and `B`. All matrices must have the same shape. Aliasing is allowed.

```c
void nmod_poly_mat_neg(nmod_poly_mat_t B, const nmod_poly_mat_t A)
```

Sets `B` to the negation of `A`. The matrices must have the same shape. Aliasing is allowed.

```c
void nmod_poly_mat_mul(nmod_poly_mat_t C, const nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical, KS and evaluation-interpolation multiplication.

```c
void nmod_poly_mat_mul_classical(nmod_poly_mat_t C, const nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`, computed using the classical algorithm. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

```c
void nmod_poly_mat_mul_KS(nmod_poly_mat_t C, const nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`, computed using Kronecker segmentation. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

```c
void nmod_poly_mat_mul_interpolate(nmod_poly_mat_t C, const nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`, computed through evaluation and interpolation. The matrices must have compatible dimensions for matrix multiplication. For interpolation to be well-defined, we require that the modulus is a prime at least as large as \( m + n - 1 \) where \( m \) and \( n \) are the maximum lengths of polynomials in the input matrices. Aliasing is allowed.

```c
void nmod_poly_mat_sqr(nmod_poly_mat_t B, const nmod_poly_mat_t A)
```

Sets `B` to the square of `A`, which must be a square matrix. Aliasing is allowed. This function automatically chooses between classical and KS squaring.

```c
void nmod_poly_mat_sqr_classical(nmod_poly_mat_t B, const nmod_poly_mat_t A)
```

Sets `B` to the square of `A`, which must be a square matrix. Aliasing is allowed. This function uses direct formulas for very small matrices, and otherwise classical matrix multiplication.
void nmod_poly_mat_sqr_KS(nmod_poly_mat_t B, const nmod_poly_mat_t A)
Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function uses Kronecker segmentation.

void nmod_poly_mat_sqr_interpolate(nmod_poly_mat_t B, const nmod_poly_mat_t A)
Sets B to the square of A, which must be a square matrix, computed through evaluation and interpolation. For interpolation to be well-defined, we require that the modulus is a prime at least as large as $2^n - 1$ where $n$ is the maximum length of polynomials in the input matrix. Aliasing is allowed.

void nmod_poly_mat_pow(nmod_poly_mat_t B, const nmod_poly_mat_t A, ulong exp)
Sets B to A raised to the power exp, where A is a square matrix. Uses exponentiation by squaring. Aliasing is allowed.

6.5.13 Row reduction

slong nmod_poly_mat_find_pivot_any(const nmod_poly_mat_t mat, slong start_row, slong end_row, slong c)
Attempts to find a pivot entry for row reduction. Returns a row index $r$ between start_row (inclusive) and stop_row (exclusive) such that column $c$ in mat has a nonzero entry on row $r$, or returns -1 if no such entry exists.
This implementation simply chooses the first nonzero entry from it encounters. This is likely to be a nearly optimal choice if all entries in the matrix have roughly the same size, but can lead to unnecessary coefficient growth if the entries vary in size.

slong nmod_poly_mat_find_pivot_partial(const nmod_poly_mat_t mat, slong start_row, slong end_row, slong c)
Attempts to find a pivot entry for row reduction. Returns a row index $r$ between start_row (inclusive) and stop_row (exclusive) such that column $c$ in mat has a nonzero entry on row $r$, or returns -1 if no such entry exists.
This implementation searches all the rows in the column and chooses the nonzero entry of smallest degree. This heuristic typically reduces coefficient growth when the matrix entries vary in size.

slong nmod_poly_mat_fflu(nmod_poly_mat_t B, nmod_poly_t den, slong *perm, const nmod_poly_mat_t A, int rank_check)
Uses fraction-free Gaussian elimination to set (B, den) to a fraction-free LU decomposition of A and returns the rank of A. Aliasing of A and B is allowed.
Pivot elements are chosen with nmod_poly_mat_find_pivot_partial. If perm is non-NULL, the permutation of rows in the matrix will also be applied to perm.
If rank_check is set, the function aborts and returns 0 if the matrix is detected not to have full rank without completing the elimination.
The denominator den is set to $\pm \det(A)$, where the sign is decided by the parity of the permutation. Note that the determinant is not generally the minimal denominator.

slong nmod_poly_mat_rref(nmod_poly_mat_t B, nmod_poly_t den, const nmod_poly_mat_t A)
Sets (B, den) to the reduced row echelon form of A and returns the rank of A. Aliasing of A and B is allowed.
The denominator den is set to $\pm \det(A)$. Note that the determinant is not generally the minimal denominator.
6.5.14 Trace

void nmod_poly_mat_trace(nmod_poly_t trace, const nmod_poly_mat_t mat)

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

6.5.15 Determinant and rank

void nmod_poly_mat_det(nmod_poly_t det, const nmod_poly_mat_t A)

Sets det to the determinant of the square matrix A. Uses a direct formula, fraction-free LU decomposition, or interpolation, depending on the size of the matrix.

void nmod_poly_mat_det_fflu(nmod_poly_t det, const nmod_poly_mat_t A)

Sets det to the determinant of the square matrix A. The determinant is computed by performing a fraction-free LU decomposition on a copy of A.

void nmod_poly_mat_det_interpolate(nmod_poly_t det, const nmod_poly_mat_t A)

Sets det to the determinant of the square matrix A. The determinant is computed by determining a bound \( n \) for its length, evaluating the matrix at \( n \) distinct points, computing the determinant of each coefficient matrix, and forming the interpolating polynomial.

If the coefficient ring does not contain \( n \) distinct points (that is, if working over \( \mathbb{Z}/p\mathbb{Z} \) where \( p < n \)), this function automatically falls back to nmod_poly_mat_det_fflu.

slong nmod_poly_mat_rank(const nmod_poly_mat_t A)

Returns the rank of A. Performs fraction-free LU decomposition on a copy of A.

6.5.16 Inverse

int nmod_poly_mat_inv(nmod_poly_mat_t Ainv, nmod_poly_t den, const nmod_poly_mat_t A)

Sets (Ainv, den) to the inverse matrix of A. Returns 1 if A is nonsingular and 0 if A is singular. Aliasing of Ainv and A is allowed.

More precisely, det will be set to the determinant of A and Ainvo will be set to the adjugate matrix of A. Note that the determinant is not necessarily the minimal denominator.

Uses fraction-free LU decomposition, followed by solving for the identity matrix.

6.5.17 Nullspace

slong nmod_poly_mat_nullspace(nmod_poly_mat_t res, const nmod_poly_mat_t mat)

Computes the right rational nullspace of the matrix mat and returns the nullity.

More precisely, assume that mat has rank \( r \) and nullity \( n \). Then this function sets the first \( n \) columns of res to linearly independent vectors spanning the nullspace of mat. As a result, we always have rank(res) = \( n \), and mat × res is the zero matrix.

The computed basis vectors will not generally be in a reduced form. In general, the polynomials in each column vector in the result will have a nontrivial common GCD.
6.5.18 Solving

```c
int nmod_poly_mat_solve(nmod_poly_mat_t X, nmod_poly_t den, const nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Solves the equation $AX = B$ for nonsingular $A$. More precisely, computes $(X, \text{den})$ such that $AX = B \times \text{den}$. Returns 1 if $A$ is nonsingular and 0 if $A$ is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```c
int nmod_poly_mat_solve_fflu(nmod_poly_mat_t X, nmod_poly_t den, const nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Solves the equation $AX = B$ for nonsingular $A$. More precisely, computes $(X, \text{den})$ such that $AX = B \times \text{den}$. Returns 1 if $A$ is nonsingular and 0 if $A$ is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```c
void nmod_poly_mat_solve_fflu_precomp(nmod_poly_mat_t X, const slong *perm, const nmod_poly_mat_t FFLU, const nmod_poly_mat_t B)
```

Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation.

6.6 nmod_poly_factor.h – factorisation of univariate polynomials over integers mod n (word-size n)

6.6.1 Types, macros and constants

```c
type nmod_poly_factor_struct

type nmod_poly_factor_t
```

6.6.2 Factorisation

```c
void nmod_poly_factor_init(nmod_poly_factor_t fac)
```

Initialises `fac` for use. An `nmod_poly_factor_t` represents a polynomial in factorised form as a product of polynomials with associated exponents.

```c
void nmod_poly_factor_clear(nmod_poly_factor_t fac)
```

Frees all memory associated with `fac`.

```c
void nmod_poly_factor_realloc(nmod_poly_factor_t fac, slong alloc)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```c
void nmod_poly_factor_fit_length(nmod_poly_factor_t fac, slong len)
```

Ensures that the factor structure has space for at least `len` factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

```c
void nmod_poly_factor_set(nmod_poly_factor_t res, const nmod_poly_factor_t fac)
```

Sets `res` to the same factorisation as `fac`.

```c
void nmod_poly_factor_print(const nmod_poly_factor_t fac)
```

Prints the entries of `fac` to standard output.
void nmod_poly_factor_insert(nmod_poly_factor_t fac, const nmod_poly_t poly, slong exp)
    Inserts the factor poly with multiplicity exp into the factorisation fac.
    If fac already contains poly, then exp simply gets added to the exponent of the existing entry.

void nmod_poly_factor_concat(nmod_poly_factor_t res, const nmod_poly_factor_t fac)
    Concatenates two factorisations. This is equivalent to calling nmod_poly_factor_insert() repeatedly with the individual factors of fac.
    Does not support aliasing between res and fac.

void nmod_poly_factor_pow(nmod_poly_factor_t fac, slong exp)
    Raises fac to the power exp.

int nmod_poly_is_irreducible(const nmod_poly_t f)
    Returns 1 if the polynomial f is irreducible, otherwise returns 0.

int nmod_poly_is_irreducible_ddf(const nmod_poly_t f)
    Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

int nmod_poly_is_irreducible_rabin(const nmod_poly_t f)
    Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses Rabin irreducibility test.

int _nmod_poly_is_squarefree(mp_srcptr f, slong len, nmod_t mod)
    Returns 1 if (f, len) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

int nmod_poly_is_squarefree(const nmod_poly_t f)
    Returns 1 if f is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

void nmod_poly_factor_squarefree(nmod_poly_factor_t res, const nmod_poly_t f)
    Sets res to a square-free factorization of f.

int nmod_poly_factor_equal_deg_prob(nmod_poly_t factor, flint_rand_t state, const nmod_poly_t pol, slong d)
    Probabilistic equal degree factorisation of pol into irreducible factors of degree d. If it passes, a factor is placed in factor and 1 is returned, otherwise 0 is returned and the value of factor is undetermined.
    Requires that pol be monic, non-constant and squarefree.

void nmod_poly_factor_equal_deg(nmod_poly_factor_t factors, const nmod_poly_t pol, slong d)
    Assuming pol is a product of irreducible factors all of degree d, finds all those factors and places them in factors. Requires that pol be monic, non-constant and squarefree.

void nmod_poly_factor_distinct_deg(nmod_poly_factor_t res, const nmod_poly_t poly, slong *const *degs)
    Factorises a monic non-constant squarefree polynomial poly of degree n into factors f[d] such that for 1 \leq d \leq n f[d] is the product of the monic irreducible factors of poly of degree d. Factors f[d] are stored in res, and the degree d of the irreducible factors is stored in degs in the same order as the factors.
    Requires that degs has enough space for (n/2)+1 * sizeof(slong).

void nmod_poly_factor_distinct_deg_threaded(nmod_poly_factor_t res, const nmod_poly_t poly, slong *const *degs)
    Multithreaded version of nmod_poly_factor_distinct_deg().
void nmod_poly_factor_cantor_zassenhaus(nmod_poly_factor_t res, const nmod_poly_t f)
    Factorises a non-constant polynomial f into monic irreducible factors using the Cantor-Zassenhaus algorithm.

void nmod_poly_factor_berlekamp(nmod_poly_factor_t res, const nmod_poly_t f)
    Factorises a non-constant, squarefree polynomial f into monic irreducible factors using the Berlekamp algorithm.

void nmod_poly_factor_kaltofen_shoup(nmod_poly_factor_t res, const nmod_poly_t poly)
    Factorises a non-constant polynomial f into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a “baby step/giant step” strategy for the distinct-degree factorization step. If flint_get_num_threads() is greater than one nmod_poly_factor_distinct_deg_threaded() is used.

mp_limb_t nmod_poly_factor_with_berlekamp(nmod_poly_factor_t res, const nmod_poly_t f)
    Factorises a general polynomial f into monic irreducible factors and returns the leading coefficient of f, or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs Berlekamp on all the individual square-free factors.

mp_limb_t nmod_poly_factor_with_cantor_zassenhaus(nmod_poly_factor_t res, const nmod_poly_t f)
    Factorises a general polynomial f into monic irreducible factors and returns the leading coefficient of f, or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

mp_limb_t nmod_poly_factor_with_kaltofen_shoup(nmod_poly_factor_t res, const nmod_poly_t f)
    Factorises a general polynomial f into monic irreducible factors and returns the leading coefficient of f, or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

mp_limb_t nmod_poly_factor(nmod_poly_factor_t res, const nmod_poly_t f)
    Factorises a general polynomial f into monic irreducible factors and returns the leading coefficient of f, or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs either Cantor-Zassenhaus or Berlekamp on all the individual square-free factors. Currently Cantor-Zassenhaus is used by default unless the modulus is 2, in which case Berlekamp is used.

void _nmod_poly_interval_poly_worker(void **arg_ptr)
    Worker function to compute interval polynomials in distinct degree factorisation. Input/output is stored in nmod_poly_interval_poly_arg_t.
6.7 nmod_mpoly.h – multivariate polynomials over integers mod n (word-size n)

The exponents follow the mpoly interface. A coefficient may be referenced as a mp_limb_t *.

6.7.1 Types, macros and constants

type nmod_mpoly_struct
A structure holding a multivariate polynomial over the integers modulo n for word-sized n.

type nmod_mpoly_t
An array of length 1 of nmod_mpoly_struct.

type nmod_mpoly_ctx_struct
Context structure representing the parent ring of an nmod_mpoly.

type nmod_mpoly_ctx_t
An array of length 1 of nmod_mpoly_ctx_struct.

6.7.2 Context object

void nmod_mpoly_ctx_init(nmod_mpoly_ctx_t ctx, slong nvars, const ordering_t ord, mp_limb_t n)
Initialise a context object for a polynomial ring with the given number of variables and the given ordering. It will have coefficients modulo n. Setting n = 0 will give undefined behavior. The possibilities for the ordering are ORD_LEX, ORD_DEGLEX and ORD_DEGREVLEX.

slong nmod_mpoly_ctx_nvars(const nmod_mpoly_ctx_t ctx)
Return the number of variables used to initialize the context.

ordering_t nmod_mpoly_ctx_ord(const nmod_mpoly_ctx_t ctx)
Return the ordering used to initialize the context.

mp_limb_t nmod_mpoly_ctx_modulus(const nmod_mpoly_ctx_t ctx)
Return the modulus used to initialize the context.

void nmod_mpoly_ctx_clear(nmod_mpoly_ctx_t ctx)
Release any space allocated by ctx.

6.7.3 Memory management

void nmod_mpoly_init(nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
Initialise A for use with the given an initialised context object. Its value is set to zero.

void nmod_mpoly_init2(nmod_mpoly_t A, slong alloc, const nmod_mpoly_ctx_t ctx)
Initialise A for use with the given an initialised context object. Its value is set to zero. It is allocated with space for alloc terms and at least MPOLY_MIN_BITS bits for the exponent widths.

void nmod_mpoly_init3(nmod_mpoly_t A, slong alloc, flint_bitcnt_t bits, const nmod_mpoly_ctx_t ctx)
Initialise A for use with the given an initialised context object. Its value is set to zero. It is allocated with space for alloc terms and bits bits for the exponents.

void nmod_mpoly_fit_length(nmod_mpoly_t A, slong len, const nmod_mpoly_ctx_t ctx)
Ensure that A has space for at least len terms.
void nmod_mpoly_realloc(nmod_mpoly_t A, slong alloc, const nmod_mpoly_ctx_t ctx)
    Reallocate A to have space for alloc terms. Assumes the current length of the polynomial is not greater than alloc.

void nmod_mpoly_clear(nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Release any space allocated for A.

6.7.4 Input/Output

The variable strings in x start with the variable of most significance at index 0. If x is NULL, the variables are named x1, x2, etc.

char *nmod_mpoly_get_str_pretty(const nmod_mpoly_t A, const char **x, const nmod_mpoly_ctx_t ctx)
    Return a string, which the user is responsible for cleaning up, representing A, given an array of variable strings x.

int nmod_mpoly_fprint_pretty(FILE *file, const nmod_mpoly_t A, const char **x, const nmod_mpoly_ctx_t ctx)
    Print a string representing A to file.

int nmod_mpoly_print_pretty(const nmod_mpoly_t A, const char **x, const nmod_mpoly_ctx_t ctx)
    Print a string representing A to stdout.

int nmod_mpoly_set_str_pretty(nmod_mpoly_t A, const char *str, const char **x, const nmod_mpoly_ctx_t ctx)
    Set A to the polynomial in the null-terminates string str given an array x of variable strings. If parsing str fails, A is set to zero, and -1 is returned. Otherwise, 0 is returned. The operations +, -, *, and / are permitted along with integers and the variables in x. The character ^ must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

6.7.5 Basic manipulation

void nmod_mpoly_gen(nmod_mpoly_t A, slong var, const nmod_mpoly_ctx_t ctx)
    Set A to the variable of index var, where var = 0 corresponds to the variable with the most significance with respect to the ordering.

int nmod_mpoly_is_gen(const nmod_mpoly_t A, slong var, const nmod_mpoly_ctx_t ctx)
    If var ≥ 0, return 1 if A is equal to the var-th generator, otherwise return 0. If var < 0, return 1 if the polynomial is equal to any generator, otherwise return 0.

void nmod_mpoly_set(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)
    Set A to B.

int nmod_mpoly_equal(const nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)
    Return 1 if A is equal to B, else return 0.

void nmod_mpoly_swap(nmod_mpoly_t A, nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)
    Efficiently swap A and B.
6.7.6 Constants

```c
int nmod_mpoly_is_ui(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Return 1 if A is a constant, else return 0.

ulong nmod_mpoly_get_ui(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Assuming that A is a constant, return this constant. This function throws if A is not a constant.

void nmod_mpoly_set_ui(nmod_mpoly_t A, ulong c, const nmod_mpoly_ctx_t ctx)
    Set A to the constant c.

void nmod_mpoly_zero(nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Set A to the constant 0.

void nmod_mpoly_one(nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Set A to the constant 1.

int nmod_mpoly_equal_ui(const nmod_mpoly_t A, ulong c, const nmod_mpoly_ctx_t ctx)
    Return 1 if A is equal to the constant c, else return 0.

int nmod_mpoly_is_zero(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Return 1 if A is the constant 0, else return 0.

int nmod_mpoly_is_one(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Return 1 if A is the constant 1, else return 0.
```

6.7.7 Degrees

```c
int nmod_mpoly_degrees_fit_si(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Return 1 if the degrees of A with respect to each variable fit into an slong, otherwise return 0.

void nmod_mpoly_degrees_fmpz(fmpz **degs, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
void nmod_mpoly_degrees_si(slong *degs, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Set degs to the degrees of A with respect to each variable. If A is zero, all degrees are set to −1.

void nmod_mpoly_degree_fmpz(fmpz_t deg, const nmod_mpoly_t A, slong var, const nmod_mpoly_ctx_t ctx)

slong nmod_mpoly_degree_si(const nmod_mpoly_t A, slong var, const nmod_mpoly_ctx_t ctx)
    Either return or set deg to the degree of A with respect to the variable of index var. If A is zero, the degree is defined to be −1.

int nmod_mpoly_total_degree_fits_si(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Return 1 if the total degree of A fits into an slong, otherwise return 0.

void nmod_mpoly_total_degree_fmpz(fmpz_t tdeg, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)

slong nmod_mpoly_total_degree_si(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Either return or set tdeg to the total degree of A. If A is zero, the total degree is defined to be −1.

void nmod_mpoly_used_vars(int *used, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    For each variable index i, set used[i] to nonzero if the variable of index i appears in A and to zero otherwise.
```
6.7.8 Coefficients

ulong nmod_mpoly_get_coeff_ui_monomial(const nmod_mpoly_t A, const nmod_mpoly_t M, const nmod_mpoly_ctx_t ctx)

Assuming that $M$ is a monomial, return the coefficient of the corresponding monomial in $A$. This function throws if $M$ is not a monomial.

void nmod_mpoly_set_coeff_ui_monomial(nmod_mpoly_t A, ulong c, const nmod_mpoly_t M, const nmod_mpoly_ctx_t ctx)

Assuming that $M$ is a monomial, set the coefficient of the corresponding monomial in $A$ to $c$. This function throws if $M$ is not a monomial.

ulong nmod_mpoly_get_coeff_ui_fmpz(const nmod_mpoly_t A, fmpz *const *exp, const nmod_mpoly_ctx_t ctx)

ulong nmod_mpoly_get_coeff_ui_ui(const nmod_mpoly_t A, const ulong *exp, const nmod_mpoly_ctx_t ctx)

Return the coefficient of the monomial with exponent exp.

void nmod_mpoly_set_coeff_ui_fmpz(nmod_mpoly_t A, ulong c, fmpz *const *exp, const nmod_mpoly_ctx_t ctx)

void nmod_mpoly_set_coeff_ui_ui(nmod_mpoly_t A, ulong c, const ulong *exp, const nmod_mpoly_ctx_t ctx)

Set the coefficient of the monomial with exponent exp to c.

void nmod_mpoly_get_coeff_vars_ui(nmod_mpoly_t C, const nmod_mpoly_t A, const slong *vars, const ulong *exps, slong length, const nmod_mpoly_ctx_t ctx)

Set $C$ to the coefficient of $A$ with respect to the variables in $vars$ with powers in the corresponding array $exps$. Both $vars$ and $exps$ point to array of length $length$. It is assumed that $0 < length \le nvars(A)$ and that the variables in $vars$ are distinct.

6.7.9 Comparison

int nmod_mpoly_cmp(const nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)

Return 1 (resp. −1, or 0) if $A$ is after (resp. before, same as) $B$ in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when $A$ and $B$ are both monomials.

6.7.10 Container operations

These functions deal with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw.

mp_limb_t *nmp_mpoly_term_coeff_ref(nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)

Return a reference to the coefficient of index $i$ of $A$.

int nmp_mpoly_is_canonical(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)

Return 1 if $A$ is in canonical form. Otherwise, return 0. To be in canonical form, all of the terms must have nonzero coefficients, and the terms must be sorted from greatest to least.

slong nmp_mpoly_length(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)

Return the number of terms in $A$. If the polynomial is in canonical form, this will be the number of nonzero coefficients.

void nmp_mpoly_resize(nmod_mpoly_t A, slong new_length, const nmod_mpoly_ctx_t ctx)

Set the length of $A$ to new_length. Terms are either deleted from the end, or new zero terms are appended.
ulong nmod_mpoly_get_term_coeff_ui(const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)

Return the coefficient of the term of index $i$.

void nmod_mpoly_set_term_coeff_ui(nmod_mpoly_t A, slong i, ulong c, const nmod_mpoly_ctx_t ctx)

Set the coefficient of the term of index $i$ to $c$.

int nmod_mpoly_term_exp_fits_si(const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
int nmod_mpoly_term_exp_fits_ui(const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)

Return 1 if all entries of the exponent vector of the term of index $i$ fit into an slong (resp. a ulong). Otherwise, return 0.

void nmod_mpoly_get_term_exp_fmpz(fmpz **exp, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
void nmod_mpoly_get_term_exp_ui(ulong *exp, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)

void nmod_mpoly_get_term_exp_si(slong *exp, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)

Set $exp$ to the exponent vector of the term of index $i$. The _ui (resp. _si) version throws if any entry does not fit into a ulong (resp. slong).

ulong nmod_mpoly_get_term_var_exp_ui(const nmod_mpoly_t A, slong i, slong var, const nmod_mpoly_ctx_t ctx)
slong nmod_mpoly_get_term_var_exp_si(const nmod_mpoly_t A, slong i, slong var, const nmod_mpoly_ctx_t ctx)

Return the exponent of the variable $var$ of the term of index $i$. This function throws if the exponent does not fit into a ulong (resp. slong).

void nmod_mpoly_set_term_exp_fmpz(nmod_mpoly_t A, slong i, fmpz *const *exp, const nmod_mpoly_ctx_t ctx)
void nmod_mpoly_set_term_exp_ui(nmod_mpoly_t A, slong i, const ulong *exp, const nmod_mpoly_ctx_t ctx)

Set the exponent of the term of index $i$ to $exp$.

void nmod_mpoly_get_term(nmod_mpoly_t M, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)

Set $M$ to the term of index $i$ in $A$.

void nmod_mpoly_get_term_monomial(nmod_mpoly_t M, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)

Set $M$ to the monomial of the term of index $i$ in $A$. The coefficient of $M$ will be one.

void nmod_mpoly_push_term_ui_fmpz(nmod_mpoly_t A, ulong c, fmpz *const *exp, const nmod_mpoly_ctx_t ctx)
void nmod_mpoly_push_term_ui_ffmpz(nmod_mpoly_t A, ulong c, const fmpz *exp, const nmod_mpoly_ctx_t ctx)
void nmod_mpoly_push_term_ui_ui(nmod_mpoly_t A, ulong c, const ulong *exp, const nmod_mpoly_ctx_t ctx)

Append a term to $A$ with coefficient $c$ and exponent vector $exp$. This function runs in constant average time.

void nmod_mpoly_sort_terms(nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)

Sort the terms of $A$ into the canonical ordering dictated by the ordering in $ctx$. This function simply reorders the terms: It does not combine like terms, nor does it delete terms with coefficient zero. This function runs in linear time in the bit size of $A$. 

392 Chapter 6. Integers mod n
void nmod_mpoly_combine_like_terms(nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Combine adjacent like terms in A and delete terms with coefficient zero. If the terms of A were
    sorted to begin with, the result will be in canonical form. This function runs in linear time in the
    bit size of A.

void nmod_mpoly_reverse(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)
    Set A to the reversal of B.

6.7.11 Random generation

void nmod_mpoly_randtest_bound(nmod_mpoly_t A, flint_rand_t state, slong length, ulong
    exp_bound, const nmod_mpoly_ctx_t ctx)
    Generate a random polynomial with length up to length and exponents in the range [0, exp_bound
    - 1]. The exponents of each variable are generated by calls to n_randint(state, exp_bound).

void nmod_mpoly_randtest_bounds(nmod_mpoly_t A, flint_rand_t state, slong length, ulong
    *exp_bounds, const nmod_mpoly_ctx_t ctx)
    Generate a random polynomial with length up to length and exponents in the range [0, 
    exp_bounds[i] - 1]. The exponents of the variable of index i are generated by calls to
    n_randint(state, exp_bounds[i]).

void nmod_mpoly_randtest_bits(nmod_mpoly_t A, flint_rand_t state, slong length, mp_limb_t
    exp_bits, const nmod_mpoly_ctx_t ctx)
    Generate a random polynomial with length up to length and exponents whose packed form does
    not exceed the given bit count.

6.7.12 Addition/Subtraction

void nmod_mpoly_add_ui(nmod_mpoly_t A, const nmod_mpoly_t B, ulong c, const
    nmod_mpoly_ctx_t ctx)
    Set A to B + c.

void nmod_mpoly_sub_ui(nmod_mpoly_t A, const nmod_mpoly_t B, ulong c, const
    nmod_mpoly_ctx_t ctx)
    Set A to B − c.

void nmod_mpoly_add(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C, const
    nmod_mpoly_ctx_t ctx)
    Set A to B + C.

void nmod_mpoly_sub(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C, const
    nmod_mpoly_ctx_t ctx)
    Set A to B − C.

6.7.13 Scalar operations

void nmod_mpoly_neg(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)
    Set A to −B.

void nmod_mpoly_scalar_mul_ui(nmod_mpoly_t A, const nmod_mpoly_t B, ulong c, const
    nmod_mpoly_ctx_t ctx)
    Set A to B × c.
void nmod_mpoly_make_monic(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)

Set A to B divided by the leading coefficient of B. This throws if B is zero or the leading coefficient is not invertible.

### 6.7.14 Differentiation

void nmod_mpoly_derivative(nmod_mpoly_t A, const nmod_mpoly_t B, slong var, const nmod_mpoly_ctx_t ctx)

Set A to the derivative of B with respect to the variable of index var.

### 6.7.15 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

ulong nmod_mpoly_evaluate_all_ui(const nmod_mpoly_t A, const ulong *vals, const nmod_mpoly_ctx_t ctx)

Return the evaluation of A where the variables are replaced by the corresponding elements of the array vals.

void nmod_mpoly_evaluate_one_ui(nmod_mpoly_t A, const nmod_mpoly_t B, slong var, ulong val, const nmod_mpoly_ctx_t ctx)

Set A to the evaluation of B where the variable of index var is replaced by val.

int nmod_mpoly_compose_nmod_poly(nmod_poly_t A, const nmod_mpoly_t B, nmod_poly_struct *const *C, const nmod_mpoly_ctx_t ctxB, const nmod_mpoly_ctx_t ctxAC)

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C. The context object of B is ctxB. Return 1 for success and 0 for failure.

int nmod_mpoly_compose_nmod_mpoly_geobucket(nmod_mpoly_t A, const nmod_mpoly_t B, nmod_mpoly_struct *const *C, const nmod_mpoly_ctx_t ctxB, const nmod_mpoly_ctx_t ctxAC)

int nmod_mpoly_compose_nmod_mpoly_horner(nmod_mpoly_t A, const nmod_mpoly_t B, nmod_mpoly_struct *const *C, const nmod_mpoly_ctx_t ctxB, const nmod_mpoly_ctx_t ctxAC)

int nmod_mpoly_compose_nmod_mpoly(nmod_mpoly_t A, const nmod_mpoly_t B, nmod_mpoly_struct *const *C, const nmod_mpoly_ctx_t ctxB, const nmod_mpoly_ctx_t ctxAC)

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C. Both A and the elements of C have context object ctxAC, while B has context object ctxB. Neither of A and B is allowed to alias any other polynomial. Return 1 for success and 0 for failure. The main method attempts to perform the calculation using matrices and chooses heuristically between the geobucket and horner methods if needed.

void nmod_mpoly_compose_nmod_mpoly_gen(nmod_mpoly_t A, const nmod_mpoly_t B, const slong *c, const nmod_mpoly_ctx_t ctxB, const nmod_mpoly_ctx_t ctxAC)

Set A to the evaluation of B where the variable of index i in ctxB is replaced by the variable of index c[i] in ctxAC. The length of the array C is the number of variables in ctxB. If any c[i] is negative, the corresponding variable of B is replaced by zero. Otherwise, it is expected that c[i] is less than the number of variables in ctxAC.
6.7.16 Multiplication

void nmod_mpoly_mul(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C, const nmod_mpoly_ctx_t ctx)

Set A to $B \times C$.

void nmod_mpoly_mul_johnson(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C, const nmod_mpoly_ctx_t ctx)

Set A to $B \times C$ using Johnson’s heap-based method. The first version always uses one thread.

int nmod_mpoly_mul_array(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C, const nmod_mpoly_ctx_t ctx)

Try to set A to $B \times C$ using arrays. If the return is 0, the operation was unsuccessful. Otherwise, it was successful, and the return is 1. The first version always uses one thread.

int nmod_mpoly_mul_dense(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C, const nmod_mpoly_ctx_t ctx)

Try to set A to $B \times C$ using univariate arithmetic. If the return is 0, the operation was unsuccessful. Otherwise, it was successful and the return is 1.

6.7.17 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

int nmod_mpoly_pow_fmpz(nmod_mpoly_t A, const nmod_mpoly_t B, const fmpz_t k, const nmod_mpoly_ctx_t ctx)

Set A to $B^k$. Return 1 for success and 0 for failure.

int nmod_mpoly_pow_ui(nmod_mpoly_t A, const nmod_mpoly_t B, ulong k, const nmod_mpoly_ctx_t ctx)

Set A to $B^k$. Return 1 for success and 0 for failure.

6.7.18 Division

The division functions assume that the modulus is prime.

int nmod_mpoly_divides(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)

If A is divisible by B, set Q to the exact quotient and return 1. Otherwise, set Q to zero and return 0. Note that the function nmod_mpoly_div below may be faster if the quotient is known to be exact.

void nmod_mpoly_div(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)

Set Q to the quotient of A by B, discarding the remainder.

void nmod_mpoly_divrem(nmod_mpoly_t Q, nmod_mpoly_t R, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)

Set Q and R to the quotient and remainder of A divided by B.
void nmod_mpoly_divrem_ideal(nmod_mpoly_struct **Q, nmod_mpoly_t R, const nmod_mpoly_t A, nmod_mpoly_struct *const *B, slong len, const
nmod_mpoly_ctx_t ctx)

This function is as per nmod_mpoly_divrem() except that it takes an array of divisor polynomials $B$ and it returns an array of quotient polynomials $Q$. The number of divisor (and hence quotient) polynomials, is given by $len$.

int nmod_mpoly_divides_dense(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)

Try to do the operation of nmod_mpoly_divides using univariate arithmetic. If the return is $-1$, the operation was unsuccessful. Otherwise, it was successful and the return is 0 or 1.

int nmod_mpoly_divides_monagan_pearce(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)

Do the operation of nmod_mpoly_divides using the algorithm of Michael Monagan and Roman Pearce.

int nmod_mpoly_divides_heap_threaded(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)

Do the operation of nmod_mpoly_divides using the heap and multiple threads. This function should only be called once global_thread_pool has been initialized.

Note: This function is only defined if the machine is known to be strongly ordered during the configuration. To check whether this function is defined during compilation-time, use the C preprocessor macro #ifndef nmod_mpoly_divides_heap_threaded.

Note that, if the system is known to be strongly ordered, the underlying algorithm for this function is utilized in nmod_mpoly_divides(). Hence, you may find it easier to use this function instead if the C preprocessor is not available.

### 6.7.19 Greatest Common Divisor

The greatest common divisor functions assume that the modulus is prime.

void nmod_mpoly_term_content(nmod_mpoly_t M, const nmod_mpoly_t A, const
nmod_mpoly_ctx_t ctx)

Set $M$ to the GCD of the terms of $A$. If $A$ is zero, $M$ will be zero. Otherwise, $M$ will be a monomial with coefficient one.

int nmod_mpoly_content_vars(nmod_mpoly_t g, const nmod_mpoly_t A, slong *vars, slong
vars_length, const nmod_mpoly_ctx_t ctx)

Set $g$ to the GCD of the coefficients of $A$ when viewed as a polynomial in the variables $vars$. Return 1 for success and 0 for failure. Upon success, $g$ will be independent of the variables $vars$.

int nmod_mpoly_gcd(nmod_mpoly_t G, const nmod_mpoly_t A, const nmod_mpoly_t B, const
nmod_mpoly_ctx_t ctx)

Try to set $G$ to the monic GCD of $A$ and $B$. The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and $G$ is left untouched.

int nmod_mpoly_gcd_cofactors(nmod_mpoly_t G, nmod_mpoly_t Abar, nmod_mpoly_t Bbar, const
nmod_mpoly_t A, const nmod_mpoly_t B, const
nmod_mpoly_ctx_t ctx)

Do the operation of nmod_mpoly_gcd() and also compute $Abar = A/G$ and $Bbar = B/G$ if successful.

int nmod_mpoly_gcd_brown(nmod_mpoly_t G, const nmod_mpoly_t A, const nmod_mpoly_t B,
const nmod_mpoly_ctx_t ctx)
int nmod_mpoly_gcd_hensel(nmod_mpoly_t G, const nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)
int nmod_mpoly_gcd_zippel(nmod_mpoly_t G, const nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)

Try to set $G$ to the GCD of $A$ and $B$ using various algorithms.

int nmod_mpoly_resultant(nmod_mpoly_t R, const nmod_mpoly_t A, const nmod_mpoly_t B, slong var, const nmod_mpoly_ctx_t ctx)

Try to set $R$ to the resultant of $A$ and $B$ with respect to the variable of index $\text{var}$.

int nmod_mpoly_discriminant(nmod_mpoly_t D, const nmod_mpoly_t A, slong var, const nmod_mpoly_ctx_t ctx)

Try to set $D$ to the discriminant of $A$ with respect to the variable of index $\text{var}$.

6.7.20 Square Root

The square root functions assume that the modulus is prime for correct operation.

int nmod_mpoly_sqrt(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)

If $Q^2 = A$ has a solution, set $Q$ to a solution and return 1, otherwise return 0 and set $Q$ to zero.

int nmod_mpoly_is_square(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)

Return 1 if $A$ is a perfect square, otherwise return 0.

int nmod_mpoly_quadratic_root(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)

If $Q^2 + AQ = B$ has a solution, set $Q$ to a solution and return 1, otherwise return 0.

6.7.21 Univariate Functions

An nmod_mpoly_univar_t holds a univariate polynomial in some main variable with nmod_mpoly_t coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of $\mathbb{Z}/n\mathbb{Z}[x_1,\ldots,x_m]$ as an element of $(\mathbb{Z}/n\mathbb{Z}[x_1,\ldots,x_{\text{var}-1},x_{\text{var}+1},\ldots,x_m])[x_{\text{var}}]$ and vice versa.

void nmod_mpoly_univar_init(nmod_mpoly_univar_t A, const nmod_mpoly_ctx_t ctx)

Initialize $A$.

void nmod_mpoly_univar_clear(nmod_mpoly_univar_t A, const nmod_mpoly_ctx_t ctx)

Clear $A$.

void nmod_mpoly_univar_swap(nmod_mpoly_univar_t A, nmod_mpoly_univar_t B, const nmod_mpoly_ctx_t ctx)

Swap $A$ and $B$.

void nmod_mpoly_to_univar(nmod_mpoly_univar_t A, const nmod_mpoly_t B, slong var, const nmod_mpoly_ctx_t ctx)

Set $A$ to a univariate form of $B$ by pulling out the variable of index $\text{var}$. The coefficients of $A$ will still belong to the content $\text{ctx}$ but will not depend on the variable of index $\text{var}$.

void nmod_mpoly_from_univar(nmod_mpoly_t A, const nmod_mpoly_univar_t B, slong var, const nmod_mpoly_ctx_t ctx)

Set $A$ to the normal form of $B$ by putting in the variable of index $\text{var}$. This function is undefined if the coefficients of $B$ depend on the variable of index $\text{var}$.

int nmod_mpoly_univar_degree_fits_si(const nmod_mpoly_univar_t A, const nmod_mpoly_ctx_t ctx)

Return 1 if the degree of $A$ with respect to the main variable fits an slong. Otherwise, return 0.
### Internal Functions

- **nmod_mpoly_pow_rmul**
  
  ```c
  void nmod_mpoly_pow_rmul(nmod_mpoly_t A, const nmod_mpoly_t B, ulong k, const nmod_mpoly_ctx_t ctx)
  ```

  Set $A$ to $B$ raised to the $k$-th power using repeated multiplications.

- **nmod_mpoly_div_monagan_pearce**
  
  ```c
  void nmod_mpoly_div_monagan_pearce(nmod_mpoly_t polyq, const nmod_mpoly_t poly2, const nmod_mpoly_t poly3, const nmod_mpoly_ctx_t ctx)
  ```

  Set $polyq$ to the quotient of $poly2$ by $poly3$, discarding the remainder (with notional remainder coefficients reduced modulo the leading coefficient of $poly3$). Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce. This function is exceptionally efficient if the division is known to be exact.

- **nmod_mpoly_divrem_monagan_pearce**
  
  ```c
  void nmod_mpoly_divrem_monagan_pearce(nmod_mpoly_t q, nmod_mpoly_t r, const nmod_mpoly_t poly2, const nmod_mpoly_t poly3, const nmod_mpoly_ctx_t ctx)
  ```

  Set $polyq$ and $polyr$ to the quotient and remainder of $poly2$ divided by $poly3$, (with remainder coefficients reduced modulo the leading coefficient of $poly3$). Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce.

- **nmod_mpoly_divrem_ideal_monagan_pearce**
  
  ```c
  void nmod_mpoly_divrem_ideal_monagan_pearce(nmod_mpoly_struct **q, nmod_mpoly_t r, const nmod_mpoly_t poly2, nmod_mpoly_struct *const *poly3, slong len, const nmod_mpoly_ctx_t ctx)
  ```

  This function is as per **nmod_mpoly_divrem_monagan_pearce** except that it takes an array of divisor polynomials $poly3$, and it returns an array of quotient polynomials $q$. The number of divisor (and hence quotient) polynomials, is given by $len$. The function computes polynomials $q_i = q[i]$ such that $poly2 = r + \sum_{i=0}^{len} q_i b_i$, where $b_i = poly3[i]$.

### nmod_mpoly_factor.h – factorisation of multivariate polynomials over integers mod $n$ (word-size $n$)

#### 6.8.1 Types, macros and constants

- **nmod_mpoly_factor_struct**
  
  A struct for holding a factored polynomial. There is a single constant and a product of bases to corresponding exponents.

- **nmod_mpoly_factor_t**
  
  An array of length 1 of **nmod_mpoly_factor_struct**.
6.8.2 Memory management

void nmod_mpoly_factor_init(nmod_mpoly_factor_t f, const nmod_mpoly_ctx_t ctx)
    Initialise f.

void nmod_mpoly_factor_clear(nmod_mpoly_factor_t f, const nmod_mpoly_ctx_t ctx)
    Clear f.

6.8.3 Basic manipulation

void nmod_mpoly_factor_swap(nmod_mpoly_factor_t f, nmod_mpoly_factor_t g, const nmod_mpoly_ctx_t ctx)
    Efficiently swap f and g.

slong nmod_mpoly_factor_length(const nmod_mpoly_factor_t f, const nmod_mpoly_ctx_t ctx)
    Return the length of the product in f.

ulong nmod_mpoly_factor_get_constant_ui(const nmod_mpoly_factor_t f, const nmod_mpoly_ctx_t ctx)
    Return the constant of f.

void nmod_mpoly_factor_get_base(nmod_mpoly_t p, const nmod_mpoly_factor_t f, slong i, const nmod_mpoly_ctx_t ctx)
    Set (resp. swap) B to (resp. with) the base of the term of index i in A.

void nmod_mpoly_factor_swap_base(nmod_mpoly_t p, nmod_mpoly_factor_t f, slong i, const nmod_mpoly_ctx_t ctx)
    Set (resp. swap) B to (resp. with) the base of the term of index i in A.

slong nmod_mpoly_factor_get_exp_si(nmod_mpoly_factor_t f, slong i, const nmod_mpoly_ctx_t ctx)
    Return the exponent of the term of index i in A. It is assumed to fit an slong.

void nmod_mpoly_factor_sort(nmod_mpoly_factor_t f, const nmod_mpoly_ctx_t ctx)
    Sort the product of f first by exponent and then by base.

6.8.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and f is undefined. None of these functions multiply f by A: f is simply set to a factorisation of A, and thus these functions should not depend on the initial value of the output f.

int nmod_mpoly_factor_squarefree(nmod_mpoly_factor_t f, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Set f to a factorization of A where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call nmod_mpoly_factor_sort() and then multiply the bases with the desired exponent.

int nmod_mpoly_factor(nmod_mpoly_factor_t f, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
    Set f to a factorization of A where the bases are irreducible.
6.9 fmpz_mod.h – arithmetic modulo integers

6.9.1 Types, macros and constants

type fmpz_mod_ctx_struct

type fmpz_mod_ctx_t

The context object for arithmetic modulo integers.

6.9.2 Context object

void fmpz_mod_ctx_init(fmpz_mod_ctx_t ctx, const fmpz_t n)

Initialise ctx for arithmetic modulo n, which is expected to be positive.

void fmpz_mod_ctx_clear(fmpz_mod_ctx_t ctx)

Free any memory used by ctx.

void fmpz_mod_ctx_set_modulus(fmpz_mod_ctx_t ctx, const fmpz_t n)

Reconfigure ctx for arithmetic modulo n.

6.9.3 Conversions

void fmpz_mod_set_fmpz(fmpz_t a, const fmpz_t b, const fmpz_mod_ctx_t ctx)

Set a to b after reduction modulo the modulus.

6.9.4 Arithmetic

Unless specified otherwise all functions here expect their relevant arguments to be in the canonical range \([0, n)\). Comparison of elements against each other or against zero can be accomplished with func::fmpz_equal or func::fmpz_is_zero without a context.

int fmpz_mod_is_canonical(const fmpz_t a, const fmpz_mod_ctx_t ctx)

Return 1 if a is in the canonical range \([0, n)\) and 0 otherwise.

int fmpz_mod_is_one(const fmpz_t a, const fmpz_mod_ctx_t ctx)

Return 1 if a is 1 modulo n and return 0 otherwise.

void fmpz_mod_add(fmpz_t a, const fmpz_t b, const fmpz_t c, const fmpz_mod_ctx_t ctx)

Set a to b + c modulo n.

void fmpz_mod_add_fmpz(fmpz_t a, const fmpz_t b, const fmpz_t c, const fmpz_mod_ctx_t ctx)

void fmpz_mod_add_ui(fmpz_t a, const fmpz_t b, ulong c, const fmpz_mod_ctx_t ctx)

void fmpz_mod_add_si(fmpz_t a, const fmpz_t b, slong c, const fmpz_mod_ctx_t ctx)

Set a to b + c modulo n where only b is assumed to be canonical.

void fmpz_mod_sub(fmpz_t a, const fmpz_t b, const fmpz_t c, const fmpz_mod_ctx_t ctx)

Set a to b − c modulo n.

void fmpz_mod_sub_fmpz(fmpz_t a, const fmpz_t b, const fmpz_t c, const fmpz_mod_ctx_t ctx)

void fmpz_mod_sub_ui(fmpz_t a, const fmpz_t b, ulong c, const fmpz_mod_ctx_t ctx)

void fmpz_mod_sub_si(fmpz_t a, const fmpz_t b, slong c, const fmpz_mod_ctx_t ctx)

Set a to b − c modulo n where only b is assumed to be canonical.

void fmpz_mod_fmpz_sub(fmpz_t a, const fmpz_t b, const fmpz_t c, const fmpz_mod_ctx_t ctx)

void fmpz_mod_ui_sub(fmpz_t a, ulong b, const fmpz_t c, const fmpz_mod_ctx_t ctx)
void fmpz_mod_si_sub(fmpz_t a, slong b, const fmpz_t c, const fmpz_mod_ctx_t ctx)
    Set a to \( b - c \) modulo \( n \) where only \( c \) is assumed to be canonical.

void fmpz_mod_neg(fmpz_t a, const fmpz_t b, const fmpz_mod_ctx_t ctx)
    Set a to \(-b\) modulo \( n\).

void fmpz_mod_mul(fmpz_t a, const fmpz_t b, const fmpz_t c, const fmpz_mod_ctx_t ctx)
    Set a to \( b \cdot c \) modulo \( n \).

void fmpz_mod_inv(fmpz_t a, const fmpz_t b, const fmpz_mod_ctx_t ctx)
    Set a to \( b^{-1} \) modulo \( n \). This function expects that \( b \) is invertible modulo \( n \) and throws if this not the case. Invertibility may be tested with \( \text{fmpz_mod_pow_fmpz()} \) or \( \text{fmpz_mod_divides()} \).

int fmpz_mod_divides(fmpz_t a, const fmpz_t b, const fmpz_t c, const fmpz_mod_ctx_t ctx)
    If \( a \cdot c = b \mod n \) has a solution for \( a \) return 1 and set \( a \) to such a solution. Otherwise return 0 and leave \( a \) undefined.

void fmpz_mod_pow_ui(fmpz_t a, const fmpz_t b, ulong e, const fmpz_mod_ctx_t ctx)
    Set a to \( b^e \) modulo \( n \).

int fmpz_mod_pow_fmpz(fmpz_t a, const fmpz_t b, const fmpz_t e, const fmpz_mod_ctx_t ctx)
    Try to set \( a \) to \( b^e \) modulo \( n \). If \( e < 0 \) and \( b \) is not invertible modulo \( n \), the return is 0. Otherwise, the return is 1.

6.9.5 Discrete Logarithms via Pohlig-Hellman

void fmpz_mod_discrete_log_pohlig_hellman_init(fmpz_mod_discrete_log_pohlig_hellman_t L)
    Initialize \( L \). Upon initialization \( L \) is not ready for computation.

void fmpz_mod_discrete_log_pohlig_hellman_clear(fmpz_mod_discrete_log_pohlig_hellman_t L)
    Free any space used by \( L \).

double fmpz_mod_discrete_log_pohlig_hellman_precompute_prime(fmpz_mod_discrete_log_pohlig_hellman_t L, const fmpz_t p)
    Configure \( L \) for discrete logarithms modulo \( p \) to an internally chosen base. It is assumed that \( p \) is prime. The return is an estimate on the number of multiplications needed for one run.

const fmpz *fmpz_mod_discrete_log_pohlig_hellman_primitive_root(fmpz_mod_discrete_log_pohlig_hellman_t L)
    Return the internally stored base.

void fmpz_mod_discrete_log_pohlig_hellman_run(fmpz_t x, const fmpz_mod_discrete_log_pohlig_hellman_t L, const fmpz_t y)
    Set \( x \) to the logarithm of \( y \) with respect to the internally stored base. \( y \) is expected to be reduced modulo \( p \). The function is undefined if the logarithm does not exist.

int fmpz_next_smooth_prime(fmpz_t a, const fmpz_t b)
    Either return 1 and set \( a \) to a smooth prime strictly greater than \( b \), or return 0 and set \( a \) to 0. The smooth primes returned by this function currently have no prime factor of \( a - 1 \) greater than 23, but this should not be relied upon.

6.9. fmpz_mod.h – arithmetic modulo integers
6.10 fmpz_mod_vec.h – vectors over integers mod n

6.10.1 Conversions

void _fmpz_mod_vec_set_fmpz_vec (fmpz *A, const fmpz *B, slong len, const fmpz_mod_ctx_t ctx)
Set the fmpz vector (A, len) to the fmpz vector (B, len) after reduction of each entry modulo the modulus.

6.10.2 Arithmetic

void _fmpz_mod_vec_neg (fmpz *A, const fmpz *B, slong len, const fmpz_mod_ctx_t ctx)
Set (A, len) to −(B, len).

void _fmpz_mod_vec_add (fmpz *a, const fmpz *b, const fmpz *c, slong n, const fmpz_mod_ctx_t ctx)
Set (A, len) to (B, len) + (C, len).

void _fmpz_mod_vec_sub (fmpz *a, const fmpz *b, const fmpz *c, slong n, const fmpz_mod_ctx_t ctx)
Set (A, len) to (B, len) − (C, len).

6.10.3 Scalar Multiplication

void _fmpz_mod_vec_scalar_mul_fmpz_mod (fmpz *A, const fmpz *B, slong len, const fmpz_t c, const fmpz_mod_ctx_t ctx)
Set (A, len) to (B, len) * c.

void _fmpz_mod_vec_scalar_addmul_fmpz_mod (fmpz *A, const fmpz *B, slong len, const fmpz_t c, const fmpz_mod_ctx_t ctx)
Set (A, len) to (A, len) + (B, len) * c.

void _fmpz_mod_vec_scalar_div_fmpz_mod (fmpz *A, const fmpz *B, slong len, const fmpz_t c, const fmpz_mod_ctx_t ctx)
Set (A, len) to (B, len)/c assuming c is nonzero.

6.10.4 Dot Product

void _fmpz_mod_vec_dot (fmpz_t d, const fmpz *A, const fmpz *B, slong len, const fmpz_mod_ctx_t ctx)
Set d to the dot product of (A, len) with (B, len).

void _fmpz_mod_vec_dot_rev (fmpz_t d, const fmpz *A, const fmpz *B, slong len, const fmpz_mod_ctx_t ctx)
Set d to the dot product of (A, len) with the reverse of the vector (B, len).

6.10.5 Multiplication

void _fmpz_mod_vec_mul (fmpz *A, const fmpz *B, const fmpz *C, slong len, const fmpz_mod_ctx_t ctx)
Set (A, len) the pointwise multiplication of (B, len) and (C, len).
6.11 fmpz_mod_mat.h – matrices over integers mod n

6.11.1 Types, macros and constants

**type** `fmpz_mod_mat_struct`

**type** `fmpz_mod_mat_t`

6.11.2 Element access

```c
fmpz *fmpz_mod_mat_entry(const fmpz_mod_mat_t mat, slong i, slong j)
```

Return a reference to the element at row `i` and column `j` of `mat`.

```c
void fmpz_mod_mat_set_entry(fmpz_mod_mat_t mat, slong i, slong j, const fmpz_t val)
```

Set the entry at row `i` and column `j` of `mat` to `val`.

6.11.3 Memory management

```c
void fmpz_mod_mat_init(fmpz_mod_mat_t mat, slong rows, slong cols, const fmpz_t n)
```

Initialise `mat` as a matrix with the given number of `rows` and `cols` and modulus `n`.

```c
void fmpz_mod_mat_init_set(fmpz_mod_mat_t mat, const fmpz_mod_mat_t src)
```

Initialise `mat` and set it equal to the matrix `src`, including the number of rows and columns and the modulus.

```c
void fmpz_mod_mat_clear(fmpz_mod_mat_t mat)
```

Clear `mat` and release any memory it used.

Basic manipulation

```c
slong fmpz_mod_mat_nrows(const fmpz_mod_mat_t mat)
```

Return the number of rows of `mat`.

```c
slong fmpz_mod_mat_ncols(const fmpz_mod_mat_t mat)
```

Return the number of columns of `mat`.

```c
void _fmpz_mod_mat_set_mod(fmpz_mod_mat_t mat, const fmpz_t n)
```

Set the modulus of the matrix `mat` to `n`.

```c
void fmpz_mod_mat_one(fmpz_mod_mat_t mat)
```

Set `mat` to the identity matrix (ones down the diagonal).

```c
void fmpz_mod_mat_zero(fmpz_mod_mat_t mat)
```

Set `mat` to the zero matrix.

```c
void fmpz_mod_mat_swap(fmpz_mod_mat_t mat1, fmpz_mod_mat_t mat2)
```

Efficiently swap the matrices `mat1` and `mat2`.

```c
void fmpz_mod_mat_swap_entrywise(fmpz_mod_mat_t mat1, fmpz_mod_mat_t mat2)
```

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

```c
int fmpz_mod_mat_is_empty(const fmpz_mod_mat_t mat)
```

Return 1 if `mat` has either zero rows or columns.

```c
int fmpz_mod_mat_is_square(const fmpz_mod_mat_t mat)
```

Return 1 if `mat` has the same number of rows and columns.

```c
void _fmpz_mod_mat_reduce(fmpz_mod_mat_t mat)
```

Reduce all the entries of `mat` by the modulus `n`. This function is only needed internally.
6.11.4 Random generation

```c
void fmpz_mod_mat_randtest(fmpz_mod_mat_t mat, flint_rand_t state)
```
Generate a random matrix with the existing dimensions and entries in $[0, n)$ where $n$ is the modulus.

6.11.5 Windows and concatenation

```c
void fmpz_mod_mat_window_init(fmpz_mod_mat_t window, const fmpz_mod_mat_t mat, slong r1, slong c1, slong r2, slong c2)
```
Initializes the matrix `window` to be an $r_2 - r_1$ by $c_2 - c_1$ submatrix of `mat` whose $(0, 0)$ entry is the $(r_1, c_1)$ entry of `mat`. The memory for the elements of `window` is shared with `mat`.

```c
void fmpz_mod_mat_window_clear(fmpz_mod_mat_t window)
```
Clears the matrix `window` and releases any memory that it uses. Note that the memory to the underlying matrix that `window` points to is not freed.

```c
void fmpz_mod_mat_concat_horizontal(fmpz_mod_mat_t res, const fmpz_mod_mat_t mat1, const fmpz_mod_mat_t mat2)
```
Sets `res` to vertical concatenation of `(mat1, mat2)` in that order. Matrix dimensions: `mat1`: $m \times n$, `mat2`: $k \times n$, `res`: $(m + k) \times n$.

```c
void fmpz_mod_mat_concat_vertical(fmpz_mod_mat_t res, const fmpz_mod_mat_t mat1, const fmpz_mod_mat_t mat2)
```
Sets `res` to horizontal concatenation of `(mat1, mat2)` in that order. Matrix dimensions: `mat1`: $m \times n$, `mat2`: $m \times k$, `res`: $m \times (n + k)$.

6.11.6 Input and output

```c
void fmpz_mod_mat_print_pretty(const fmpz_mod_mat_t mat)
```
Prints the given matrix to `stdout`. The format is an opening square bracket then on each line a row of the matrix, followed by a closing square bracket. Each row is written as an opening square bracket followed by a space separated list of coefficients followed by a closing square bracket.

6.11.7 Comparison

```c
int fmpz_mod_mat_is_zero(const fmpz_mod_mat_t mat)
```
Return 1 if `mat` is the zero matrix.

6.11.8 Set and transpose

```c
void fmpz_mod_mat_set(fmpz_mod_mat_t B, const fmpz_mod_mat_t A)
```
Set `B` to equal `A`.

```c
void fmpz_mod_mat_transpose(fmpz_mod_mat_t B, const fmpz_mod_mat_t A)
```
Set `B` to the transpose of `A`.
6.11.9 Conversions

void fmpz_mod_mat_set_fmpz_mat(fmpz_mod_mat_t A, const fmpz_mat_t B)
    Set A to the matrix B reducing modulo the modulus of A.

void fmpz_mod_mat_get_fmpz_mat(fmpz_mat_t A, const fmpz_mod_mat_t B)
    Set A to a lift of B.

6.11.10 Addition and subtraction

void fmpz_mod_mat_add(fmpz_mod_mat_t C, const fmpz_mod_mat_t A, const fmpz_mod_mat_t B)
    Set C to A + B.

void fmpz_mod_mat_sub(fmpz_mod_mat_t C, const fmpz_mod_mat_t A, const fmpz_mod_mat_t B)
    Set C to A – B.

void fmpz_mod_mat_neg(fmpz_mod_mat_t B, const fmpz_mod_mat_t A)
    Set B to –A.

6.11.11 Scalar arithmetic

void fmpz_mod_mat_scalar_mul_si(fmpz_mod_mat_t B, const fmpz_mod_mat_t A, slong c)
    Set B to cA where c is a constant.

void fmpz_mod_mat_scalar_mul_ui(fmpz_mod_mat_t B, const fmpz_mod_mat_t A, ulong c)
    Set B to cA where c is a constant.

void fmpz_mod_mat_scalar_mul_fmpz(fmpz_mod_mat_t B, const fmpz_mod_mat_t A, fmpz_t c)
    Set B to cA where c is a constant.

6.11.12 Matrix multiplication

void fmpz_mod_mat_mul(fmpz_mod_mat_t C, const fmpz_mod_mat_t A, const fmpz_mod_mat_t B)
    Set C to A \times B. The number of rows of B must match the number of columns of A.

void _fmpz_mod_mat_mul_classical_threaded_pool_op(fmpz_mod_mat_t D, const fmpz_mod_mat_t C, const fmpz_mod_mat_t A, const fmpz_mod_mat_t B, int op, thread_pool_handle *threads, slong num_threads)
    Set D to A \times B + op*C where op is +1, -1 or 0.

void _fmpz_mod_mat_mul_classical_threaded_op(fmpz_mod_mat_t D, const fmpz_mod_mat_t C, const fmpz_mod_mat_t A, const fmpz_mod_mat_t B, int op)
    Set D to A \times B + op*C where op is +1, -1 or 0.

void fmpz_mod_mat_mul_classical_threaded(fmpz_mod_mat_t C, const fmpz_mod_mat_t A, const fmpz_mod_mat_t B)
    Set C to A \times B. The number of rows of B must match the number of columns of A.
void fmpz_mod_mat_sqr(fmpz_mod_mat_t B, const fmpz_mod_mat_t A)

Set $B$ to $A^2$. The matrix $A$ must be square.

void fmpz_mod_mat_mul_fmpz_vec(fmpz *c, const fmpz_mod_mat_t A, const fmpz *b, slong blen)
void fmpz_mod_mat_mul_fmpz_vec_ptr(fmpz *const *c, const fmpz_mod_mat_t A, const fmpz *const *b, slong blen)

Compute a matrix-vector product of $A$ and $(b, blen)$ and store the result in $c$. The vector $(b, blen)$ is either truncated or zero-extended to the number of columns of $A$. The number entries written to $c$ is always equal to the number of rows of $A$.

void fmpz_mod_mat_fmpz_vec_mul(fmpz *c, const fmpz *a, slong alen, const fmpz_mod_mat_t B)
void fmpz_mod_mat_fmpz_vec_mul_ptr(fmpz *const *c, const fmpz *const *a, slong alen, const fmpz_mod_mat_t B)

Compute a vector-matrix product of $(a, alen)$ and $B$ and store the result in $c$. The vector $(a, alen)$ is either truncated or zero-extended to the number of rows of $B$. The number entries written to $c$ is always equal to the number of columns of $B$.

6.11.13 Trace

void fmpz_mod_mat_trace(fmpz_t trace, const fmpz_mod_mat_t mat)

Set $trace$ to the trace of the matrix $mat$.

6.11.14 Gaussian elimination

slong fmpz_mod_mat_rref(slong *perm, fmpz_mod_mat_t mat)

Uses Gauss-Jordan elimination to set $mat$ to its reduced row echelon form and returns the rank of $mat$.

If $perm$ is non-NULL, the permutation of rows in the matrix will also be applied to $perm$.

The modulus is assumed to be prime.

6.11.15 Strong echelon form and Howell form

void fmpz_mod_mat_strong_echelon_form(fmpz_mod_mat_t mat)

Transforms $mat$ into the strong echelon form of $mat$. The Howell form and the strong echelon form are equal up to permutation of the rows, see [FieHof2014] for a definition of the strong echelon form and the algorithm used here.

$mat$ must have at least as many rows as columns.

slong fmpz_mod_mat_howell_form(fmpz_mod_mat_t mat)

Transforms $mat$ into the Howell form of $mat$. For a definition of the Howell form see [StoMul1998]. The Howell form is computed by first putting $mat$ into strong echelon form and then ordering the rows.

$mat$ must have at least as many rows as columns.
6.11.16 Inverse

```c
int fmpz_mod_mat_inv(fmpz_mod_mat_t B, fmpz_mod_mat_t A)
```
Sets $B = A^{-1}$ and returns 1 if $A$ is invertible. If $A$ is singular, returns 0 and sets the elements of $B$ to undefined values.

$A$ and $B$ must be square matrices with the same dimensions.

The modulus is assumed to be prime.

6.11.17 LU decomposition

```c
slong fmpz_mod_mat_lu(slong *P, fmpz_mod_mat_t A, int rank_check)
```
Computes a generalised LU decomposition $LU = PA$ of a given matrix $A$, returning the rank of $A$.

If $A$ is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix $L$ and an upper triangular matrix $U$ (the diagonal of $L$ will not be stored explicitly).

If $A$ is an arbitrary matrix of rank $r$, $U$ will be in row echelon form having $r$ nonzero rows, and $L$ will be lower triangular but truncated to $r$ columns, having implicit ones on the $r$ first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if $A$ is detected to be rank-deficient.

The modulus is assumed to be prime.

6.11.18 Triangular solving

```c
void fmpz_mod_mat_solve_tril(fmpz_mod_mat_t X, const fmpz_mod_mat_t L, const fmpz_mod_mat_t B, int unit)
```
Sets $X = L^{-1}B$ where $L$ is a full rank lower triangular square matrix. If `unit = 1`, $L$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

The modulus is assumed to be prime.

```c
void fmpz_mod_mat_solve_triu(fmpz_mod_mat_t X, const fmpz_mod_mat_t U, const fmpz_mod_mat_t B, int unit)
```
Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If `unit = 1`, $U$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

The modulus is assumed to be prime.

6.11.19 Solving

```c
int fmpz_mod_mat_solve(fmpz_mod_mat_t X, const fmpz_mod_mat_t A, const fmpz_mod_mat_t B)
```
Solves the matrix-matrix equation $AX = B$.

Returns 1 if $A$ has full rank; otherwise returns 0 and sets the elements of $X$ to undefined values.

The matrix $A$ must be square.

The modulus is assumed to be prime.
int fmpz_mod_mat_can_solve(fmpz_mod_mat_t X, const fmpz_mod_mat_t A, const fmpz_mod_mat_t B)

Solves the matrix-matrix equation $AX = B$ over $F_p$.

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of $X$ to zero. If more than one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of $A$ and it may be singular.

The modulus is assumed to be prime.

### 6.11.20 Transforms

void fmpz_mod_mat_similarity(fmpz_mod_mat_t M, slong r, fmpz_t d)

Applies a similarity transform to the $n \times n$ matrix $M$ in-place.

If $P$ is the $n \times n$ identity matrix the zero entries of whose row $r$ (0-indexed) have been replaced by $d$, this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

The value $d$ is required to be reduced modulo the modulus of the entries in the matrix.

The modulus is assumed to be prime.

### 6.11.21 Characteristic polynomial

void fmpz_mod_mat_charpoly(fmpz_mod_poly_t p, const fmpz_mod_mat_t M, const fmpz_mod_ctx_t ctx)

Compute the characteristic polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised.

### 6.11.22 Minimal polynomial

void fmpz_mod_mat_minpoly(fmpz_mod_poly_t p, const fmpz_mod_mat_t M, const fmpz_mod_ctx_t ctx)

Compute the minimal polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised.

The modulus is assumed to be prime.

### 6.12 fmpz_mod_poly.h – polynomials over integers mod n

The `fmpz_mod_poly_t` data type represents elements of $\mathbb{Z}/n\mathbb{Z}[x]$ for a fixed modulus $n$. The `fmpz_mod_poly` module provides routines for memory management, basic arithmetic and some higher level functions such as GCD, etc.

Each coefficient of an `fmpz_mod_poly_t` is of type `fmpz` and represents an integer reduced modulo the fixed modulus $n$ in the range $[0, n)$.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

The `fmpz_mod_poly_t` type is a typedef for an array of length 1 of `fmpz_mod_poly_struct`'s. This permits passing parameters of type `fmpz_mod_poly_t` by reference.
In reality one never deals directly with the struct and simply deals with objects of type fmpz_mod_poly_t. For simplicity we will think of an fmpz_mod_poly_t as a struct, though in practice to access fields of this struct, one needs to dereference first, e.g. to access the length field of an fmpz_mod_poly_t called poly1 one writes poly1->length.

An fmpz_mod_poly_t is said to be normalised if either length is zero, or if the leading coefficient of the polynomial is non-zero. All fmpz_mod_poly functions expect their inputs to be normalised and all coefficients to be reduced modulo n, and unless otherwise specified they produce output that is normalised with coefficients reduced modulo n.

### 6.12.1 Simple example

The following example computes the square of the polynomial $5x^3 + 6$ in $\mathbb{Z}/7\mathbb{Z}[x]$.

```c
#include "fmpz_mod_poly.h"
int main()
{
    fmpz_t n;
    fmpz_mod_poly_t x, y;

    fmpz_init_set_ui(n, 7);
    fmpz_mod_poly_init(x, n);
    fmpz_mod_poly_init(y, n);
    fmpz_mod_poly_set_coeff_ui(x, 3, 5);
    fmpz_mod_poly_set_coeff_ui(x, 0, 6);
    fmpz_mod_poly_sqr(y, x);
    fmpz_mod_poly_print(x); flint_printf("\n");
    fmpz_mod_poly_print(y); flint_printf("\n");
    fmpz_mod_poly_clear(x);
    fmpz_mod_poly_clear(y);
    fmpz_clear(n);
}
```

The output is:

```
4 7 6 0 0 5
7 7 1 0 0 4 0 0 4
```

### 6.12.2 Types, macros and constants

**type** fmpz_mod_poly_struct

A structure holding a polynomial over the integers modulo n.

**type** fmpz_mod_poly_t

An array of length 1 of fmpz_mod_poly_struct.
6.12.3 Memory management

```c
void fmpz_mod_poly_init(fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```
Initialises `poly` for use with context `ctx` and set it to zero. A corresponding call to `fmpz_mod_poly_clear()` must be made to free the memory used by the polynomial.

```c
void fmpz_mod_poly_init2(fmpz_mod_poly_t poly, slong alloc, const fmpz_mod_ctx_t ctx)
```
Initialises `poly` with space for at least `alloc` coefficients and sets the length to zero. The allocated coefficients are all set to zero.

```c
void fmpz_mod_poly_clear(fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```
Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

```c
void fmpz_mod_poly_realloc(fmpz_mod_poly_t poly, slong alloc, const fmpz_mod_ctx_t ctx)
```
Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` the polynomial is first truncated to length `alloc`.

```c
void fmpz_mod_poly_fit_length(fmpz_mod_poly_t poly, slong len, const fmpz_mod_ctx_t ctx)
```
If `len` is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least `len` coefficients. No data is lost when calling this function. The function efficiently deals with the case where it is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

```c
void _fmpz_mod_poly_normalise(fmpz_mod_poly_t poly)
```
Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```c
void _fmpz_mod_poly_set_length(fmpz_mod_poly_t poly, slong len)
```
Demotes the coefficients of `poly` beyond `len` and sets the length of `poly` to `len`.

```c
void fmpz_mod_poly_truncate(fmpz_mod_poly_t poly, slong len, const fmpz_mod_ctx_t ctx)
```
If the current length of `poly` is greater than `len`, it is truncated to have the given length. Discarded coefficients are not necessarily set to zero.

```c
void fmpz_mod_poly_set_trunc(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, slong n, const fmpz_mod_ctx_t ctx)
```
Notionally truncate `poly` to length `n` and set `res` to the result. The result is normalised.

6.12.4 Randomisation

```c
void fmpz_mod_poly_randtest(fmpz_mod_poly_t f, flint_rand_t state, slong len, const fmpz_mod_ctx_t ctx)
```
Sets the polynomial~`f` to a random polynomial of length up~`len`.

```c
void fmpz_mod_poly_randtest_irreducible(fmpz_mod_poly_t f, flint_rand_t state, slong len, const fmpz_mod_ctx_t ctx)
```
Sets the polynomial~`f` to a random irreducible polynomial of length up~`len`, assuming `len` is positive.

```c
void fmpz_mod_poly_randtest_not_zero(fmpz_mod_poly_t f, flint_rand_t state, slong len, const fmpz_mod_ctx_t ctx)
```
Sets the polynomial~`f` to a random polynomial of length up~`len`, assuming `len` is positive.
void \texttt{fmpz\_mod\_poly\_randtest\_monic}(\texttt{fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{flint\_rand\_t} \texttt{state}, \texttt{slong} \texttt{len}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Generates a random monic polynomial with length \texttt{len}.

void \texttt{fmpz\_mod\_poly\_randtest\_monic\_irreducible}(\texttt{fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{flint\_rand\_t} \texttt{state},
\texttt{slong} \texttt{len}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Generates a random monic irreducible polynomial with length \texttt{len}.

void \texttt{fmpz\_mod\_poly\_randtest\_monic\_primitive}(\texttt{fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{flint\_rand\_t} \texttt{state}, \texttt{slong} \texttt{len}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Generates a random monic irreducible primitive polynomial with length \texttt{len}.

void \texttt{fmpz\_mod\_poly\_randtest\_trinomial}(\texttt{fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{flint\_rand\_t} \texttt{state}, \texttt{slong} \texttt{len}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Generates a random monic trinomial of length \texttt{len}.

\begin{verbatim}
int \texttt{fmpz\_mod\_poly\_randtest\_trinomial\_irreducible}(\texttt{fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{flint\_rand\_t} \texttt{state},
\texttt{slong} \texttt{len}, \texttt{slong max\_attempts}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Attempts to set \texttt{poly} to a monic irreducible trinomial of length \texttt{len}. It will generate up to \texttt{max\_attempts} trinomials in attempt to find an irreducible one. If \texttt{max\_attempts} is 0, then it will keep generating trinomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.
\end{verbatim}

void \texttt{fmpz\_mod\_poly\_randtest\_pentomial}(\texttt{fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{flint\_rand\_t} \texttt{state}, \texttt{slong} \texttt{len}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Generates a random monic pentomial of length \texttt{len}.

\begin{verbatim}
int \texttt{fmpz\_mod\_poly\_randtest\_pentomial\_irreducible}(\texttt{fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{flint\_rand\_t} \texttt{state},
\texttt{slong} \texttt{len}, \texttt{slong max\_attempts}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Attempts to set \texttt{poly} to a monic irreducible pentomial of length \texttt{len}. It will generate up to \texttt{max\_attempts} pentomials in attempt to find an irreducible one. If \texttt{max\_attempts} is 0, then it will keep generating pentomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.
\end{verbatim}

void \texttt{fmpz\_mod\_poly\_randtest\_sparse\_irreducible}(\texttt{fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{flint\_rand\_t} \texttt{state},
\texttt{slong} \texttt{len}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Attempts to set \texttt{poly} to a sparse, monic irreducible polynomial with length \texttt{len}. It attempts to find an irreducible trinomial. If that does not succeed, it attempts to find a irreducible pentomial. If that fails, then \texttt{poly} is just set to a random monic irreducible polynomial.

\section{6.12.5 Attributes}

\begin{verbatim}
slong \texttt{fmpz\_mod\_poly\_degree}(\texttt{const fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Returns the degree of the polynomial. The degree of the zero polynomial is defined to be \(-1\).
\end{verbatim}

\begin{verbatim}
slong \texttt{fmpz\_mod\_poly\_length}(\texttt{const fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Returns the length of the polynomial, which is one more than its degree.
\end{verbatim}

\begin{verbatim}
\texttt{fmpz* \texttt{fmpz\_mod\_poly\_lead}(\texttt{const fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Returns a pointer to the first leading coefficient of \texttt{poly} if this is non-zero, otherwise returns NULL.
\end{verbatim}
6.12.6 Assignment and basic manipulation

void fmpz_mod_poly_set(fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, const fmpz_mod_ctx_t ctx)

Sets the polynomial poly1 to the value of poly2.

void fmpz_mod_poly_swap(fmpz_mod_poly_t poly1, fmpz_mod_poly_t poly2, const fmpz_mod_ctx_t ctx)

Swaps the two polynomials. This is done efficiently by swapping pointers rather than individual coefficients.

void fmpz_mod_poly_zero(fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

Sets poly to the zero polynomial.

void fmpz_mod_poly_one(fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

Sets poly to the constant polynomial 1.

void fmpz_mod_poly_zero_coeffs(fmpz_mod_poly_t poly, slong i, slong j, const fmpz_mod_ctx_t ctx)

Sets the coefficients of $X^k$ for $k \in [i, j)$ in the polynomial to zero.

void fmpz_mod_poly_reverse(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, slong n, const fmpz_mod_ctx_t ctx)

This function considers the polynomial poly to be of length n, notionally truncating and zero padding if required, and reverses the result. Since the function normalises its result res may be of length less than n.

6.12.7 Conversion

void fmpz_mod_poly_set_ui(fmpz_mod_poly_t f, ulong c, const fmpz_mod_ctx_t ctx)

Sets the polynomial $f$ to the constant $c$ reduced modulo $p$.

void fmpz_mod_poly_set_fmpz(fmpz_mod_poly_t f, const fmpz_t c, const fmpz_mod_ctx_t ctx)

Sets the polynomial $f$ to the constant $c$ reduced modulo $p$.

void fmpz_mod_poly_set_fmpz_poly(fmpz_mod_poly_t f, const fmpz_poly_t g, const fmpz_mod_ctx_t ctx)

Sets $f$ to $g$ reduced modulo $p$, where $p$ is the modulus that is part of the data structure of $f$.

void fmpz_mod_poly_get_fmpz_poly(fmpz_poly_t f, const fmpz_mod_poly_t g, const fmpz_mod_ctx_t ctx)

Sets $f$ to $g$. This is done simply by lifting the coefficients of $g$ taking representatives $[0, p) \subset \mathbb{Z}$.

void fmpz_mod_poly_get_nmod_poly(nmod_poly_t f, const fmpz_mod_poly_t g)

Sets $f$ to $g$ assuming the modulus of both polynomials is the same (no checking is performed).

void fmpz_mod_poly_set_nmod_poly(fmpz_mod_poly_t f, const nmod_poly_t g)

Sets $f$ to $g$ assuming the modulus of both polynomials is the same (no checking is performed).
6.12.8 Comparison

```c
int fmpz_mod_poly_equal(const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, const fmpz_mod_ctx_t ctx)

Returns non-zero if the two polynomials are equal, otherwise returns zero.
```

```c
int fmpz_mod_poly_equal_trunc(const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, slong n, const fmpz_mod_ctx_t ctx)

Notionally truncates the two polynomials to length \( n \) and returns non-zero if the two polynomials are equal, otherwise returns zero.
```

```c
int fmpz_mod_poly_is_zero(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

Returns non-zero if the polynomial is zero.
```

```c
int fmpz_mod_poly_is_one(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

Returns non-zero if the polynomial is the constant 1.
```

```c
int fmpz_mod_poly_is_gen(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

Returns non-zero if the polynomial is the degree 1 polynomial \( x \).
```

6.12.9 Getting and setting coefficients

```c
void fmpz_mod_poly_set_coeff_fmpz(fmpz_mod_poly_t poly, slong n, const fmpz_t x, const fmpz_mod_ctx_t ctx)

Sets the coefficient of \( X^n \) in the polynomial to \( x \), assuming \( n \geq 0 \).
```

```c
void fmpz_mod_poly_set_coeff_ui(fmpz_mod_poly_t poly, slong n, ulong x, const fmpz_mod_ctx_t ctx)

Sets the coefficient of \( X^n \) in the polynomial to \( x \), assuming \( n \geq 0 \).
```

```c
void fmpz_mod_poly_get_coeff_fmpz(fmpz_t x, const fmpz_mod_poly_t poly, slong n, const fmpz_mod_ctx_t ctx)

Sets \( x \) to the coefficient of \( X^n \) in the polynomial, assuming \( n \geq 0 \).
```

```c
void fmpz_mod_poly_set_coeff_mpz(fmpz_mod_poly_t poly, slong n, const mpz_t x, const fmpz_mod_ctx_t ctx)

Sets the coefficient of \( X^n \) in the polynomial to \( x \), assuming \( n \geq 0 \).
```

```c
void fmpz_mod_poly_get_coeff_mpz(mpz_t x, const fmpz_mod_poly_t poly, slong n, const fmpz_mod_ctx_t ctx)

Sets \( x \) to the coefficient of \( X^n \) in the polynomial, assuming \( n \geq 0 \).
```

6.12.10 Shifting

```c
void _fmpz_mod_poly_shift_left(fmpz *res, const fmpz *poly, slong len, slong n)

Sets \( (\text{res}, \text{len} + n) \) to \( (\text{poly}, \text{len}) \) shifted left by \( n \) coefficients.
```

```c
void fmpz_mod_poly_shift_left(fmpz_mod_poly_t f, const fmpz_mod_poly_t g, slong n, const fmpz_mod_ctx_t ctx)

Sets \text{res} to \text{poly} shifted left by \( n \) coeffs. Zero coefficients are inserted.
```
void _fmpz_mod_poly_shift_right(fmpz *res, const fmpz *poly, slong len, slong n)
Sets (res, len - n) to (poly, len) shifted right by n coefficients.
Assumes that len and n are positive, that len > n, and that res fits len - n elements. Supports aliasing between res and poly, although in this case the top coefficients of poly are not set to zero.

void fmpz_mod_poly_shift_right(fmpz_mod_poly_t f, const fmpz_mod_poly_t g, slong n, const fmpz_mod_ctx_t ctx)
Sets res to poly shifted right by n coefficients. If n is equal to or greater than the current length of poly, res is set to the zero polynomial.

6.12.11 Addition and subtraction

void _fmpz_mod_poly_add(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, const fmpz_mod_ctx_t ctx)
Sets res to the sum of (poly1, len1) and (poly2, len2). It is assumed that res has sufficient space for the longer of the two polynomials.

void fmpz_mod_poly_add(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, const fmpz_mod_ctx_t ctx)
Sets res to the sum of poly1 and poly2.

void fmpz_mod_poly_add_series(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, slong n, const fmpz_mod_ctx_t ctx)
Notionally truncate poly1 and poly2 to length n and set res to the sum.

void _fmpz_mod_poly_sub(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, const fmpz_mod_ctx_t ctx)
Sets res to (poly1, len1) minus (poly2, len2). It is assumed that res has sufficient space for the longer of the two polynomials.

void fmpz_mod_poly_sub(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, const fmpz_mod_ctx_t ctx)
Sets res to poly1 minus poly2.

void fmpz_mod_poly_sub_series(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, slong n, const fmpz_mod_ctx_t ctx)
Notionally truncate poly1 and poly2 to length n and set res to the difference.

void _fmpz_mod_poly_neg(fmpz *res, const fmpz *poly, slong len, const fmpz_mod_ctx_t ctx)
Sets (res, len) to the negative of (poly, len) modulo p.

void fmpz_mod_poly_neg(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
Sets res to the negative of poly modulo p.

6.12.12 Scalar multiplication and division

void _fmpz_mod_poly_scalar_mul_fmpz(fmpz *res, const fmpz *poly, slong len, const fmpz_t x, const fmpz_mod_ctx_t ctx)
Sets (res, len) to (poly, len) multiplied by x, reduced modulo p.

void fmpz_mod_poly_scalar_mul_fmpz(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const fmpz_t x, const fmpz_mod_ctx_t ctx)
Sets res to poly multiplied by x.
void fmpz_mod_poly_scalar_addmul_fmpz(fmpz_mod_poly_t rop, const fmpz_mod_poly_t op, const fmpz_t x, const fmpz_mod_ctx_t ctx)

Adds to rop the product of op by the scalar x.

void _fmpz_mod_poly_scalar_div_fmpz(fmpz *res, const fmpz *poly, slong len, const fmpz_t x, const fmpz_mod_ctx_t ctx)

Sets (res, len) to (poly, len) divided by x (i.e. multiplied by the inverse of x (mod p)). The result is reduced modulo p.

void fmpz_mod_poly_scalar_div_fmpz(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const fmpz_t x, const fmpz_mod_ctx_t ctx)

Sets res to poly divided by x, (i.e. multiplied by the inverse of x (mod p)). The result is reduced modulo p.

6.12.13 Multiplication

void _fmpz_mod_poly_mul(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, const fmpz_mod_ctx_t ctx)

Sets (res, len1 + len2 - 1) to the product of (poly1, len1) and (poly2, len2). Assumes len1 >= len2 > 0. Allows zero-padding of the two input polynomials.

void fmpz_mod_poly_mul(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, const fmpz_mod_ctx_t ctx)

Sets res to the product of poly1 and poly2.

void _fmpz_mod_poly_mullow(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n, const fmpz_mod_ctx_t ctx)

Sets (res, n) to the lowest n coefficients of the product of (poly1, len1) and (poly2, len2). Assumes len1 >= len2 > 0 and 0 < n <= len1 + len2 - 1. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

void fmpz_mod_poly_mullow(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, slong n, const fmpz_mod_ctx_t ctx)

Sets res to the lowest n coefficients of the product of poly1 and poly2.

void _fmpz_mod_poly_sqr(fmpz *res, const fmpz *poly, slong len, const fmpz_mod_ctx_t ctx)

Sets res to the square of poly.

Computes res as the square of poly.

void fmpz_mod_poly_mulhigh(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, slong start, const fmpz_mod_ctx_t ctx)

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary.

void _fmpz_mod_poly_mulmod(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, const fmpz *f, slong lenf, const fmpz_mod_ctx_t ctx)

Sets res, len1 + len2 - 1 to the remainder of the product of poly1 and poly2 upon polynomial division by f.

It is required that len1 + len2 - lenf > 0, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use _fmpz_mod_poly_mul instead.

Aliasing of f and res is not permitted.

6.12. fmpz_mod_poly.h – polynomials over integers mod n
6.12.14 Products

void _fmpz_mod_poly_product_roots_fmpz_vec(fmpz *poly, const fmpz *xs, slong n, const fmpz_mod_ctx_t ctx)

Sets \( \text{poly} \) to the monic polynomial which is the product of \( (x-x_0)(x-x_1)\cdots(x-x_{n-1}) \), the roots \( x_i \) being given by \( xs \). It is required that the roots are canonical.

Aliasing of the input and output is not allowed.

void fmpz_mod_poly_product_roots_fmpz_vec(fmpz_mod_poly_t poly, const fmpz *xs, slong n, const fmpz_mod_ctx_t ctx)

Sets \( \text{poly} \) to the monic polynomial which is the product of \( (x-x_0)(x-x_1)\cdots(x-x_{n-1}) \), the roots \( x_i \) being given by \( xs \). It is required that the roots are canonical.

int fmpz_mod_poly_find_distinct_nonzero_roots(fmpz *roots, const fmpz_mod_poly_t A, const fmpz_mod_ctx_t ctx)

If \( A \) has \( \deg(A) \) distinct nonzero roots in \( \mathbb{F}_p \), write these roots out to \( \text{roots}[0] \) to \( \text{roots}[\deg(A)-1] \) and return 1. Otherwise, return 0. It is assumed that \( A \) is nonzero and that the modulus of \( A \) is prime. This function uses Rabin’s probabilistic method via gcd’s with \( (x+\delta)^{\frac{p-1}{2}} - 1 \).

Powering

void _fmpz_mod_poly_pow(fmpz *rop, const fmpz *op, slong len, ulong e, const fmpz_mod_ctx_t ctx)

Sets \( rop = poly \cdot e \), assuming that \( e > 1 \) and \( \text{elen} > 0 \), and that \( \text{res} \) has space for \( e \cdot (\text{len} - 1) + 1 \) coefficients. Does not support aliasing.

void fmpz_mod_poly_pow(fmpz_mod_poly_t rop, const fmpz_mod_poly_t op, slong e, const fmpz_mod_ctx_t ctx)

Computes \( rop = poly \cdot e \). If \( e \) is zero, returns one, so that in particular \( 0^0 = 1 \).

void _fmpz_mod_poly_pow_trunc(fmpz *res, const fmpz *poly, ulong e, slong trunc, const fmpz_mod_ctx_t ctx)

Sets \( res \) to the low \( \text{trunc} \) coefficients of \( \text{poly} \) (assumed to be zero padded if necessary to length \( \text{trunc} \)) to the power \( e \). This is equivalent to doing a powering followed by a truncation. We require
that \( \text{res} \) has enough space for \( \text{trunc} \) coefficients, that \( \text{trunc} > 0 \) and that \( e > 1 \). Aliasing is not permitted.

```c
void fmpz_mod_poly_powtrunc(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, ulong e, slong trunc, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to the low \( \text{trunc} \) coefficients of \( \text{poly} \) to the power \( e \). This is equivalent to doing a powering followed by a truncation.

```c
void _fmpz_mod_poly_powtrunc_binexp(fmpz *res, const fmpz *poly, ulong e, slong trunc, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to the low \( \text{trunc} \) coefficients of \( \text{poly} \) (assumed to be zero padded if necessary to length \( \text{trunc} \)) to the power \( e \). This is equivalent to doing a powering followed by a truncation. We require that \( \text{res} \) has enough space for \( \text{trunc} \) coefficients, that \( \text{trunc} > 0 \) and that \( e > 1 \). Aliasing is not permitted. Uses the binary exponentiation method.

```c
void fmpz_mod_poly_powmod_ui_binexp(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, ulong e, slong trunc, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to the low \( \text{trunc} \) coefficients of \( \text{poly} \) to the power \( e \). This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.

```c
void _fmpz_mod_poly_powmod_ui_binexp_preinv(fmpz *res, const fmpz *poly, ulong e, const fmpz *f, slong lenf, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to \( \text{poly} \) raised to the power \( e \) modulo \( f \), using binary exponentiation. We require \( e > 0 \). We require \( \text{lenf} > 1 \). It is assumed that \( \text{poly} \) is already reduced modulo \( f \) and zero-padded as necessary to have length exactly \( \text{lenf} - 1 \). The output \( \text{res} \) must have room for \( \text{lenf} - 1 \) coefficients.

```c
void fmpz_mod_poly_powmod_ui_binexp(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, ulong e, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to \( \text{poly} \) raised to the power \( e \) modulo \( f \), using binary exponentiation. We require \( e >= 0 \). The output \( \text{res} \) must have room for \( \text{lenf} - 1 \) coefficients.

```c
void _fmpz_mod_poly_powmod_ui_binexp_preinv(fmpz *res, const fmpz *poly, ulong e, const fmpz *f, slong lenf, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to \( \text{poly} \) raised to the power \( e \) modulo \( f \), using binary exponentiation. We require \( e > 0 \). We require \( \text{lenf} > 1 \). It is assumed that \( \text{poly} \) is already reduced modulo \( f \) and zero-padded as necessary to have length exactly \( \text{lenf} - 1 \). The output \( \text{res} \) must have room for \( \text{lenf} - 1 \) coefficients.

```c
void fmpz_mod_poly_powmod_ui_binexp(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly,ulong e, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to \( \text{poly} \) raised to the power \( e \) modulo \( f \), using binary exponentiation. We require \( e >= 0 \).
void _fmpz_mod_poly_powmod_fmpz_binexp_preinv(fmpz *res, const fmpz *poly, const fmpz_t e,  
  const fmpz *f, slong lenf, const fmpz *finv, slong lenfinv, const fmpz_mod_ctx_t_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0.  
We require finv to be the inverse of the reverse of f.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded  
as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1  
coefficients.

void fmpz_mod_poly_powmod_fmpz_binexp_preinv(fmpz_mod_poly_t res, const fmpz_mod_poly_t  
  poly, const fmpz_t e, const fmpz_mod_poly_t_t f,  
  const fmpz_mod_poly_t_t finv, const fmpz_mod_ctx_t_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e  
>= 0. We require finv to be the inverse of the reverse of f.

void _fmpz_mod_poly_powmod_x_fmpz_preinv(fmpz *res, const fmpz_t_t e, const fmpz *f, slong lenf,  
  const fmpz *finv, slong lenfinv, const fmpz_mod_ctx_t_t ctx)

Sets res to x raised to the power e modulo f, using sliding window exponentiation. We require e >  
0. We require finv to be the inverse of the reverse of f.

We require lenf > 2. The output res must have room for lenf - 1 coefficients.

void fmpz_mod_poly_powmod_x_fmpz_preinv(fmpz_mod_poly_t res, const fmpz_t_t e, const  
  fmpz_mod_poly_t_t f, const fmpz_mod_poly_t_t finv, const fmpz_mod_ctx_t_t ctx)

Sets res to x raised to the power e modulo f, using sliding window exponentiation. We require e  
>= 0. We require finv to be the inverse of the reverse of x.

void _fmpz_mod_poly_powers_mod_preinv_naive(fmpz **res, const fmpz *f, slong flen, slong n, const  
  fmpz *g, slong glen, const fmpz *ginv, slong ginven, const fmpz_mod_ctx_t_t ctx)

Compute $f^0, f^1, \ldots, f^{(n-1)} \mod g$, where g has length glen and f is reduced mod g and  
has length flen (possibly zero spaced). Assumes res is an array of n arrays each with space for  
at least glen - 1 coefficients and that flen > 0. We require that ginv of length ginven is set  
to the power series inverse of the reverse of g.

void fmpz_mod_poly_powers_mod_preinv_naive(fmpz_mod_poly_t_struct *res, const fmpz_mod_poly_t_t f,  
  slong n, const fmpz_mod_poly_t_t g, const fmpz_mod_ctx_t_t ctx)

Set the entries of the array res to $f^0, f^1, \ldots, f^{(n-1)} \mod g$. No aliasing is permitted  
between the entries of res and either of the inputs.

void _fmpz_mod_poly_powers_mod_preinv_threaded_pool(fmpz **res, const fmpz *f, slong flen,  
  slong n, const fmpz *g, slong glen, const fmpz *ginv, slong ginven, const  
  fmpz_mod_ctx_t_t p, thread_pool_handle *threads, slong num_threads)

Compute $f^0, f^1, \ldots, f^{(n-1)} \mod g$, where g has length glen and f is reduced mod g and  
has length flen (possibly zero spaced). Assumes res is an array of n arrays each with space for  
at least glen - 1 coefficients and that flen > 0. We require that ginv of length ginven is set  
to the power series inverse of the reverse of g.

void fmpz_mod_poly_powers_mod_bsgs(fmpz_mod_poly_t_struct *res, const fmpz_mod_poly_t_t f, slong  
  n, const fmpz_mod_poly_t_t g, const fmpz_mod_ctx_t_t ctx)

Set the entries of the array res to $f^0, f^1, \ldots, f^{(n-1)} \mod g$. No aliasing is permitted  
between the entries of res and either of the inputs.
void fmpz_mod_poly_frobenius_powers_2exp_precomp(fmpz_mod_poly_frobenius_powers_2exp_t pow, const fmpz_mod_poly_t f, const fmpz_mod_poly_t finv, ulong m, const fmpz_mod_ctx_t ctx)

If $p = f\rightarrow p$, compute $x(p^1), x(p^2), x(p^4), \ldots, x(p^{2^l}) \quad (\text{mod } f)$ where $2^l$ is the greatest power of 2 less than or equal to $m$.

Allows construction of $x(p^k)$ for $k = 0, 1, \ldots, x(p^m) \quad (\text{mod } f)$ using fmpz_mod_poly_frobenius_power().

Requires precomputed inverse of $f$, i.e. newton inverse.

void fmpz_mod_poly_frobenius_powers_2exp_clear(fmpz_mod_poly_frobenius_powers_2exp_t pow, const fmpz_mod_ctx_t ctx)

Clear resources used by the fmpz_mod_poly_frobenius_powers_2exp_t struct.

void fmpz_mod_poly_frobenius_power(fmpz_mod_poly_t res, fmpz_mod_poly_frobenius_powers_2exp_t pow, const fmpz_mod_poly_t f, ulong m, const fmpz_mod_ctx_t ctx)

If $p = f\rightarrow p$, compute $x(p^m) \quad (\text{mod } f)$.

Requires precomputed frobenius powers supplied by fmpz_mod_poly_frobenius_powers_2exp_precomp.

If $m == 0$ and $f$ has degree 0 or 1, this performs a division. However an impossible inverse by the leading coefficient of $f$ will have been caught by fmpz_mod_poly_frobenius_powers_2exp_precomp.

void fmpz_mod_poly_frobenius_powers_precomp(fmpz_mod_poly_frobenius_powers_t pow, const fmpz_mod_poly_t f, const fmpz_mod_poly_t finv, ulong m, const fmpz_mod_ctx_t ctx)

If $p = f\rightarrow p$, compute $x(p^0), x(p^1), x(p^2), x(p^3), \ldots, x(p^m) \quad (\text{mod } f)$.

Requires precomputed inverse of $f$, i.e. newton inverse.

void fmpz_mod_poly_frobenius_powers_clear(fmpz_mod_poly_frobenius_powers_t pow, const fmpz_mod_ctx_t ctx)

Clear resources used by the fmpz_mod_poly_frobenius_powers_t struct.

6.12.15 Division

void _fmpz_mod_poly_divrem_basecase(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_t invB, const fmpz_mod_ctx_t ctx)

Computes $(Q, \text{len}A - \text{len}B + 1), (R, \text{len}A)$ such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of $B$ is invertible modulo $p$, and that invB is the inverse.

Assumes that len$(A), \text{len}(B) > 0$. Allows zero-padding in $(A, \text{len}A)$. $R$ and $A$ may be aliased, but apart from this no aliasing of input and output operands is allowed.

void fmpz_mod_poly_divrem_basecase(fmpz_mod_poly_t Q, fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)

Computes $Q, R$ such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of $B$ is invertible modulo $p$.

void _fmpz_mod_poly_divrem_newton_n_preinv(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz *Binv, slong lenBinv, const fmpz_mod_ctx_t ctx)
void \texttt{fmpz\_mod\_poly\_divrem\_newton\_n\_preinv}(\texttt{fmpz\_mod\_poly\_t} Q, \texttt{fmpz\_mod\_poly\_t} R, \texttt{const fmpz\_mod\_poly\_t} A, \texttt{const fmpz\_mod\_poly\_t} B, \texttt{const fmpz\_mod\_poly\_t} Binv, \texttt{const fmpz\_mod\_ctx\_t} ctx)

Computes $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}\,B$, where $A$ is of length $\text{len}A$ and $B$ is of length $\text{len}B$. We require that $Q$ have space for $\text{len}A - \text{len}B + 1$ coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of $B \mod x^{\text{len}(B)}$. The algorithm used is to call \texttt{div\_newton\_n\_preinv()} and then multiply out and compute the remainder.

void \texttt{fmpz\_mod\_poly\_div\_newton\_n\_preinv}(\texttt{fmpz\_mod\_poly\_t} A, \texttt{const fmpz\_mod\_poly\_t} B, \texttt{const fmpz\_mod\_poly\_t} Binv, \texttt{const fmpz\_mod\_ctx\_t} ctx)

Computes $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume $Binv$ is the inverse of the reverse of $B \mod x^{\text{len}(B)}$.

It is required that the length of $A$ is less than or equal to $2*\text{the length of }B - 2$.

The algorithm used is to call \texttt{div\_newton\_n()} and then multiply out and compute the remainder.

void \texttt{fmpz\_mod\_poly\_div\_newton\_n\_preinv}(\texttt{fmpz\_mod\_poly\_t} Q, \texttt{const fmpz\_mod\_poly\_t} A, \texttt{const fmpz\_mod\_poly\_t} B, \texttt{const fmpz\_mod\_poly\_t} Binv, \texttt{const fmpz\_mod\_ctx\_t} ctx)

Notionally computes polynomials $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where $A$ is of length $\text{len}A$ and $B$ is of length $\text{len}B$, but return only $Q$.

We require that $Q$ have space for $\text{len}A - \text{len}B + 1$ coefficients and assume that the leading coefficient of $B$ is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of $B \mod x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

void \texttt{fmpz\_mod\_poly\_div\_newton\_n\_preinv}(\texttt{fmpz\_mod\_poly\_t} Q, \texttt{const fmpz\_mod\_poly\_t} A, \texttt{const fmpz\_mod\_poly\_t} B, \texttt{const fmpz\_mod\_poly\_t} Binv, \texttt{const fmpz\_mod\_ctx\_t} ctx)

Notionally computes $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only $Q$.

We assume that the leading coefficient of $B$ is a unit and that $Binv$ is the inverse of the reverse of $B \mod x^{\text{len}(B)}$.

It is required that the length of $A$ is less than or equal to $2*\text{the length of }B - 2$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

ulong \texttt{fmpz\_mod\_poly\_remove}(\texttt{fmpz\_mod\_poly\_t} f, \texttt{const fmpz\_mod\_poly\_t} g, \texttt{const fmpz\_mod\_ctx\_t} ctx)

Removes the highest possible power of $g$ from $f$ and returns the exponent.

void \texttt{fmpz\_mod\_poly\_rem\_basecase}(\texttt{fmpz\_t} *R, \texttt{const fmpz\_t} *A, \texttt{const fmpz\_t} *B, \texttt{const fmpz\_t} invB, \texttt{const fmpz\_mod\_ctx\_t} ctx)

Notionally, computes $Q, R$ such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets $(R, \text{len}B - 1)$.

Allows aliasing only between $A$ and $R$. Allows zero-padding in $A$ but not in $B$. Assumes that the leading coefficient of $B$ is a unit modulo $p$.

void \texttt{fmpz\_mod\_poly\_rem\_basecase}(\texttt{fmpz\_mod\_poly\_t} R, \texttt{const fmpz\_mod\_poly\_t} A, \texttt{const fmpz\_mod\_poly\_t} B, \texttt{const fmpz\_mod\_ctx\_t} ctx)

Notionally, computes $Q, R$ such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ assuming that the leading term of $B$ is a unit.

void \texttt{fmpz\_mod\_poly\_div}(\texttt{fmpz\_t} *Q, \texttt{const fmpz\_t} *A, \texttt{const fmpz\_t} *B, \texttt{const fmpz\_t} invB, \texttt{const fmpz\_mod\_ctx\_t} ctx)

Notionally, computes $Q, R$ such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets $(Q, \text{len}A - \text{len}B + 1)$.
Assumes that the leading coefficient of $B$ is a unit modulo $p$.

```c
void fmpz_mod_poly_div(fmpz_mod_poly_t Q, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Notationally, computes $Q, R$ such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ assuming that the leading term of $B$ is a unit.

```c
void _fmpz_mod_poly_divrem(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)
```

Computes $(Q, \text{len}A - \text{len}B + 1), (R, \text{len}B - 1)$ such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that $B$ is non-zero, that the leading coefficient of $B$ is invertible modulo $p$ and that $\text{inv}B$ is the inverse.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in $(A, \text{len}A)$. No aliasing of input and output operands is allowed.

```c
void fmpz_mod_poly_divrem(fmpz_mod_poly_t Q, fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Computes $Q, R$ such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that $B$ is non-zero and that the leading coefficient of $B$ is invertible modulo $p$.

```c
void fmpz_mod_poly_divrem_f(fmpz_t f, fmpz_mod_poly_t Q, fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Either finds a non-trivial factor of the modulus $p$, or computes $Q, R$ such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

If the leading coefficient of $B$ is invertible in $\mathbb{Z}/p\mathbb{Z}$, the division with remainder operation is carried out, $Q$ and $R$ are computed correctly, and $f$ is set to 1. Otherwise, $f$ is set to a non-trivial factor of $p$ and $Q$ and $R$ are not touched.

Assumes that $B$ is non-zero.

```c
void _fmpz_mod_poly_rem(fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)
```

Notationally, computes $(Q, \text{len}A - \text{len}B + 1), (R, \text{len}B - 1)$ such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$, returning only the remainder part.

Assumes that $B$ is non-zero, that the leading coefficient of $B$ is invertible modulo $p$.

```c
void fmpz_mod_poly_rem(fmpz_mod_poly_t Q, fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

If $f$ returns with the value 1 then the function operates as _fmpz_mod_poly_rem, otherwise $f$ will be set to a non-trivial factor of $p$.

```c
void fmpz_mod_poly_rem(fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Notationally, computes $Q, R$ such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$, returning only the remainder part.

Assumes that $B$ is non-zero and that the leading coefficient of $B$ is invertible modulo $p$.  

6.12. fmpz_mod_poly.h – polynomials over integers mod $n$ 421
6.12.16 Divisibility testing

```c
int _fmpz_mod_poly_divides_classical(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)
```

Returns 1 if \((B, \text{len}B)\) divides \((A, \text{len}A)\) and sets \((Q, \text{len}A - \text{len}B + 1)\) to the quotient. Otherwise, returns 0 and sets \((Q, \text{len}A - \text{len}B + 1)\) to zero. We require that \(\text{len}A > = \text{len}B > 0\).

```c
int fmpz_mod_poly_divides_classical(fmpz_mod_poly_t Q, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Returns 1 if \(B\) divides \(A\) and sets \(Q\) to the quotient. Otherwise returns 0 and sets \(Q\) to zero.

```c
int _fmpz_mod_poly_divides(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)
```

Returns 1 if \((B, \text{len}B)\) divides \((A, \text{len}A)\) and sets \((Q, \text{len}A - \text{len}B + 1)\) to the quotient. Otherwise, returns 0 and sets \((Q, \text{len}A - \text{len}B + 1)\) to zero. We require that \(\text{len}A > = \text{len}B > 0\).

```c
int fmpz_mod_poly_divides(fmpz_mod_poly_t Q, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Returns 1 if \(B\) divides \(A\) and sets \(Q\) to the quotient. Otherwise returns 0 and sets \(Q\) to zero.

6.12.17 Power series inversion

```c
void _fmpz_mod_poly_inv_series(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n, const fmpz_mod_ctx_t ctx)
```

Sets \((Qinv, n)\) to the inverse of \((Q, n)\) modulo \(x^n\), where \(n \geq 1\), assuming that the bottom coefficient of \(Q\) is invertible modulo \(p\) and that its inverse is \(\text{cin}v\).

```c
void fmpz_mod_poly_inv_series(fmpz_mod_poly_t Qinv, const fmpz_mod_poly_t Q, slong n, const fmpz_mod_ctx_t ctx)
```

Sets \(Qinv\) to the inverse of \(Q\) modulo \(x^n\), where \(n \geq 1\), assuming that the bottom coefficient of \(Q\) is a unit.

```c
void fmpz_mod_poly_inv_series_f(fmpz_t f, fmpz_mod_poly_t Qinv, const fmpz_mod_poly_t Q, slong n, const fmpz_mod_ctx_t ctx)
```

Either sets \(f\) to a nontrivial factor of \(p\) with the value of \(Qinv\) undefined, or sets \(Qinv\) to the inverse of \(Q\) modulo \(x^n\), where \(n \geq 1\).

6.12.18 Power series division

```c
void _fmpz_mod_poly_div_series(fmpz *Q, const fmpz *A, slong Alen, const fmpz *B, slong Blen, slong n, const fmpz_mod_ctx_t ctx)
```

Set \((Q, n)\) to the quotient of the series \((A, \text{Alen})\) and \((B, \text{Blen})\) assuming \(\text{Alen}, \text{Blen} <= n\). We assume the bottom coefficient of \(B\) is invertible modulo \(p\).

```c
void fmpz_mod_poly_div_series(fmpz_mod_poly_t Q, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, slong n, const fmpz_mod_ctx_t ctx)
```

Set \(Q\) to the quotient of the series \(A\) by \(B\), thinking of the series as though they were of length \(n\). We assume that the bottom coefficient of \(B\) is a unit.
6.12.19 Greatest common divisor

void fmpz_mod_poly_make_monic(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

If poly is non-zero, sets res to poly divided by its leading coefficient. This assumes that the leading coefficient of poly is invertible modulo p.

Otherwise, if poly is zero, sets res to zero.

void fmpz_mod_poly_make_monic_f(fmpz_t f, fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

Either set f to 1 and res to poly divided by its leading coefficient or set f to a non-trivial factor of p and leave res undefined.

slong fmpz_mod_poly_gcd(fmpz *G, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)

Sets G to the greatest common divisor of (A, len(A)) and (B, len(B)) and returns its length. Assumes that len(A) > len(B) > 0 and that the vector G has space for sufficiently many coefficients. Assumes that invB is the inverse of the leading coefficients of B modulo the prime number p.

void fmpz_mod_poly_gcd(fmpz_mod_poly_t G, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)

Sets G to the greatest common divisor of A and B. In general, the greatest common divisor is defined in the polynomial ring \((\mathbb{Z}/(p\mathbb{Z})])[X]\) if and only if \(p\) is a prime number. Thus, this function assumes that \(p\) is prime.

slong fmpz_mod_poly_gcd_euclidean_f(fmpz_t f, fmpz *G, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)

Either sets \(f = 1\) and \(G\) to the greatest common divisor of \((A, len(A))\) and \((B, len(B))\) and returns its length, or sets \(f \in (1,p)^{-}\) to a non-trivial factor of \(p\) and leaves the contents of the vector \((G, lenB)\) undefined.

Assumes that len(A) > len(B) > 0 and that the vector G has space for sufficiently many coefficients. Does not support aliasing of any of the input arguments with any of the output arguments.

void fmpz_mod_poly_gcd_euclidean_f(fmpz_t f, fmpz_mod_poly_t G, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)

Either sets \(f = 1\) and \(G\) to the greatest common divisor of \((A, len(A))\) and \((B, len(B))\) and returns its length, or sets \(f \in (1,p)^{-}\) to a non-trivial factor of \(p\).

In general, the greatest common divisor is defined in the polynomial ring \((\mathbb{Z}/(p\mathbb{Z})])[X]\) if and only if \(p\) is a prime number.

6.12. fmpz_mod_poly.h – polynomials over integers mod n
Computes the HGC\( \text{D} \) of \( a \) and \( b \), that is, a matrix\( ^{\sim} M \), a sign\( ^{\sim} \sigma \) and two polynomials \( A \) and \( B \) such that
\[
(A, B)^{\prime} = \sigma M^{-1}(a, b)^{\prime}.
\]
Assumes that \( \text{len}(a) > \text{len}(b) > 0 \).
Assumes that \( A \) and \( B \) have space of size at least \( \text{len}(a) \) and \( \text{len}(b) \), respectively. On exit, \*\text{len}A
and \*\text{len}B will contain the correct lengths of \( A \) and \( B \).
Assumes that \( M[0], M[1], M[2], \text{and } M[3] \) each point to a vector of size at least \( \text{len}(a) \).

If \( f \) returns with the value 1 then the function operates as per \text{fmpz\_mod\_poly\_xgcd\_euclidean}, otherwise \( f \) is set to a nontrivial factor of \( p \).

If \( f \) returns with the value 1 then the function operates as per \text{fmpz\_mod\_poly\_xgcd\_euclidean}, otherwise \( f \) is set to a nontrivial factor of \( p \).

Computes the GCD of \( A \) and \( B \) together with cofactors \( S \) and \( T \) such that \( SA + TB = G \). Returns the length of \( G \).
Assumes that \( \text{len}(A) \geq \text{len}(B) \geq 1 \) and \( (\text{len}(A), \text{len}(B)) \neq (1, 1) \).
No attempt is made to make the GCD monic.
Requires that \( G \) have space for \( \text{len}(B) \) coefficients. Writes \( \text{len}(B) - 1 \) and \( \text{len}(A) - 1 \) coefficients to \( S \) and \( T \), respectively. Note that, in fact, \( \text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1) \) and \( \text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1) \).
No aliasing of input and output operands is permitted.

Computes the GCD of \( A \) and \( B \). The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \( P \) is defined to be \( P \). Except in the case where the GCD is zero, the GCD \( G \) is made monic.

Polynomials \( S \) and \( T \) are computed such that \( S \times A + T \times B = G \). The length of \( S \) will be at most \lenB and the length of \( T \) will be at most \lenA.

If \( f \) returns with the value 1 then the function operates as per \text{fmpz\_mod\_poly\_xgcd}, otherwise \( f \) is set to a nontrivial factor of \( p \).

Computes \( G \), \lenA, \( (S, \text{len}B - 1) \) such that \( G \cong SA \mod B \), returning the actual length of \( G \).
Assumes that $0 < \text{len}(A) < \text{len}(B)$.

```c
void fmpz_mod_poly_gcdinv_euclidean(fmpz_mod_poly_t G, fmpz_mod_poly_t S, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Computes polynomials $G$ and $S$, both reduced modulo B, such that $G \equiv SA \pmod{B}$, where $B$ is assumed to have $\text{len}(B) \geq 2$.

In the case that $A = 0 \pmod{B}$, returns $G = S = 0$.

```c
slong _fmpz_mod_poly_gcdinv_euclidean_f(fmpz_t f, fmpz *G, fmpz *S, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_t invA, const fmpz_mod_ctx_t ctx)
```

If $f$ returns with value 1 then the function operates as per `_fmpz_mod_poly_gcdinv_euclidean()`, otherwise $f$ is set to a nontrivial factor of $p$.

```c
void fmpz_mod_poly_gcdinv_euclidean_f(fmpz_t f, fmpz_mod_poly_t G, fmpz_mod_poly_t S, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

If $f$ returns with value 1 then the function operates as per `fmpz_mod_poly_gcdinv_euclidean()`, otherwise $f$ is set to a nontrivial factor of the modulus of $A$.

```c
slong _fmpz_mod_poly_gcdinv(fmpz *G, fmpz *S, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)
```

Computes $(G, \text{len}A), (S, \text{len}B-1)$ such that $G \equiv SA \pmod{B}$, returning the actual length of $G$.

Assumes that $0 < \text{len}(A) < \text{len}(B)$.

```c
slong _fmpz_mod_poly_gcdinv_f(fmpz_t f, fmpz *G, fmpz *S, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)
```

If $f$ returns with value 1 then the function operates as per `_fmpz_mod_poly_gcdinv()`, otherwise $f$ will be set to a nontrivial factor of $p$.

```c
void fmpz_mod_poly_gcdinv(fmpz_mod_poly_t G, fmpz_mod_poly_t S, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Computes polynomials $G$ and $S$, both reduced modulo B, such that $G \equiv SA \pmod{B}$, where $B$ is assumed to have $\text{len}(B) \geq 2$.

In the case that $A = 0 \pmod{B}$, returns $G = S = 0$.

```c
void fmpz_mod_poly_gcdinv_f(fmpz_t f, fmpz_mod_poly_t G, fmpz_mod_poly_t S, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

If $f$ returns with value 1 then the function operates as per `fmpz_mod_poly_gcdinv()`, otherwise $f$ will be set to a nontrivial factor of $p$.

```c
int _fmpz_mod_poly_invmod(fmpz *A, const fmpz *B, slong lenB, const fmpz *P, slong lenP, const fmpz_mod_ctx_t ctx)
```

Attempts to set $(A, \text{len}P-1)$ to the inverse of $(B, \text{len}B)$ modulo the polynomial $(P, \text{len}P)$. Returns 1 if $(B, \text{len}B)$ is invertible and 0 otherwise.

Assumes that $0 < \text{len}(B) < \text{len}(P)$, and hence also $\text{len}(P) \geq 2$, but supports zero-padding in $(B, \text{len}B)$.

Does not support aliasing.

Assumes that $p$ is a prime number.

```c
int _fmpz_mod_poly_invmod_f(fmpz_t f, fmpz *A, const fmpz *B, slong lenB, const fmpz *P, slong lenP, const fmpz_mod_ctx_t ctx)
```

If $f$ returns with the value 1, then the function operates as per `_fmpz_mod_poly_invmod()`. Otherwise $f$ is set to a nontrivial factor of $p$. 

---

**6.12. fmpz_mod_poly.h – polynomials over integers mod n**

425
int fmpz_mod_poly_invmod(fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_poly_t P, const fmpz_mod_ctx_t ctx)

Attempts to set $A$ to the inverse of $B$ modulo $P$ in the polynomial ring $(\mathbb{Z}/p\mathbb{Z})[X]$, where we assume that $p$ is a prime number.

If $\deg(P) < 2$, raises an exception.

If the greatest common divisor of $B$ and $P$ is~`1` , returns~`1` and sets $A$ to the inverse of $B$. Otherwise, returns~`0` and the value of $A$ on exit is undefined.

int fmpz_mod_poly_invmod_f(fmpz_t f, fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_poly_t P, const fmpz_mod_ctx_t ctx)

If $f$ returns with the value 1, then the function operates as per fmpz_mod_poly_invmod(). Otherwise $f$ is set to a nontrivial factor of $p$.

### 6.12.20 Minpoly

**slong _fmpz_mod_poly_minpoly_bm**(fmpz *poly, const fmpz *seq, slong len, const fmpz_mod_ctx_t ctx)

Sets poly to the coefficients of a minimal generating polynomial for sequence $(seq, len)$ modulo $p$.

The return value equals the length of poly.

It is assumed that $p$ is prime and poly has space for at least $len + 1$ coefficients. No aliasing between inputs and outputs is allowed.

**void fmpz_mod_poly_minpoly_bm**(fmpz_mod_poly_t poly, const fmpz *seq, slong len, const fmpz_mod_ctx_t ctx)

Sets poly to a minimal generating polynomial for sequence seq of length len.

Assumes that the modulus is prime.

This version uses the Berlekamp-Massey algorithm, whose running time is proportional to $len$ times the size of the minimal generator.

**slong _fmpz_mod_poly_minpoly_hgcd**(fmpz *poly, const fmpz *seq, slong len, const fmpz_mod_ctx_t ctx)

Sets poly to the coefficients of a minimal generating polynomial for sequence $(seq, len)$ modulo $p$.

The return value equals the length of poly.

It is assumed that $p$ is prime and poly has space for at least $len + 1$ coefficients. No aliasing between inputs and outputs is allowed.

**void fmpz_mod_poly_minpoly_hgcd**(fmpz_mod_poly_t poly, const fmpz *seq, slong len, const fmpz_mod_ctx_t ctx)

Sets poly to a minimal generating polynomial for sequence seq of length len.

Assumes that the modulus is prime.

This version uses the HGCD algorithm, whose running time is $O(n \log^2 n)$ field operations, regardless of the actual size of the minimal generator.

**slong _fmpz_mod_poly_minpoly**(fmpz *poly, const fmpz *seq, slong len, const fmpz_mod_ctx_t ctx)

Sets poly to the coefficients of a minimal generating polynomial for sequence $(seq, len)$ modulo $p$.

The return value equals the length of poly.

It is assumed that $p$ is prime and poly has space for at least $len + 1$ coefficients. No aliasing between inputs and outputs is allowed.
void \texttt{fmpz\_mod\_poly\_minpoly}(\texttt{fmpz\_mod\_poly\_t} \texttt{poly}, \texttt{const fmpz *} \texttt{seq}, \texttt{slong} \texttt{len}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Sets \texttt{poly} to a minimal generating polynomial for sequence \texttt{seq} of length \texttt{len}.

A minimal generating polynomial is a monic polynomial \( f = x^d + c_{d-1}x^{d-1} + \cdots + c_1x + c_0 \), of minimal degree \( d \), that annihilates any consecutive \( d + 1 \) terms in \texttt{seq}. That is, for any \( i < \texttt{len} - d \),

\[
\text{seq}_i = - \sum_{j=0}^{d-1} \text{seq}_{i+j} \ast f_j.
\]

Assumes that the modulus is prime.

This version automatically chooses the fastest underlying implementation based on \texttt{len} and the size of the modulus.

### Resultant

void \texttt{_fmpz\_mod\_poly\_resultant\_euclidean}(\texttt{fmpz_t} \texttt{res}, \texttt{const fmpz *} \texttt{poly1}, \texttt{slong} \texttt{len1}, \texttt{const fmpz *} \texttt{poly2}, \texttt{slong} \texttt{len2}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Sets \( r \) to the resultant of \((\texttt{poly1}, \texttt{len1})\) and \((\texttt{poly2}, \texttt{len2})\) using the Euclidean algorithm.

Assumes that \( \texttt{len1} \geq \texttt{len2} > 0 \).

Assumes that the modulus is prime.

void \texttt{fmpz\_mod\_poly\_resultant\_euclidean}(\texttt{fmpz\_t} \texttt{r}, \texttt{const fmpz\_mod\_poly\_t} \texttt{f}, \texttt{const fmpz\_mod\_poly\_t} \texttt{g}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Computes the resultant of \( f \) and \( g \) using the Euclidean algorithm.

For two non-zero polynomials \( f(x) = a_m x^m + \cdots + a_0 \) and \( g(x) = b_n x^n + \cdots + b_0 \) of degrees \( m \) and \( n \), the resultant is defined to be

\[
\prod_{(x,y): f(x) = g(y) = 0} (x - y).
\]

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

void \texttt{_fmpz\_mod\_poly\_resultant\_hugcd}(\texttt{fmpz\_t} \texttt{res}, \texttt{const fmpz *} \texttt{A}, \texttt{slong} \texttt{lenA}, \texttt{const fmpz *} \texttt{B}, \texttt{slong} \texttt{lenB}, \texttt{const fmpz\_mod\_ctx\_t} \texttt{ctx})

Sets \texttt{res} to the resultant of \((\texttt{A}, \texttt{lenA})\) and \((\texttt{B}, \texttt{lenB})\) using the half-gcd algorithm.

This algorithm computes the half-gcd as per \texttt{fmpz\_mod\_poly\_gcd\_hugcd()} but additionally updates the resultant every time a division occurs. The half-gcd algorithm computes the GCD recursively. Given inputs \( a \) and \( b \) it lets \( m = \texttt{len}(a)/2 \) and (recursively) performs all quotients in the Euclidean algorithm which do not require the low \( m \) coefficients of \( a \) and \( b \).

This performs quotients in exactly the same order as the ordinary Euclidean algorithm except that the low \( m \) coefficients of the polynomials in the remainder sequence are not computed. A correction step after \texttt{hugcd} has been called computes these low \( m \) coefficients (by matrix multiplication by a transformation matrix also computed by \texttt{hugcd}).

This means that from the point of view of the resultant, all but the last quotient performed by a recursive call to \texttt{hugcd} is an ordinary quotient as per the usual Euclidean algorithm. However, the final quotient may give a remainder of less than \( m + 1 \) coefficients, which won’t be corrected until the \texttt{hugcd} correction step is performed afterwards.

To compute the adjustments to the resultant coming from this corrected quotient, we save the relevant information in an \texttt{nmmod\_poly\_res\_t} struct at the time the quotient is performed so that when the correction step is performed later, the adjustments to the resultant can be computed at that time also.

The only time an adjustment to the resultant is not required after a call to \texttt{hugcd} is if \texttt{hugcd} does nothing (the remainder may already have had less than \( m + 1 \) coefficients when \texttt{hugcd} was called).

Assumes that \( \texttt{lenA} \geq \texttt{lenB} > 0 \).
Assumes that the modulus is prime.

```c
void fmpz_mod_poly_resultant_hgcd(fmpz_t res, const fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const fmpz_mod_ctx_t ctx)
```

Computes the resultant of $f$ and $g$ using the half-gcd algorithm.

For two non-zero polynomials $f(x) = a_m x^m + \cdots + a_0$ and $g(x) = b_n x^n + \cdots + b_0$ of degrees $m$ and $n$, the resultant is defined to be

$$a^n b^m \prod_{(x,y):f(x)=g(y)=0} (x-y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

```c
void _fmpz_mod_poly_resultant(fmpz_t res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, const fmpz_mod_ctx_t ctx)
```

Returns the resultant of $(poly1, len1)$ and $(poly2, len2)$.

Assumes that $len1 \geq len2 > 0$.

Assumes that the modulus is prime.

```c
void fmpz_mod_poly_resultant(fmpz_t res, const fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const fmpz_mod_ctx_t ctx)
```

Computes the resultant of $f$ and $g$.

For two non-zero polynomials $f(x) = a_m x^m + \cdots + a_0$ and $g(x) = b_n x^n + \cdots + b_0$ of degrees $m$ and $n$, the resultant is defined to be

$$a^n b^m \prod_{(x,y):f(x)=g(y)=0} (x-y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

### 6.12.22 Discriminant

```c
void _fmpz_mod_poly_discriminant(fmpz_t d, const fmpz *poly, slong len, const fmpz_mod_ctx_t ctx)
```

Set $d$ to the discriminant of $(poly, len)$. Assumes $len > 1$.

```c
void fmpz_mod_poly_discriminant(fmpz_t d, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

Set $d$ to the discriminant of $f$. We normalise the discriminant so that

$$\text{disc}(f) = (-1)^{\frac{n}{2}} \frac{n(n-1)/2 \text{res}(f,f')/\text{lc}(f)n-m-2)}{\prod_{i<j}(r_i-r_j)^2},$$

where $n = \text{len}(f)$ and $m = \text{len}(f')$. Thus $\text{disc}(f) = \text{lc}(f)^{1/2}n-2')^{1/2}n-m-2]) \prod_{i<j}(r_i-r_j)^2$, where $\text{lc}(f)$ is the leading coefficient of $f$ and $r_i$ are the roots of $f$.

### 6.12.23 Derivative

```c
void _fmpz_mod_poly_derivative(fmpz_t res, const fmpz *poly, slong len, const fmpz_mod_ctx_t ctx)
```

Sets $(res, len - 1)$ to the derivative of $(poly, len)$. Also handles the cases where $len$ is 0 or 1 correctly. Supports aliasing of $res$ and $poly$.

```c
void fmpz_mod_poly_derivative(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Sets $res$ to the derivative of $poly$.  

428 Chapter 6. Integers mod n
6.12.24 Evaluation

void _fmpz_mod_poly_evaluate_fmpz(fmpz_t res, const fmpz *poly, slong len, const fmpz_t a, const fmpz_mod_ctx_t ctx)

Evaluates the polynomial (poly, len) at the integer a and sets res to the result. Aliasing between res and a or any of the coefficients of poly is not supported.

void fmpz_mod_poly_evaluate_fmpz(fmpz_t res, const fmpz_mod_poly_t poly, const fmpz_t a, const fmpz_mod_ctx_t ctx)

Evaluates the polynomial poly at the integer a and sets res to the result. As expected, aliasing between res and a is supported. However, res may not be aliased with a coefficient of poly.

6.12.25 Multipoint evaluation

void _fmpz_mod_poly_evaluate_fmpz_vec_iter(fmpz *ys, const fmpz *coeffs, slong len, const fmpz *xs, slong n, const fmpz_mod_ctx_t ctx)

Evaluates (coeffs, len) at the n values given in the vector xs, writing the output values to ys. The values in xs should be reduced modulo the modulus. Uses Horner’s method iteratively.

void fmpz_mod_poly_evaluate_fmpz_vec_iter(fmpz *ys, const fmpz_mod_poly_t poly, const fmpz *xs, slong n, const fmpz_mod_ctx_t ctx)

Evaluates poly at the n values given in the vector xs, writing the output values to ys. The values in xs should be reduced modulo the modulus. Uses Horner’s method iteratively.

void _fmpz_mod_poly_evaluate_fmpz_vec_fast_precomp(fmpz *vs, const fmpz *poly, slong plen, fmpz_poly_struct *const *tree, slong len, const fmpz_mod_ctx_t ctx)

Evaluates (poly, plen) at the len values given by the precomputed subproduct tree.

void _fmpz_mod_poly_evaluate_fmpz_vec_fast(fmpz *ys, const fmpz *poly, slong len, const fmpz *xs, slong n, const fmpz_mod_ctx_t ctx)

Evaluates (coeffs, len) at the n values given in the vector xs, writing the output values to ys. The values in xs should be reduced modulo the modulus. Uses fast multipoint evaluation, building a temporary subproduct tree.

void fmpz_mod_poly_evaluate_fmpz_vec_fast(fmpz *ys, const fmpz_mod_poly_t poly, const fmpz *xs, slong n, const fmpz_mod_ctx_t ctx)

Evaluates poly at the n values given in the vector xs, writing the output values to ys. The values in xs should be reduced modulo the modulus. Uses fast multipoint evaluation, building a temporary subproduct tree.

void _fmpz_mod_poly_evaluate_fmpz_vec(fmpz *ys, const fmpz *coeffs, slong len, const fmpz *xs, slong n, const fmpz_mod_ctx_t ctx)

Evaluates (coeffs, len) at the n values given in the vector xs, writing the output values to ys. The values in xs should be reduced modulo the modulus.

void fmpz_mod_poly_evaluate_fmpz_vec(fmpz *ys, const fmpz_mod_poly_t poly, const fmpz *xs, slong n, const fmpz_mod_ctx_t ctx)

Evaluates poly at the n values given in the vector xs, writing the output values to ys. The values in xs should be reduced modulo the modulus.
6.12.26 Composition

void _fmpz_mod_poly_compose(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, const fmpz_mod_ctx_t ctx)

Sets res to the composition of (poly1, len1) and (poly2, len2).

Assumes that res has space for (len1-1)*(len2-1) + 1 coefficients, although in $\mathbb{Z}_p[X]$ this might not actually be the length of the resulting polynomial when p is not a prime.

Assumes that poly1 and poly2 are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

void fmpz_mod_poly_compose(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, const fmpz_mod_ctx_t ctx)

Sets res to the composition of poly1 and poly2.

To be precise about the order of composition, denoting res, poly1, and poly2 by $f$, $g$, and $h$, respectively, sets $f(t) = g(h(t))$.

6.12.27 Square roots

The series expansions for $\sqrt{h}$ and $1/\sqrt{h}$ are defined by means of the generalised binomial theorem $h^r = (1+y)^r = \sum_{\{k=0\}}^{\infty} \{r \text{ choose } k\} y^k$. It is assumed that $h$ has constant term 1 and that the coefficients $2^{-k}$ exist in the coefficient ring (i.e. 2 must be invertible).

void _fmpz_mod_poly_invsqrt_series(fmpz *g, const fmpz *h, slong hlen, slong n, const fmpz_mod_ctx_t ctx)

Set the first $n$ terms of $g$ to the series expansion of $1/\sqrt{h}$. It is assumed that $n > 0$ and $h > 0$. Aliasing is not permitted.

void fmpz_mod_poly_invsqrt_series(fmpz_mod_poly_t g, const fmpz_mod_poly_t h, slong n, const fmpz_mod_ctx_t ctx)

Set $g$ to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that $h$ has constant term 1.

void _fmpz_mod_poly_sqrt_series(fmpz *g, const fmpz *h, slong hlen, slong n, const fmpz_mod_ctx_t ctx)

Set the first $n$ terms of $g$ to the series expansion of $\sqrt{h}$. It is assumed that $n > 0$ and $h > 0$. Aliasing is not permitted.

void fmpz_mod_poly_sqrt_series(fmpz_mod_poly_t g, const fmpz_mod_poly_t h, slong n, const fmpz_mod_ctx_t ctx)

Set $g$ to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that $h$ has constant term 1.

int _fmpz_mod_poly_sqrt(fmpz *s, const fmpz *p, slong n, const fmpz_mod_ctx_t ctx)

If $(p, n)$ is a perfect square, sets $(s, n / 2 + 1)$ to a square root of $p$ and returns 1. Otherwise returns 0.

int fmpz_mod_poly_sqrt(fmpz_mod_poly_t s, const fmpz_mod_poly_t p, const fmpz_mod_ctx_t ctx)

If $p$ is a perfect square, sets $s$ to a square root of $p$ and returns 1. Otherwise returns 0.
6.12.28 Modular composition

void _fmpz_mod_poly_compose_mod(fmpz *res, const fmpz *f, slong lenf, const fmpz *g, const fmpz *h, slong lenh, const fmpz_mod_ctx_t ctx)

Sets res to the composition \( f(g) \pmod{h} \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

void fmpz_mod_poly_compose_mod(fmpz_mod_poly_t res, const fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const fmpz_mod_poly_t h, const fmpz_mod_ctx_t ctx)

Sets res to the composition \( f(g) \pmod{h} \). We require that \( h \) is nonzero.

void _fmpz_mod_poly_compose_mod_horner(fmpz *res, const fmpz *f, slong lenf, const fmpz *g, const fmpz *h, slong lenh, const fmpz_mod_ctx_t ctx)

Sets res to the composition \( f(g) \pmod{h} \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner’s rule.

void fmpz_mod_poly_compose_mod_horner(fmpz_mod_poly_t res, const fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const fmpz_mod_poly_t h, const fmpz_mod_ctx_t ctx)

Sets res to the composition \( f(g) \pmod{h} \). We require that \( h \) is nonzero. The algorithm used is Horner’s rule.

void _fmpz_mod_poly_compose_mod_brent_kung(fmpz *res, const fmpz *f, slong lenf, const fmpz *g, const fmpz *h, slong lenh, const fmpz_mod_ctx_t ctx)

Sets res to the composition \( f(g) \pmod{h} \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). We also require that the length of \( f \) is less than the length of \( h \). The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

void fmpz_mod_poly_compose_mod_brent_kung(fmpz_mod_poly_t res, const fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const fmpz_mod_poly_t h, const fmpz_mod_ctx_t ctx)

Sets res to the composition \( f(g) \pmod{h} \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). The algorithm used is the Brent-Kung matrix algorithm.

void _fmpz_mod_poly_reduce_matrix_mod_poly(fmpz_mat_t A, const fmpz_mat_t B, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)

Sets the \( i \)th row of \( A \) to the reduction of the \( i \)th row of \( B \) modulo \( f \) for \( i = 1, \ldots, \sqrt{\deg(f)} \). We require \( B \) to be at least a \( \sqrt{\deg(f)} \times \deg(f) \) matrix and \( f \) to be nonzero.

void _fmpz_mod_poly_precompute_matrix_worker(void *arg_ptr)

Worker function version of _fmpz_mod_poly_precompute_matrix. Input/output is stored in fmpz_mod_poly_matrix_precompute_arg_t.

void _fmpz_mod_poly_precompute_matrix(fmpz_mat_t A, const fmpz *f, const fmpz *g, slong leng, const fmpz *ginv, slong lenginv, const fmpz_mod_ctx_t ctx)

Sets the \( i \)th row of \( A \) to \( f^i \pmod{g} \) modulo \( g \) for \( i = 1, \ldots, \sqrt{\deg(g)} \). We require \( A \) to be a \( \sqrt{\deg(g)} \times \deg(g) \) matrix. We require \( ginv \) to be the inverse of the reverse of \( g \) and \( g \) to be nonzero. \( f \) has to be reduced modulo \( g \) and of length one less than \( leng \) (possibly with zero padding).
Sets the ith row of A to \( f^i \) modulo g for \( i = 1, \ldots, \sqrt{\text{deg}(g)} \). We require A to be a \( \sqrt{\text{deg}(g)} \times \text{deg}(g) \) matrix. We require \( \text{ginv} \) to be the inverse of the reverse of g.

```c
void _fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv_worker(void *arg_ptr)

Worker function version of _fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv(). Input/output is stored in fmpz_mod_poly_compose_mod_precomp_preinv_arg_t.
```

```c
void _fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv(fmpz *res, const fmpz *f, slong lenf, const fmpz_mat_t A, const fmpz *h, slong l, const fmpz *hinv, slong lenhinv, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo h. We require that h is nonzero. We require that the ith row of A contains \( g^i \) for \( i = 1, \ldots, \sqrt{\text{deg}(h)} \), i.e. A is a \( \sqrt{\text{deg}(h)} \times \text{deg}(h) \) matrix. We also require that the length of \( f \) is less than the length of h. Furthermore, we require \( \text{hinv} \) to be the inverse of the reverse of h. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```c
void fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv(fmpz_mod_poly_t res, const fmpz_mod_poly_t f, const fmpz_mat_t A, const fmpz_mod_poly_t h, const fmpz_mod_poly_t hinv, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo h. We require that the ith row of A contains \( g^i \) for \( i = 1, \ldots, \sqrt{\text{deg}(h)} \), i.e. A is a \( \sqrt{\text{deg}(h)} \times \text{deg}(h) \) matrix. We require that h is nonzero and that \( f \) has smaller degree than h. Furthermore, we require \( \text{hinv} \) to be the inverse of the reverse of h. This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form \( f(g) \) modulo h for fixed g and h.

```c
void _fmpz_mod_poly_compose_mod_brent_kung_preinv(fmpz *res, const fmpz *f, slong lenf, const fmpz *g, const fmpz *h, slong lenh, const fmpz *hinv, slong lenhinv, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo h. We require that h is nonzero and that the length of \( g \) is one less than the length of h (possibly with zero padding). We also require that the length of \( f \) is less than the length of h. Furthermore, we require \( \text{hinv} \) to be the inverse of the reverse of h. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```c
void fmpz_mod_poly_compose_mod_brent_kung_preinv(fmpz_mod_poly_t res, const fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const fmpz_mod_poly_t h, const fmpz_mod_poly_t hinv, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo h. We require that h is nonzero and that \( f \) has smaller degree than h. Furthermore, we require \( \text{hinv} \) to be the inverse of the reverse of h. The algorithm used is the Brent-Kung matrix algorithm.

```c
void _fmpz_mod_poly_compose_mod_brent_kung_vec_preinv(fmpz_mod_poly_struct *res, const fmpz_mod_poly_struct *polys, slong lenl, slong l, const fmpz *g, slong lenl, const fmpz *h, slong lenh, const fmpz *hinv, slong lenhinv, const fmpz_mod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f_i(g) \) modulo h for \( 1 \leq i \leq l \), where \( f_i \) are the 1 elements of polys.
We require that $h$ is nonzero and that the length of $g$ is less than the length of $h$. We also require that the length of $f_i$ is less than the length of $h$. We require $\text{res}$ to have enough memory allocated to hold $l$ $\text{fmpz\_mod\_poly\_struct}$'s. The entries of $\text{res}$ need to be initialised and $l$ needs to be less than $\text{len1}$. Furthermore, we require $h\text{inv}$ to be the inverse of the reverse of $h$. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```c
void fmpz_mod_poly_compose_mod_brent_kung_vec_preinv(fmpz_mod_poly_struct *res, const fmpz_mod_poly_struct *polys, slong len1, slong n, const fmpz_mod_poly_t g, const fmpz_mod_poly_t h, const fmpz_mod_poly_t hinv, const fmpz_mod_ctx_t ctx)
```

Sets $\text{res}$ to the composition $f_i(g)$ modulo $h$ for $1 \leq i \leq n$ where $f_i$ are the $n$ elements of $\text{polys}$. We require $\text{res}$ to have enough memory allocated to hold $n$ $\text{fmpz\_mod\_poly\_struct}$'s. The entries of $\text{res}$ need to be initialised and $n$ needs to be less than $\text{len1}$. We require that $h$ is nonzero and that $f_i$ and $g$ have smaller degree than $h$. Furthermore, we require $h\text{inv}$ to be the inverse of the reverse of $h$. No aliasing of $\text{res}$ and $\text{polys}$ is allowed. The algorithm used is the Brent-Kung matrix algorithm.

```c
void _fmpz_mod_poly_compose_mod_brent_kung_vec_preinv_threaded_pool(fmpz_mod_poly_struct *res, const fmpz_mod_poly_struct *polys, slong lenpolys, slong l, const fmpz *g, slong glen, const fmpz *poly, slong len, const fmpz *polyinv, slong leninv, const fmpz_mod_ctx_t ctx, thread_pool_handle *threads, slong num_threads)
```

Multithreaded version of $\text{_fmpz\_mod\_poly\_compose\_mod\_brent\_kung\_vec\_preinv()}$. Distributing the Horner evaluations across $\text{flint\_get\_num\_threads()}$ threads.

```c
void fmpz_mod_poly_compose_mod_brent_kung_vec_preinv_threaded_pool(fmpz_mod_poly_struct *res, const fmpz_mod_poly_struct *polys, slong len1, slong n, const fmpz_mod_poly_t g, const fmpz_mod_poly_t poly, const fmpz_mod_poly_t polyinv, const fmpz_mod_ctx_t ctx, thread_pool_handle *threads, slong num_threads)
```

Multithreaded version of $\text{fmpz\_mod\_poly\_compose\_mod\_brent\_kung\_vec\_preinv()}$. Distributing the Horner evaluations across $\text{flint\_get\_num\_threads()}$ threads.
void fmpz_mod_poly_compose_mod_brent_kung_vec_preinv_threaded(fmpz_mod_poly_struct *res, const fmpz_mod_poly_struct *polys, slong len1, slong n, const fmpz_mod_poly_t g, const fmpz_mod_poly_t poly, const fmpz_mod_poly_t polyinv, const fmpz_mod_ctx_t ctx)

Multithreaded version of fmpz_mod_poly_compose_mod_brent_kung_vec_preinv(). Distributing the Horner evaluations across flint_get_num_threads() threads.

6.12.29 Subproduct trees

fmpz_poly_struct **fmpz_mod_poly_tree_alloc(slong len)

Allocates space for a subproduct tree of the given length, having linear factors at the lowest level.

void _fmpz_mod_poly_tree_free(fmpz_poly_struct **tree, slong len)

Free the allocated space for the subproduct.

void _fmpz_mod_poly_tree_build(fmpz_poly_struct **tree, const fmpz *roots, slong len, const fmpz_mod_ctx_t ctx)

Builds a subproduct tree in the preallocated space from the len monic linear factors \((x - r_i)\) where \(r_i\) are given by roots. The top level product is not computed.

6.12.30 Radix conversion

The following functions provide the functionality to solve the radix conversion problems for polynomials, which is to express a polynomial \(f(X)\) with respect to a given radix \(r(X)\) as

\[
f(X) = \sum_{i=0}^{N} b_i(X)r(X)^i
\]

where \(N = \lceil \deg(f)/\deg(r) \rceil\). The algorithm implemented here is a recursive one, which performs Euclidean divisions by powers of \(r\) of the form \(r^{2^i}\), and it has time complexity \(\Theta(\deg(f) \log \deg(f))\). It facilitates the repeated use of precomputed data, namely the powers of \(r\) and their power series inverses. This data is stored in objects of type fmpz_mod_poly_radix_t and it is computed using the function fmpz_mod_poly_radix_init(), which only depends on \(r\) and an upper bound on the degree of \(f\).

void _fmpz_mod_poly_radix_init(fmpz **Rpow, fmpz **Rinv, const fmpz *R, slong lenR, slong k, const fmpz_t invL, const fmpz_mod_ctx_t ctx)

Computes powers of \(R\) of the form \(R^{2^i}\) and their Newton inverses modulo \(x^{2^i \deg(R)}\) for \(i = 0, \ldots, k-1\).

Assumes that the vectors Rpow[i] and Rinv[i] have space for \(2^i \deg(R) + 1\) and \(2^i \deg(R)\) coefficients, respectively.

Assumes that the polynomial \(R\) is non-constant, i.e. \(\deg(R) \geq 1\).

Assumes that the leading coefficient of \(R\) is a unit and that the argument invL is the inverse of the coefficient modulo \(p\).

The argument \(p\) is the modulus, which in \(p\)-adic applications is typically a prime power, although this is not necessary. Here, we only assume that \(p \geq 2\).

Note that this precomputed data can be used for any \(F\) such that \(\text{len}(F) \leq 2^k \deg(R)\).
void fmpz_mod_poly_radix_init(fmpz_mod_poly_radix_t D, const fmpz_mod_poly_t R, slong degF, const fmpz_mod_ctx_t ctx)

Carries out the precomputation necessary to perform radix conversion to radix~`R` for polynomials~`F` of degree at most~`degF`.

Assumes that~`R` is non-constant, i.e.~`deg(R) ≥ 1`, and that the leading coefficient is a unit.

void _fmpz_mod_poly_radix(fmpz *B, const fmpz *F, fmpz **Rpow, fmpz **Rinv, slong degR, slong k, slong i, fmpz *W, const fmpz_mod_ctx_t ctx)

This is the main recursive function used by the function~`fmpz_mod_poly_radix()`.

Assumes that, for all~`i = 0,...,N`, the vector~`B[i]` has space for~`deg(R)` coefficients.

The variable~`k` denotes the factors of~`r` that have previously been counted for the polynomial~`F`, which is assumed to have length~`2^{i+1}deg(R)`, possibly including zero-padding.

Assumes that~`W` is a vector providing temporary space of length~`len(F) = 2^{i+1}deg(R)`.

The entire computation takes place over~`Z/pZ`, where~`p ≥ 2` is a natural number.

Thus, the top level call will have~`F` as in the original problem, and~`k = 0`.

void fmpz_mod_poly_radix(fmpz_mod_poly_t **B, const fmpz_mod_poly_t F, const fmpz_mod_poly_radix_t D, const fmpz_mod_ctx_t ctx)

Given a polynomial~`F` and the precomputed data~`D` for the radix~`R`, computes polynomials~`B_0,...,B_N` of degree less than~`deg(R)` such that

\[ F = B_0 + B_1R + \cdots + B_NR^N, \]

where necessarily~`N = \lceil \frac{deg(F)}{deg(R)} \rceil`.

Assumes that~`R` is non-constant, i.e.~`deg(R) ≥ 1`, and that the leading coefficient is a unit.

### 6.12.31 Input and output

The printing options supported by this module are very similar to what can be found in the two related modules~`fmpz_poly` and~`nmod_poly`. Consider, for example, the polynomial~`f(x) = 5x^3 + 2x + 1` in~`(Z/6Z)[x]`. Its simple string representation is~"4 6 1 2 0 5", where the first two numbers denote the length of the polynomial and the modulus. The pretty string representation is~"5*x^3+2*x+1".

int _fmpz_mod_poly_fprint(FILE *file, const fmpz *poly, slong len, const fmpz_t p)

Prints the polynomial (poly, len) to the stream file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fmpz_mod_poly_fprint(FILE *file, const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

Prints the polynomial to the stream file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fmpz_mod_poly_fprint_pretty(FILE *file, const fmpz_mod_poly_t poly, const char *x, const fmpz_mod_ctx_t ctx)

Prints the pretty representation of (poly, len) to the stream file, using the string~`x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fmpz_mod_poly_print(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

Prints the polynomial to stdout.

In case of success, returns a positive value. In case of failure, returns a non-positive value.
int fmpz_mod_poly_print_pretty(const fmpz_mod_poly_t poly, const char *x, const fmpz_mod_ctx_t ctx)

Prints the pretty representation of poly to stdout, using the string x to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

6.12.32 Inflation and deflation

void fmpz_mod_poly_inflate(fmpz_mod_poly_t result, const fmpz_mod_poly_t input, ulong inflation, const fmpz_mod_ctx_t ctx)

Sets result to the inflated polynomial $p(x^n)$ where $p$ is given by input and $n$ is given by inflation.

void fmpz_mod_poly_deflate(fmpz_mod_poly_t result, const fmpz_mod_poly_t input, ulong deflation, const fmpz_mod_ctx_t ctx)

Sets result to the deflated polynomial $p(x^{1/n})$ where $p$ is given by input and $n$ is given by deflation. Requires $n > 0$.

ulong fmpz_mod_poly_deflation(const fmpz_mod_poly_t input, const fmpz_mod_ctx_t ctx)

Returns the largest integer by which input can be deflated. As special cases, returns 0 if input is the zero polynomial and 1 if input is a constant polynomial.

6.12.33 Berlekamp-Massey Algorithm

The fmpz_mod_berlekamp_massey_t manages an unlimited stream of points $a_1, a_2, \ldots$. At any point in time, after, say, $n$ points have been added, a call to fmpz_mod_berlekamp_massey_reduce() will calculate the polynomials $U$, $V$ and $R$ in the extended euclidean remainder sequence with

$$U \cdot x^n + V \cdot (a_1 \cdot x^{n-1} + \cdots + a_{n-1} \cdot x + a_n) = R,$$

$$\deg(U) < \deg(V) \leq n/2, \quad \deg(R) < n/2.$$ 

The polynomials $V$ and $R$ may be obtained with fmpz_mod_berlekamp_massey_V_poly() and fmpz_mod_berlekamp_massey_R_poly(). This class differs from fmpz_mod_poly_minpoly() in the following respect. Let $v_i$ denote the coefficient of $x^i$ in $V$. fmpz_mod_poly_minpoly() will return a polynomial $V$ of lowest degree that annihilates the whole sequence $a_1, \ldots, a_n$ as

$$\sum_i v_i a_{j+i} = 0, \quad 1 \leq j \leq n - \deg(V).$$

The cost is that a polynomial of degree $n-1$ might be returned and the return is not generally uniquely determined by the input sequence. For the fmpz_mod_berlekamp_massey_t we have

$$\sum_{i,j} v_i a_{j+i} x^{-j} = -U + R \cdot x^n,$$

and it can be seen that $\sum_{i,j} v_i a_{j+i}$ is zero for $1 \leq j < n - \deg(R)$. Thus whether or not $V$ has annihilated the whole sequence may be checked by comparing the degrees of $V$ and $R$.

void fmpz_mod_berlekamp_massey_init(fmpz_mod_berlekamp_massey_t B, const fmpz_mod_ctx_t ctx)

Initialize B with an empty stream.

void fmpz_mod_berlekamp_massey_clear(fmpz_mod_berlekamp_massey_t B, const fmpz_mod_ctx_t ctx)

Free any space used by B.
void fmpz_mod_berlekamp_massey_start_over(fmpz_mod_berlekamp_massey_t B, const
   fmpz_mod_ctx_t ctx)
   
   Empty the stream of points in B.

void fmpz_mod_berlekamp_massey_add_points(fmpz_mod_berlekamp_massey_t B, const fmpz *a,
   slong count, const fmpz_mod_ctx_t ctx)

void fmpz_mod_berlekamp_massey_add_zeros(fmpz_mod_berlekamp_massey_t B, slong count,
   const fmpz_mod_ctx_t ctx)

void fmpz_mod_berlekamp_massey_add_point(fmpz_mod_berlekamp_massey_t B, const
   fmpz_t a, const fmpz_mod_ctx_t ctx)

   Add point(s) to the stream processed by B. The addition of any number of points will not update
   the V and R polynomial.

int fmpz_mod_berlekamp_massey_reduce(fmpz_mod_berlekamp_massey_t B, const
   fmpz_mod_ctx_t ctx)

   Ensure that the polynomials V and R are up to date. The return value is 1 if this function changed
   V and 0 otherwise. For example, if this function is called twice in a row without adding any points
   in between, the return of the second call should be 0. As another example, suppose the object is
   emptied, the points 1,1,2,3 are added, then reduce is called. This reduce should return 1 with
   \( \deg(R) < \deg(V) = 2 \) because the Fibonacci sequence has been recognized. The further addition
   of the two points 5,8 and a reduce will result in a return value of 0.

slong fmpz_mod_berlekamp_massey_point_count(const fmpz_mod_berlekamp_massey_t B)

   Return the number of points stored in B.

cost fmpz *fmpz_mod_berlekamp_massey_points(const fmpz_mod_berlekamp_massey_t B)

   Return a pointer the array of points stored in B. This may be \texttt{NULL} if
   func::fmpz_mod_berlekamp_massey_point_count returns 0.

cost fmpz_mod_poly_struct *fmpz_mod_berlekamp_massey_V_poly(const
   fmpz_mod_berlekamp_massey_t B)

   Return the polynomial V in B.

cost fmpz_mod_poly_struct *fmpz_mod_berlekamp_massey_R_poly(const
   fmpz_mod_berlekamp_massey_t B)

   Return the polynomial R in B.

6.13 fmpz_mod_poly_factor.h – factorisation of polynomials over integers mod n

6.13.1 Types, macros and constants

type fmpz_mod_poly_factor_struct

   A structure representing a polynomial in factorised form as a product of polynomials with associated
   exponents.

type fmpz_mod_poly_factor_t

   An array of length 1 of fmpz_mpoly_factor_struct.
6.13.2 Factorisation

```c
void fmpz_mod_poly_factor_init(fmpz_mod_poly_factor_t fac, const fmpz_mod_ctx_t ctx)
    Initialises fac for use.
void fmpz_mod_poly_factor_clear(fmpz_mod_poly_factor_t fac, const fmpz_mod_ctx_t ctx)
    Frees all memory associated with fac.
void fmpz_mod_poly_factor_realloc(fmpz_mod_poly_factor_t fac, slong alloc, const fmpz_mod_ctx_t ctx)
    Reallocates the factor structure to provide space for precisely alloc factors.
void fmpz_mod_poly_factor_fit_length(fmpz_mod_poly_factor_t fac, slong len, const fmpz_mod_ctx_t ctx)
    Ensures that the factor structure has space for at least len factors. This function takes care of the
    case of repeated calls by always at least doubling the number of factors the structure can hold.
void fmpz_mod_poly_factor_set(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_factor_t fac, const fmpz_mod_ctx_t ctx)
    Sets res to the same factorisation as fac.
void fmpz_mod_poly_factor_print(const fmpz_mod_poly_factor_t fac, const fmpz_mod_ctx_t ctx)
    Prints the entries of fac to standard output.
void fmpz_mod_poly_factor_insert(fmpz_mod_poly_factor_t fac, const fmpz_mod_poly_t poly,
    slong exp, const fmpz_mod_ctx_t ctx)
    Inserts the factor poly with multiplicity exp into the factorisation fac.
    If fac already contains poly, then exp simply gets added to the exponent of the existing entry.
void fmpz_mod_poly_factor_concat(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_factor_t fac,
    const fmpz_mod_ctx_t ctx)
    Concatenates two factorisations.
    This is equivalent to calling fmpz_mod_poly_factor_insert() repeatedly with the individual
    factors of fac.
    Does not support aliasing between res and fac.
void fmpz_mod_poly_factor_pow(fmpz_mod_poly_factor_t fac, slong exp, const fmpz_mod_ctx_t ctx)
    Raises fac to the power exp.
int fmpz_mod_poly_is_irreducible(const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
    Returns 1 if the polynomial f is irreducible, otherwise returns 0.
int fmpz_mod_poly_is_irreducible_ddf(const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
    Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.
int fmpz_mod_poly_is_irreducible_rabin(const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
    Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses Rabin irreducibility test.
int fmpz_mod_poly_is_irreducible_rabin_f(fmpz_t r, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
    Either sets r to 1 and returns 1 if the polynomial f is irreducible or 0 otherwise, or sets r to a
    nontrivial factor of p.
    This algorithm correctly determines whether f is irreducible over \( \mathbb{Z}/p\mathbb{Z} \), even for composite f, or
    it finds a factor of p.
```
int _fmpz_mod_poly_is_squarefree(const fmpz *f, slong len, const fmpz_mod_ctx_t ctx)

Returns 1 if \((f, \text{len})\) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

int _fmpz_mod_poly_is_squarefree_f(fmpz_t fac, const fmpz *f, slong len, const fmpz_mod_ctx_t ctx)

If \(f_{\text{ac}}\) returns with the value 1 then the function operates as per \(_fmpz_mod_poly_is_squarefree()\), otherwise \(f\) is set to a nontrivial factor of \(p\).

int fmpz_mod_poly_is_squarefree(const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)

Returns 1 if \(f\) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

int fmpz_mod_poly_is_squarefree_f(fmpz_t fac, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)

If \(f_{\text{ac}}\) returns with the value 1 then the function operates as per \(fmpz_mod_poly_is_squarefree()\), otherwise \(f\) is set to a nontrivial factor of \(p\).

int fmpz_mod_poly_factor_equal_deg_prob(fmpz_mod_poly_t factor, flint_rand_t state, const fmpz_mod_poly_t pol, slong d, const fmpz_mod_ctx_t ctx)

Probabilistic equal degree factorisation of \(\text{pol}\) into irreducible factors of degree \(d\). If it passes, a factor is placed in \(\text{factor}\) and 1 is returned, otherwise 0 is returned and the value of \(\text{factor}\) is undetermined.

Requires that \(\text{pol}\) be monic, non-constant and squarefree.

void fmpz_mod_poly_factor_equal_deg(fmpz_mod_poly_factor_t factors, const fmpz_mod_poly_t pol, slong d, const fmpz_mod_ctx_t ctx)

Assuming \(\text{pol}\) is a product of irreducible factors all of degree \(d\), finds all those factors and places them in \(\text{factors}\). Requires that \(\text{pol}\) be monic, non-constant and squarefree.

void fmpz_mod_poly_factor_distinct_deg(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t poly, slong *const *degs, const fmpz_mod_ctx_t ctx)

Factorises a monic non-constant squarefree polynomial \(\text{poly}\) of degree \(n\) into factors \(f[d]\) such that for \(1 \leq d \leq n\) \(f[d]\) is the product of the monic irreducible factors of \(\text{poly}\) of degree \(d\). Factors \(f[d]\) are stored in \(\text{res}\), and the degree \(d\) of the irreducible factors is stored in \(\text{degs}\) in the same order as the factors.

Requires that \(\text{degs}\) has enough space for \((n/2) + 1 \times \text{sizeof}(\text{slong})\).

void fmpz_mod_poly_factor_distinct_deg_threaded(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t poly, slong *const *degs, const fmpz_mod_ctx_t ctx)

Multithreaded version of \(fmpz_mod_poly_factor_distinct_deg()\).

void fmpz_mod_poly_factor_squarefree(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)

Sets \(\text{res}\) to a squarefree factorization of \(\text{f}\).

void fmpz_mod_poly_factor(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)

Factorises a non-constant polynomial \(\text{f}\) into monic irreducible factors choosing the best algorithm for given modulo and degree. Choice is based on heuristic measurements.

void fmpz_mod_poly_factor_cantor_zassenhaus(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)

Factorises a non-constant polynomial \(\text{f}\) into monic irreducible factors using the Cantor-Zassenhaus algorithm.
void fmpz_mod_poly_factor_kaltofen_shoup(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)

Factorises a non-constant polynomial \( \text{poly} \) into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a baby step/giant step strategy for the distinct-degree factorization step. If \( \text{flint\_get\_num\_threads()} \) is greater than one \( \text{fmpz\_mod\_poly\_factor\_distinct\_deg\_threaded()} \) is used.

void fmpz_mod_poly_factor_berlekamp(fmpz_mod_poly_factor_t factors, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)

Factorises a non-constant polynomial \( f \) into monic irreducible factors using the Berlekamp algorithm.

void _fmpz_mod_poly_interval_poly_worker(void *arg_ptr)

Worker function to compute interval polynomials in distinct degree factorisation. Input/output is stored in \( \text{fmpz\_mod\_poly\_interval\_poly\_arg\_t} \).

6.13.3 Root Finding

void fmpz_mod_poly_roots(fmpz_mod_poly_factor_t r, const fmpz_mod_poly_t f, int with_multiplicity, const fmpz_mod_ctx_t ctx)

Fill \( r \) with factors of the form \( x - r_i \) where the \( r_i \) are the distinct roots of a nonzero \( f \) in \( \mathbb{Z}/p\mathbb{Z} \). It is expected and not checked that the modulus of \( \text{ctx} \) is prime. If \( \text{with\_multiplicity} \) is zero, the exponent \( e_i \) of the factor \( x - r_i \) is 1. Otherwise, it is the largest \( e_i \) such that \( (x - r_i)^{e_i} \) divides \( f \). This function throws if \( f \) is zero, but is otherwise always successful.

int fmpz_mod_poly_roots_factored(fmpz_mod_poly_factor_t r, const fmpz_mod_poly_t f, int with_multiplicity, const fmpz_factor_t n, const fmpz_mod_ctx_t ctx)

Fill \( r \) with factors of the form \( x - r_i \) where the \( r_i \) are the distinct roots of a nonzero \( f \) in \( \mathbb{Z}/n\mathbb{Z} \). It is expected and not checked that \( n \) is a prime factorization of the modulus of \( \text{ctx} \). If \( \text{with\_multiplicity} \) is zero, the exponent \( e_i \) of the factor \( x - r_i \) is 1. Otherwise, it is the largest \( e_i \) such that \( (x - r_i)^{e_i} \) divides \( f \). The roots are first found modulo the primes in \( n \), then lifted to the corresponding prime powers, then combined into roots of the original polynomial \( f \). A return of 1 indicates the function was successful. A return of 0 indicates the function was not able to find the roots, possibly because there are too many of them. This function throws if \( f \) is zero.

6.14 fmpz_mod_mpoly.h – polynomials over the integers mod n

The exponents follow the \texttt{mpoly} interface. A coefficient may be referenced as a \texttt{fmpz *}, but this may disappear in a future version.

6.14.1 Types, macros and constants

type fmpz_mod_mpoly_struct

A structure holding a multivariate polynomial over the integers mod n.

type fmpz_mod_mpoly_t

An array of length 1 of \texttt{fmpz\_mod\_mpoly\_ctx\_struct}.

type fmpz_mod_mpoly_ctx_struct

Context structure representing the parent ring of an \texttt{fmpz\_mod\_mpoly}.

type fmpz_mod_mpoly_ctx_t

An array of length 1 of \texttt{fmpz\_mod\_mpoly\_struct}.
6.14.2 Context object

```c
void fmpz_mod_mpoly_ctx_init(fmpz_mod_mpoly_ctx_t ctx, slong nvars, const ordering_t ord, const fmpz_t p)
```

Initialise a context object for a polynomial ring modulo \( n \) with \( nvars \) variables and ordering \( ord \). The possibilities for the ordering are \( \text{ORD_LEX} \), \( \text{ORD_DEGLEX} \) and \( \text{ORD_DEGREVLEX} \).

```c
slong fmpz_mod_mpoly_ctx_nvars(const fmpz_mod_mpoly_ctx_t ctx)
```

Return the number of variables used to initialize the context.

```c
ordering_t fmpz_mod_mpoly_ctx_ord(const fmpz_mod_mpoly_ctx_t ctx)
```

Return the ordering used to initialize the context.

```c
void fmpz_mod_mpoly_ctx_get_modulus(fmpz_t n, const fmpz_mod_mpoly_ctx_t ctx)
```

Set \( n \) to the modulus used to initialize the context.

```c
void fmpz_mod_mpoly_ctx_clear(fmpz_mod_mpoly_ctx_t ctx)
```

Release up any space allocated by an \( ctx \).

6.14.3 Memory management

```c
void fmpz_mod_mpoly_init(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Initialise \( A \) for use with the given an initialised context object. Its value is set to zero.

```c
void fmpz_mod_mpoly_init2(fmpz_mod_mpoly_t A, slong alloc, const fmpz_mod_mpoly_ctx_t ctx)
```

Initialise \( A \) for use with the given an initialised context object. Its value is set to zero. It is allocated with space for \( alloc \) terms and at least \( \text{MPOLY_MIN_BITS} \) bits for the exponents.

```c
void fmpz_mod_mpoly_init3(fmpz_mod_mpoly_t A, slong alloc, flint_bitcnt_t bits, const fmpz_mod_mpoly_ctx_t ctx)
```

Initialise \( A \) for use with the given an initialised context object. Its value is set to zero. It is allocated with space for \( alloc \) terms and \( bits \) bits for the exponents.

```c
void fmpz_mod_mpoly_clear(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Release any space allocated for \( A \).

6.14.4 Input/Output

The variable strings in \( x \) start with the variable of most significance at index 0. If \( x \) is \( \text{NULL} \), the variables are named \( x_1, x_2, \) etc.

```c
char *fmpz_mod_mpoly_get_str_pretty(const fmpz_mod_mpoly_t A, const char **x, const fmpz_mod_mpoly_ctx_t ctx)
```

Return a string, which the user is responsible for cleaning up, representing \( A \), given an array of variable strings \( x \).

```c
int fmpz_mod_mpoly_fprint_pretty(FILE *file, const fmpz_mod_mpoly_t A, const char **x, const fmpz_mod_mpoly_ctx_t ctx)
```

Print a string representing \( A \) to \( file \).

```c
int fmpz_mod_mpoly_print_pretty(const fmpz_mod_mpoly_t A, const char **x, const fmpz_mod_mpoly_ctx_t ctx)
```

Print a string representing \( A \) to \( \text{stdout} \).

```c
int fmpz_mod_mpoly_set_str_pretty(fmpz_mod_mpoly_t A, const char *str, const char **x, const fmpz_mod_mpoly_ctx_t ctx)
```

```c
6.14. fmpz_mod_mpoly.h – polynomials over the integers mod n 441
```
Set \( A \) to the polynomial in the null-terminates string \( str \) given an array \( x \) of variable strings. If parsing \( str \) fails, \( A \) is set to zero, and \(-1\) is returned. Otherwise, \( 0 \) is returned. The operations \(+\), \(-\), \(*\), and \(/\) are permitted along with integers and the variables in \( x \). The character \(^\ast\) must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

### 6.14.5 Basic manipulation

```c
void fmpz_mod_mpoly_gen(fmpz_mod_mpoly_t A, slong var, const fmpz_mod_mpoly_ctx_t ctx)
```

Set \( A \) to the variable of index \( var \), where \( var = 0 \) corresponds to the variable with the most significance with respect to the ordering.

```c
int fmpz_mod_mpoly_is_gen(const fmpz_mod_mpoly_t A, slong var, const fmpz_mod_mpoly_ctx_t ctx)
```

If \( var \geq 0 \), return \( 1 \) if \( A \) is equal to the \( var \)-th generator, otherwise return \( 0 \). If \( var < 0 \), return \( 1 \) if the polynomial is equal to any generator, otherwise return \( 0 \).

```c
void fmpz_mod_mpoly_set(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_ctx_t ctx)
```

Set \( A \) to \( B \).

```c
int fmpz_mod_mpoly_equal(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_ctx_t ctx)
```

Return \( 1 \) if \( A \) is equal to \( B \), else return \( 0 \).

```c
void fmpz_mod_mpoly_swap(fmpz_mod_mpoly_t poly1, fmpz_mod_mpoly_t poly2, const fmpz_mod_mpoly_ctx_t ctx)
```

Efficiently swap \( A \) and \( B \).

### 6.14.6 Constants

```c
int fmpz_mod_mpoly_is_fmpz(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Return \( 1 \) if \( A \) is a constant, else return \( 0 \).

```c
void fmpz_mod_mpoly_get_fmpz(fmpz_t c, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Assuming that \( A \) is a constant, set \( c \) to this constant. This function throws if \( A \) is not a constant.

```c
void fmpz_mod_mpoly_set_fmpz(fmpz_mod_mpoly_t A, const fmpz_t c, const fmpz_mod_mpoly_ctx_t ctx)
```

```c
void fmpz_mod_mpoly_set_ui(fmpz_mod_mpoly_t A, ulong c, const fmpz_mod_mpoly_ctx_t ctx)
```

```c
void fmpz_mod_mpoly_set_si(fmpz_mod_mpoly_t A, slong c, const fmpz_mod_mpoly_ctx_t ctx)
```

Set \( A \) to the constant \( c \).

```c
void fmpz_mod_mpoly_zero(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Set \( A \) to the constant \( 0 \).

```c
void fmpz_mod_mpoly_one(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Set \( A \) to the constant \( 1 \).

```c
int fmpz_mod_mpoly_equal_fmpz(const fmpz_mod_mpoly_t A, const fmpz_t c, const fmpz_mod_mpoly_ctx_t ctx)
```

```c
int fmpz_mod_mpoly_equal_ui(const fmpz_mod_mpoly_t A, ulong c, const fmpz_mod_mpoly_ctx_t ctx)
```

442 Chapter 6. Integers mod n
int fmpz_mod_mpoly_equal_si(const fmpz_mod_mpoly_t A, slong c, const fmpz_mod_mpoly_ctx_t ctx)
    Return 1 if A is equal to the constant c, else return 0.

int fmpz_mod_mpoly_is_zero(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
    Return 1 if A is the constant 0, else return 0.

int fmpz_mod_mpoly_is_one(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
    Return 1 if A is the constant 1, else return 0.

6.14.7 Degrees

int fmpz_mod_mpoly_degrees_fit_si(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
    Return 1 if the degrees of A with respect to each variable fit into an slong, otherwise return 0.

void fmpz_mod_mpoly_degrees_fmpz(fmpz **degs, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_degrees_si(slong *degs, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
    Set degs to the degrees of A with respect to each variable. If A is zero, all degrees are set to −1.

void fmpz_mod_mpoly_degree_fmpz(fmpz_t deg, const fmpz_mod_mpoly_t A, slong var, const fmpz_mod_mpoly_ctx_t ctx)

slong fmpz_mod_mpoly_degree_si(const fmpz_mod_mpoly_t A, slong var, const fmpz_mod_mpoly_ctx_t ctx)
    Either return or set deg to the degree of A with respect to the variable of index var. If A is zero, the degree is defined to be −1.

int fmpz_mod_mpoly_total_degree_fits_si(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
    Return 1 if the total degree of A fits into an slong, otherwise return 0.

void fmpz_mod_mpoly_total_degree_fmpz(fmpz_t tdeg, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)

slong fmpz_mod_mpoly_total_degree_si(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
    Either return or set tdeg to the total degree of A. If A is zero, the total degree is defined to be −1.

void fmpz_mod_mpoly_used_vars(int *used, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
    For each variable index i, set used[i] to nonzero if the variable of index i appears in A and to zero otherwise.

6.14.8 Coefficients

void fmpz_mod_mpoly_get_coeff_fmpz_monomial(fmpz_t c, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t M, const fmpz_mod_mpoly_ctx_t ctx)
    Assuming that M is a monomial, set c to the coefficient of the corresponding monomial in A. This function throws if M is not a monomial.

void fmpz_mod_mpoly_set_coeff_fmpz_monomial(fmpz_mod_mpoly_t A, const fmpz_t c, const fmpz_mod_mpoly_t M, const fmpz_mod_mpoly_ctx_t ctx)
Assuming that $M$ is a monomial, set the coefficient of the corresponding monomial in $A$ to $c$. This function throws if $M$ is not a monomial.

```c
void fmpz_mod_mpoly_get_coeff_fmpz_fmpz
(fmpz_t c, const fmpz_mod_mpoly_t A, fmpz *const *exp, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_get_coeff_fmpz_ui
(fmpz_t c, const fmpz_mod_mpoly_t A, const ulong *exp, const fmpz_mod_mpoly_ctx_t ctx)
```

Set $c$ to the coefficient of the monomial with exponent vector $exp$.

```c
void fmpz_mod_mpoly_set_coeff_fmpz_fmpz
(fmpz_mod_mpoly_t A, const fmpz_t c, fmpz *const *exp, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_set_coeff_ui_fmpz
(fmpz_mod_mpoly_t A, ulong c, fmpz *const *exp, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_set_coeff_si_fmpz
(fmpz_mod_mpoly_t A, slong c, fmpz *const *exp, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_set_coeff_fmpz_ui
(fmpz_mod_mpoly_t A, const fmpz_t c, const ulong *exp, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_set_coeff_ui_ui
(fmpz_mod_mpoly_t A, ulong c, const ulong *exp, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_set_coeff_si_ui
(fmpz_mod_mpoly_t A, slong c, const ulong *exp, const fmpz_mod_mpoly_ctx_t ctx)
```

Set the coefficient of the monomial with exponent vector $exp$ to $c$.

```c
void fmpz_mod_mpoly_get_coeff_vars_ui
(fmpz_mod_mpoly_t C, const fmpz_mod_mpoly_t A, const slong *vars, const ulong *exps, slong length, const fmpz_mod_mpoly_ctx_t ctx)
```

Set $C$ to the coefficient of $A$ with respect to the variables in $vars$ with powers in the corresponding array $exps$. Both $vars$ and $exps$ point to array of length $length$. It is assumed that $0 < length \leq nvars(A)$ and that the variables in $vars$ are distinct.

### 6.14.9 Comparison

```c
int fmpz_mod_mpoly_cmp
(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 (resp. $-1$, or 0) if $A$ is after (resp. before, same as) $B$ in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when $A$ and $B$ are both monomials.

### 6.14.10 Container operations

These functions deal with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw.

```c
int fmpz_mod_mpoly_is_canonical
(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)

slong fmpz_mod_mpoly_length
(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 if $A$ is in canonical form. Otherwise, return 0. To be in canonical form, all of the terms must have nonzero coefficient, and the terms must be sorted from greatest to least.

```c
slong fmpz_mod_mpoly_length
(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Return the number of terms in $A$. If the polynomial is in canonical form, this will be the number of nonzero coefficients.

```c
void fmpz_mod_mpoly_resize
(fmpz_mod_mpoly_t A, slong new_length, const fmpz_mod_mpoly_ctx_t ctx)
```

Set the length of $A$ to $new_length$. Terms are either deleted from the end, or new zero terms are appended.
void \texttt{fmpz\_mod\_mpoly\_get\_term\_coeff\_fmpz}(fmpz\_t \texttt{c}, const fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

Set \texttt{c} to the coefficient of the term of index \texttt{i}.

void \texttt{fmpz\_mod\_mpoly\_set\_term\_coeff\_fmpz}(fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, const fmpz\_t \texttt{c}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

void \texttt{fmpz\_mod\_mpoly\_set\_term\_coeff\_ui}(fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, ulong \texttt{c}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

void \texttt{fmpz\_mod\_mpoly\_set\_term\_coeff\_si}(fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, slong \texttt{c}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

Set the coefficient of the term of index \texttt{i} to \texttt{c}.

int \texttt{fmpz\_mod\_mpoly\_term\_exp\_fits\_si}(const fmpz\_mod\_mpoly\_t \texttt{poly}, slong \texttt{i}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

int \texttt{fmpz\_mod\_mpoly\_term\_exp\_fits\_ui}(const fmpz\_mod\_mpoly\_t \texttt{poly}, slong \texttt{i}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

Return 1 if all entries of the exponent vector of the term of index \texttt{i} fit into an \texttt{slong} (resp. a \texttt{ulong}). Otherwise, return 0.

void \texttt{fmpz\_mod\_mpoly\_get\_term\_exp\_fmpz}(fmpz\_t** \texttt{exp}, const fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

void \texttt{fmpz\_mod\_mpoly\_get\_term\_exp\_ui}(ulong* \texttt{exp}, const fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

void \texttt{fmpz\_mod\_mpoly\_get\_term\_exp\_si}(slong* \texttt{exp}, const fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

Set \texttt{exp} to the exponent vector of the term of index \texttt{i}. The _ui (resp. _si) version throws if any entry does not fit into a \texttt{ulong} (resp. \texttt{slong}).

ulong \texttt{fmpz\_mod\_mpoly\_get\_term\_var\_exp\_ui}(const fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, ulong \texttt{var}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

slong \texttt{fmpz\_mod\_mpoly\_get\_term\_var\_exp\_si}(const fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, slong \texttt{var}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

Return the exponent of the variable \texttt{var} of the term of index \texttt{i}. This function throws if the exponent does not fit into a \texttt{ulong} (resp. \texttt{slong}).

void \texttt{fmpz\_mod\_mpoly\_set\_term\_exp\_fmpz}(fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, fmpz* const* \texttt{exp}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

void \texttt{fmpz\_mod\_mpoly\_set\_term\_exp\_ui}(fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, const ulong* \texttt{exp}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

Set the exponent vector of the term of index \texttt{i} to \texttt{exp}.

void \texttt{fmpz\_mod\_mpoly\_get\_term}(fmpz\_mod\_mpoly\_t \texttt{M}, const fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

Set \texttt{M} to the term of index \texttt{i} in \texttt{A}.

void \texttt{fmpz\_mod\_mpoly\_get\_term\_monomial}(fmpz\_mod\_mpoly\_t \texttt{M}, const fmpz\_mod\_mpoly\_t \texttt{A}, slong \texttt{i}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

Set \texttt{M} to the monomial of the term of index \texttt{i} in \texttt{A}. The coefficient of \texttt{M} will be one.

void \texttt{fmpz\_mod\_mpoly\_push\_term\_fmpz\_fmpz}(fmpz\_mod\_mpoly\_t \texttt{A}, const fmpz\_t \texttt{c}, fmpz* const* \texttt{exp}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

void \texttt{fmpz\_mod\_mpoly\_push\_term\_fmpz\_ffmpz}(fmpz\_mod\_mpoly\_t \texttt{A}, const fmpz\_t \texttt{c}, const fmpz* \texttt{exp}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

void \texttt{fmpz\_mod\_mpoly\_push\_term\_ui\_fmpz}(fmpz\_mod\_mpoly\_t \texttt{A}, ulong \texttt{c}, fmpz* const* \texttt{exp}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

void \texttt{fmpz\_mod\_mpoly\_push\_term\_ui\_ffmpz}(fmpz\_mod\_mpoly\_t \texttt{A}, ulong \texttt{c}, const fmpz* \texttt{exp}, const fmpz\_mod\_mpoly\_ctx\_t \texttt{ctx})

6.14. \texttt{fmpz\_mod\_mpoly\_h} – polynomials over the integers mod n
void fmpz_mod_mpoly_push_term_si_fmpz(fmpz_mod_mpoly_t A, slong c, fmpz *const *exp, const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_push_term_si_ffmpz(fmpz_mod_mpoly_t A, slong c, const fmpz *exp, const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_push_term_fmpz_ui(fmpz_mod_mpoly_t A, const fmpz_t c, const ulong *exp, const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_push_term_ui_ui(fmpz_mod_mpoly_t A, ulong c, const ulong *exp, const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_push_term_si_ui(fmpz_mod_mpoly_t A, slong c, const ulong *exp, const fmpz_mod_mpoly_ctx_t ctx)

Append a term to A with coefficient c and exponent vector exp. This function runs in constant average time.

void fmpz_mod_mpoly_sort_terms(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
Sort the terms of A into the canonical ordering dictated by the ordering in ctx. This function simply reorders the terms: It does not combine like terms, nor does it delete terms with coefficient zero. This function runs in linear time in the size of A.

void fmpz_mod_mpoly_combine_like_terms(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
Combine adjacent like terms in A and delete terms with coefficient zero. If the terms of A were sorted to begin with, the result will be in canonical form. This function runs in linear time in the size of A.

void fmpz_mod_mpoly_reverse(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_ctx_t ctx)
Set A to the reversal of B.

6.14.11 Random generation

void fmpz_mod_mpoly_randtest_bound(fmpz_mod_mpoly_t A, flint_rand_t state, slong length, ulong exp_bound, const fmpz_mod_mpoly_ctx_t ctx)
Generate a random polynomial with length up to length and exponents in the range \([0, \text{exp\_bound} - 1]\). The exponents of each variable are generated by calls to \text{n\_randint(state, exp\_bound)}.

void fmpz_mod_mpoly_randtest_bounds(fmpz_mod_mpoly_t A, flint_rand_t state, slong length, ulong *exp\_bounds, const fmpz_mod_mpoly_ctx_t ctx)
Generate a random polynomial with length up to length and exponents in the range \([0, \text{exp\_bounds[i]} - 1]\). The exponents of the variable of index i are generated by calls to \text{n\_randint(state, exp\_bounds[i])}.

void fmpz_mod_mpoly_randtest_bits(fmpz_mod_mpoly_t A, flint_rand_t state, slong length, mp_limb_t exp\_bits, const fmpz_mod_mpoly_ctx_t ctx)
Generate a random polynomial with length up to length and exponents whose packed form does not exceed the given bit count.
6.14.12 Addition/Subtraction

void fmpz_mod_mpoly_add_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_t c, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_add_ui(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, ulong c, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_add_si(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, slong c, const fmpz_mod_mpoly_ctx_t ctx)

   Set A to \( B + c \).

void fmpz_mod_mpoly_sub_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_t c, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_sub_ui(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, ulong c, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_sub_si(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, slong c, const fmpz_mod_mpoly_ctx_t ctx)

   Set A to \( B - c \).

void fmpz_mod_mpoly_add(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_t C, const fmpz_mod_mpoly_ctx_t ctx)

   Set A to \( B + C \).

void fmpz_mod_mpoly_sub(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_t C, const fmpz_mod_mpoly_ctx_t ctx)

   Set A to \( B - C \).

6.14.13 Scalar operations

void fmpz_mod_mpoly_neg(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_ctx_t ctx)

   Set A to \( -B \).

void fmpz_mod_mpoly_scalar_mul_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_t c, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_scalar_mul_ui(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, ulong c, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_scalar_mul_si(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, slong c, const fmpz_mod_mpoly_ctx_t ctx)

   Set A to \( B \times c \).

void fmpz_mod_mpoly_scalar_addmul_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_t C, const fmpz_t d, const fmpz_mod_mpoly_ctx_t ctx)

   Sets A to \( B + C \times d \).

void fmpz_mod_mpoly_make_monic(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_ctx_t ctx)

   Set A to \( B \) divided by the leading coefficient of \( B \). This throws if \( B \) is zero or the leading coefficient is not invertible.
6.14.14 Differentiation

```c
void fmpz_mod_mpoly_derivative(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, slong var, const fmpz_mod_mpoly_ctx_t ctx)
```

Set $A$ to the derivative of $B$ with respect to the variable of index $var$.

6.14.15 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

```c
void fmpz_mod_mpoly_evaluate_all_fmpz(fmpz_t eval, const fmpz_mod_mpoly_t A, fmpz *const *vals, const fmpz_mod_mpoly_ctx_t ctx)
```

Set $ev$ to the evaluation of $A$ where the variables are replaced by the corresponding elements of the array $vals$.

```c
void fmpz_mod_mpoly_evaluate_one_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, slong var, const fmpz_t val, const fmpz_mod_mpoly_ctx_t ctx)
```

Set $A$ to the evaluation of $B$ where the variable of index $var$ is replaced by $val$. Return 1 for success and 0 for failure.

```c
int fmpz_mod_mpoly_compose_fmpz_poly(fmpz_poly_t A, const fmpz_mod_mpoly_t B, fmpz_poly_struct *const *C, const fmpz_mod_mpoly_ctx_t ctxB)
```

Set $A$ to the evaluation of $B$ where the variables are replaced by the corresponding elements of the array $C$. The context object of $B$ is $ctxB$. Return 1 for success and 0 for failure.

```c
int fmpz_mod_mpoly_compose_fmpz_mod_mpoly_geobucket(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, fmpz_mod_mpoly_struct *const *C, const fmpz_mod_mpoly_ctx_t ctxB, const fmpz_mod_mpoly_ctx_t ctxAC)
```

Set $A$ to the evaluation of $B$ where the variables are replaced by the corresponding elements of the array $C$. Both $A$ and the elements of $C$ have context object $ctxAC$, while $B$ has context object $ctxB$. The length of the array $C$ is the number of variables in $ctxB$. Neither $A$ nor $B$ is allowed to alias any other polynomial. Return 1 for success and 0 for failure. The main method attempts to perform the calculation using matrices and chooses heuristically between the geobucket and horner methods if needed.

```c
void fmpz_mod_mpoly_compose_fmpz_mod_mpoly_gen(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const slong *c, const fmpz_mod_mpoly_ctx_t ctxB, const fmpz_mod_mpoly_ctx_t ctxAC)
```

Set $A$ to the evaluation of $B$ where the variable of index $i$ in $ctxB$ is replaced by the variable of index $c[i]$ in $ctxAC$. The length of the array $C$ is the number of variables in $ctxB$. If any $c[i]$ is negative, the corresponding variable of $B$ is replaced by zero. Otherwise, it is expected that $c[i]$ is less than the number of variables in $ctxAC$.
6.14.16 Multiplication

\[
\text{void } \text{fmpz\_mod\_mpoly\_mul} (\text{fmpz\_mod\_mpoly\_t } A, \text{ const } \text{fmpz\_mod\_mpoly\_t } B, \text{ const } \text{fmpz\_mod\_mpoly\_t } C, \text{ const } \text{fmpz\_mod\_mpoly\_ctx\_t } \text{ctx})
\]

Set \( A \) to \( B \times C \).

\[
\text{void } \text{fmpz\_mod\_mpoly\_mul\_johnson} (\text{fmpz\_mod\_mpoly\_t } A, \text{ const } \text{fmpz\_mod\_mpoly\_t } B, \text{ const } \text{fmpz\_mod\_mpoly\_t } C, \text{ const } \text{fmpz\_mod\_mpoly\_ctx\_t } \text{ctx})
\]

Set \( A \) to \( B \times C \) using Johnson’s heap-based method.

\[
\text{int } \text{fmpz\_mod\_mpoly\_mul\_dense} (\text{fmpz\_mod\_mpoly\_t } A, \text{ const } \text{fmpz\_mod\_mpoly\_t } B, \text{ const } \text{fmpz\_mod\_mpoly\_t } C, \text{ const } \text{fmpz\_mod\_mpoly\_ctx\_t } \text{ctx})
\]

Try to set \( A \) to \( B \times C \) using dense arithmetic. If the return is 0, the operation was unsuccessful. Otherwise, it was successful and the return is 1.

6.14.17 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

\[
\text{int } \text{fmpz\_mod\_mpoly\_pow\_fmpz} (\text{fmpz\_mod\_mpoly\_t } A, \text{ const } \text{fmpz\_mod\_mpoly\_t } B, \text{ const } \text{fmpz\_t } k, \text{ const } \text{fmpz\_mod\_mpoly\_ctx\_t } \text{ctx})
\]

Set \( A \) to \( B \) raised to the \( k \)-th power. Return 1 for success and 0 for failure.

\[
\text{int } \text{fmpz\_mod\_mpoly\_pow\_ui} (\text{fmpz\_mod\_mpoly\_t } A, \text{ const } \text{fmpz\_mod\_mpoly\_t } B, \text{ ulong } k, \text{ const } \text{fmpz\_mod\_mpoly\_ctx\_t } \text{ctx})
\]

Set \( A \) to \( B \) raised to the \( k \)-th power. Return 1 for success and 0 for failure.

6.14.18 Division

The division functions assume that the modulus is prime.

\[
\text{int } \text{fmpz\_mod\_mpoly\_divides} (\text{fmpz\_mod\_mpoly\_t } Q, \text{ const } \text{fmpz\_mod\_mpoly\_t } A, \text{ const } \text{fmpz\_mod\_mpoly\_t } B, \text{ const } \text{fmpz\_mod\_mpoly\_ctx\_t } \text{ctx})
\]

If \( A \) is divisible by \( B \), set \( Q \) to the exact quotient and return 1. Otherwise, set \( Q \) to zero and return 0.

\[
\text{void } \text{fmpz\_mod\_mpoly\_div} (\text{fmpz\_mod\_mpoly\_t } Q, \text{ const } \text{fmpz\_mod\_mpoly\_t } A, \text{ const } \text{fmpz\_mod\_mpoly\_t } B, \text{ const } \text{fmpz\_mod\_mpoly\_ctx\_t } \text{ctx})
\]

Set \( Q \) to the quotient of \( A \) by \( B \), discarding the remainder.

\[
\text{void } \text{fmpz\_mod\_mpoly\_divrem} (\text{fmpz\_mod\_mpoly\_t } Q, \text{ fmpz\_mod\_mpoly\_t } R, \text{ const } \text{fmpz\_mod\_mpoly\_t } A, \text{ const } \text{fmpz\_mod\_mpoly\_t } B, \text{ const } \text{fmpz\_mod\_mpoly\_ctx\_t } \text{ctx})
\]

Set \( Q \) and \( R \) to the quotient and remainder of \( A \) divided by \( B \).

\[
\text{void } \text{fmpz\_mod\_mpoly\_divrem\_ideal} (\text{fmpz\_mod\_mpoly\_struct } **Q, \text{ fmpz\_mod\_mpoly\_t } R, \text{ const } \text{fmpz\_mod\_mpoly\_struct } *\text{const } *B, \text{ slong } \text{len}, \text{ const } \text{fmpz\_mod\_mpoly\_ctx\_t } \text{ctx})
\]

This function is as per \text{fmpz\_mod\_mpoly\_divrem()} except that it takes an array of divisor polynomials \( B \) and it returns an array of quotient polynomials \( Q \). The number of divisor (and hence quotient) polynomials, is given by \( \text{len} \).
6.14.19 Greatest Common Divisor

void \texttt{fmpz\_mod\_mpoly\_term\_content}(fmpz\_mod\_mpoly\_t M, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_ctx\_t ctx)

Set \( M \) to the GCD of the terms of \( A \). If \( A \) is zero, \( M \) will be zero. Otherwise, \( M \) will be a monomial with coefficient one.

int \texttt{fmpz\_mod\_mpoly\_content\_vars}(fmpz\_mod\_mpoly\_t g, const fmpz\_mod\_mpoly\_t A, slong *vars, slong vars\_length, const fmpz\_mod\_mpoly\_ctx\_t ctx)

Set \( g \) to the GCD of the coefficients of \( A \) when viewed as a polynomial in the variables \( \text{vars} \). Return 1 for success and 0 for failure. Upon success, \( g \) will be independent of the variables \( \text{vars} \).

int \texttt{fmpz\_mod\_mpoly\_gcd}(fmpz\_mod\_mpoly\_t G, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_t B, const fmpz\_mod\_mpoly\_ctx\_t ctx)

Try to set \( G \) to the monic GCD of \( A \) and \( B \). The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and \( G \) is left untouched.

int \texttt{fmpz\_mod\_mpoly\_gcd\_cofactors}(fmpz\_mod\_mpoly\_t G, fmpz\_mod\_mpoly\_t Abar, fmpz\_mod\_mpoly\_t Bbar, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_t B, const fmpz\_mod\_mpoly\_ctx\_t ctx)

Do the operation of \texttt{fmpz\_mod\_mpoly\_gcd()} and also compute \( \text{A}_\text{bar} = A/G \) and \( \text{B}_\text{bar} = B/G \) if successful.

int \texttt{fmpz\_mod\_mpoly\_gcd\_brown}(fmpz\_mod\_mpoly\_t G, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_t B, const fmpz\_mod\_mpoly\_ctx\_t ctx)

int \texttt{fmpz\_mod\_mpoly\_gcd\_hensel}(fmpz\_mod\_mpoly\_t G, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_t B, const fmpz\_mod\_mpoly\_ctx\_t ctx)

int \texttt{fmpz\_mod\_mpoly\_gcd\_subresultant}(fmpz\_mod\_mpoly\_t G, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_t B, const fmpz\_mod\_mpoly\_ctx\_t ctx)

int \texttt{fmpz\_mod\_mpoly\_gcd\_zippel}(fmpz\_mod\_mpoly\_t G, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_t B, const fmpz\_mod\_mpoly\_ctx\_t ctx)

int \texttt{fmpz\_mod\_mpoly\_gcd\_zippel2}(fmpz\_mod\_mpoly\_t G, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_t B, const fmpz\_mod\_mpoly\_ctx\_t ctx)

Try to set \( G \) to the GCD of \( A \) and \( B \) using various algorithms.

int \texttt{fmpz\_mod\_mpoly\_resultant}(fmpz\_mod\_mpoly\_t R, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_t B, slong var, const fmpz\_mod\_mpoly\_ctx\_t ctx)

Try to set \( R \) to the resultant of \( A \) and \( B \) with respect to the variable of index \( \text{var} \).

int \texttt{fmpz\_mod\_mpoly\_discriminant}(fmpz\_mod\_mpoly\_t D, const fmpz\_mod\_mpoly\_t A, slong var, const fmpz\_mod\_mpoly\_ctx\_t ctx)

Try to set \( D \) to the discriminant of \( A \) with respect to the variable of index \( \text{var} \).

6.14.20 Square Root

The square root functions assume that the modulus is prime for correct operation.

int \texttt{fmpz\_mod\_mpoly\_sqrt}(fmpz\_mod\_mpoly\_t Q, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_ctx\_t ctx)

If \( Q^2 = A \) has a solution, set \( Q \) to a solution and return 1, otherwise return 0 and set \( Q \) to zero.

int \texttt{fmpz\_mod\_mpoly\_is\_square}(const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_ctx\_t ctx)

Return 1 if \( A \) is a perfect square, otherwise return 0.

int \texttt{fmpz\_mod\_mpoly\_quadratic\_root}(fmpz\_mod\_mpoly\_t Q, const fmpz\_mod\_mpoly\_t A, const fmpz\_mod\_mpoly\_t B, const fmpz\_mod\_mpoly\_ctx\_t ctx)

If \( Q^2 + AQ = B \) has a solution, set \( Q \) to a solution and return 1, otherwise return 0.
6.14.21 Univariate Functions

An `fmpz_mod_mpoly_univar_t` holds a univariate polynomial in some main variable with `fmpz_mod_mpoly_t` coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of \( \mathbb{Z}/n\mathbb{Z}[x_1, \ldots, x_m] \) as an element of \( (\mathbb{Z}/n\mathbb{Z}[x_1, \ldots, x_{v-1}, x_{v+1}, \ldots, x_m])[x_v] \) and vice versa.

```c
void fmpz_mod_mpoly_univar_init(fmpz_mod_mpoly_univar_t A, const fmpz_mod_mpoly_ctx_t ctx)
Initialize A.

void fmpz_mod_mpoly_univar_clear(fmpz_mod_mpoly_univar_t A, const fmpz_mod_mpoly_ctx_t ctx)
Clear A.

void fmpz_mod_mpoly_univar_swap(fmpz_mod_mpoly_univar_t A, fmpz_mod_mpoly_univar_t B, const fmpz_mod_mpoly_ctx_t ctx)
Swap A and B.

void fmpz_mod_mpoly_to_univar(fmpz_mod_mpoly_univar_t A, const fmpz_mod_mpoly_t B, slong var, const fmpz_mod_mpoly_ctx_t ctx)
Set A to a univariate form of B by pulling out the variable of index var. The coefficients of A will still belong to the content ctx but will not depend on the variable of index var.

void fmpz_mod_mpoly_from_univar(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_univar_t B, slong var, const fmpz_mod_mpoly_ctx_t ctx)
Set A to the normal form of B by putting in the variable of index var. This function is undefined if the coefficients of B depend on the variable of index var.

int fmpz_mod_mpoly_univar_degree_fits_si(const fmpz_mod_mpoly_univar_t A, const fmpz_mod_mpoly_ctx_t ctx)
Return 1 if the degree of A with respect to the main variable fits an `slong`. Otherwise, return 0.

slong fmpz_mod_mpoly_univar_length(const fmpz_mod_mpoly_univar_t A, const fmpz_mod_mpoly_ctx_t ctx)
Return the number of terms in A with respect to the main variable.

slong fmpz_mod_mpoly_univar_get_term_exp_si(fmpz_mod_mpoly_univar_t A, slong i, const fmpz_mod_mpoly_ctx_t ctx)
Return the exponent of the term of index i of A.

void fmpz_mod_mpoly_univar_get_term_coeff(fmpz_mod_mpoly_t c, const fmpz_mod_mpoly_univar_t A, slong i, const fmpz_mod_mpoly_ctx_t ctx)
Set (resp. swap) c to (resp. with) the coefficient of the term of index i of A.

void fmpz_mod_mpoly_univar_swap_term_coeff(fmpz_mod_mpoly_t c, fmpz_mod_mpoly_univar_t A, slong i, const fmpz_mod_mpoly_ctx_t ctx)

void fmpz_mod_mpoly_univar_set_coeff_ui(fmpz_mod_mpoly_univar_t Ax, ulong e, const fmpz_mod_mpoly_t c, const fmpz_mod_mpoly_ctx_t ctx)
Set the coefficient of \( X^e \) in Ax to c.

int fmpz_mod_mpoly_univar_resultant(fmpz_mod_mpoly_t R, const fmpz_mod_mpoly_univar_t Ax, const fmpz_mod_mpoly_univar_t Bx, const fmpz_mod_mpoly_ctx_t ctx)
Try to set R to the resultant of Ax and Bx.
```
int fmpz_mod_mpoly_univar_discriminant(fmpz_mod_mpoly_t D, const fmpz_mod_mpoly_univar_t Ax, const fmpz_mod_mpoly_ctx_t ctx)

Try to set $D$ to the discriminant of $Ax$.

### 6.14.22 Internal Functions

void fmpz_mod_mpoly_inflate(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz *shift, const fmpz *stride, const fmpz_mod_mpoly_ctx_t ctx)

Apply the function $e \rightarrow \text{shift}[v] + \text{stride}[v] \times e$ to each exponent $e$ corresponding to the variable $v$. It is assumed that each shift and stride is not negative.

void fmpz_mod_mpoly_deflate(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz *shift, const fmpz *stride, const fmpz_mod_mpoly_ctx_t ctx)

Apply the function $e \rightarrow (e - \text{shift}[v]) \div \text{stride}[v]$ to each exponent $e$ corresponding to the variable $v$. If any $\text{stride}[v]$ is zero, the corresponding numerator $e - \text{shift}[v]$ is assumed to be zero, and the quotient is defined as zero. This allows the function to undo the operation performed by fmpz_mod_mpoly_inflate() when possible.

void fmpz_mod_mpoly_deflation(fmpz *shift, fmpz *stride, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)

For each variable $v$ let $S_v$ be the set of exponents appearing on $v$. Set $\text{shift}[v]$ to $\text{min}(S_v)$ and set $\text{stride}[v]$ to $\gcd(S - \text{min}(S_v))$. If $A$ is zero, all shifts and strides are set to zero.

### 6.15 fmpz_mod_mpoly_factor.h – factorisation of multivariate polynomials over the integers mod $n$

#### 6.15.1 Types, macros and constants

type fmpz_mod_mpoly_factor_struct

A struct for holding a factored polynomial over the integers mod $n$. There is a single constant and a product of bases to corresponding exponents.

type fmpz_mod_mpoly_factor_t

An array of length 1 of fmpz_mod_mpoly_factor_struct.

#### 6.15.2 Memory management

void fmpz_mod_mpoly_factor_init(fmpz_mod_mpoly_factor_t f, const fmpz_mod_mpoly_ctx_t ctx)

Initialise $f$.

void fmpz_mod_mpoly_factor_clear(fmpz_mod_mpoly_factor_t f, const fmpz_mod_mpoly_ctx_t ctx)

Clear $f$. 

Chapter 6. Integers mod $n$
### 6.15.3 Basic manipulation

```c
void fmpz_mod_mpoly_factor_swap(fmpz_mod_mpoly_factor_t f, fmpz_mod_mpoly_factor_t g, const fmpz_mod_mpoly_ctx_t ctx)
```

Efficiently swap \( f \) and \( g \).

```c
slong fmpz_mod_mpoly_factor_length(const fmpz_mod_mpoly_factor_t f, const fmpz_mod_mpoly_ctx_t ctx)
```

Return the length of the product in \( f \).

```c
void fmpz_mod_mpoly_factor_get_constant_fmpz(fmpz_t c, const fmpz_mod_mpoly_factor_t f, const fmpz_mod_mpoly_ctx_t ctx)
```

Set \( c \) to the constant of \( f \).

```c
void fmpz_mod_mpoly_factor_get_base(fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_factor_t f, slong i, const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_factor_swap_base(fmpz_mod_mpoly_t B, fmpz_mod_mpoly_factor_t f, slong i, const fmpz_mod_mpoly_ctx_t ctx)
```

Set (resp. swap) \( B \) to (resp. with) the base of the term of index \( i \) in \( f \).

```c
slong fmpz_mod_mpoly_factor_get_exp_si(fmpz_mod_mpoly_factor_t f, slong i, const fmpz_mod_mpoly_ctx_t ctx)
```

Return the exponent of the term of index \( i \) in \( f \). It is assumed to fit an \texttt{slong}.

```c
void fmpz_mod_mpoly_factor_sort(fmpz_mod_mpoly_factor_t f, const fmpz_mod_mpoly_ctx_t ctx)
```

Sort the product of \( f \) first by exponent and then by base.

### 6.15.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and \( f \) is undefined. None of these functions multiply \( f \) by \( A \): \( f \) is simply set to a factorisation of \( A \), and thus these functions should not depend on the initial value of the output \( f \).

```c
int fmpz_mod_mpoly_factor_squarefree(fmpz_mod_mpoly_factor_t f, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Set \( f \) to a factorization of \( A \) where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call \texttt{fmpz_mod_mpoly_factor_sort()} and then multiply the bases with the desired exponent.

```c
int fmpz_mod_mpoly_factor(fmpz_mod_mpoly_factor_t f, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Set \( f \) to a factorization of \( A \) where the bases are irreducible.
7.1 perm.h – permutations

7.1.1 Memory management

`slong *perm_init(slong n)`
Initialises the permutation for use.

`void _perm_clear(slong *vec)`
Clears the permutation.

7.1.2 Assignment

`void _perm_set(slong *res, const slong *vec, slong n)`
Sets the permutation `res` to the same as the permutation `vec`.

`void _perm_one(slong *vec, slong n)`
Sets the permutation to the identity permutation.

`void _perm_inv(slong *res, const slong *vec, slong n)`
Sets `res` to the inverse permutation of `vec`. Allows aliasing of `res` and `vec`.

7.1.3 Composition

`void _perm_compose(slong *res, const slong *vec1, const slong *vec2, slong n)`
Forms the composition $\pi_1 \circ \pi_2$ of two permutations $\pi_1$ and $\pi_2$. Here, $\pi_2$ is applied first, that is, $(\pi_1 \circ \pi_2)(i) = \pi_1(\pi_2(i))$.

Allows aliasing of `res`, `vec1` and `vec2`.

7.1.4 Parity

`int _perm_parity(const slong *vec, slong n)`
Returns the parity of `vec`, 0 if the permutation is even and 1 if the permutation is odd.
### 7.1.5 Randomisation

```c
int _perm_randtest(slong *vec, slong n, flint_rand_t state)
```
Generates a random permutation vector of length $n$ and returns its parity, 0 or 1.
This function uses the Knuth shuffle algorithm to generate a uniformly random permutation without retries.

### 7.2 qfb.h – binary quadratic forms

Authors:
- William Hart
- Håvard Damm-Johnsen (updated documentation)

#### 7.2.1 Introduction

This module contains functionality for creating, listing and reducing binary quadratic forms. A `qfb` struct consists of three `fmpz_t` $a$, $b$ and $c$, and basic algorithms for operations such as reduction, composition and enumerating are implemented and described below.
Currently the code only works for definite binary quadratic forms.

#### 7.2.2 Memory management

```c
void qfb_init(qfb_t q)
```
Initialise a `qfb_t` $q$ for use.

```c
void qfb_clear(qfb_t q)
```
Clear a `qfb_t` after use. This releases any memory allocated for $q$ back to flint.

```c
void qfb_array_clear(qfb **forms, slong num)
```
Clean up an array of `qfb` structs allocated by a qfb function. The parameter `num` must be set to the length of the array.

#### 7.2.3 Hash table

```c
qfb_hash_t *qfb_hash_init(slong depth)
```
Initialises a hash table of size $2^{\text{depth}}$.

```c
void qfb_hash_clear(qfb_hash_t *qhash, slong depth)
```
Frees all memory used by a hash table of size $2^{\text{depth}}$.

```c
void qfb_hash_insert(qfb_hash_t *qhash, qfb_t q, qfb_t q2, slong iter, slong depth)
```
Insert the binary quadratic form $q$ into the given hash table of size $2^{\text{depth}}$ in the field $q$ of the hash structure. Also store the second binary quadratic form $q2$ (if not `NULL`) in the similarly named field and `iter` in the similarly named field of the hash structure.

```c
slong qfb_hash_find(qfb_hash_t *qhash, qfb_t q, slong depth)
```
Search for the given binary quadratic form or its inverse in the given hash table of size $2^{\text{depth}}$. If it is found, return the index in the table (which is an array of `qfb_hash_t` structs), otherwise return -1.
7.2.4 Basic manipulation

```c
void qfb_set(qfb_t f, qfb_t g)
```

Set the binary quadratic form \( f \) to be equal to \( g \).

7.2.5 Comparison

```c
int qfb_equal(qfb_t f, qfb_t g)
```

Returns 1 if \( f \) and \( g \) are identical binary quadratic forms, otherwise returns 0.

7.2.6 Input/output

```c
void qfb_print(qfb_t q)
```

Print a binary quadratic form \( q \) in the format \((a, b, c)\) where \( a, b, c \) are the entries of \( q \).

7.2.7 Computing with forms

```c
void qfb_discriminant(fmpz_t D, qfb_t f)
```

Set \( D \) to the discriminant of the binary quadratic form \( f \), i.e. to \( b^2 - 4ac \), where \( f = (a, b, c) \).

```c
void qfb_reduce(qfb_t r, qfb_t f, fmpz_t D)
```

Set \( r \) to a reduced form equivalent to the binary quadratic form \( f \) of discriminant \( D \).

```c
int qfb_is_reduced(qfb_t r)
```

Returns 1 if \( q \) is a reduced binary quadratic form, otherwise returns 0. Note that this only tests for definite quadratic forms, so a form \( r = (a, b, c) \) is reduced if and only if \(|b| \leq a \leq c\) and if either inequality is an equality, then \( b \geq 0 \).

```c
slong qfb_reduced_forms(qfb **forms, slong d)
```

Given a discriminant \( d \) (negative for negative definite forms), compute all the reduced binary quadratic forms of that discriminant. The function allocates space for these and returns it in the variable \( \text{forms} \) (the user is responsible for cleaning this up by a single call to \( \text{qfb_array_clear} \) on \( \text{forms} \), after use.) The function returns the number of forms generated (the form class number). The forms are stored in an array of \( \text{qfb} \) structs, which contain fields \( a, b, c \) corresponding to \( (a, b, c) \).

```c
slong qfb_reduced_forms_large(qfb **forms, slong d)
```

As for \( \text{qfb_reduced_forms} \). However, for small \(|d|\) it requires fewer primes to be computed at a small cost in speed. It is called automatically by \( \text{qfb_reduced_forms} \) for large \(|d|\) so that \( \text{flint_primes} \) is not exhausted.

```c
void qfb_nucomp(qfb_t r, const qfb_t f, const qfb_t g, fmpz_t D, fmpz_t L)
```

Shanks’ NUCOMP as described in \cite{JvdP2002}.

Computes the near reduced composition of forms \( f \) and \( g \) given \( L = \lceil |D|^{1/4} \rceil \) where \( D \) is the common discriminant of \( f \) and \( g \). The result is returned in \( r \).

We require that \( f \) is a primitive form.

```c
void qfb_nudupl(qfb_t r, const qfb_t f, fmpz_t D, fmpz_t L)
```

As for \( \text{nucomp} \) except that the form \( f \) is composed with itself. We require that \( f \) is a primitive form.

```c
void qfb_pow_ui(qfb_t r, qfb_t f, fmpz_t D, ulong exp)
```

Compute the near reduced form \( r \) which is the result of composing the principal form (identity) with \( f \) \( \text{exp} \) times.

We require \( D \) to be set to the discriminant of \( f \) and that \( f \) is a primitive form.
void qfb_pow(qfb_t r, qfb_t f, fmpz_t D, fmpz_t exp)
    As per qfb_pow_ui.

void qfb_inverse(qfb_t r, qfb_t f)
    Set r to the inverse of the binary quadratic form f.

int qfb_is_principal_form(qfb_t f, fmpz_t D)
    Return 1 if f is the reduced principal form of discriminant D, i.e. the identity in the form class group, else 0.

void qfb_principal_form(qfb_t f, fmpz_t D)
    Set f to the principal form of discriminant D, i.e. the identity in the form class group.

int qfb_is_primitive(qfb_t f)
    Return 1 if f is primitive, i.e. the greatest common divisor of its three coefficients is 1. Otherwise the function returns 0.

void qfb_prime_form(qfb_t r, fmpz_t D, fmpz_t p)
    Sets r to the unique prime \((p, b, c)\) of discriminant \(D\), i.e. with \(0 < b \leq p\). We require that \(p\) is a prime.

int qfb_exponent_element(fmpz_t exponent, qfb_t f, fmpz_t n, ulong B1, ulong B2_sqrt)
    Find the exponent of the element \(f\) in the form class group of forms of discriminant \(n\), doing a stage 1 with primes up to at least \(B1\) and a stage 2 for a single large prime up to at least the square of \(B2_{\text{sqrt}}\). If the function fails to find the exponent it returns 0, otherwise the function returns 1 and \(\text{exponent}\) is set to the exponent of \(f\), i.e. the minimum power of \(f\) which gives the identity.
    It is assumed that the form \(f\) is reduced. We require that \(\text{iters}\) is a power of 2 and that \(\text{iters} \geq 1024\).
    The function performs a stage 2 which stores up to \(4 \times \text{iters}\) binary quadratic forms, and \(12 \times \text{iters}\) additional limbs of data in a hash table, where \(\text{iters}\) is the square root of \(B2\).

int qfb_exponent(fmpz_t exponent, fmpz_t n, ulong B1, ulong B2_sqrt, slong c)
    Compute the exponent of the class group of discriminant \(n\), doing a stage 1 with primes up to at least \(B1\) and a stage 2 for a single large prime up to at least the square of \(B2_{\text{sqrt}}\), and with probability at least \(1 - 2^{-c}\). If the prime limits are exhausted without finding the exponent, the function returns 0, otherwise it returns 1 and \(\text{exponent}\) is set to the computed exponent, i.e. the minimum power to which every element of the class group has to be raised in order to get the identity.
    The function performs a stage 2 which stores up to \(4 \times \text{iters}\) binary quadratic forms, and \(12 \times \text{iters}\) additional limbs of data in a hash table, where \(\text{iters}\) is the square root of \(B2\).
    We use algorithm 8.1 of [Sut2007].

int qfb_exponent_grh(fmpz_t exponent, fmpz_t n, ulong B1, ulong B2_sqrt)
    Similar to qfb_exponent except that the bound \(c\) is automatically generated such that the exponent is guaranteed to be correct, if found, assuming the GRH, namely that the class group is generated by primes less than \(6 \log^2(|n|)\) as described in [BD1992].
7.3 dirichlet.h – Dirichlet characters

Warning: the interfaces in this module are experimental and may change without notice.

This module allows working with Dirichlet characters algebraically. For evaluations of characters as complex numbers, see acb_dirichlet.h – Dirichlet L-functions, Riemann zeta and related functions.

7.3.1 Dirichlet characters

Working with Dirichlet characters mod $q$ consists mainly in going from residue classes mod $q$ to exponents on a set of generators of the group.

This implementation relies on the Conrey numbering scheme introduced in the L-functions and Modular Forms Database, which is an explicit choice of generators allowing to represent Dirichlet characters via the pairing

$$(\mathbb{Z}/q\mathbb{Z})^\times \times (\mathbb{Z}/q\mathbb{Z})^\times \rightarrow \bigoplus_{Z/\phi_i \mathbb{Z}} (\mathbb{Z}/\phi_i \mathbb{Z})^\times \rightarrow \mathbb{C}$$

$$\chi_q(m,n) = \exp(2\pi \sum \frac{a_i b_i}{\phi_i})$$

We call number a residue class $m$ modulo $q$, and log the corresponding vector $(a_i)$ of exponents of Conrey generators.

Going from a log to the corresponding number is a cheap operation we call exponential, while the converse requires computing discrete logarithms.

7.3.2 Multiplicative group modulo $q$

```c

typedef dirichlet_group_struct

type dirichlet_group_t

Represents the group of Dirichlet characters mod $q$.

An dirichlet_group_t is defined as an array of dirichlet_group_struct of length 1, permitting it to be passed by reference.

int dirichlet_group_init(dirichlet_group_t G, ulong q)

Initializes $G$ to the group of Dirichlet characters mod $q$.

This method computes a canonical decomposition of $G$ in terms of cyclic groups, which are the mod $p^e$ subgroups for $p^e || q$, plus the specific generator described by Conrey for each subgroup.

In particular $G$ contains:

- the number num of components
- the generators
- the exponent expo of the group

It does not automatically precompute lookup tables of discrete logarithms or numerical roots of unity, and can therefore safely be called even with large $q$.

For implementation reasons, the largest prime factor of $q$ must not exceed $10^{16}$. This restriction could be removed in the future. The function returns 1 on success and 0 if a factor is too large.

void dirichlet_subgroup_init(dirichlet_group_t H, const dirichlet_group_t G, ulong h)

Given an already computed group $G$ mod $q$, initialize its subgroup $H$ defined mod $h | q$. Precomputed discrete log tables are inherited.

void dirichlet_group_clear(dirichlet_group_t G)

Clears $G$. Remark this function does not clear the discrete logarithm tables stored in $G$ (which may be shared with another group).

```
ulong dirichlet_group_size(const dirichlet_group_t G)
    Returns the number of elements in G, i.e. $\varphi(q)$.

ulong dirichlet_group_num_primitive(const dirichlet_group_t G)
    Returns the number of primitive elements in G.

void dirichlet_group_dlog_precompute(dirichlet_group_t G, ulong num)
    Precompute decomposition and tables for discrete log computations in G, so as to minimize the complexity of num calls to discrete logarithms.
    If num gets very large, the entire group may be indexed.

void dirichlet_group_dlog_clear(dirichlet_group_t G)
    Clear discrete logarithm tables in G. When discrete logarithm tables are shared with subgroups, those subgroups must be cleared before clearing the tables.

### 7.3.3 Character type

type dirichlet_char_struct

type dirichlet_char_t
    Represents a Dirichlet character. This structure contains both a number (residue class) and the corresponding log (exponents on the group generators).

    An dirichlet_char_t is defined as an array of dirichlet_char_struct of length 1, permitting it to be passed by reference.

void dirichlet_char_init(dirichlet_char_t chi, const dirichlet_group_t G)
    Initializes chi to an element of the group G and sets its value to the principal character.

void dirichlet_char_clear(dirichlet_char_t chi)
    Clears chi.

void dirichlet_char_print(const dirichlet_group_t G, const dirichlet_char_t chi)
    Prints the array of exponents representing this character.

void dirichlet_char_log(dirichlet_char_t x, const dirichlet_group_t G, ulong m)
    Sets x to the character of number $m$, computing its log using discrete logarithm in G.

ulong dirichlet_char_exp(const dirichlet_group_t G, const dirichlet_char_t x)
    Returns the number $m$ corresponding to exponents in $x$.

ulong _dirichlet_char_exp(dirichlet_char_t x, const dirichlet_group_t G)
    Computes and returns the number $m$ corresponding to exponents in $x$. This function is for internal use.

void dirichlet_char_one(dirichlet_char_t x, const dirichlet_group_t G)
    Sets x to the principal character in G, having log $[0,\ldots0]$.

void dirichlet_char_first_primitive(dirichlet_char_t x, const dirichlet_group_t G)
    Sets x to the first primitive character of G, having log $[1,\ldots1]$, or $[0,1,\ldots1]$ if $8 | q$.

void dirichlet_char_set(dirichlet_char_t x, const dirichlet_group_t G, const dirichlet_char_t y)
    Sets x to the element y.

int dirichlet_char_next(dirichlet_char_t x, const dirichlet_group_t G)
    Sets x to the next character in G according to lexicographic ordering of log.

    The return value is the index of the last updated exponent of x, or -1 if the last element has been reached.

    This function allows to iterate on all elements of G looping on their log. Note that it produces elements in seemingly random number order.
The following template can be used for such a loop:

```c
dirichlet_char_one(chi, G);
  do {
    /* use character chi */
  } while (dirichlet_char_next(chi, G) >= 0);
```

```c
int dirichlet_char_next_primitive(dirichlet_char_t x, const dirichlet_group_t G)
  Same as dirichlet_char_next(), but jumps to the next primitive character of G.

ulong dirichlet_index_char(const dirichlet_group_t G, const dirichlet_char_t x)
  Returns the lexicographic index of the log of x as an integer in \(0 \ldots \varphi(q)\).

void dirichlet_char_index(dirichlet_char_t x, const dirichlet_group_t G, ulong j)
  Sets x to the character whose log has lexicographic index j.

int dirichlet_char_eq(const dirichlet_char_t x, const dirichlet_char_t y)

int dirichlet_char_eq_deep(const dirichlet_group_t G, const dirichlet_char_t x, const dirichlet_char_t y)
  Return 1 if x equals y. The second version checks every byte of the representation and is intended for testing only.
```

### 7.3.4 Character properties

As a consequence of the Conrey numbering, all these numbers are available at the level of `number` and `char` object. Both case require no discrete log computation.

```c
int dirichlet_char_is_principal(const dirichlet_group_t G, const dirichlet_char_t chi)
  Returns 1 if chi is the principal character mod q.

ulong dirichlet_conductor_ui(const dirichlet_group_t G, ulong a)
  Returns the conductor of \(\chi_q(a, \cdot)\), that is the smallest r dividing q such \(\chi_q(a, \cdot)\) can be obtained as a character mod r.

int dirichlet_conductor_char(const dirichlet_group_t G, const dirichlet_char_t x)
  Returns the conductor of \(\chi_q(a, \cdot)\), that is the smallest r dividing q such \(\chi_q(a, \cdot)\) can be obtained as a character mod r.

int dirichlet_parity_ui(const dirichlet_group_t G, ulong a)
  Returns the parity \(\lambda\) in \(\{0, 1\}\) of \(\chi_q(a, \cdot)\), such that \(\chi_q(a, -1) = (-1)^\lambda\).

ulong dirichlet_order_ui(const dirichlet_group_t G, ulong a)
  Returns the order of \(\chi_q(a, \cdot)\) which is the order of a mod q.

int dirichlet_parity_char(const dirichlet_group_t G, const dirichlet_char_t x)

ulong dirichlet_order_char(const dirichlet_group_t G, const dirichlet_char_t x)
  Returns the order of \(\chi_q(a, \cdot)\) which is the order of a mod q.

int dirichlet_char_is_real(const dirichlet_group_t G, const dirichlet_char_t chi)
  Returns 1 if chi is a real character (iff it has order \(\leq 2\)).

int dirichlet_char_is_primitive(const dirichlet_group_t G, const dirichlet_char_t chi)
  Returns 1 if chi is primitive (iff its conductor is exactly q).
7.3.5 Character evaluation

Dirichlet characters take value in a finite cyclic group of roots of unity plus zero.

Evaluation functions return a `ulong`, this number corresponds to the power of a primitive root of unity, the special value `DIRICHLET_CHI_NULL` encoding the zero value.

```c
ulong dirichlet_pairing(const dirichlet_group_t G, ulong m, ulong n)
ulong dirichlet_pairing_char(const dirichlet_group_t G, const dirichlet_char_t chi, const dirichlet_char_t psi)
```

Compute the value of the Dirichlet pairing on numbers `m` and `n`, as exponent modulo `G->expo`.

The `char` variant takes as input two characters, so that no discrete logarithm is computed.

The returned value is the numerator of the actual value exponent mod the group exponent `G->expo`.

```c
ulong dirichlet_chi(const dirichlet_group_t G, const dirichlet_char_t chi, ulong n)
```

Compute the value $\chi(n)$ as the exponent modulo `G->expo`.

```c
void dirichlet_chi_vec(ulong *v, const dirichlet_group_t G, const dirichlet_char_t chi, slong nv)
void dirichlet_chi_vec_order(ulong *v, const dirichlet_group_t G, const dirichlet_char_t chi, ulong order, slong nv)
```

Compute the list of exponent values $v[k]$ for $0 \leq k < nv$, as exponents modulo `G->expo`.

7.3.6 Character operations

```c
void dirichlet_char_mul(dirichlet_char_t chi2, const dirichlet_group_t G, const dirichlet_char_t chi1, const dirichlet_char_t chi2)
void dirichlet_char_pow(dirichlet_char_t c, const dirichlet_group_t G, const dirichlet_char_t a, ulong n)
void dirichlet_char_lift(dirichlet_char_t chi_G, const dirichlet_group_t G, const dirichlet_char_t chi_H, const dirichlet_group_t H)
void dirichlet_char_lower(dirichlet_char_t chi_H, const dirichlet_group_t H, const dirichlet_char_t chi_G, const dirichlet_group_t G)
```

If `H` is a subgroup of `G`, computes the character in `G` corresponding to `chi_H` in `H`.

If `chi_G` is a character of `G` which factors through `H`, sets `chi_H` to the corresponding restriction in `H`.

This requires $c(\chi_G) \mid q_H \mid q_G$, where $c(\chi_G)$ is the conductor of $\chi_G$ and $q_G, q_H$ are the moduli of `G` and `H`. 
7.4 dlog.h – discrete logarithms mod ulong primes

This module implements discrete logarithms, with the application to Dirichlet characters in mind.

In particular, this module defines a `dlog_precomp_t` structure permitting to describe a discrete log problem in some subgroup of $(\mathbb{Z}/p^e\mathbb{Z})^\times$ for primepower moduli $p^e$, and store precomputed data for faster computation of several such discrete logarithms.

When initializing this data, the user provides both a group description and the expected number of subsequent discrete logarithms calls. The choice of algorithm and the amount of stored data depend both on the structure of the group and this number.

No particular effort has been made towards single discrete logarithm computation. Currently only machine size primepower moduli are supported.

7.4.1 Types, macros and constants

DLOG_NONE

Return value when the discrete logarithm does not exist

type `dlog_precomp_struct`

type `dlog_precomp_t`

Structure for discrete logarithm precomputed data.

A `dlog_precomp_t` is defined as an array of length one of type `dlog_precomp_struct`, permitting a `dlog_precomp_t` to be passed by reference.

7.4.2 Single evaluation

ulong `dlog_once(ulong b, ulong a, const nmod_t mod, ulong n)`

Return $x$ such that $b = a^x$ in $(\mathbb{Z}/mod\mathbb{Z})^\times$, where $a$ is known to have order $n$.

7.4.3 Precomputations

void `dlog_precomp_n_init(dlog_precomp_t pre, ulong a, ulong mod, ulong n, ulong num)`

Precompute data for $num$ discrete logarithms evaluations in the subgroup generated by $a$ modulo $mod$, where $a$ is known to have order $n$.

ulong `dlog_precomp(const dlog_precomp_t pre, ulong b)`

Return $\log(b)$ for the group described in `pre`.

void `dlog_precomp_clear(dlog_precomp_t pre)`

Clears $t$.

Specialized versions of `dlog_precomp_n_init()` are available when specific information is known about the group:

void `dlog_precomp_modpe_init(dlog_precomp_t pre, ulong a, ulong p, ulong e, ulong pe, ulong num)`

Assume that $a$ generates the group of residues modulo $pe$ equal $p^e$ for prime $p$.

void `dlog_precomp_p_init(dlog_precomp_t pre, ulong a, ulong mod, ulong p, ulong num)`

Assume that $a$ has prime order $p$.

void `dlog_precomp_pe_init(dlog_precomp_t pre, ulong a, ulong mod, ulong p, ulong e, ulong pe, ulong num)`

Assume that $a$ has primepower order $pe p^e$. 

7.4. dlog.h – discrete logarithms mod ulong primes
Make a complete lookup table of size $n$. If $\text{mod}$ is small, this is done using an element-indexed array (see $dlog_table_t$), otherwise with a sorted array allowing binary search.

### 7.4.4 Vector evaluations

These functions compute all logarithms of successive integers $1 \ldots n$.

- **$dlog_vec_fill(ulong *v, ulong nv, ulong x)$**
  
  Sets values $v[k]$ to $x$ for all $k$ less than $nv$.

- **$dlog_vec_set_not_found(ulong *v, ulong nv, nmod_t mod)$**
  
  Sets values $v[k]$ to $DLOG_NONE$ for all $k$ not coprime to $\text{mod}$.

- **$dlog_vec(ulong *v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)$**
  
  Sets $v[k]$ to $\log(k, a)$ times value $va$ for $0 \leq k < nv$, where $a$ has order $na$. $va$ should be 1 for usual log computation.

- **$dlog_vec_add(ulong *v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)$**
  
  Same parameters as before, but adds $\log(k, a) \times v_a$ to $v[k]$ and reduce modulo $\text{order}$ instead of replacing the value. Indices $k$ such that $v[k]$ equals $DLOG_NONE$ are ignored.

Depending on the relative size of $nv$ and $na$, these two $dlog_vec$ functions call one of the following functions.

- **$dlog_vec_loop(ulong *v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)$**

  Perform a complete loop of size $na$ on powers of $a$ to fill the logarithm values, discarding powers outside the bounds of $v$. This requires no discrete logarithm computation.

- **$dlog_vec_loop_add(ulong *v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)$**

  Compute discrete logarithms of prime numbers less than $nv$ and propagate to composite numbers.

- **$dlog_vec_sieve_add(ulong *v, ulong nv, ulong a, ulong va, nmod_t mod, ulong na, nmod_t order)$**

  Compute the discrete logarithms of the first few prime numbers, then use them as a factor base to obtain the logarithms of larger primes by sieving techniques.

In the present implementation, the full index-calculus method is not implemented.

### 7.4.5 Internal discrete logarithm strategies

Several discrete logarithms strategies are implemented:

- Complete lookup table for small groups.
- Baby-step giant-step table.

combined with mathematical reductions:

- Pohlig-Hellman decomposition (Chinese remainder decomposition on the order of the group and base $p$ decomposition for primepower order).
- $p$-adic log for primepower modulus $p^e$.

The $dlog_precomp$ structure makes recursive use of the following method-specific structures.
**Complete table**

**Type** \texttt{dlog\_table\_struct}  
**Type** \texttt{dlog\_table\_t}  
Structure for complete lookup table.

\texttt{ulong dlog\_table\_init(dlog\_table\_t t, ulong a, ulong mod)}  
Initialize a table of powers of \(a\) modulo \(mod\), storing all elements in an array of size \(mod\).

\texttt{void dlog\_table\_clear(dlog\_table\_t t)}  
Clears \(t\).

\texttt{ulong dlog\_table(const dlog\_table\_t t, ulong b)}  
Return \(\log(b, a)\) using the precomputed data \(t\).

**Baby-step giant-step table**

**Type** \texttt{dlog\_bsgs\_struct}  
**Type** \texttt{dlog\_bsgs\_t}  
Structure for Baby-Step Giant-Step decomposition.

\texttt{ulong dlog\_bsgs\_init(dlog\_bsgs\_t t, ulong a, ulong mod, ulong n, ulong m)}  
Initialize \(t\) and store the first \(m\) powers of \(a\) in a sorted array. The return value is a rough measure of the cost of each logarithm using this table. The user should take \(m \approx \sqrt{kn}\) to compute \(k\) logarithms in a group of size \(n\).

\texttt{void dlog\_bsgs\_clear(dlog\_bsgs\_t t)}  
Clears \(t\).

\texttt{ulong dlog\_bsgs(const dlog\_bsgs\_t t, ulong b)}  
Return \(\log(b, a)\) using the precomputed data \(t\).

**Prime-power modulus decomposition**

**Type** \texttt{dlog\_modpe\_struct}  
**Type** \texttt{dlog\_modpe\_t}  
Structure for discrete logarithm modulo primepower \(p^e\).

A \texttt{dlog\_modpe\_t} is defined as an array of length one of type \texttt{dlog\_modpe\_struct}, permitting a \texttt{dlog\_modpe\_t} to be passed by reference.

\texttt{ulong dlog\_modpe\_init(dlog\_modpe\_t t, ulong a, ulong p, ulong e, ulong pe, ulong num)}  

\texttt{void dlog\_modpe\_clear(dlog\_modpe\_t t)}  
Clears \(t\).

\texttt{ulong dlog\_modpe(const dlog\_modpe\_t t, ulong b)}  
Return \(\log(b, a)\) using the precomputed data \(t\).
CRT decomposition

**type** `dlog_crt_struct`

**type** `dlog_crt_t`

Structure for discrete logarithm for groups of composite order. A `dlog_crt_t` is defined as an array of length one of type `dlog_crt_struct`, permitting a `dlog_crt_t` to be passed by reference.

```c
ulong dlog_crt_init(dlog_crt_t t, ulong a, ulong mod, ulong n, ulong num)
```

Precompute data for `num` evaluations of discrete logarithms in base `a` modulo `mod`, where `a` has composite order `n`, using chinese remainder decomposition.

```c
void dlog_crt_clear(dlog_crt_t t)
```

Clears `t`.

```c
ulong dlog_crt(const dlog_crt_t t, ulong b)
```

Return $\log(b, a)$ using the precomputed data `t`.

**padic decomposition**

**type** `dlog_power_struct`

**type** `dlog_power_t`

Structure for discrete logarithm for groups of primepower order. A `dlog_power_t` is defined as an array of length one of type `dlog_power_struct`, permitting a `dlog_power_t` to be passed by reference.

```c
ulong dlog_power_init(dlog_power_t t, ulong a, ulong mod, ulong p, ulong e, ulong num)
```

Precompute data for `num` evaluations of discrete logarithms in base `a` modulo `mod`, where `a` has prime power order $p^e$ equals $p^e$, using decomposition in base `p`.

```c
void dlog_power_clear(dlog_power_t t)
```

Clears `t`.

```c
ulong dlog_power(const dlog_power_t t, ulong b)
```

Return $\log(b, a)$ using the precomputed data `t`.

**Pollard rho method**

**type** `dlog_rho_struct`

**type** `dlog_rho_t`

Structure for discrete logarithm using Pollard rho. A `dlog_rho_t` is defined as an array of length one of type `dlog_rho_struct`, permitting a `dlog_rho_t` to be passed by reference.

```c
void dlog_rho_init(dlog_rho_t t, ulong a, ulong mod, ulong n)
```

Initialize random walks for evaluations of discrete logarithms in base `a` modulo `mod`, where `a` has order `n`.

```c
void dlog_rho_clear(dlog_rho_t t)
```

Clears `t`.

```c
ulong dlog_rho(const dlog_rho_t t, ulong b)
```

Return $\log(b, a)$ by the rho method in the group described by `t`. 
7.5 bool_mat.h – matrices over booleans

A bool_mat_t represents a dense matrix over the boolean semiring \( \{0, 1\}, \lor, \land \) implemented as an array of entries of type int.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

7.5.1 Types, macros and constants

type bool_mat_struct

type bool_mat_t

Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An bool_mat_t is defined as an array of length one of type bool_mat_struct, permitting an bool_mat_t to be passed by reference.

int bool_mat_get_entry(const bool_mat_t mat, slong i, slong j)

Returns the entry of matrix mat at row i and column j.

void bool_mat_set_entry(bool_mat_t mat, slong i, slong j, int x)

Sets the entry of matrix mat at row i and column j to x.

bool_mat_nrows(mat)

Returns the number of rows of the matrix.

bool_mat_ncols(mat)

Returns the number of columns of the matrix.

7.5.2 Memory management

void bool_mat_init(bool_mat_t mat, slong r, slong c)

Initializes the matrix, setting it to the zero matrix with r rows and c columns.

void bool_mat_clear(bool_mat_t mat)

Clears the matrix, deallocating all entries.

int bool_mat_is_empty(const bool_mat_t mat)

Returns nonzero iff the number of rows or the number of columns in mat is zero. Note that this does not depend on the entry values of mat.

int bool_mat_is_square(const bool_mat_t mat)

Returns nonzero iff the number of rows is equal to the number of columns in mat.

7.5.3 Conversions

void bool_mat_set(bool_mat_t dest, const bool_mat_t src)

Sets dest to src. The operands must have identical dimensions.
### 7.5.4 Input and output

```c
void bool_mat_print(const bool_mat_t mat)
    Prints each entry in the matrix.

void bool_mat_fprint(FILE *file, const bool_mat_t mat)
    Prints each entry in the matrix to the stream file.
```

### 7.5.5 Value comparisons

```c
int bool_mat_equal(const bool_mat_t mat1, const bool_mat_t mat2)
    Returns nonzero iff the matrices have the same dimensions and identical entries.

int bool_mat_any(const bool_mat_t mat)
    Returns nonzero iff mat has a nonzero entry.

int bool_mat_all(const bool_mat_t mat)
    Returns nonzero iff all entries of mat are nonzero.

int bool_mat_is_diagonal(const bool_mat_t A)
    Returns nonzero iff $i \neq j \Rightarrow A_{ij}$.  

int bool_mat_is_lower_triangular(const bool_mat_t A)
    Returns nonzero iff $i < j \Rightarrow A_{ij}$.

int bool_mat_is_transitive(const bool_mat_t mat)
    Returns nonzero iff $A_{ij} \land A_{jk} \Rightarrow A_{ik}$.

int bool_mat_is_nilpotent(const bool_mat_t A)
    Returns nonzero iff some positive matrix power of $A$ is zero.
```

### 7.5.6 Random generation

```c
void bool_mat_randtest(bool_mat_t mat, flint_rand_t state)
    Sets mat to a random matrix.

void bool_mat_randtest_diagonal(bool_mat_t mat, flint_rand_t state)
    Sets mat to a random diagonal matrix.

void bool_mat_randtest_nilpotent(bool_mat_t mat, flint_rand_t state)
    Sets mat to a random nilpotent matrix.
```

### 7.5.7 Special matrices

```c
void bool_mat_zero(bool_mat_t mat)
    Sets all entries in mat to zero.

void bool_mat_one(bool_mat_t mat)
    Sets the entries on the main diagonal to ones, and all other entries to zero.

void bool_mat_directed_path(bool_mat_t A)
    Sets $A_{ij}$ to $j = i + 1$. Requires that $A$ is a square matrix.

void bool_mat_directed_cycle(bool_mat_t A)
    Sets $A_{ij}$ to $j = (i + 1) \mod n$ where $n$ is the order of the square matrix $A$.
```
### 7.5.8 Transpose

```c
void bool_mat_transpose(bool_mat_t dest, const bool_mat_t src)
```

Sets `dest` to the transpose of `src`. The operands must have compatible dimensions. Aliasing is allowed.

### 7.5.9 Arithmetic

```c
void bool_mat_complement(bool_mat_t B, const bool_mat_t A)
```

Sets `B` to the logical complement of `A`. That is, $B_{ij}$ is set to $\overline{A_{ij}}$. The operands must have the same dimensions.

```c
void bool_mat_add(bool_mat_t res, const bool_mat_t mat1, const bool_mat_t mat2)
```

Sets `res` to the sum of `mat1` and `mat2`. The operands must have the same dimensions.

```c
void bool_mat_mul(bool_mat_t res, const bool_mat_t mat1, const bool_mat_t mat2)
```

Sets `res` to the matrix product of `mat1` and `mat2`. The operands must have compatible dimensions for matrix multiplication.

```c
void bool_mat_mul_entrywise(bool_mat_t res, const bool_mat_t mat1, const bool_mat_t mat2)
```

Sets `res` to the entrywise product of `mat1` and `mat2`. The operands must have the same dimensions.

```c
void bool_mat_sqr(bool_mat_t B, const bool_mat_t A)
```

Sets `B` to the matrix square of `A`. The operands must both be square with the same dimensions.

```c
void bool_mat_pow_ui(bool_mat_t B, const bool_mat_t A, ulong exp)
```

Sets `B` to $A$ raised to the power `exp`. Requires that `A` is a square matrix.

### 7.5.10 Special functions

```c
int bool_mat_trace(const bool_mat_t mat)
```

Returns the trace of the matrix, i.e. the sum of entries on the main diagonal of `mat`. The matrix is required to be square. The sum is in the boolean semiring, so this function returns nonzero iff any entry on the diagonal of `mat` is nonzero.

```c
slong bool_mat_nilpotency_degree(const bool_mat_t A)
```

Returns the nilpotency degree of the $n \times n$ matrix $A$. It returns the smallest positive $k$ such that $A^k = 0$. If no such $k$ exists then the function returns $-1$ if $n$ is positive, and otherwise it returns 0.

```c
void bool_mat_transitive_closure(bool_mat_t B, const bool_mat_t A)
```

Sets `B` to the transitive closure $\sum_{k=1}^{\infty} A^k$. The matrix `A` is required to be square.

```c
slong bool_mat_get_strongly_connected_components(slong *p, const bool_mat_t A)
```

Partitions the $n$ row and column indices of the $n \times n$ matrix `A` according to the strongly connected components (SCC) of the graph for which `A` is the adjacency matrix. If the graph has $k$ SCCs then the function returns $k$, and for each vertex $i \in [0,n - 1]$, $p_i$ is set to the index of the SCC to which the vertex belongs. The SCCs themselves can be considered as nodes in a directed acyclic graph (DAG), and the SCCs are indexed in postorder with respect to that DAG.

```c
slong bool_mat_all_pairs_longest_walk(fmpz_mat_t B, const bool_mat_t A)
```

Sets $B_{ij}$ to the length of the longest walk with endpoint vertices $i$ and $j$ in the graph whose adjacency matrix is `A`. The matrix `A` must be square. Empty walks with zero length which begin and end at the same vertex are allowed. If $j$ is not reachable from $i$ then no walk from $i$ to $j$ exists and $B_{ij}$ is set to the special value $-1$. If arbitrarily long walks from $i$ to $j$ exist then $B_{ij}$ is set to the special value $-2$. 

## 7.5. `bool_mat.h` – matrices over booleans

469
The function returns \(-2\) if any entry of \(B_{ij}\) is \(-2\), and otherwise it returns the maximum entry in \(B\), except if \(A\) is empty in which case \(-1\) is returned. Note that the returned value is one less than that of \texttt{nilpotency\_degree()}.

This function can help quantify entrywise errors in a truncated evaluation of a matrix power series. If \(A\) is an indicator matrix with the same sparsity pattern as a matrix \(M\) over the real or complex numbers, and if \(B_{ij}\) does not take the special value \(-2\), then the tail \([\sum_{k=N}^{\infty} a_k M^k]_{ij}\) vanishes when \(N > B_{ij}\).
8.1 nf.h – number fields

type _nf_struct_

type _nf_t_
    Represents a number field.

void _nf_init(nf_t nf, const _fmpq_poly_t pol)_
    Perform basic initialisation of a number field (for element arithmetic) given a defining polynomial over $\mathbb{Q}$.

void _nf_clear(nf_t nf)_
    Release resources used by a number field object. The object will need initialisation again before it can be used.

8.2 nf_elem.h – number field elements

Authors:

• William Hart

8.2.1 Initialisation

type _nf_elem_struct_

type _nf_elem_t_
    Represents a number field element.

void _nf_elem_init(nf_elem_t a, const nf_t nf)_
    Initialise a number field element to belong to the given number field nf. The element is set to zero.

void _nf_elem_clear(nf_elem_t a, const nf_t nf)_
    Clear resources allocated by the given number field element in the given number field.

void _nf_elem_randtest(nf_elem_t a, flint_rand_t state, mp_bitcnt_t bits, const nf_t nf)_
    Generate a random number field element $a$ in the number field $nf$ whose coefficients have up to the given number of bits.

void _nf_elem_canonicalise(nf_elem_t a, const nf_t nf)_
    Canonicalise a number field element, i.e. reduce numerator and denominator to lowest terms. If the numerator is 0, set the denominator to 1.
void _nf_elem_reduce(nf_elem_t a, const nf_t nf)
Reduce a number field element modulo the defining polynomial. This is used with functions such as nf_elem_mul_red which allow reduction to be delayed. Does not canonicalise.

void nf_elem_reduce(nf_elem_t a, const nf_t nf)
Reduce a number field element modulo the defining polynomial. This is used with functions such as nf_elem_mul_red which allow reduction to be delayed.

int _nf_elem_invertible_check(nf_elem_t a, const nf_t nf)
Whilst the defining polynomial for a number field should by definition be irreducible, it is not enforced. Thus in test code, it is convenient to be able to check that a given number field element is invertible modulo the defining polynomial of the number field. This function does precisely this.
If \(a\) is invertible modulo the defining polynomial of \(nf\) the value 1 is returned, otherwise 0 is returned.
The function is only intended to be used in test code.

8.2.2 Conversion

void nf_elem_set_fmpz_mat_row(nf_elem_t b, const fmpz_mat_t M, const slong i, fmpz_t den, const nf_t nf)
Set \(b\) to the element specified by row \(i\) of the matrix \(M\) and with the given denominator \(d\). Column 0 of the matrix corresponds to the constant coefficient of the number field element.

void nf_elem_get_fmpz_mat_row(fmpz_mat_t M, const slong i, fmpz_t den, const nf_elem_t b, const nf_t nf)
Set the row \(i\) of the matrix \(M\) to the coefficients of the numerator of the element \(b\) and \(d\) to the denominator of \(b\). Column 0 of the matrix corresponds to the constant coefficient of the number field element.

void nf_elem_set_fmpq_poly(nf_elem_t a, const fmpq_poly_t pol, const nf_t nf)
Set \(a\) to the element corresponding to the polynomial \(pol\).

void nf_elem_get_fmpq_poly(fmpq_poly_t pol, const nf_elem_t a, const nf_t nf)
Set \(pol\) to a polynomial corresponding to \(a\), reduced modulo the defining polynomial of \(nf\).

void nf_elem_get_nmod_poly_den(nmod_poly_t pol, const nf_elem_t a, const nf_t nf, int den)
Set \(pol\) to the reduction of the polynomial corresponding to the numerator of \(a\). If \(den == 1\), the result is multiplied by the inverse of the denominator of \(a\). In this case it is assumed that the reduction of the denominator of \(a\) is invertible.

void nf_elem_get_nmod_poly(nmod_poly_t pol, const nf_elem_t a, const nf_t nf)
Set \(pol\) to the reduction of the polynomial corresponding to the numerator of \(a\). The result is multiplied by the inverse of the denominator of \(a\). It is assumed that the reduction of the denominator of \(a\) is invertible.

void nf_elem_get_fmpz_mod_poly_den(fmpz_mod_poly_t pol, const nf_elem_t a, const nf_t nf, int den, const fmpz_mod_ctx_t ctx)
Set \(pol\) to the reduction of the polynomial corresponding to the numerator of \(a\). If \(den == 1\), the result is multiplied by the inverse of the denominator of \(a\). In this case it is assumed that the reduction of the denominator of \(a\) is invertible.

void nf_elem_get_fmpz_mod_poly(fmpz_mod_poly_t pol, const nf_elem_t a, const nf_t nf, const fmpz_mod_ctx_t ctx)
Set \(pol\) to the reduction of the polynomial corresponding to the numerator of \(a\). The result is multiplied by the inverse of the denominator of \(a\). It is assumed that the reduction of the denominator of \(a\) is invertible.
8.2.3 Basic manipulation

```c
void nf_elem_set_den(nf_elem_t b, fmpz_t d, const nf_t nf)
    Set the denominator of the nf_elem_t b to the given integer d. Assumes d > 0.

void nf_elem_get_den(fmpz_t d, const nf_elem_t b, const nf_t nf)
    Set d to the denominator of the nf_elem_t b.

void _nf_elem_set_coeff_num_fmpz(nf_elem_t a, slong i, const fmpz_t d, const nf_t nf)
    Set the i-th coefficient of the denominator of a to the given integer d.
```

8.2.4 Comparison

```c
int _nf_elem_equal(const nf_elem_t a, const nf_elem_t b, const nf_t nf)
    Return 1 if the given number field elements are equal in the given number field nf. This function does not assume a and b are canonicalised.

int nf_elem_equal(const nf_elem_t a, const nf_elem_t b, const nf_t nf)
    Return 1 if the given number field elements are equal in the given number field nf. This function assumes a and b are canonicalised.

int nf_elem_is_zero(const nf_elem_t a, const nf_t nf)
    Return 1 if the given number field element is equal to zero, otherwise return 0.

int nf_elem_is_one(const nf_elem_t a, const nf_t nf)
    Return 1 if the given number field element is equal to one, otherwise return 0.
```

8.2.5 I/O

```c
void nf_elem_print_pretty(const nf_elem_t a, const nf_t nf, const char *var)
    Print the given number field element to stdout using the null-terminated string var not equal to "\0" as the name of the primitive element.
```

8.2.6 Arithmetic

```c
void nf_elem_zero(nf_elem_t a, const nf_t nf)
    Set the given number field element to zero.

void nf_elem_one(nf_elem_t a, const nf_t nf)
    Set the given number field element to one.

void nf_elem_set(nf_elem_t a, const nf_elem_t b, const nf_t nf)
    Set the number field element a to equal the number field element b, i.e. set a = b.

void nf_elem_neg(nf_elem_t a, const nf_elem_t b, const nf_t nf)
    Set the number field element a to minus the number field element b, i.e. set a = −b.

void nf_elem_swap(nf_elem_t a, nf_elem_t b, const nf_t nf)
    Efficiently swap the two number field elements a and b.

void nf_elem_mul_gen(nf_elem_t a, const nf_elem_t b, const nf_t nf)
    Multiply the element b with the generator of the number field.

void _nf_elem_add(nf_elem_t r, const nf_elem_t a, const nf_elem_t b, const nf_t nf)
    Add two elements of a number field nf, i.e. set r = a + b. Canonicalisation is not performed.
```
void **nf_elem_add\(nf\_elem\_t r, const nf\_elem\_t a, const nf\_elem\_t b, const nf\_t nf\)  
Add two elements of a number field \(nf\), i.e. set \(r = a + b\).

void **nf_elem_sub\(nf\_elem\_t r, const nf\_elem\_t a, const nf\_elem\_t b, const nf\_t nf\)  
Subtract two elements of a number field \(nf\), i.e. set \(r = a - b\). Canonicalisation is not performed.

void **nf_elem_sub\(nf\_elem\_t r, const nf\_elem\_t a, const nf\_elem\_t b, const nf\_t nf\)  
Subtract two elements of a number field \(nf\), i.e. set \(r = a - b\).

void **nf_elem_mul\(nf\_elem\_t a, const nf\_elem\_t b, const nf\_elem\_t c, const nf\_t nf\)  
Multiply two elements of a number field \(nf\), i.e. set \(r = a \cdot b\). Does not canonicalise. Aliasing of inputs with output is not supported.

void **nf_elem_mul\(nf\_elem\_t a, const nf\_elem\_t b, const nf\_elem\_t c, const nf\_t nf\)  
Multiply two elements of a number field \(nf\), i.e. set \(r = a \cdot b\).

void **nf_elem_add\(nf\_elem\_t a, const nf\_elem\_t b, const nf\_elem\_t c, const nf\_t nf\)  
Multiply two elements of a number field \(nf\), i.e. set \(r = a \cdot b\).

void **nf_elem_pow\(nf\_elem\_t a, const nf\_elem\_t b, const nf\_elem\_t c, const nf\_t nf\)  
As per \_nf_elem_mul, but reduction modulo the defining polynomial of the number field is only carried out if \(\text{red} == 1\). Assumes both inputs are reduced.

void **nf_elem_pow\(nf\_elem\_t a, const nf\_elem\_t b, const nf\_elem\_t c, const nf\_t nf\)  
As per \_nf_elem_mul, but reduction modulo the defining polynomial of the number field is only carried out if \(\text{red} == 1\). Assumes both inputs are reduced.

void **nf_elem_inv\(nf\_elem\_t r, const nf\_elem\_t a, const nf\_t nf\)  
Invert an element of a number field \(nf\), i.e. set \(r = a^{-1}\). Aliasing of the input with the output is not supported.

void **nf_elem_inv\(nf\_elem\_t r, const nf\_elem\_t a, const nf\_t nf\)  
Invert an element of a number field \(nf\), i.e. set \(r = a^{-1}\).

void **nf_elem_div\(nf\_elem\_t a, const nf\_elem\_t b, const nf\_elem\_t c, const nf\_t nf\)  
Set \(a\) to \(b/c\) in the given number field. Aliasing of \(a\) and \(b\) is not permitted.

void **nf_elem_div\(nf\_elem\_t a, const nf\_elem\_t b, const nf\_elem\_t c, const nf\_t nf\)  
Set \(a\) to \(b/c\) in the given number field.

void **nf_elem_pow\(nf\_elem\_t res, const nf\_elem\_t a, ulong e, const nf\_t nf\)  
Set \(res\) to \(a^e\) using left-to-right binary exponentation as described on p. 461 of [Knu97].

void **nf_elem_pow\(nf\_elem\_t res, const nf\_elem\_t a, ulong e, const nf\_t nf\)  
Set \(res\) to \(a^e\) using the binary exponentiation algorithm. If \(e\) is zero, returns one, so that in particular \(0^0 = 1\).

void **nf_elem_norm\(fmpz\_t rnum, fmpz\_t rden, const nf\_elem\_t a, const nf\_t nf\)  
Set \(rnum, rden\) to the absolute norm of the given number field element \(a\).

void **nf_elem_norm\(fmpz\_t res, const nf\_elem\_t a, const nf\_t nf\)  
Set \(res\) to the absolute norm of the given number field element \(a\).

void **nf_elem_norm\(fmpz\_t res, const nf\_elem\_t a, const nf\_t nf, const fmpz\_t div, slong nbits\)  
Set \(res\) to the absolute norm of the given number field element \(a\), divided by \(\text{div}\). Assumes the result to be an integer and having at most \(nbits\) bits.

void **nf_elem_norm\(fmpz\_t res, const nf\_elem\_t a, const nf\_t nf, const fmpz\_t divisor, slong nbits\)  
Set \(rnum, rden\) to the absolute norm of the given number field element \(a\), divided by \(\text{div}\). Assumes the result to be an integer and having at most \(nbits\) bits.

void **nf_elem_trace\(fmpz\_t rnum, fmpz\_t rden, const nf\_elem\_t a, const nf\_t nf\)  
Set \(rnum, rden\) to the absolute trace of the given number field element \(a\).
void \texttt{nf\_elem\_trace}(\texttt{fmpq\_t res, const nf\_elem\_t a, const nf\_t nf})

Set \texttt{res} to the absolute trace of the given number field element \(a\).

### 8.2.7 Representation matrix

void \texttt{nf\_elem\_rep\_mat}(\texttt{fmpq\_mat\_t res, const nf\_elem\_t a, const nf\_t nf})

Set \texttt{res} to the matrix representing the multiplication with \(a\) with respect to the basis \(1, a, \ldots, a^{d-1}\), where \(a\) is the generator of the number field of \(d\) is its degree.

void \texttt{nf\_elem\_rep\_mat\_fmpz\_mat\_den}(\texttt{fmpz\_mat\_t res, fmpz\_t den, const nf\_elem\_t a, const nf\_t nf})

Return a tuple \(M, d\) such that \(M/d\) is the matrix representing the multiplication with \(a\) with respect to the basis \(1, a, \ldots, a^{d-1}\), where \(a\) is the generator of the number field of \(d\) is its degree. The integral matrix \(M\) is primitive.

### 8.2.8 Modular reduction

void \texttt{nf\_elem\_mod\_fmpz\_den}(\texttt{nf\_elem\_t z, const nf\_elem\_t a, const fmpz\_t mod, const nf\_t nf, int den})

If \texttt{den} == 0, return an element \(z\) with denominator 1, such that the coefficients of \(z - da\) are divisible by \texttt{mod}, where \(d\) is the denominator of \(a\). The coefficients of \(z\) are reduced modulo \texttt{mod}.

If \texttt{den} == 1, return an element \(z\), such that \(z - a\) has denominator 1 and the coefficients of \(z - a\) are divisible by \texttt{mod}. The coefficients of \(z\) are reduced modulo \(\texttt{mod} \cdot d\), where \(d\) is the denominator of \(a\).

Reduction takes place with respect to the positive residue system.

void \texttt{nf\_elem\_smod\_fmpz\_den}(\texttt{nf\_elem\_t z, const nf\_elem\_t a, const fmpz\_t mod, const nf\_t nf, int den})

If \texttt{den} == 0, return an element \(z\) with denominator 1, such that the coefficients of \(z - da\) are divisible by \texttt{mod}, where \(d\) is the denominator of \(a\). The coefficients of \(z\) are reduced modulo \texttt{mod}.

If \texttt{den} == 1, return an element \(z\), such that \(z - a\) has denominator 1 and the coefficients of \(z - a\) are divisible by \texttt{mod}. The coefficients of \(z\) are reduced modulo \(\texttt{mod} \cdot d\), where \(d\) is the denominator of \(a\).

Reduction takes place with respect to the symmetric residue system.

void \texttt{nf\_elem\_mod\_fmpz}(\texttt{nf\_elem\_t res, const nf\_elem\_t a, const fmpz\_t mod, const nf\_t nf})

Return an element \(z\) such that \(z - a\) has denominator 1 and the coefficients of \(z - a\) are divisible by \texttt{mod}. The coefficients of \(z\) are reduced modulo \(\texttt{mod} \cdot d\), where \(d\) is the denominator of \(b\).

Reduction takes place with respect to the positive residue system.

void \texttt{nf\_elem\_smod\_fmpz}(\texttt{nf\_elem\_t res, const nf\_elem\_t a, const fmpz\_t mod, const nf\_t nf})

Return an element \(z\) such that \(z - a\) has denominator 1 and the coefficients of \(z - a\) are divisible by \texttt{mod}. The coefficients of \(z\) are reduced modulo \(\texttt{mod} \cdot d\), where \(d\) is the denominator of \(b\).

Reduction takes place with respect to the symmetric residue system.

void \texttt{nf\_elem\_coprime\_den}(\texttt{nf\_elem\_t res, const nf\_elem\_t a, const fmpz\_t mod, const nf\_t nf})

Return an element \(z\) such that the denominator of \(z - a\) is coprime to \texttt{mod}.

Reduction takes place with respect to the positive residue system.

void \texttt{nf\_elem\_coprime\_den\_signed}(\texttt{nf\_elem\_t res, const nf\_elem\_t a, const fmpz\_t mod, const nf\_t nf})

Return an element \(z\) such that the denominator of \(z - a\) is coprime to \texttt{mod}.

Reduction takes place with respect to the symmetric residue system.

8.2. \texttt{nf\_elem.h} – number field elements
8.3 fmpzi.h – Gaussian integers

This module allows working with elements of the ring \( \mathbb{Z}[i] \). At present, only a minimal interface is provided.

8.3.1 Types, macros and constants

type fmpzi_struct

type fmpzi_t
  Contains a pairs of integers representing the real and imaginary parts. An \( \text{fmpzi}_t \) is defined as an array of length one of type \( \text{fmpzi}_\text{struct} \), permitting an \( \text{fmpzi}_t \) to be passed by reference.

\text{fmpzi_realref}(x)
  Macro giving a pointer to the real part of \( x \).

\text{fmpzi_imagref}(x)
  Macro giving a pointer to the imaginary part of \( x \).

8.3.2 Basic manipulation

void \text{fmpzi_init}(\text{fmpzi}_t x)

void \text{fmpzi_clear}(\text{fmpzi}_t x)

void \text{fmpzi_swap}(\text{fmpzi}_t x, \text{fmpzi}_t y)

void \text{fmpzi_zero}(\text{fmpzi}_t x)

void \text{fmpzi_one}(\text{fmpzi}_t x)

void \text{fmpzi_set}(\text{fmpzi}_t res, const \text{fmpzi}_t x)

void \text{fmpzi_set_si_si}(\text{fmpzi}_t res, \text{slong} a, \text{slong} b)

8.3.3 Input and output

void \text{fmpzi_print}(\text{const fmpzi}_t x)

8.3.4 Random number generation

void \text{fmpzi_randtest}(\text{fmpzi}_t res, \text{flint_rand}_t state, mp\_bitcnt_t bits)

8.3.5 Properties

int \text{fmpzi_equal}(\text{const fmpzi}_t x, \text{const fmpzi}_t y)

int \text{fmpzi_is_zero}(\text{const fmpzi}_t x)

int \text{fmpzi_is_one}(\text{const fmpzi}_t x)
8.3.6 Units

int fmpzi_is_unit(const fmpzi_t x)

slong fmpzi_canonical_unit_i_pow(const fmpzi_t x)

void fmpzi_canonicalise_unit(fmpzi_t res, const fmpzi_t x)

8.3.7 Norms

slong fmpzi_bits(const fmpzi_t x)

void fmpzi_norm(fmpz_t res, const fmpzi_t x)

8.3.8 Arithmetic

void fmpzi_conj(fmpzi_t res, const fmpzi_t x)

void fmpzi_neg(fmpzi_t res, const fmpzi_t x)

void fmpzi_add(fmpzi_t res, const fmpzi_t x, const fmpzi_t y)

void fmpzi_sub(fmpzi_t res, const fmpzi_t x, const fmpzi_t y)

void fmpzi_sqr(fmpzi_t res, const fmpzi_t x)

void fmpzi_mul(fmpzi_t res, const fmpzi_t x, const fmpzi_t y)

void fmpzi_pow_ui(fmpzi_t res, const fmpzi_t x, ulong exp)

8.3.9 Division

void fmpzi_divexact(fmpzi_t q, const fmpzi_t x, const fmpzi_t y)

Sets q to the quotient of x and y, assuming that the division is exact.

void fmpzi_divrem(fmpzi_t q, fmpzi_t r, const fmpzi_t x, const fmpzi_t y)

Computes a quotient and remainder satisfying \(x = qy + r\) with \(N(r) \leq N(y)/2\), with a canonical choice of remainder when breaking ties.

void fmpzi_divrem_approx(fmpzi_t q, fmpzi_t r, const fmpzi_t x, const fmpzi_t y)

Computes a quotient and remainder satisfying \(x = qy + r\) with \(N(r) < N(y)\), with an implementation-defined, non-canonical choice of remainder.

slong fmpzi_remove_one_plus_i(fmpzi_t res, const fmpzi_t x)

Divide \(x\) exactly by the largest possible power \((1 + i)^k\) and return the exponent \(k\).

8.3.10 GCD

void fmpzi_gcd_euclidean(fmpzi_t res, const fmpzi_t x, const fmpzi_t y)

void fmpzi_gcd_euclidean_improved(fmpzi_t res, const fmpzi_t x, const fmpzi_t y)

void fmpzi_gcd_binary(fmpzi_t res, const fmpzi_t x, const fmpzi_t y)

void fmpzi_gcd_shortest(fmpzi_t res, const fmpzi_t x, const fmpzi_t y)
void \texttt{fmpzi\_gcd}(\texttt{fmpzi\_t} \texttt{res}, \texttt{const fmpzi\_t} \texttt{x}, \texttt{const fmpzi\_t} \texttt{y})

Computes the GCD of \( x \) and \( y \). The result is in canonical unit form.

The \texttt{euclidean} version is a straightforward implementation of Euclid’s algorithm. The \texttt{euclidean\_improved} version is optimized by performing approximate divisions. The \texttt{binary} version uses a \((1+i)\)-ary analog of the binary GCD algorithm for integers [Wei2000]. The \texttt{shortest} version finds the GCD as the shortest vector in a lattice. The default version chooses an algorithm automatically.

### 8.3.11 Primality testing

\begin{verbatim}
int fmpzi\_is\_prime(const fmpzi\_t \texttt{n})
    Check whether \( n \) is a Gaussian prime.

int fmpzi\_is\_probabprime(const fmpzi\_t \texttt{n})
    Check whether \( n \) is a probable Gaussian prime.
\end{verbatim}

### 8.4 \texttt{qqbar.h} – algebraic numbers represented by minimal polynomials

A \texttt{qqbar\_t} represents a real or complex algebraic number (an element of \( \mathbb{Q} \)) by its unique reduced minimal polynomial in \( \mathbb{Z}[x] \) and an isolating complex interval. The precision of isolating intervals is maintained automatically to ensure that all operations on \texttt{qqbar\_t} instances are exact.

This representation is useful for working with individual algebraic numbers of moderate degree (up to 100, say). Arithmetic in this representation is expensive: an arithmetic operation on numbers of degrees \( m \) and \( n \) involves computing and then factoring an annihilating polynomial of degree \( mn \) and potentially also performing numerical root-finding. For doing repeated arithmetic, it is generally more efficient to work with the \texttt{ca\_t} type in a fixed number field. The \texttt{qqbar\_t} type is used internally by the \texttt{ca\_t} type to represent the embedding of number fields in \( \mathbb{R} \) or \( \mathbb{C} \) and to decide predicates for algebraic numbers.

### 8.4.1 Types and macros

\begin{verbatim}
type qqbar\_struct

type qqbar\_t
    A \texttt{qqbar\_struct} consists of an \texttt{fmpz\_poly\_struct} and an \texttt{acb\_struct}. A \texttt{qqbar\_t} is defined as an array of length one of type \texttt{qqbar\_struct}, permitting a \texttt{qqbar\_t} to be passed by reference.

type qqbar\_ptr
    Alias for \texttt{qqbar\_struct \*}, used for \texttt{qqbar} vectors.

type qqbar\_srcptr
    Alias for \texttt{const qqbar\_struct \*}, used for \texttt{qqbar} vectors when passed as readonly input to functions.

QQBAR\_POLY(x)
    Macro returning a pointer to the minimal polynomial of \( x \) which can be used as an \texttt{fmpz\_poly\_t}.

QQBAR\_COEFFS(x)
    Macro returning a pointer to the array of \texttt{fmpz} coefficients of the minimal polynomial of \( x \).

QQBAR\_ENCLOSURE(x)
    Macro returning a pointer to the enclosure of \( x \) which can be used as an \texttt{acb\_t}.
\end{verbatim}
### 8.4.2 Memory management

- **void qqbar_init(qqbar_t res)**
  - Initializes the variable res for use, and sets its value to zero.

- **void qqbar_clear(qqbar_t res)**
  - Clears the variable res, freeing or recycling its allocated memory.

- **qqbar_prt _qqbar_vec_init(slong len)**
  - Returns a pointer to an array of len initialized qbar_structs.

- **void _qqbar_vec_clear(qqbar_ptr vec, slong len)**
  - Clears all len entries in the vector vec and frees the vector itself.

### 8.4.3 Assignment

- **void qqbar_swap(qqbar_t x, qqbar_t y)**
  - Swaps the values of x and y efficiently.

- **void qqbar_set(qqbar_t res, const qqbar_t x)**

- **void qqbar_set_si(qqbar_t res, slong x)**

- **void qqbar_set_ui(qqbar_t res, ulong x)**

- **void qqbar_set_fmpz(qqbar_t res, const fmpz_t x)**

- **void qqbar_set_fmpq(qqbar_t res, const fmpq_t x)**
  - Sets res to the value x.

- **void qqbar_set_re_im(qqbar_t res, const qqbar_t x, const qqbar_t y)**
  - Sets res to the value x + yi.

- **int qqbar_set_d(qqbar_t res, double x)**

- **int qqbar_set_re_im_d(qqbar_t res, double x, double y)**
  - Sets res to the value x or x + yi respectively. These functions performs error handling: if x and y are finite, the conversion succeeds and the return flag is 1. If x or y is non-finite (infinity or NaN), the conversion fails and the return flag is 0.

### 8.4.4 Properties

- **slong qqbar_degree(const qqbar_t x)**
  - Returns the degree of x, i.e. the degree of the minimal polynomial.

- **int qqbar_is_rational(const qqbar_t x)**
  - Returns whether x is a rational number.

- **int qqbar_is_integer(const qqbar_t x)**
  - Returns whether x is an integer (an element of Z).

- **int qqbar_is_algebraic_integer(const qqbar_t x)**
  - Returns whether x is an algebraic integer, i.e. whether its minimal polynomial has leading coefficient 1.

- **int qqbar_is_zero(const qqbar_t x)**

- **int qqbar_is_one(const qqbar_t x)**

- **int qqbar_is_neg_one(const qqbar_t x)**
  - Returns whether x is the number 0, 1, −1.

- **int qqbar_is_i(const qqbar_t x)**
int qqbar_is_neg_i(const qqbar_t x)
    Returns whether \( x \) is the imaginary unit \( i \) (respectively \(-i\)).

int qqbar_is_real(const qqbar_t x)
    Returns whether \( x \) is a real number.

void qqbar_height(fmpz_t res, const qqbar_t x)
    Sets \( res \) to the height of \( x \) (the largest absolute value of the coefficients of the minimal polynomial of \( x \)).

slong qqbar_height_bits(const qqbar_t x)
    Returns the height of \( x \) (the largest absolute value of the coefficients of the minimal polynomial of \( x \)) measured in bits.

int qqbar_within_limits(const qqbar_t x, slong deg_limit, slong bits_limit)
    Checks if \( x \) has degree bounded by \( \text{deg}_\text{limit} \) and height bounded by \( \text{bits}_\text{limit} \) bits, returning 0 (false) or 1 (true). If \( \text{deg}_\text{limit} \) is set to 0, the degree check is skipped, and similarly for \( \text{bits}_\text{limit} \).

int qqbar_binop_within_limits(const qqbar_t x, const qqbar_t y, slong deg_limit, slong bits_limit)
    Checks if \( x + y \), \( x - y \), \( x \cdot y \) and \( x/y \) certainly have degree bounded by \( \text{deg}_\text{limit} \) (by multiplying the degrees for \( x \) and \( y \) to obtain a trivial bound). For \( \text{bits}_\text{limit} \), the sum of the bit heights of \( x \) and \( y \) is checked against the bound (this is only a heuristic). If \( \text{deg}_\text{limit} \) is set to 0, the degree check is skipped, and similarly for \( \text{bits}_\text{limit} \).

8.4.5 Conversions

void _qqbar_get_fmpq(fmpz_t num, fmpz_t den, const qqbar_t x)
    Sets \( \text{num} \) and \( \text{den} \) to the numerator and denominator of \( x \). Aborts if \( x \) is not a rational number.

void qqbar_get_fmpq(fmpq_t res, const qqbar_t x)
    Sets \( \text{res} \) to \( x \). Aborts if \( x \) is not a rational number.

void qqbar_get_fmpz(fmpz_t res, const qqbar_t x)
    Sets \( \text{res} \) to \( x \). Aborts if \( x \) is not an integer.

8.4.6 Special values

void qqbar_zero(qqbar_t res)
    Sets \( \text{res} \) to the number 0.

void qqbar_one(qqbar_t res)
    Sets \( \text{res} \) to the number 1.

void qqbar_i(qqbar_t res)
    Sets \( \text{res} \) to the imaginary unit \( i \).

void qqbar_phi(qqbar_t res)
    Sets \( \text{res} \) to the golden ratio \( \varphi = \frac{1}{2}(\sqrt{5} + 1) \).
### 8.4.7 Input and output

void *qqbar_print(const qqbar_t x)

Prints res to standard output. The output shows the degree and the list of coefficients of the minimal polynomial followed by a decimal representation of the enclosing interval. This function is mainly intended for debugging.

void *qqbar_printn(const qqbar_t x, slong n)

Prints res to standard output. The output shows a decimal approximation to n digits.

void *qqbar_printnd(const qqbar_t x, slong n)

Prints res to standard output. The output shows a decimal approximation to n digits, followed by the degree of the number.

For example, print, printn and printnd with \( n = 6 \) give the following output for the numbers 0, 1, \( i \), \( \varphi \), \( \sqrt{2} - \sqrt{3}i \):

<table>
<thead>
<tr>
<th>deg 1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>deg 1</td>
<td>[-1, 1] 1.00000</td>
</tr>
<tr>
<td>deg 2</td>
<td>[1, 0, 1] 1.00000*I</td>
</tr>
<tr>
<td>deg 2</td>
<td>[-1, -1, 1] [1.6180339887499484820458683436563811772 +/- 6.00e-39]</td>
</tr>
<tr>
<td>deg 4</td>
<td>[25, 0, 2, 0, 1] [1.414213562373095048801688724209680786 +/- 8.67e-38] + [-1.73205087568877293527446341505872367 +/- 1.10e-37]*I</td>
</tr>
</tbody>
</table>

0
1.00000
1.00000*I
1.61803
1.41421 - 1.73205*I

0 (deg 1)
1.00000 (deg 1)
1.00000*I (deg 2)
1.61803 (deg 2)
1.41421 - 1.73205*I (deg 4)

### 8.4.8 Random generation

void *qqbar_randtest(qqbar_t res, flint_rand_t state, slong deg, slong bits)

Sets res to a random algebraic number with degree up to deg and with height (measured in bits) up to bits.

void *qqbar_randtest_real(qqbar_t res, flint_rand_t state, slong deg, slong bits)

Sets res to a random real algebraic number with degree up to deg and with height (measured in bits) up to bits.

void *qqbar_randtest_nonreal(qqbar_t res, flint_rand_t state, slong deg, slong bits)

Sets res to a random nonreal algebraic number with degree up to deg and with height (measured in bits) up to bits. Since all algebraic numbers of degree 1 are real, deg must be at least 2.
### 8.4.9 Comparisons

```c
int qqbar_equal(const qqbar_t x, const qqbar_t y)
    Returns whether x and y are equal.
```

```c
int qqbar_equal_fmpq_poly_val(const qqbar_t x, const fmpq_poly_t f, const qqbar_t y)
    Returns whether x is equal to f(y). This function is more efficient than evaluating f(y) and comparing the results.
```

```c
int qqbar_cmp_re(const qqbar_t x, const qqbar_t y)
    Compares the real parts of x and y, returning -1, 0 or +1.
```

```c
int qqbar_cmp_im(const qqbar_t x, const qqbar_t y)
    Compares the imaginary parts of x and y, returning -1, 0 or +1.
```

```c
int qqbar_cmpabs_re(const qqbar_t x, const qqbar_t y)
    Compares the absolute values of the real parts of x and y, returning -1, 0 or +1.
```

```c
int qqbar_cmpabs_im(const qqbar_t x, const qqbar_t y)
    Compares the absolute values of the imaginary parts of x and y, returning -1, 0 or +1.
```

```c
int qqbar_cmpabs(const qqbar_t x, const qqbar_t y)
    Compares the absolute values of x and y, returning -1, 0 or +1.
```

```c
int qqbar_cmp_root_order(const qqbar_t x, const qqbar_t y)
    Compares x and y using an arbitrary but convenient ordering defined on the complex numbers. This is useful for sorting the roots of a polynomial in a canonical order.

    We define the root order as follows: real roots come first, in descending order. Nonreal roots are subsequently ordered first by real part in descending order, then in ascending order by the absolute value of the imaginary part, and then in descending order of the sign. This implies that complex conjugate roots are adjacent, with the root in the upper half plane first.
```

```c
ulong qqbar_hash(const qqbar_t x)
    Returns a hash of x. As currently implemented, this function only hashes the minimal polynomial of x. The user should mix in some bits based on the numerical value if it is critical to distinguish between conjugates of the same minimal polynomial. This function is also likely to produce serial runs of values for lexicographically close minimal polynomials. This is not necessarily a problem for use in hash tables, but if it is important that all bits in the output are random, the user should apply an integer hash function to the output.
```

### 8.4.10 Complex parts

```c
void qqbar_conj(qqbar_t res, const qqbar_t x)
    Sets res to the complex conjugate of x.
```

```c
void qqbar_re(qqbar_t res, const qqbar_t x)
    Sets res to the real part of x.
```

```c
void qqbar_im(qqbar_t res, const qqbar_t x)
    Sets res to the imaginary part of x.
```

```c
void qqbar_re_im(qqbar_t res1, qqbar_t res2, const qqbar_t x)
    Sets res1 to the real part of x and res2 to the imaginary part of x.
```

```c
void qqbar_abs(qqbar_t res, const qqbar_t x)
    Sets res to the absolute value of x.
```

```c
void qqbar_abs2(qqbar_t res, const qqbar_t x)
    Sets res to the square of the absolute value of x.
```
void qqbar_sgn(qqbar_t res, const qqbar_t x)
    Sets res to the complex sign of x, defined as 0 if x is zero and as x/|x| otherwise.

int qqbar_sgn_re(const qqbar_t x)
    Returns the sign of the real part of x (-1, 0 or +1).

int qqbar_sgn_im(const qqbar_t x)
    Returns the sign of the imaginary part of x (-1, 0 or +1).

int qqbar_csgn(const qqbar_t x)
    Returns the extension of the real sign function taking the value 1 for x strictly in the right half plane, -1 for x strictly in the left half plane, and the sign of the imaginary part when x is on the imaginary axis. Equivalently, csgn(x) = x/√x² except that the value is 0 when x is zero.

8.4.11 Integer parts

void qqbar_floor(fmpz_t res, const qqbar_t x)
    Sets res to the floor function of x. If x is not real, the value is defined as the floor function of the real part of x.

void qqbar_ceil(fmpz_t res, const qqbar_t x)
    Sets res to the ceiling function of x. If x is not real, the value is defined as the ceiling function of the real part of x.

8.4.12 Arithmetic

void qqbar_neg(qqbar_t res, const qqbar_t x)
    Sets res to the negation of x.

void qqbar_add(qqbar_t res, const qqbar_t x, const qqbar_t y)

void qqbar_add_fmpq(qqbar_t res, const qqbar_t x, const fmpq_t y)

void qqbar_add_fmpz(qqbar_t res, const qqbar_t x, const fmpz_t y)

void qqbar_add_ui(qqbar_t res, const qqbar_t x, ulong y)

void qqbar_add_si(qqbar_t res, const qqbar_t x, slong y)
    Sets res to the sum of x and y.

void qqbar_sub(qqbar_t res, const qqbar_t x, const qqbar_t y)

void qqbar_sub_fmpq(qqbar_t res, const qqbar_t x, const fmpq_t y)

void qqbar_sub_fmpz(qqbar_t res, const qqbar_t x, const fmpz_t y)

void qqbar_sub_ui(qqbar_t res, const qqbar_t x, ulong y)

void qqbar_sub_si(qqbar_t res, const qqbar_t x, slong y)

void qqbar_fmpq_sub(qqbar_t res, const fmpq_t x, const qqbar_t y)

void qqbar_fmpz_sub(qqbar_t res, const fmpz_t x, const qqbar_t y)

void qqbar_ui_sub(qqbar_t res, ulong x, const qqbar_t y)

void qqbar_si_sub(qqbar_t res, slong x, const qqbar_t y)
    Sets res to the difference of x and y.

void qqbar_mul(qqbar_t res, const qqbar_t x, const qqbar_t y)

void qqbar_mul_fmpq(qqbar_t res, const qqbar_t x, const fmpq_t y)

void qqbar_mul_fmpz(qqbar_t res, const qqbar_t x, const fmpz_t y)

void qqbar_mul_ui(qqbar_t res, const qqbar_t x, ulong y)

void qqbar_mul_si(qqbar_t res, const qqbar_t x, slong y)
    Sets res to the product of x and y.
void qqbar_mul_2exp_si(qqbar_t res, const qqbar_t x, slong e)
    Sets res to x multiplied by $2^e$.

void qqbar_sqr(qqbar_t res, const qqbar_t x)
    Sets res to the square of x.

void qqbar_inv(qqbar_t res, const qqbar_t x)
    Sets res to the multiplicative inverse of y. Division by zero calls flint_abort.

void qqbar_div(qqbar_t res, const qqbar_t x, const qqbar_t y)

void qqbar_div_fmpq(qqbar_t res, const qqbar_t x, const fmpq_t y)

void qqbar_div_fmpz(qqbar_t res, const qqbar_t x, const fmpz_t y)

void qqbar_div_ui(qqbar_t res, const qqbar_t x, ulong y)

void qqbar_div_si(qqbar_t res, const qqbar_t x, slong y)

void qqbar_fmpq_div(qqbar_t res, const fmpq_t x, const qqbar_t y)

void qqbar_fmpz_div(qqbar_t res, const fmpz_t x, const qqbar_t y)

void qqbar_ui_div(qqbar_t res, ulong x, const qqbar_t y)

void qqbar_si_div(qqbar_t res, slong x, const qqbar_t y)
    Sets res to the quotient of x and y. Division by zero calls flint_abort.

void qqbar_scalar_op(qqbar_t res, const qqbar_t x, const fmpz_t a, const fmpz_t b, const fmpz_t c)
    Sets res to the rational affine transformation $(ax + b)/c$, performed as a single operation. There are no restrictions on a, b and c except that c must be nonzero. Division by zero calls flint_abort.

8.4.13 Powers and roots

void qqbar_sqrt(qqbar_t res, const qqbar_t x)
void qqbar_sqrt_ui(qqbar_t res, ulong x)
    Sets res to the principal square root of x.

void qqbar_rsqrt(qqbar_t res, const qqbar_t x)
    Sets res to the reciprocal of the principal square root of x. Division by zero calls flint_abort.

void qqbar_pow_ui(qqbar_t res, const qqbar_t x, ulong n)
void qqbar_pow_si(qqbar_t res, const qqbar_t x, slong n)
void qqbar_pow_fmpz(qqbar_t res, const qqbar_t x, const fmpz_t n)
void qqbar_pow_fmpq(qqbar_t res, const qqbar_t x, const fmpq_t n)
    Sets res to $x^n$. Raising zero to a negative power aborts.

void qqbar_root_ui(qqbar_t res, const qqbar_t x, ulong n)
void qqbar_fmpq_root_ui(qqbar_t res, const fmpq_t x, ulong n)
    Sets res to the principal $n$-th root of x. The order $n$ must be positive.

void qqbar_fmpq_pow_ui(qqbar_t res, const fmpq_t x, slong m, ulong n)
    Sets res to the principal branch of $x^{m/n}$. The order $n$ must be positive. Division by zero calls flint_abort.

int qqbar_pow(qqbar_t res, const qqbar_t x, const qqbar_t y)
    General exponentiation: if $x^y$ is an algebraic number, sets res to this value and returns 1. If $x^y$ is transcendental or undefined, returns 0. Note that this function returns 0 instead of aborting on division zero.
8.4.14 Numerical enclosures

The following functions guarantee a polished output in which both the real and imaginary parts are accurate to prec bits and exact when exactly representable (that is, when a real or imaginary part is a sufficiently small dyadic number). In some cases, the computations needed to polish the output may be expensive. When polish is unnecessary, qqbar_enclosure_raw() may be used instead. Alternatively, qqbar_cache_enclosure() can be used to avoid recomputations.

void qqbar_get_acb(acb_t res, const qqbar_t x, slong prec)
Sets res to an enclosure of x rounded to prec bits.

void qqbar_get_arb(arb_t res, const qqbar_t x, slong prec)
Sets res to an enclosure of x rounded to prec bits, assuming that x is a real number. If x is not real, res is set to [NaN ±∞].

void qqbar_get_arb_re(arb_t res, const qqbar_t x, slong prec)
Sets res to an enclosure of the real part of x rounded to prec bits.

void qqbar_get_arb_im(arb_t res, const qqbar_t x, slong prec)
Sets res to an enclosure of the imaginary part of x rounded to prec bits.

void qqbar_cache_enclosure(qqbar_t res, slong prec)
Polishes the internal enclosure of res to at least prec bits of precision in-place. Normally, qqbar operations that need high-precision enclosures compute them on the fly without caching the results; if res will be used as an invariant operand for many operations, calling this function as a precomputation step can improve performance.

8.4.15 Numerator and denominator

void qqbar_denominator(fmpz_t res, const qqbar_t y)
Sets res to the denominator of y, i.e. the leading coefficient of the minimal polynomial of y.

void qqbar_numerator(qqbar_t res, const qqbar_t y)
Sets res to the numerator of y, i.e. y multiplied by its denominator.

8.4.16 Conjugates

void qqbar_conjugates(qqbar_ptr res, const qqbar_t x)
Sets the entries of the vector res to the d algebraic conjugates of x, including x itself, where d is the degree of x. The output is sorted in a canonical order (as defined by qqbar_cmp_root_order()).

8.4.17 Polynomial evaluation

void _qqbar_evaluate_fmpq_poly(qqbar_t res, const fmpz *poly, const fmpz_t den, slong len, const qqbar_t x)
void qqbar_evaluate_fmpq_poly(qqbar_t res, const fmpq_poly_t poly, const qqbar_t x)
void _qqbar_evaluate_fmpz_poly(qqbar_t res, const fmpz *poly, slong len, const qqbar_t x)
void qqbar_evaluate_fmpz_poly(qqbar_t res, const fmpz_poly_t poly, const qqbar_t x)
Sets res to the value of the given polynomial poly evaluated at the algebraic number x. These methods detect simple special cases and automatically reduce poly if its degree is greater or equal to that of the minimal polynomial of x. In the generic case, evaluation is done by computing minimal polynomials of representation matrices.

int qqbar_evaluate_fmpz_mpoly_iter(qqbar_t res, const fmpz_mpoly_t poly, qqbar_srcptr x, slong deg_limit, slong bits_limit, const fmpz_mpoly_ctx_t ctx)
int qqbar_evaluate_fmpz_mpoly_horner(qqbar_t res, const fmpz_mpoly_t poly, qqbar_srcptr x, slong deg_limit, slong bits_limit, const fmpz_mpoly_ctx_t ctx)

int qqbar_evaluate_fmpz_mpoly(qqbar_t res, const fmpz_mpoly_t poly, qqbar_srcptr x, slong deg_limit, slong bits_limit, const fmpz_mpoly_ctx_t ctx)

Sets res to the value of poly evaluated at the algebraic numbers given in the vector x. The number of variables is defined by the context object ctx.

The parameters deg_limit and bits_limit define evaluation limits: if any temporary result exceeds these limits (not necessarily the final value, in case of cancellation), the evaluation is aborted and 0 (failure) is returned. If evaluation succeeds, 1 is returned.

The iter version iterates over all terms in succession and computes the powers that appear. The horner version uses a multivariate implementation of the Horner scheme. The default algorithm currently uses the Horner scheme.

8.4.18 Polynomial roots

void qqbar_roots_fmpz_poly(qqbar_ptr res, const fmpz_poly_t poly, int flags)
void qqbar_roots_fmpq_poly(qqbar_ptr res, const fmpq_poly_t poly, int flags)

Sets the entries of the vector res to the d roots of the polynomial poly. Roots with multiplicity appear with repetition in the output array. By default, the roots will be sorted in a convenient canonical order (as defined by qqbar_cmp_root_order()). Instances of a repeated root always appear consecutively.

The following flags are supported:

- QQBAR_ROOTS_IRREDUCIBLE - if set, poly is assumed to be irreducible (it may still have constant content), and no polynomial factorization is performed internally.
- QQBAR_ROOTS_UNSORTED - if set, the roots will not be guaranteed to be sorted (except for repeated roots being listed consecutively).

void qqbar_eigenvalues_fmpz_mat(qqbar_ptr res, const fmpz_mat_t mat, int flags)
void qqbar_eigenvalues_fmpq_mat(qqbar_ptr res, const fmpq_mat_t mat, int flags)

Sets the entries of the vector res to the eigenvalues of the square matrix mat. These functions compute the characteristic polynomial of mat and then call qqbar_roots_fmpz_poly() with the same flags.

8.4.19 Roots of unity and trigonometric functions

The following functions use word-size integers p and q instead of fmpq_t instances to express rational numbers. This is to emphasize that the computations are feasible only with small q in this representation of algebraic numbers since the associated minimal polynomials have degree $O(q)$. The input p and q do not need to be reduced a priori, but should not be close to the word boundaries (they may be added and subtracted internally).

void qqbar_root_of_unity(qqbar_t res, slong p, ulong q)

Sets res to the root of unity $e^{2\pi ip/q}$.

int qqbar_is_root_of_unity(slong *p, ulong *q, const qqbar_t x)

If x is not a root of unity, returns 0. If x is a root of unity, returns 1. If p and q are not NULL and x is a root of unity, this also sets p and q to the minimal integers with $0 \leq p < q$ such that $x = e^{2\pi ip/q}$.

void qqbar_exp_pi_i(qqbar_t res, slong p, ulong q)

Sets res to the root of unity $e^{\pi i/p}$.

void qqbar_cos_pi(qqbar_t res, slong p, ulong q)
void qqbar_sin_pi(qqbar_t res, slong p, ulong q)
int qqbar_tan_pi(qqbar_t res, slong p, ulong q)
int qqbar_cot_pi(qqbar_t res, slong p, ulong q)
int qqbar_sec_pi(qqbar_t res, slong p, ulong q)
int qqbar_csc_pi(qqbar_t res, slong p, ulong q)
int qqbar_log_pi_i(qqbar_t res, const qqbar_t x)

Attempts to find an algebraic number \( p/q \) of degree at most \( \text{max\_deg} \) and height at most \( \text{max\_bits} \) bits matching the numerical enclosure \( z \). The return flag indicates success. This is only a heuristic method, and the return flag neither implies a rigorous proof that \( z \) is the correct result, nor a rigorous proof that no suitable algebraic number with the given \( \text{max\_deg} \) and \( \text{max\_bits} \) exists. (Proof of nonexistence could in principle be computed, but this is not yet implemented.)

The working precision \( \text{prec} \) should normally be the same as the precision used to compute \( z \). It does not make much sense to run this algorithm with precision smaller than \( O(\text{max\_deg} \cdot \text{max\_bits}) \).

This function does a single iteration at the target \( \text{max\_deg} \), \( \text{max\_bits} \), and \( \text{prec} \). For best performance, one should invoke this function repeatedly with successively larger parameters when the size of the intended solution is unknown or may be much smaller than a worst-case bound.

Attempts to express \( x \) in the number field generated by \( \alpha \), returning success (0 or 1). On success, \( \text{res} \) is set to a polynomial \( f \) of degree less than the degree of \( \alpha \) and with height (counting both the numerator and the denominator, when the coefficients of \( y \) are put on a common denominator) bounded by \( \text{max\_bits} \) bits, such that \( f(\alpha) = x \).
(Exception: the max_bits parameter is currently ignored if x is rational, in which case res is just set to the value of x.)

This function looks for a linear relation heuristically using a working precision of prec bits. If x is expressible in terms of alpha, then this function is guaranteed to succeed when prec is taken large enough. The identity \( f(\alpha) = x \) is checked rigorously, i.e. a return value of 1 implies a proof of correctness. In principle, choosing a sufficiently large prec can be used to prove that \( x \) does not lie in the field generated by alpha, but the present implementation does not support doing so automatically.

This function does a single iteration at the target max_bits and prec. For best performance, one should invoke this function repeatedly with successively larger parameters when the size of the intended solution is unknown or may be much smaller than a worst-case bound.

### 8.4.21 Symbolic expressions and conversion to radicals

```c
void qqbar_get_quadratic(fmpz_t a, fmpz_t b, fmpz_t c, fmpz_t q, const qqbar_t x, int factoring)
```

Assuming that \( x \) has degree 1 or 2, computes integers \( a \), \( b \), \( c \) and \( q \) such that

\[
x = \frac{a + b\sqrt{c}}{q}
\]

and such that \( c \) is not a perfect square, \( q \) is positive, and \( q \) has no content in common with both \( a \) and \( b \). In other words, this determines a quadratic field \( \mathbb{Q}(\sqrt{c}) \) containing \( x \), and then finds the canonical reduced coefficients \( a \), \( b \) and \( q \) expressing \( x \) in this field. For convenience, this function supports rational \( x \), for which \( b \) and \( c \) will both be set to zero. The following remarks apply to irrationals.

The radicand \( c \) will not be a perfect square, but will not automatically be squarefree since this would require factoring the discriminant. As a special case, \( c \) will be set to \(-1\) if \( x \) is a Gaussian rational number. Otherwise, behavior is controlled by the factoring parameter.

- If `factoring` is 0, no factorization is performed apart from removing powers of two.
- If `factoring` is 1, a complete factorization is performed (\( c \) will be minimal). This can be very expensive if the discriminant is large.
- If `factoring` is 2, a smooth factorization is performed to remove small factors from \( c \). This is a tradeoff that provides pretty output in most cases while avoiding extreme worst-case slowdown. The smooth factorization guarantees finding all small factors (up to some trial division limit determined internally by Flint), but large factors are only found heuristically.

```c
int qqbar_set_fexpr(qqbar_t res, const fexpr_t expr)
```

Sets \( res \) to the algebraic number represented by the symbolic expression \( expr \), returning 1 on success and 0 on failure.

This function performs a “static” evaluation using qqbar arithmetic, supporting only closed-form expressions with explicitly algebraic subexpressions. It can be used to recover values generated by `qqbar_get_expr_formula()` and variants. For evaluating more complex expressions involving other types of values or requiring symbolic simplifications, the user should preprocess \( expr \) so that it is in a form which can be parsed by `qqbar_set_fexpr()`.

The following expressions are supported:

- Integer constants
- Arithmetic operations with algebraic operands
- Square roots of algebraic numbers
- Powers with algebraic base and exponent an explicit rational number
- `NumberI`, `GoldenRatio`, `RootOfUnity`
- Floor, Ceil, Abs, Sign, Csgn, Conjugate, Re, Im, Max, Min
- Trigonometric functions with argument an explicit rational number times Pi
- Exponentials with argument an explicit rational number times Pi * NumberI
- The Decimal() constructor
- AlgebraicNumberSerialized() (assuming valid data, which is not checked)
- PolynomialRootIndexed()
- PolynomialRootNearest()

Examples of formulas that are not supported, despite the value being an algebraic number:

- Pi - Pi (general transcendental simplifications are not performed)
- 1 / Infinity (only numbers are handled)
- Sum(n, For(n, 1, 10)) (only static evaluation is performed)

```
void qqbar_get_fexpr_repr(fexpr_t res, const.qqbar_t x)
Sets res to a symbolic expression reflecting the exact internal representation of x. The output
will have the form AlgebraicNumberSerialized(List(coeffs), enclosure). The output can
be converted back to a qqbar_t value using qqbar_set_fexpr(). This is the recommended format
for serializing algebraic numbers as it requires minimal computation, but it has the disadvantage
of not being human-readable.
```

```
void qqbar_get_fexpr_root_nearest(fexpr_t res, const.qqbar_t x)
Sets res to a symbolic expression unambiguously describing x in the form
PolynomialRootNearest(List(coeffs), point) where point is an approximation of x guaran-
teed to be closer to x than any conjugate root. The output can be converted back to a qqbar_t
value using qqbar_set_fexpr(). This is a useful format for human-readable presentation, but
serialization and deserialization can be expensive.
```

```
void qqbar_get_fexpr_root_indexed(fexpr_t res, const.qqbar_t x)
Sets res to a symbolic expression unambiguously describing x in the form
PolynomialRootIndexed(List(coeffs), index) where index is the index of x among its
conjugate roots in the builtin root sort order. The output can be converted back to a qqbar_t
value using qqbar_set_fexpr(). This is a useful format for human-readable presentation when
the numerical value is important, but serialization and deserialization can be expensive.
```

```
int qqbar_get_fexpr_formula(fexpr_t res, const.qqbar_t x, ulong flags)
Attempts to express the algebraic number x as a closed-form expression using arithmetic op-
erations, radicals, and possibly exponentials or trigonometric functions, but without using
PolynomialRootNearest or PolynomialRootIndexed. Returns 0 on failure and 1 on success.
```

The flags parameter toggles different methods for generating formulas. It can be set to any com-
bination of the following. If flags is 0, only rational numbers will be handled.

QQBAR_FORMULA_ALL
  Toggles all methods (potentially expensive).

QQBAR_FORMULA_GAUSSIANS
  Detect Gaussian rational numbers a + bi.

QQBAR_FORMULA_QUADRATICS
  Solve quadratics in the form a + b√d.

QQBAR_FORMULA_CYCLOTOMICS
  Detect elements of cyclotomic fields. This works by trying plausible cyclotomic fields (based
  on the degree of the input), using LLL to find candidate number field elements, and certifying
  candidates through an exact computation. Detection is heuristic and is not guaranteed to
  find all cyclotomic numbers.
Solve polynomials of degree 3, 4 and (where applicable) 5 using cubic, quartic and quintic formulas (not yet implemented).

Use depression to try to generate simpler numbers.

Use deflation to try to generate simpler numbers. This allows handling number of the form $a^{1/n}$ where $a$ can be represented in closed form.

Try separating real and imaginary parts or sign and magnitude of complex numbers. This allows handling numbers of the form $a + bi$ or $m \cdot s$ (with $m > 0, |s| = 1$) where $a$ and $b$ or $m$ and $s$ can be represented in closed form. This is only attempted as a fallback after other methods fail: if an explicit Cartesian or magnitude-sign represented is desired, the user should manually separate the number into complex parts before calling `qqbar_get_fexpr_formula()`.

Select output form for cyclotomic numbers. The `auto` form (equivalent to no flags being set) results in radicals for numbers of low degree, trigonometric functions for real numbers, and complex exponentials for nonreal numbers. The other flags (not fully implemented) can be used to force exponential form, trigonometric form, or radical form.

8.4.22 Internal functions

void `qqbar_fmpz_poly_composed_op`(fmpz_poly_t res, const fmpz_poly_t A, const fmpz_poly_t B, int op)

Given nonconstant polynomials $A$ and $B$, sets $res$ to a polynomial whose roots are $a + b, a - b, ab$ or $a/b$ for all roots $a$ of $A$ and all roots $b$ of $B$. The parameter $op$ selects the arithmetic operation: 0 for addition, 1 for subtraction, 2 for multiplication and 3 for division. If $op$ is 3, $B$ must not have zero as a root.

void `qqbar_binary_op`(qqbar_t res, const qqbar_t x, const qqbar_t y, int op)

Performs a binary operation using a generic algorithm. This does not check for special cases.

int `_qqbar_validate_uniqueness`(acb_t res, const fmpz_poly_t poly, const acb_t z, slong max_prec)

Given $z$ known to be an enclosure of at least one root of $poly$, certifies that the enclosure contains a unique root, and in that case sets $res$ to a new (possibly improved) enclosure for the same root, returning 1. Returns 0 if uniqueness cannot be certified.

The enclosure is validated by performing a single step with the interval Newton method. The working precision is determined from the accuracy of $z$, but limited by $max\_prec$ bits.

This method slightly inflates the enclosure $z$ to improve the chances that the interval Newton step will succeed. Uniqueness on this larger interval implies uniqueness of the original interval, but not existence; when existence has not been ensured a priori, `_qqbar_validate_existence_uniqueness()` should be used instead.

int `_qqbar_validate_existence_uniqueness`(acb_t res, const fmpz_poly_t poly, const acb_t z, slong max_prec)

Given any complex interval $z$, certifies that the enclosure contains a unique root of $poly$, and in
that case sets res to a new (possibly improved) enclosure for the same root, returning 1. Returns 0 if existence and uniqueness cannot be certified.

The enclosure is validated by performing a single step with the interval Newton method. The working precision is determined from the accuracy of $z$, but limited by $\text{max} \_\text{prec}$ bits.

```c
void _qqbar_enclosure_raw(acb_t res, const fmpz_poly_t poly, const acb_t z, slong prec)
void qqbar_enclosure_raw(acb_t res, const qqbar_t x, slong prec)
```

Sets res to an enclosure of $x$ accurate to about $\text{prec}$ bits (the actual accuracy can be slightly lower, or higher).

This function uses repeated interval Newton steps to polish the initial enclosure $z$, doubling the working precision each time. If any step fails to improve the accuracy significantly, the root is recomputed from scratch to higher precision.

If the initial enclosure is accurate enough, res is set to this value without rounding and without further computation.

```c
int _qqbar_acb_lindep(fmpz *rel, acb_srcptr vec, slong len, int check, slong prec)
```

Attempts to find an integer vector $rel$ giving a linear relation between the elements of the real or complex vector $vec$, using the LLL algorithm.

The working precision is set to the minimum of $\text{prec}$ and the relative accuracy of $vec$ (that is, the difference between the largest magnitude and the largest error magnitude within $vec$). 95% of the bits within the working precision are used for the LLL matrix, and the remaining 5% bits are used to validate the linear relation by evaluating the linear combination and checking that the resulting interval contains zero. This validation does not prove the existence or nonexistence of a linear relation, but it provides a quick heuristic way to eliminate spurious relations.

If check is set, the return value indicates whether the validation was successful; otherwise, the return value simply indicates whether the algorithm was executed normally (failure may occur, for example, if the input vector is non-finite).

In principle, this method can be used to produce a proof that no linear relation exists with coefficients up to a specified bit size, but this has not yet been implemented.
REAL AND COMPLEX NUMBERS

9.1 Feature overview

Ball arithmetic, also known as mid-rad interval arithmetic, is an extension of floating-point arithmetic in which an error bound is attached to each variable. This allows computing with real and complex numbers in a mathematically rigorous way.

With plain floating-point arithmetic, the user must do an error analysis to guarantee that results are correct. Manual error analysis is time-consuming and bug-prone. Ball arithmetic effectively makes error analysis automatic.

In traditional (inf-sup) interval arithmetic, both endpoints of an interval \([a, b]\) are full-precision numbers, which makes interval arithmetic twice as expensive as floating-point arithmetic. In ball arithmetic, only the midpoint \(m\) of an interval \([m \pm r]\) is a full-precision number, and a few bits suffice for the radius \(r\). At high precision, ball arithmetic is therefore not more expensive than plain floating-point arithmetic.

Joris van der Hoeven’s paper [Hoe2009] is a good introduction to the subject.

Other implementations of ball arithmetic include iRRAM and Mathemagix. Arb differs from earlier implementations in technical aspects of the implementation, which makes certain computations more efficient. It also provides a more comprehensive low-level interface, giving the user full access to the internals. Finally, it implements a wider range of transcendental functions, covering a large portion of the special functions in standard reference works such as [NIST2012].

The ball arithmetic routines in FLINT (formerly the standalone Arb library) are designed for computer algebra and computational number theory, but may be useful in any area demanding reliable or precise numerical computing. The contents include:

- A module \((arf)\) for correctly rounded arbitrary-precision floating-point arithmetic. Arb’s floating-point numbers have a few special features, such as arbitrary-size exponents (useful for combinatorics and asymptotics) and dynamic allocation (facilitating implementation of hybrid integer/floating-point and mixed-precision algorithms).

- A module \((mag)\) for representing magnitudes (error bounds) more efficiently than with an arbitrary-precision floating-point type.

- A module \((arb)\) for real ball arithmetic, where a ball is implemented as an arf midpoint and a mag radius.

- A module \((acb)\) for complex numbers in rectangular form, represented as pairs of real balls.

- Modules \((arb\_poly, acb\_poly)\) for polynomials or power series over the real and complex numbers, implemented using balls as coefficients, with asymptotically fast polynomial multiplication and many other operations.

- Modules \((arb\_mat, acb\_mat)\) for matrices over the real and complex numbers, implemented using balls as coefficients. At the moment, only rudimentary linear algebra operations are provided.

- Functions for high-precision evaluation of various mathematical constants and special functions, implemented using ball arithmetic with rigorous error bounds.
9.2 Using ball arithmetic

This section gives an introduction to working with real numbers in Arb (see *arb.h* – real numbers for the API and technical documentation). The general principles carry over to complex numbers, polynomials and matrices.

9.2.1 Ball semantics

Let \( f : A \to B \) be a function. A ball implementation of \( f \) is a function \( F \) that maps sets \( X \subseteq A \) to sets \( F(X) \subseteq B \) subject to the following rule:

For all \( x \in X \), we have \( f(x) \in F(X) \).

In other words, \( F(X) \) is an enclosure for the set \( \{ f(x) : x \in X \} \). This rule is sometimes called the inclusion principle.

Throughout the documentation (except where otherwise noted), we will simply write \( f(x) \) instead of \( F(X) \) when describing ball implementations of pointwise-defined mathematical functions, understanding that the input is a set of point values and that the output is an enclosure.

General subsets of \( \mathbb{R} \) are not possible to represent on a computer. Instead, we work with subsets of the form \( [m \pm r] = [m-r, m+r] \) where the midpoint \( m \) and radius \( r \) are binary floating-point numbers, i.e. numbers of the form \( u2^v \) with \( u, v \in \mathbb{Z} \) (to make this scheme complete, we also need to adjoin the special floating-point values \( -\infty, +\infty \) and NaN).

Given a ball \( [m \pm r] \) with \( m \in \mathbb{R} \) (not necessarily a floating-point number), we can always round \( m \) to a nearby floating-point number that has at most most \( \text{prec} \) bits in the component \( u \), and add an upper bound for the rounding error to \( r \). In Arb, ball functions that take a \( \text{prec} \) argument as input (e.g. \( \text{arb_add()} \)) always round their output to \( \text{prec} \) bits. Some functions are always exact (e.g. \( \text{arb_neg()} \)), and thus do not take a \( \text{prec} \) argument.

The programming interface resembles that of GMP. Each \( \text{arb}_t \) variable must be initialized with \( \text{arb_init()} \) before use (this also sets its value to zero), and deallocated with \( \text{arb_clear()} \) after use. Variables have pass-by-reference semantics. In the list of arguments to a function, output variables come first, followed by input variables, and finally the precision:

```c
#include "arb.h"

int main()
{
    arb_t x, y;
    arb_init(x); arb_init(y);
    arb_set_ui(x, 3); /* x = 3 */
    arb_const_pi(y, 128); /* y = \pi, to 128 bits */
    arb_sub(y, y, x, 53); /* y = y - x, to 53 bits */
    arb_clear(x); arb_clear(y);
}
```

9.2.2 Binary and decimal

While the internal representation uses binary floating-point numbers, it is usually preferable to print numbers in decimal. The binary-to-decimal conversion generally requires rounding. Three different methods are available for printing a number to standard output:

- \( \text{arb_print()} \) shows the exact internal representation of a ball, with binary exponents.
- \( \text{arb_printd()} \) shows an inexact view of the internal representation, approximated by decimal floating-point numbers.
arb_printn() shows a decimal ball that is guaranteed to be an enclosure of the binary floating-point ball. By default, it only prints digits in the midpoint that are certain to be correct, up to an error of at most one unit in the last place. Converting from binary to decimal is generally inexact, and the output of this method takes this rounding into account when printing the radius.

This snippet computes a 53-bit enclosure of \( \pi \) and prints it in three ways:

```c
arb_const_pi(x, 53);
arb_print(x); printf("\n");
arb_printd(x, 20); printf("\n");
arb_printn(x, 20, 0); printf("\n");
```

The output is:

```
(884279719003555 * 2^-48) +/- (536870913 * 2^-80)
3.141592653589793116 +/- 4.4409e-16
[3.141592653589793 +/- 5.61e-16]
```

The arb_get_str() and arb_set_str() methods are useful for converting rigorously between decimal strings and binary balls (arb_get_str() produces the same string as arb_printn(), and arb_set_str() can parse such strings back).

A potential mistake is to create a ball from a double constant such as 2.3, when this actually represents 2.29999999999999982236431605997495353221893310546875. To produce a ball containing the rational number 23/10, one of the following can be used:

```c
arb_set_str(x, "2.3", prec)
arb_set_ui(x, 23);
arb_div_ui(x, x, 10, prec)
fmpq_set_si(q, 23, 10); /* q is a FLINT fmpq_t */
arb_set_fmpq(x, q, prec);
```

### 9.2.3 Quality of enclosures

The main problem when working with ball arithmetic (or interval arithmetic) is overestimation. In general, the enclosure of a value or set of values as computed with ball arithmetic will be larger than the smallest possible enclosure.

Overestimation results naturally from rounding errors and cancellations in the individual steps of a calculation. As a general principle, formula rewriting techniques that make floating-point code more numerically stable also make ball arithmetic code more numerically stable, in the sense of producing tighter enclosures.

As a result of the dependency problem, ball or interval arithmetic can produce error bounds that are much larger than the actual numerical errors resulting from doing floating-point arithmetic. Consider the expression \((x + 1) - x\) as an example. When evaluated in floating-point arithmetic, \(x\) may have a large initial error. However, that error will cancel itself out in the subtraction, so that the result equals 1 (except perhaps for a small rounding error left from the operation \(x + 1\)). In ball arithmetic, dependent errors add up instead of cancelling out. If \(x = [3 \pm 0.1]\), the result will be \([1 \pm 0.2]\), where the error bound has doubled. In unfavorable circumstances, error bounds can grow exponentially with the number of steps.

If all inputs to a calculation are “point values”, i.e. exact numbers and known mathematical constants that can be approximated arbitrarily closely (such as \(\pi\)), then an error of order \(2^n\) can typically be overcome by working with \(n\) extra bits of precision, increasing the computation time by an amount that is polynomial in \(n\). In certain situations, however, overestimation leads to exponential slowdown or even failure of an algorithm to converge. For example, root-finding algorithms that refine the result iteratively may fail to converge in ball arithmetic, even if they do converge in plain floating-point arithmetic.
Therefore, ball arithmetic is not a silver bullet: there will always be situations where some amount of numerical or mathematical analysis is required. Some experimentation may be required to find whether (and how) it can be used effectively for a given problem.

## 9.2.4 Predicates

A ball implementation of a predicate $f : \mathbb{R} \to \{\text{True}, \text{False}\}$ would need to be able to return a third logical value indicating that the result could be either True or False. In most cases, predicates in Arb are implemented as functions that return the `int` value 1 to indicate that the result certainly is True, and the `int` value 0 to indicate that the result could be either True or False. To test whether a predicate certainly is False, the user must test whether the negated predicate certainly is True.

For example, the following code would *not* be correct in general:

```c
if (arb_is_positive(x))
{
    ... /* do things assuming that $x > 0$ */
}
else
{
    ... /* do things assuming that $x \leq 0$ */
}
```

Instead, the following can be used:

```c
if (arb_is_positive(x))
{
    ... /* do things assuming that $x > 0$ */
}
else if (arb_is_nonpositive(x))
{
    ... /* do things assuming that $x \leq 0$ */
}
else
{
    ... /* do things assuming that the sign of $x$ is unknown */
}
```

Likewise, we will write $x \leq y$ in mathematical notation with the meaning that $x \leq y$ holds for all $x \in X, y \in Y$ where $X$ and $Y$ are balls.

Note that some predicates such as `arb_overlaps()` and `arb_contains()` actually are predicates on balls viewed as sets, and not ball implementations of pointwise predicates.

Some predicates are also complementary. For example `arb_contains_zero()` tests whether the input ball contains the point zero. Negated, it is equivalent to `arb_is_nonzero()`, and complementary to `arb_is_zero()` as a pointwise predicate:

```c
if (arb_is_zero(x))
{
    ... /* do things assuming that $x = 0$ */
}
#if 1
else if (arb_is_nonzero(x))
#else
else if (!arb_contains_zero(x)) /* equivalent */
#endif
{
    ... /* do things assuming that $x \neq 0$ */
}
```

(continues on next page)
9.2.5 A worked example: the sine function

We implement the function $\sin(x)$ naively using the Taylor series $\sum_{k=0}^{\infty} (-1)^k x^{2k+1} / (2k+1)!$ and \texttt{arb_t} arithmetic. Since there are infinitely many terms, we need to split the series in two parts: a finite sum that can be evaluated directly, and a tail that has to be bounded.

We stop as soon as we reach a term $t$ bounded by $|t| \leq 2^{-\text{prec}} < 1$. The terms are alternating and must have decreasing magnitude from that point, so the tail of the series is bounded by $|t|$. We add this magnitude to the radius of the output. Since ball arithmetic automatically bounds the numerical errors resulting from all arithmetic operations, the output $\text{res}$ is a ball guaranteed to contain $\sin(x)$.

```c
#include "arb.h"

void arb_sin_naive(arb_t res, const arb_t x, slong prec)
{
    arb_t s, t, u, tol;
    slong k;
    arb_init(s); arb_init(t); arb_init(u); arb_init(tol);
    arb_one(tol);
    arb_mul_2exp_si(tol, tol, -prec); /* tol = 2^{-\text{prec}} */
    for (k = 0; ; k++)
    {
        arb_pow_ui(t, x, 2 * k + 1, prec);
        arb_fac_ui(u, 2 * k + 1, prec);
        arb_div(t, t, u, prec); /* t = x^{2k+1} / (2k+1)! */
        arb_abs(u, t);
        if (arb_le(u, tol)) /* if |t| <= 2^{-\text{prec}} */
        {
            arb_add_error(s, u); /* add |t| to the radius and stop */
            break;
        }
        if (k % 2 == 0)
            arb_add(s, s, t, prec);
        else
            arb_sub(s, s, t, prec);
    }
    arb_set(res, s);
    arb_clear(s); arb_clear(t); arb_clear(u); arb_clear(tol);
}
```

This algorithm is naive, because the Taylor series is slow to converge and suffers from catastrophic cancellation when $|x|$ is large (we could also improve the efficiency of the code slightly by computing the terms using recurrence relations instead of computing $x^k$ and $k!$ from scratch each iteration).

As a test, we compute $\sin(2016.1)$. The largest term in the Taylor series for $\sin(x)$ reaches a magnitude
of about $x^7/x!$, or about $10^{873}$ in this case. Therefore, we need over 873 digits (about 3000 bits) of precision to overcome the catastrophic cancellation and determine the result with sufficient accuracy to tell whether it is positive or negative.

```c
int main()
{
    arb_t x, y;
slong prec;
arb_init(x); arb_init(y);

    for (prec = 64; ; prec *= 2)
    {
        arb_set_str(x, "2016.1", prec);
arb_sin_naive(y, x, prec);
printf("Using %5ld bits, sin(x) = ", prec);
arb_printn(y, 10, 0); printf("n");
        if (!arb_contains_zero(y)) /* stopping condition */
            break;
    }

    arb_clear(x); arb_clear(y);
}
```

The program produces the following output:

<table>
<thead>
<tr>
<th>Precision</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bits</td>
<td>+/- 2.67e+859</td>
</tr>
<tr>
<td>128 bits</td>
<td>+/- 1.30e+840</td>
</tr>
<tr>
<td>256 bits</td>
<td>+/- 3.60e+801</td>
</tr>
<tr>
<td>512 bits</td>
<td>+/- 3.01e+724</td>
</tr>
<tr>
<td>1024 bits</td>
<td>+/- 2.18e+570</td>
</tr>
<tr>
<td>2048 bits</td>
<td>+/- 1.22e+262</td>
</tr>
<tr>
<td>4096 bits</td>
<td>[-0.7190842207 +/- 1.20e-11]</td>
</tr>
</tbody>
</table>

As an exercise, the reader may improve the naive algorithm by making it subtract a well-chosen multiple of $2\pi$ from $x$ before invoking the Taylor series (hint: use `arb_const_pi()`, `arb_div()` and `arf_get_fmpz()`). If done correctly, 64 bits of precision should be more than enough to compute $\sin(2016.1)$, and with minor adjustments to the code, the user should be able to compute $\sin(\exp(2016.1))$ quite easily as well.

This example illustrates how ball arithmetic can be used to perform nontrivial calculations. To evaluate an infinite series, the user needs to know how to bound the tail of the series, but everything else is automatic. When evaluating a finite formula that can be expressed completely using built-in functions, all error bounding is automatic from the point of view of the user. In particular, the `arb_sin()` method should be used to compute the sine of a real number; it uses a much more efficient algorithm than the naive code above.

This example also illustrates the “guess-and-verify” paradigm: instead of determining a priori the floating-point precision necessary to get a correct result, we guess some initial precision, use ball arithmetic to verify that the result is accurate enough, and restart with higher precision (or signal failure) if it is not.

If we think of rounding errors as essentially random processes, then a floating-point computation is analogous to a Monte Carlo algorithm. Using ball arithmetic to get a verified result effectively turns it into the analog of a Las Vegas algorithm, which is a randomized algorithm that always gives a correct result if it terminates, but may fail to terminate (alternatively, instead of actually looping forever, it might signal failure after a certain number of iterations).

The loop will fail to terminate if we attempt to determine the sign of $\sin(\pi)$:
The sign of a nonzero real number can be decided by computing it to sufficiently high accuracy, but the sign of an expression that is exactly equal to zero cannot be decided by a numerical computation unless the entire computation happens to be exact (in this example, we could use the \texttt{arb\_sin\_pi()} function which computes sin\((\pi x)\) in one step, with the input \(x = 1\)).

It is up to the user to implement a stopping criterion appropriate for the circumstances of a given application. For example, breaking when it is clear that \(|\sin(x)| < 10^{-10000}\) would allow the program to terminate and convey some meaningful information about the input \(x = \pi\), though this would not constitute a mathematical proof that \(\sin(\pi) = 0\).

\section*{9.2.6 More on precision and accuracy}

The relation between the working precision and the accuracy of the output is not always easy predict. The following remarks might help to choose \texttt{prec} optimally.

For a ball \([m \pm r]\) it is convenient to define the following notions:

- Absolute error: \(e_{abs} = |r|\)
- Relative error: \(e_{rel} = |r|/\max(0, |m| - |r|)\) (or \(e_{rel} = 0\) if \(r = m = 0\))
- Absolute accuracy: \(a_{abs} = 1/e_{abs}\)
- Relative accuracy: \(a_{rel} = 1/e_{rel}\)

Expressed in bits, one takes the corresponding \(\log_2\) values.

Of course, if \(x\) is the exact value being approximated, then the “absolute error” so defined is an upper bound for the actual absolute error \(|x - m|\) and “absolute accuracy” a lower bound for \(1/|x - m|\), etc.

The \texttt{prec} argument in Arb should be thought of as controlling the working precision. Generically, when evaluating a fixed expression (that is, when the sequence of operations does not depend on the precision), the absolute or relative error will be bounded by

\[
2^{O(1) - \texttt{prec}}
\]

where the \(O(1)\) term depends on the expression and implementation details of the ball functions used to evaluate it. Accordingly, for an accuracy of \(p\) bits, we need to use a working precision \(O(1) + p\). If the expression is numerically well-behaved, then the \(O(1)\) term will be small, which leads to the heuristic of “adding a few guard bits” (for most basic calculations, 10 or 20 guard bits is enough). If the \(O(1)\) term is unknown, then increasing the number of guard bits in exponential steps until the result is accurate enough is generally a good heuristic.

Sometimes, a partially accurate result can be used to estimate the \(O(1)\) term. For example, if the goal is to achieve 100 bits of accuracy and a precision of 120 bits yields 80 bits of accuracy, then it is plausible that a precision of just over 140 bits yields 100 bits of accuracy.

Built-in functions in Arb can roughly be characterized as belonging to one of two extremes (though there is actually a spectrum):
In short, the fundamental contract in Arb. Enclosures computed by built-in functions may or may not be tight enough to be useful, but the hope is that they will be sufficient for most purposes. Tightening the error bounds for more complex operations is a long term optimization goal, which in many cases will require a fair amount of research. A tradeoff also has to be made for efficiency: tighter error bounds allow the user to work with a lower precision, but they may also be much more expensive to compute.

9.2.7 Polynomial time guarantee

Arb provides a soft guarantee that the time used to evaluate a ball function will depend polynomially on \( \text{prec} \) and the bit size of the input, uniformly regardless of the numerical value of the input.

The idea behind this soft guarantee is to allow Arb to be used as a black box to evaluate expressions numerically without potentially slowing down, hanging indefinitely or crashing because of “bad” input such as nested exponentials. By controlling the precision, the user can cancel a computation before it uses up an unreasonable amount of resources, without having to rely on other timeout or exception mechanisms. A result that is feasible but very expensive to compute can still be forced by setting the precision high enough.

As motivation, consider evaluating \( \sin(x) \) or \( \exp(x) \) with the exact floating-point number \( x = 2^n \) as input. The time and space required to compute an accurate floating-point approximation of \( \sin(x) \) or \( \exp(x) \) increases as \( 2^n \), in the first case because of the need to subtract an accurate multiple of \( 2\pi \) and in the second case due to the size of the output exponent and the internal subtraction of an accurate multiple of \( \log(2) \). This is despite the fact that the size of \( x \) as an object in memory only increases linearly with \( n \). Already \( n = 33 \) would require at least 1 GB of memory, and \( n = 100 \) would be physically impossible to process. For functions that are computed by direct use of power series expansions, e.g. \( f(x) = \sum_{n=0}^{\infty} c_n x^n \), without having fast argument-reduction techniques like those for elementary functions, the time would be exponential in \( n \) already when \( x = 2^n \).

Therefore, Arb caps internal work parameters (the internal working precision, the number terms of an infinite series to add, etc.) by polynomial, usually linear, functions of \( \text{prec} \). When the limit is exceeded, the output is set to a crude bound. For example, if \( x \) is too large, \( \text{arb}_\sin() \) will simply return \([\pm 1]\), and \( \text{arb}_\exp() \) will simply return \([\pm \infty]\) if \( x \) is positive or \([\pm 2^{-m}] \) if \( x \) is negative.

This is not just a failsafe, but occasionally a useful optimization. It is not entirely uncommon to have formulas where one term is modest and another term decreases exponentially, such as:

\[
\log(x) + \sin(x) \exp(-x).
\]

For example, the reflection formula of the digamma function has a similar structure. When \( x \) is large, the right term would be expensive to compute to high relative accuracy. Doing so is unnecessary, however, since a crude bound of \([\pm 1] : [\pm 2^{-m}] \) is enough to evaluate the expression as a whole accurately.

The polynomial time guarantee is “soft” in that there are a few exceptions. For example, the complexity of computing the Riemann zeta function \( \zeta(\sigma + it) \) increases linearly with the imaginary height \(|t|\) in the
current implementation, and all known algorithms have a complexity of $|t|^\alpha$ where the best known value for $\alpha$ is about 0.3. Input with large $|t|$ is most likely to be given deliberately by users with the explicit intent of evaluating the zeta function itself, so the evaluation is not cut off automatically.

### 9.3 Technical conventions and potential issues

#### 9.3.1 Integer overflow

When machine-size integers are used for precisions, sizes of integers in bits, lengths of polynomials, and similar quantities that relate to sizes in memory, very few internal checks are performed to verify that such quantities do not overflow.

Precisions and lengths exceeding a small fraction of `LONG_MAX`, say $2^{24} \approx 10^7$ on 32-bit systems, should be regarded as resulting in undefined behavior. On 64-bit systems this should generally not be an issue, since most calculations will exhaust the available memory (or the user’s patience waiting for the computation to complete) long before running into integer overflows. However, the user needs to be wary of unintentionally passing input parameters of order `LONG_MAX` or negative parameters where positive parameters are expected, for example due to a runaway loop that repeatedly increases the precision.

Currently, no hard upper limit on the precision is defined, but $2^{24} \approx 10^7$ bits on 32-bit system and $2^{36} \approx 10^{11}$ bits on a 64-bit system can be considered safe for most purposes. The relatively low limit on 64-bit systems is due to the fact that GMP integers are used internally in some algorithms, and GMP integers are limited to $2^{37}$ bits. The minimum allowed precision is 2 bits.

This caveat does not apply to exponents of floating-point numbers, which are represented as arbitrary-precision integers, nor to integers used as numerical scalars (e.g. `arb_mul_si()`). However, it still applies to conversions and operations where the result is requested exactly and sizes become an issue. For example, trying to convert the floating-point number $2^{2^{100}}$ to an integer could result in anything from a silent wrong value to thrashing followed by a crash, and it is the user’s responsibility not to attempt such a thing.

#### 9.3.2 Aliasing

As a rule, Arb allows aliasing of operands. For example, in the function call `arb_add(z, x, y, prec)`, which performs $z \leftarrow x + y$, any two (or all three) of the variables $x$, $y$ and $z$ are allowed to be the same. Exceptions to this rule are documented explicitly.

The general rule that input and output variables can be aliased with each other only applies to variables of the same type (ignoring `const` qualifiers on input variables – a special case is that `arb_srcptr` is considered the `const` version of `arb_ptr`). This is a natural extension of the so-called strict aliasing rule in C.

For example, in `arb_poly_evaluate()` which evaluates $y = f(x)$ for a polynomial $f$, the output variable $y$ is not allowed to be a pointer to one of the coefficients of $f$ (but aliasing between $x$ and $y$ or between $x$ and the coefficients of $f$ is allowed). This also applies to `_arb_poly_evaluate()` for the purposes of aliasing, `arb_srcptr` (the type of the coefficient array within $f$) and `arb_t` (the type of $x$) are not considered to be the same type, and therefore must not be aliased with each other, even though an `arb_ptr/arb_srcptr` variable pointing to a length 1 array would otherwise be interchangeable with an `arb_t/const arb_t`.

Moreover, in functions that allow aliasing between an input array and an output array, the arrays must either be identical or completely disjoint, never partially overlapping.

There are natural exceptions to these aliasing restrictions, which may used internally without being documented explicitly. However, third party code should avoid relying on such exceptions.

An important caveat applies to aliasing of input variables. Identical pointers are understood to give permission for algebraic simplification. This assumption is made to improve performance. For
example, the call \texttt{arb_mul}(z, x, x, \texttt{prec}) sets \( z \) to a ball enclosing the set
\[
\{ t^2 : t \in x \}
\]
and not the (generally larger) set
\[
\{ tu : t \in x, u \in x \}.
\]
If the user knows that two values \( x \) and \( y \) both lie in the interval \([-1, 1]\) and wants to compute an enclosure for \( f(x, y) \), then it would be a mistake to create an \texttt{arb_t} variable \( x \) enclosing \([-1, 1]\) and reusing the same variable for \( y \), calling \( f(x, x) \). Instead, the user has to create a distinct variable \( y \) also enclosing \([-1, 1]\).

Algebraic simplification is not guaranteed to occur. For example, \texttt{arb_add}(z, x, x, \texttt{prec}) and \texttt{arb_sub}(z, x, x, \texttt{prec}) currently do not implement this optimization. It is better to use \texttt{arb_mul_2exp_si}(z, x, 1) and \texttt{arb_zero}(z), respectively.

### 9.3.3 Thread safety and caches

Arb should be fully threadsafe, provided that both MPFR and FLINT have been built in threadsafe mode. Use \texttt{flint_set_num_threads()} to set the number of threads that Arb is allowed to use internally for single computations (this is currently only exploited by a handful of operations). Please note that thread safety is only tested minimally, and extra caution when developing multithreaded code is therefore recommended.

Arb may cache some data (such as the value of \( \pi \) and Bernoulli numbers) to speed up various computations. In threadsafe mode, caches use thread-local storage. There is currently no way to save memory and avoid recomputation by having several threads share the same cache. Caches can be freed by calling the \texttt{flint_cleanup()} function. To avoid memory leaks, the user should call \texttt{flint_cleanup()} when exiting a thread. It is also recommended to call \texttt{flint_cleanup()} when exiting the main program (this should result in a clean output when running Valgrind, and can help catching memory issues).

There does not seem to be an obvious way to make sure that \texttt{flint_cleanup()} is called when exiting a thread using OpenMP. A possible solution to this problem is to use OpenMP sections, or to use C++ and create a thread-local object whose destructor invokes \texttt{flint_cleanup()}.

### 9.3.4 Use of hardware floating-point arithmetic

Arb uses hardware floating-point arithmetic (the \texttt{double} type in C) in two different ways.

First, \texttt{double} arithmetic as well as transcendental \texttt{libm} functions (such as \texttt{exp}, \texttt{log}) are used to select parameters heuristically in various algorithms. Such heuristic use of approximate arithmetic does not affect correctness: when any error bounds depend on the parameters, the error bounds are evaluated separately using rigorous methods. At worst, flaws in the floating-point arithmetic on a particular machine could cause an algorithm to become inefficient due to inefficient parameters being selected.

Second, \texttt{double} arithmetic is used internally for some rigorous error bound calculations. To guarantee correctness, we make the following assumptions. With the stated exceptions, these should hold on all commonly used platforms.

- A \texttt{double} uses the standard IEEE 754 format (with a 53-bit significand, 11-bit exponent, encoding of infinities and NaNs, etc.)
- We assume that the compiler does not perform “unsafe” floating-point optimizations, such as reordering of operations. Unsafe optimizations are disabled by default in most modern C compilers, including GCC and Clang. The exception appears to be the Intel C++ compiler, which does some unsafe optimizations by default. These must be disabled by the user.
- We do not assume that floating-point operations are correctly rounded (a counterexample is the x87 FPU), or that rounding is done in any particular direction (the rounding mode may have been changed by the user). We assume that any floating-point operation is done with at most 1.1 ulp error.
• We do not assume that underflow or overflow behaves in a particular way (we only use doubles that fit in the regular exponent range, or explicit infinities).

• We do not use transcendental \texttt{libm} functions, since these can have errors of several ulps, and there is unfortunately no way to get guaranteed bounds. However, we do use functions such as \texttt{ldexp} and \texttt{sqrt}, which we assume to be correctly implemented.

### 9.3.5 Interface changes

Most of the core API should be stable at this point, and significant compatibility-breaking changes will be specified in the release notes.

In general, Arb does not distinguish between “private” and “public” parts of the API. The implementation is meant to be transparent by design. All methods are intended to be fully documented and tested (exceptions to this are mainly due to lack of time on part of the author). The user should use common sense to determine whether a function is concerned with implementation details, making it likely to change as the implementation changes in the future. The interface of \texttt{arb_add()} is probably not going to change in the next version, but \_\_\texttt{arb_get_mpn_fixed_mod_pi4()} just might.

### 9.3.6 General note on correctness

Except where otherwise specified, Arb is designed to produce provably correct error bounds. The code has been written carefully, and the library is extensively tested. However, like any complex mathematical software, Arb is virtually certain to contain bugs, so the usual precautions are advised:

- Do sanity checks. For example, check that the result satisfies an expected mathematical relation, or compute the same result in two different ways, with different settings, and with different levels of precision. Arb’s unit tests already do such checks, but they are not guaranteed to catch every possible bug, and they provide no protection against the user accidentally using the interface incorrectly.

- Compare results with other mathematical software.

- Read the source code to verify that it really does what it is supposed to do.

All bug reports are highly appreciated.

### 9.4 Arb example programs

See \textit{Examples} for general information about example programs. Running:

```
make examples
```

will compile the programs and place the binaries in \texttt{build/examples}. The examples related to the Arb module are documented below.

#### 9.4.1 \texttt{pi.c}

This program computes $\pi$ to an accuracy of roughly $n$ decimal digits by calling the \texttt{arb_const_pi()} function with a working precision of roughly $n \log_2(10)$ bits.

Sample output, computing $\pi$ to one million digits:

```
> build/examples/pi 1000000
precision = 3321933 bits... cpu/wall(s): 0.243 0.244
virt/peak/res/peak(MB): 24.46 30.44 8.73 14.42
[3.14159265358979323846{...999959 digits...}42209010610577945815 +/- 1.38e-1000000]
```
The program prints an interval guaranteed to contain \( \pi \), and where all displayed digits are correct up to an error of plus or minus one unit in the last place (see \texttt{arb_printn()}). By default, only the first and last few digits are printed. Pass 0 as a second argument to print all digits (or pass \( m \) to print \( m+1 \) leading and \( m \) trailing digits, as above with the default \( m=20 \)).

The program can optionally compute various other constants, and can use multiple threads:

```
> build/examples/pi 1000000 -threads 4
precision = 3321933 bits... cpu/wall(s): 0.265 0.147
virt/peak/res/peak(MB): 241.95 422.15 13.33 17.54
[3.14159265358979323846{...999959 digits...}42209010610577945815 +/- 1.38e-1000000]
> build/examples/pi 1000000 -constant e
precision = 3321933 bits... cpu/wall(s): 0.09 0.09
[2.71828182845904523536{...999959 digits...}01379817644769422819 +/- 1.39e-1000000]
```

9.4.2 \texttt{zeta_zeros.c}

This program computes one or several consecutive zeros of the Riemann zeta function on the critical line:

```
> build/examples/zeta_zeros -n 1 -count 10 -digits 30
1 14.1347251417346937904572519836
2 21.0220396387715549926284795939
3 25.0108575801456887632137909926
4 30.4248761258596132103118975306
5 32.9350615877391896906623689641
6 37.5861781588256712572177634807
7 40.9187190121474951873981269146
8 43.327073280914999519461221654
9 48.0051508811671597279424727494
10 49.7738324776723021819167846786
cpu/wall(s): 0.01 0.01
virt/peak/res/peak(MB): 21.28 21.28 7.29 7.29
```

Five zeros starting with the millionth:

```
> build/examples/zeta_zeros -n 1000000 -count 5 -digits 20
1000000 600269.67701244495552
1000001 600270.30109071169886
1000002 600270.74787059436613
1000003 600271.48637367364820
1000004 600271.76148042593778
cpu/wall(s): 0.03 0.03
virt/peak/res/peak(MB): 21.41 21.41 7.41 7.41
```

The program supports the following options:


With \texttt{-platt}, Platt’s algorithm is used, which may be faster when computing many zeros of large index simultaneously.
9.4.3 bernoulli.c

This program benchmarks computing the nth Bernoulli number exactly:

```
> build/examples/bernoulli 1000000 -threads 8
cpu/wall(s): 27.227 5.836
virt/peak/res/peak(MB): 573.47 731.39 73.23 165.13
```

9.4.4 class_poly.c

This program benchmarks computing Hilbert class polynomials:

```
> build/examples/class_poly -1000004 -threads 8
cpu/wall(s): 6.932 1.478
virt/peak/res/peak(MB): 535.27 653.18 71.02 100.65
degree = 624, bits = -37823
```

9.4.5 hilbert_matrix.c

Given an input integer \( n \), this program accurately computes the determinant of the \( n \) by \( n \) Hilbert matrix. Hilbert matrices are notoriously ill-conditioned: although the entries are close to unit magnitude, the determinant \( h_n \) decreases superexponentially (nearly as \( 1/n^2 \)) as a function of \( n \). This program automatically doubles the working precision until the ball computed for \( h_n \) by \texttt{arb_mat_det()} does not contain zero.

Sample output:

```
$ build/examples/hilbert_matrix 200
prec=20: [+- 1.32e-335]
prec=40: [+- 1.63e-545]
prec=80: [+- 1.30e-933]
prec=160: [+- 3.62e-1926]
prec=320: [+- 1.81e-4129]
prec=640: [+- 3.84e-8838]
prec=1280: [2.955454297e-23924 +/- 8.29e-23935]
success!
cpu/wall(s): 8.494 8.513
virt/peak/res/peak(MB): 134.98 134.98 111.57 111.57
```

Called with \(-\text{eig } n\), instead of computing the determinant, the program computes the smallest eigenvalue of the Hilbert matrix (in fact, it isolates all eigenvalues and prints the smallest eigenvalue):

```
$ build/examples/hilbert_matrix -eig 50
prec=20: nan
prec=40: nan
prec=80: nan
prec=160: nan
prec=320: nan
prec=640: [1.459157797e-74 +/- 2.49e-84]
success!
cpu/wall(s): 1.84 1.841
virt/peak/res/peak(MB): 33.97 33.97 10.51 10.51
```
9.4.6 keiper_li.c

Given an input integer \(n\), this program rigorously computes numerical values of the Keiper-Li coefficients \(\lambda_0, \ldots, \lambda_n\). The Keiper-Li coefficients have the property that \(\lambda_n > 0\) for all \(n > 0\) if and only if the Riemann hypothesis is true. This program was used for the record computations described in [Joh2013] (the paper describes the algorithm in some more detail).

The program takes the following parameters:

\[
\text{keiper_li } n \ [-\text{prec prec}] \ [-\text{threads num_threads}] \ [-\text{out out_file}]
\]

The program prints the first and last few coefficients. It can optionally write all the computed data to a file. The working precision defaults to a value that should give all the coefficients to a few digits of accuracy, but can optionally be set higher (or lower). On a multicore system, using several threads results in faster execution.

Sample output:

```bash
> build/examples/keiper_li 1000 -threads 2
zeta: cpu/wall(s): 0.4 0.244
virt/peak/res/peak(MB): 167.98 294.69 5.09 7.43
log: cpu/wall(s): 0.02 0.016
gamma: cpu/wall(s): 0.02 0.016
binomial transform: cpu/wall(s): 0.01 0.016
0: -0.69314718055994530941723212145817656807550013436026 +/- 6.5389e-347
1: 0.02309570896612033814310247906495291621932127152051 +/- 2.0924e-345
2: 0.046172867614023351928624390639394387066108314123 +/- 1.674e-344
3: 0.0692129358180182679304973489726010689942120263932 +/- 5.0219e-344
4: 0.092197619873060496476727872409439018065541673490213 +/- 2.0089e-343
5: 0.115108542892235490486221281098577267613491323203596 +/- 1.0044e-342
6: 0.1379676871372988290416713700341666356138966078654 +/- 6.0264e-342
7: 0.1606371596529942129404028725735636292282442046163 +/- 2.1092e-341
8: 0.183219496348325790819331774721859848998089273432 +/- 8.4368e-341
9: 0.20565733870977046170289387421343304741236553410404 +/- 7.5931e-340
10: 0.22793393619315774369030467368445338074385942738 +/- 7.5931e-339
991: 2.31966179616133679283789965696825652101430813341 +/- 2.461e-11
992: 2.320926229325488403538896518325502336290971288 +/- 9.536e-11
993: 2.321092061239373228116591613332628023475592414 +/- 1.8496e-10
994: 2.32180735401884621102588261215038701274718888893 +/- 3.5907e-10
995: 2.3225217392815185726928702795122514032273358152533 +/- 6.978e-10
996: 2.323234448581462387333232260941370391258283071281 +/- 1.3574e-09
997: 2.3239447141866405142289542667580382034526509232475 +/- 2.643e-09
998: 2.324651759103270088344143240352605148856869322209 +/- 5.1524e-09
999: 2.32535482756183219112576052060526988544993162101 +/- 1.0053e-08
1000: 2.32605316168646645740560549083238185084982041782 +/- 3.927e-08
virt/peak/res/peak(MB): 170.18 294.69 7.51 7.51
```

9.4.7 logistic.c

This program computes the \(n\)-th iterate of the logistic map defined by \(x_{n+1} = r x_n (1 - x_n)\) where \(r\) and \(x_0\) are given. It takes the following parameters:

\[
\text{logistic } n \ [x_0] \ [r] \ [digits]
\]

The inputs \(x_0\), \(r\) and \(digits\) default to 0.5, 3.75 and 10 respectively. The computation is automatically restarted with doubled precision until the result is accurate to \(digits\) decimal digits.

Sample output:

```bash
506 Chapter 9. Real and complex numbers
```
> build/examples/logistic 10
Trying prec=64 bits...success!
cpu/wall(s): 0 0.001
x_10 = [0.6453672908 +/- 3.10e-11]

> build/examples/logistic 100
Trying prec=64 bits...ran out of accuracy at step 18
Trying prec=128 bits...ran out of accuracy at step 53
Trying prec=256 bits...success!
cpu/wall(s): 0 0
x_100 = [0.8882939923 +/- 1.60e-11]

> build/examples/logistic 10000
Trying prec=64 bits...ran out of accuracy at step 18
Trying prec=128 bits...ran out of accuracy at step 53
Trying prec=256 bits...ran out of accuracy at step 121
Trying prec=512 bits...ran out of accuracy at step 256
Trying prec=1024 bits...ran out of accuracy at step 525
Trying prec=2048 bits...ran out of accuracy at step 1063
Trying prec=4096 bits...ran out of accuracy at step 2139
Trying prec=8192 bits...ran out of accuracy at step 4288
Trying prec=16384 bits...ran out of accuracy at step 8584
Trying prec=32768 bits...success!
cpu/wall(s): 0.859 0.858
x_10000 = [0.8242048008 +/- 4.35e-11]

> build/examples/logistic 1234 0.1 3.99 30
Trying prec=64 bits...ran out of accuracy at step 0
Trying prec=128 bits...ran out of accuracy at step 10
Trying prec=256 bits...ran out of accuracy at step 76
Trying prec=512 bits...ran out of accuracy at step 205
Trying prec=1024 bits...ran out of accuracy at step 461
Trying prec=2048 bits...ran out of accuracy at step 974
Trying prec=4096 bits...success!
cpu/wall(s): 0.009 0.009
x_1234 = [0.2564453919585651410579677945635 +/- 3.92e-31]

9.4.8 real_roots.c

This program isolates the roots of a function on the interval \((a, b)\) (where \(a\) and \(b\) are input as double-precision literals) using the routines in the \texttt{arb} module. The program takes the following arguments:

\texttt{real_roots function a b [-refine d] [-verbose] [-maxdepth n] [-maxeval n] [-maxfound n] [-prec n]}

The following functions (specified by an integer code) are implemented:

- 0 - \(Z(x)\) (Riemann-Siegel Z-function)
- 1 - \(\sin(x)\)
- 2 - \(\sin(x^2)\)
- 3 - \(\sin(1/x)\)
- 4 - \(\text{Ai}(x)\) (Airy function)
- 5 - \(\text{Ai}'(x)\) (Airy function)
- 6 - \(\text{Bi}(x)\) (Airy function)

9.4. Arb example programs
7 - Bi′(x) (Airy function)

The following options are available:

- `-refine d`: If provided, after isolating the roots, attempt to refine the roots to \( d \) digits of accuracy using a few bisection steps followed by Newton’s method with adaptive precision, and then print them.
- `-verbose`: Print more information.
- `-maxdepth n`: Stop searching after \( n \) recursive subdivisions.
- `-maxeval n`: Stop searching after approximately \( n \) function evaluations (the actual number evaluations will be a small multiple of this).
- `-maxfound n`: Stop searching after having found \( n \) isolated roots.
- `-prec n`: Working precision to use for the root isolation.

With function 0, the program isolates roots of the Riemann zeta function on the critical line, and guarantees that no roots are missed (see `zeta-eros.c` for a far more efficient way to do this):

```
> build/examples/real_roots 0 0.0 50.0 -verbose
interval: [0, 50]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
found isolated root in: [14.111328125, 14.16015625]
found isolated root in: [20.99609375, 21.044921875]
found isolated root in: [25, 25.048828125]
found isolated root in: [30.419921875, 30.443359375]
found isolated root in: [32.91015625, 32.958984375]
found isolated root in: [37.548828125, 37.59765625]
found isolated root in: [40.91796875, 40.966796875]
found isolated root in: [43.310546875, 43.349609375]
found isolated root in: [47.998046875, 48.024609375]
found isolated root in: [49.755859375, 49.7802734375]
---------------------------------------------------------------
Found roots: 10
Subintervals possibly containing undetected roots: 0
Function evaluations: 3058
cpu/wall(s): 0.202 0.202
```

Find just one root and refine it to approximately 75 digits:

```
> build/examples/real_roots 0 0.0 50.0 -maxfound 1 -refine 75
interval: [0, 50]
maxdepth = 30, maxeval = 100000, maxfound = 1, low_prec = 30
refined root (0/8):
[14.134725141734693790457251983562470270784257115699243175685567460149963429809 +/- 2.57e-76]
---------------------------------------------------------------
Found roots: 1
Subintervals possibly containing undetected roots: 7
Function evaluations: 761
cpu/wall(s): 0.055 0.056
virt/peak/res/peak(MB): 26.12 26.14 2.75 2.75
```

Find the first few roots of an Airy function and refine them to 50 digits each:

```
> build/examples/real_roots 4 -10 0 -refine 50
interval: [-10, 0]
(continues on next page)
```
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
refined root (0/6):
[-9.022650853340980380158190839880089256524677535156083 +/- 4.85e-52]
refined root (1/6):
[-7.94413387120853123138280557926832140674396972215 +/- 1.92e-52]
refined root (2/6):
[-6.786708090071758998780246384496176966053882477393494 +/- 3.84e-52]
refined root (3/6):
[-5.520559828095551059129855512931293573797214280617525 +/- 1.05e-52]
refined root (4/6):
[-4.087949444130970616636988701457391060224764699108530 +/- 2.46e-52]
refined root (5/6):
[-2.338107410459767038489197252446735440638540145672388 +/- 1.48e-52]

Found roots: 6
Subintervals possibly containing undetected roots: 0
Function evaluations: 200
cpu/wall(s): 0.003 0.003

Find roots of \( \sin(x^2) \) on \((0,100)\). The algorithm cannot isolate the root at \( x = 0 \) (it is at the endpoint of the interval, and in any case a root of multiplicity higher than one). The failure is reported:

> build/examples/real_roots 2 0 100
interval: [0, 100]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30

---------------------------------------------------------------
Found roots: 3183
Subintervals possibly containing undetected roots: 1
Function evaluations: 34058
cpu/wall(s): 0.032 0.032

This does not miss any roots:

> build/examples/real_roots 2 1 100
interval: [1, 100]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30

---------------------------------------------------------------
Found roots: 3183
Subintervals possibly containing undetected roots: 0
Function evaluations: 34039
cpu/wall(s): 0.023 0.023
virt/peak/res/peak(MB): 26.32 26.37 2.01 2.01

Looking for roots of \( \sin(1/x) \) on \((0,1)\), the algorithm finds many roots, but will never find all of them since there are infinitely many:

> build/examples/real_roots 3 0.0 1.0
interval: [0, 1]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30

(continues on next page)
Found roots: 10198
Subintervals possibly containing undetected roots: 24695
Function evaluations: 202587
cpu/wall(s): 0.171 0.171
virt/peak/res/peak(MB): 28.39 30.38 4.05 4.05

Remark: the program always computes rigorous containing intervals for the roots, but the accuracy after refinement could be less than \(d\) digits.

### 9.4.9 poly_roots.c

This program finds the complex roots of an integer polynomial by calling `arb_fmpz_poly_complex_roots()`, which in turn calls `acb_poly_find_roots()` with increasing precision until the roots certainly have been isolated. The program takes the following arguments:

```
poly_roots [-refine d] [-print d] <poly>
```

Isolates all the complex roots of a polynomial with integer coefficients.

If `-refine d` is passed, the roots are refined to a relative tolerance better than \(10^{-d}\). By default, the roots are only computed to sufficient accuracy to isolate them. The refinement is not currently done efficiently.

If `-print d` is passed, the computed roots are printed to \(d\) decimals.

By default, the roots are not printed.

The polynomial can be specified by passing the following as `<poly>`:

- `a <n>` Easy polynomial \(1 + 2x + ... + (n+1)x^n\)
- `t <n>` Chebyshev polynomial \(T_n\)
- `u <n>` Chebyshev polynomial \(U_n\)
- `p <n>` Legendre polynomial \(P_n\)
- `c <n>` Cyclotomic polynomial \(\Phi_n\)
- `s <n>` Swinnerton-Dyer polynomial \(S_n\)
- `b <n>` Bernoulli polynomial \(B_n\)
- `w <n>` Wilkinson polynomial \(W_n\)
- `e <n>` Taylor series of \(\exp(x)\) truncated to degree \(n\)
- `m <n>` \(x^n + (100x+1)^m, n > m\)
- `coeffs <c0 c1 ... cn>` \(c_0 + c_1 x + ... + c_n x^n\)

Concatenate to multiply polynomials, e.g.: `p 5 t 6 coeffs 1 2 3` for \(P_5(x) \cdot T_6(x) \cdot (1+2x+3x^2)\)

This finds the roots of the Wilkinson polynomial with roots at the positive integers 1, 2, ..., 100:

```
> build/examples/poly_roots -print 15 w 100
computing squarefree factorization...
roots with multiplicity 1
searching for 100 roots, 100 deflated
prec=32: 0 isolated roots | cpu/wall(s): 0.098 0.098
prec=64: 0 isolated roots | cpu/wall(s): 0.247 0.247
prec=128: 0 isolated roots | cpu/wall(s): 0.498 0.497
prec=256: 0 isolated roots | cpu/wall(s): 0.713 0.713
prec=512: 100 isolated roots | cpu/wall(s): 0.104 0.105
```

(continues on next page)
This finds the roots of a Bernoulli polynomial which has both real and complex roots:

```plaintext
> build/examples/poly_roots -refine 100 -print 20 b 16
computing squarefree factorization...
cpu/wall(s): 0.001 0
roots with multiplicity 1
searching for 16 roots, 16 deflated
prec=32: 16 isolated roots | cpu/wall(s): 0.006 0.006
prec=64: 16 isolated roots | cpu/wall(s): 0.001 0.001
prec=128: 16 isolated roots | cpu/wall(s): 0.001 0.001
prec=256: 16 isolated roots | cpu/wall(s): 0.001 0.002
prec=512: 16 isolated roots | cpu/wall(s): 0.002 0.001
done!
[-0.94308706466055783383 +/- 2.02e-21]
[-0.75534059252067985752 +/- 2.70e-21]
[-0.2499975152612726002 +/- 4.27e-21]
[0.2499975152612726002 +/- 4.43e-21]
[0.75000242847487273998 +/- 4.43e-21]
[1.24999751197742101 +/- 1.43e-20]
[1.7553405925206798575 +/- 1.74e-20]
[1.9430870646605578338 +/- 3.21e-20]
[-0.99509334829229256233279 +/- 9.42e-22] + [0.44547958157103608805 +/- 3.59e-21]*I
[-0.99509334829229256233279 +/- 9.42e-22] + [-0.44547958157103608805 +/- 3.59e-21]*I
[1.9950933482922925623328 +/- 1.10e-20] + [0.44547958157103608805 +/- 3.59e-21]*I
[1.9950933482922925623328 +/- 1.10e-20] + [-0.44547958157103608805 +/- 3.59e-21]*I
[-0.9217732771429290564 +/- 4.68e-21] + [-1.0954360955079385542 +/- 1.71e-21]*I
[-0.9217732771429290564 +/- 4.68e-21] + [1.0954360955079385542 +/- 1.71e-21]*I
[1.9217732771429290564 +/- 3.54e-20] + [1.0954360955079385542 +/- 1.71e-21]*I
[1.9217732771429290564 +/- 3.54e-20] + [-1.0954360955079385542 +/- 1.71e-21]*I
cpu/wall(s): 0.011 0.012
```

Roots are automatically separated by multiplicity by performing an initial squarefree factorization:

```plaintext
> build/examples/poly_roots -print 5 p 5 p 5 t 7 coeffs 1 5 10 10 5 1
computing squarefree factorization...
cpu/wall(s): 0 0
roots with multiplicity 1
searching for 6 roots, 3 deflated
prec=32: 3 isolated roots | cpu/wall(s): 0 0.001
done!
[-0.97493 +/- 2.10e-6]
[-0.78183 +/- 1.49e-6]
```

(continues on next page)
9.4.10 zeta_zeros.c

This program finds the imaginary parts of consecutive nontrivial zeros of the Riemann zeta function by calling either `acb_dirichlet_hardy_z_zeros()` or `acb_dirichlet_platt_local_hardy_z_zeros()` depending on the height of the zeros and the number of zeros requested. The program takes the following arguments:

```
```

> build/examples/zeta_zeros -n 1048449114 -count 2
1048449114  
1048449115  
cpu/wall(s): 0.255 0.255
virt/peak/res/peak(MB): 26.77 26.77 7.88 7.88

9.4.11 complex_plot.c

This program plots one of the predefined functions over a complex interval \([x_a, x_b] + [y_a, y_b]i\) using domain coloring, at a resolution of \(xn \times yn\) pixels.

The program takes the parameters:

```
complex_plot [-range xa xb ya yb] [-size xn yn] [-color n] [-threads n] <func>
```

Defaults parameters are \([-10, 10] + [-10, 10]i\) and \(zn = yn = 512\).

A color function can be selected with -color. Valid options are 0 (phase=hue, magnitude=brightness) and 1 (phase only, white-gold-black-blue-white counterclockwise).

The output is written to `arbplot.ppm`. If you have ImageMagick, run `convert arbplot.ppm arbplot.png` to get a PNG.
Function codes <func> are:

- **gamma** - Gamma function
- **digamma** - Digamma function
- **lgamma** - Logarithmic gamma function
- **zeta** - Riemann zeta function
- **erf** - Error function
- **ai** - Airy function Ai
- **bi** - Airy function Bi
- **besselj** - Bessel function $J_0$
- **bessely** - Bessel function $Y_0$
- **besseli** - Bessel function $I_0$
- **besselk** - Bessel function $K_0$
- **modj** - Modular j-function
- **modeta** - Dedekind eta function
- **barnesg** - Barnes G-function
- **agm** - Arithmetic geometric mean

The function is just sampled at point values; no attempt is made to resolve small features by adaptive subsampling.

For example, the following plots the Riemann zeta function around a portion of the critical strip with imaginary part between 100 and 140:

```bash
> build/examples/complex_plot zeta -range -10 10 100 140 -size 256 512
```

For parallel computation on a multicore system, use `-threads n`.

### 9.4.12 lvalue.c

This program evaluates Dirichlet L-functions. It takes the following input:

```bash
> build/examples/lvalue
```

Print value of Dirichlet L-function at $s = a + bi$.

Default $a = 0.5$, $b = 0$, $p = 53$, $(q, n) = (1, 0)$ (Riemann zeta)

- `-z` - compute $Z(s)$ instead of $L(s)$
- `-deflate` - remove singular term at $s = 1$
- `-len l` - compute $l$ terms in Taylor series at $s$

Evaluating the Riemann zeta function and the Dirichlet beta function at $s = 2$:

```bash
> build/examples/lvalue -re 2 -prec 128
L(s) = [1.64493406684822643647241516664602518922 +/- 4.37e-39]
cpu/wall(s): 0.001 0.001
time/peak/res/peak(MB): 26.86 26.88 2.05 2.05

> build/examples/lvalue -character 4 3 -re 2 -prec 128
L(s) = [0.9159655941772190150546035149328411077 +/- 7.86e-39]
cpu/wall(s): 0.002 0.003
time/peak/res/peak(MB): 26.86 26.88 2.31 2.31
```
Evaluating the L-function for character number 101 modulo 1009 at \( s = 1/2 \) and \( s = 1 \):

\[
> \text{build/examples/lvalue -character 1009 101}
L(s) = [-0.459256562383872 +/- 5.24e-16] + [1.346937111206009 +/- 3.03e-16]i
\]

\[
cpu/wall(s): 0.012 0.012
\]

 virt/peak/res/peak(MB): 26.86 26.88 2.30 2.30

\[
> \text{build/examples/lvalue -character 1009 101 -re 1}
L(s) = [0.65795286112728 +/- 6.02e-16] + [1.004145273214022 +/- 3.10e-16]i
\]

\[
cpu/wall(s): 0.017 0.018
\]

 virt/peak/res/peak(MB): 26.86 26.88 2.30 2.30

Computing the first few coefficients in the Laurent series of the Riemann zeta function at \( s = 1 \):

\[
> \text{build/examples/lvalue -re 1 -deflate -len 8}
L(s) = [0.577215664901532861 +/- 5.29e-19]
L'(s) = [0.072815845483676725 +/- 2.68e-19]
[x^2] L(s+x) = [-0.004845181596436159 +/- 3.87e-19]
[x^3] L(s+x) = [-0.000342305736717224 +/- 4.20e-19]
[x^4] L(s+x) = [9.6890419394471e-5 +/- 2.40e-19]
[x^5] L(s+x) = [-6.610318108422e-6 +/- 4.51e-20]
[x^6] L(s+x) = [-3.316240908753e-7 +/- 3.85e-20]
[x^7] L(s+x) = [1.0462094584479e-7 +/- 7.78e-21]
\]

\[
cpu/wall(s): 0.003 0.004
\]

 virt/peak/res/peak(MB): 26.86 26.88 2.30 2.30

Evaluating the Riemann zeta function near the first nontrivial root:

\[
> \text{build/examples/lvalue -re 0.5 -im 14.134725}
L(s) = [1.76743e-8 +/- 1.93e-14] + [-1.110203e-7 +/- 2.84e-14]i
\]

\[
cpu/wall(s): 0.001 0.001
\]

 virt/peak/res/peak(MB): 26.86 26.88 2.31 2.31

\[
> \text{build/examples/lvalue -z -re 14.134725 -prec 200}
Z(s) = [-1.1241834983941753330011494358128257497862927935658e-7 +/- 4.62e-58]
\]

\[
cpu/wall(s): 0.001 0.001
\]

 virt/peak/res/peak(MB): 26.86 26.88 2.57 2.57

\[
> \text{build/examples/lvalue -z -re 14.134725 -len 4}
Z(s) = [-1.124184e-7 +/- 7.00e-14]
Z'(s) = [0.793160418884 +/- 4.09e-13]
[x^2] Z(s+x) = [0.065164586492 +/- 5.39e-13]
[x^3] Z(s+x) = [-0.020707762705 +/- 5.37e-13]
\]

\[
cpu/wall(s): 0.002 0.003
\]

 virt/peak/res/peak(MB): 26.86 26.88 2.57 2.57

9.4.13 lcentral.c

This program computes the central value \( L(1/2) \) for each Dirichlet L-function character modulo \( q \) for each \( q \) in the range \( q_{\text{min}} \) to \( q_{\text{max}} \). Usage:

\[
> \text{build/examples/lcentral}
\]

Computes central values (\( s = 0.5 \)) of Dirichlet L-functions.


The first few values:
> build/examples/lcentral 1 8
3,2: [0.480867557696826828621822006324 +/- 7.35e-30]
4,3: [0.66769145718960917665869092930 +/- 1.62e-30]
5,2: [0.76374788011728687822451215264 +/- 2.32e-30] + [0.21696476751886069363858659310 +/- 3.06e-30]*I
5,4: [0.2317509475040157558833866176 +/- 2.21e-30]
5,3: [0.76374788011728687822451215264 +/- 2.32e-30] + [-0.21696476751886069363858659310 +/- 3.06e-30]*I
7,3: [0.71394334376831949285993820742 +/- 1.21e-30] + [0.47490218277139938263745243935 +/- 4.52e-30]*I
7,2: [0.31008936259836766059195052534 +/- 5.29e-30] + [-0.0726419317709524562171245 +/- 5.49e-30]*I
7,6: [1.1465866669037083336712697646 +/- 1.95e-30]
7,4: [0.31008936259836766059195052534 +/- 5.29e-30] + [0.7264193177790524562171245 +/- 5.49e-30]*I
7,5: [0.71394334376831949285993820742 +/- 1.21e-30] + [-0.47490218277139938263745243935 +/- 4.52e-30]*I
8,5: [0.37369171291254730738158695300 +/- 4.01e-30]
8,3: [1.10042140952554837756713576997 +/- 3.37e-30]
cpu/wall(s): 0.002 0.003
virt/peak/res/peak(MB): 26.32 26.34 2.35 2.35

Testing a large q:

> build/examples/lcentral --quiet --check --prec 256 100000 100000
cpu/wall(s): 1.668 1.667
virt/peak/res/peak(MB): 35.67 46.66 11.67 22.61

It is conjectured that the central value never vanishes. Running with --check verifies that the interval certainly is nonzero. This can fail with insufficient precision:

> build/examples/lcentral --check --prec 15 100000 100000
100000,71877: [0.1 +/- 0.0772] + [+/- 0.136]*I
100000,90629: [2e+0 +/- 0.106] + [+/- 0.920]*I
100000,28133: [+/- 0.811] + [-2e+0 +/- 0.501]*I
100000,3141: [0.8 +/- 0.0407] + [-0.1 +/- 0.0243]*I
100000,53189: [4.0 +/- 0.0826] + [+/- 0.107]*I
100000,53253: [1.9 +/- 0.0855] + [-3.9 +/- 0.0681]*I
Value could be zero!
100000,53381: [+/- 0.0329] + [+/- 0.0413]*I
Aborted

9.4.14 integrals.c

This program computes integrals using `acb_calc_integrate()`. Invoking the program without parameters shows usage:

> build/examples/integrals
Compute integrals using acb_calc_integrate.
Usage: integrals -i n [-prec p] [-tol eps] [-twice] [...]
-heap  - use heap for subinterval queue
-verbose - show information
-verbose2 - show more information
-deg n  - use quadrature degree up to n
-eval n  - limit number of function evaluations to n
-depth n  - limit subinterval queue size to n
-threads n  - use parallel computation with n threads

Implemented integrals:
I0 = int_0^{100} \sin(x) \, dx
I1 = \frac{1}{1+100^2} \int_0^{100} \sin(x) \, dx
I2 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I3 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I4 = \int_0^{\infty} \sqrt{1-x^2} \, dx
I5 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I6 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I7 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I8 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I9 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I10 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I11 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I12 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I13 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I14 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I15 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I16 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I17 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I18 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I19 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I20 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I21 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I22 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I23 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I24 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I25 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I26 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I27 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I28 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I29 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I30 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
I31 = \int_0^{\infty} \frac{1}{1+x^2} \, dx

A few examples:

```
> build/examples/integrals -i 4
I4 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
/cpu/wall(s): 0.02 0.02
I4 = [0.34740017265725 +/- 3.95e-15]
> build/examples/integrals -i 3 -prec 333 -tol 1e-80
I3 = \int_0^{\infty} \frac{1}{1+x^2} \, dx
/cpu/wall(s): 0.024 0.024
I3 = [3.8424e-79]
```
9.4.15 fpwrap.c

This program demonstrates calling the floating-point wrapper:

```plaintext
> build/examples/fpwrap
zeta(2) = 1.644934066848226
zeta(0.5 + 123i) = 0.006252861175594465 + 0.08206030514520983i
```

9.4.16 functions_benchmark.c

This program benchmarks performance of some standard functions.

9.5 mag.h – fixed-precision unsigned floating-point numbers for bounds

The `mag_t` type holds an unsigned floating-point number with a fixed-precision mantissa (30 bits) and an arbitrary-precision exponent (represented as an `fmpz_t`), suited for representing magnitude bounds. The special values zero and positive infinity are supported, but not NaN.

Operations that involve rounding will always produce a valid upper bound, or a lower bound if the function name has the suffix `lower`. For performance reasons, no attempt is made to compute the best possible bounds: in general, a bound may be several ulps larger/smaller than the optimal bound. Some functions such as `mag_set()` and `mag_mul_2exp_si()` are always exact and therefore do not require separate `lower` versions.

A common mistake is to forget computing a lower bound for the argument of a decreasing function that is meant to be bounded from above, or vice versa. For example, to compute an upper bound for \((x + 1)/(y + 1)\), the parameter `x` should initially be an upper bound while `y` should be a lower bound, and one should do:

```plaintext
mag_add_ui(tmp1, x, 1);
mag_add_ui_lower(tmp2, y, 1);
mag_div(res, tmp1, tmp2);
```

For a lower bound of the same expression, `x` should be a lower bound while `y` should be an upper bound, and one should do:

```plaintext
mag_add_ui_lower(tmp1, x, 1);
mag_add_ui(tmp2, y, 1);
mag_div_lower(res, tmp1, tmp2);
```

Applications requiring floating-point arithmetic with more flexibility (such as correct rounding, or higher precision) should use the `arf_t` type instead. For calculations where a complex alternation between upper and lower bounds is necessary, it may be cleaner to use `arb_t` arithmetic and convert to a `mag_t` bound only in the end.
9.5.1 Types, macros and constants

type **mag_struct**
   
   A **mag_struct** holds a mantissa and an exponent. Special values are encoded by the mantissa being set to zero.


type **mag_t**
   
   A **mag_t** is defined as an array of length one of type **mag_struct**, permitting a **mag_t** to be passed by reference.

9.5.2 Memory management

void **mag_init**( **mag_t** x)
   
   Initializes the variable x for use. Its value is set to zero.

void **mag_clear**( **mag_t** x)
   
   Clears the variable x, freeing or recycling its allocated memory.

void **mag_swap**( **mag_t** x, **mag_t** y)
   
   Swaps x and y efficiently.

**mag_ptr** _**mag_vec_init**( slong n)
   
   Allocates a vector of length n. All entries are set to zero.

void _**mag_vec_clear**( **mag_ptr** v, slong n)
   
   Clears a vector of length n.

*slong** _**mag_allocated_bytes**( const **mag_t** x)
   
   Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add **sizeof(mag_struct)** to get the size of the object as a whole.

9.5.3 Special values

void **mag_zero**( **mag_t** res)
   
   Sets res to zero.

void **mag_one**( **mag_t** res)
   
   Sets res to one.

void **mag_inf**( **mag_t** res)
   
   Sets res to positive infinity.

int **mag_is_special**( const **mag_t** x)
   
   Returns nonzero iff x is zero or positive infinity.

int **mag_is_zero**( const **mag_t** x)
   
   Returns nonzero iff x is zero.

int **mag_is_inf**( const **mag_t** x)
   
   Returns nonzero iff x is positive infinity.

int **mag_is_finite**( const **mag_t** x)
   
   Returns nonzero iff x is not positive infinity (since there is no NaN value, this function is exactly the logical negation of **mag_is_inf()**).
9.5.4 Assignment and conversions

void `mag_init_set` (mag_t res, const mag_t x)
   
   Initializes res and sets it to the value of x. This operation is always exact.

void `mag_set` (mag_t res, const mag_t x)
   
   Sets res to the value of x. This operation is always exact.

void `mag_set_d` (mag_t res, double x)

void `mag_set_ui` (mag_t res, ulong x)

void `mag_set_fmpz` (mag_t res, const fmpz_t x)
   
   Sets res to an upper bound for |x|. The operation may be inexact even if x is exactly representable.

void `mag_set_d_lower` (mag_t res, double x)

void `mag_set_ui_lower` (mag_t res, ulong x)

void `mag_set_fmpz_lower` (mag_t res, const fmpz_t x)
   
   Sets res to a lower bound for |x|. The operation may be inexact even if x is exactly representable.

void `mag_set_d_2exp_fmpz` (mag_t res, double x, const fmpz_t y)

void `mag_set_fmpz_2exp_fmpz` (mag_t res, const fmpz_t x, const fmpz_t y)

void `mag_set_ui_2exp_si` (mag_t res, ulong x, slong y)
   
   Sets res to an upper bound for |x| · 2^y.

void `mag_set_d_2exp_fmpz_lower` (mag_t res, double x, const fmpz_t y)

void `mag_set_fmpz_2exp_fmpz_lower` (mag_t res, const fmpz_t x, const fmpz_t y)
   
   Sets res to a lower bound for |x| · 2^y.

double `mag_get_d` (const mag_t x)
   
   Returns a double giving an upper bound for x.

double `mag_get_d_log2_approx` (const mag_t x)
   
   Returns a double approximating \( \log_2(x) \), suitable for estimating magnitudes (warning: not a rigorous bound). The value is clamped between `COEFF_MIN` and `COEFF_MAX`.

void `mag_get_fmpq` (fmpq_t res, const mag_t x)

void `mag_get_fmpz` (fmpz_t res, const mag_t x)

void `mag_get_fmpz_lower` (fmpz_t res, const mag_t x)
   
   Sets res, respectively, to the exact rational number represented by x, the integer exactly representing the ceiling function of x, or the integer exactly representing the floor function of x.

   These functions are unsafe: the user must check in advance that x is of reasonable magnitude. If x is infinite or has a bignum exponent, an abort will be raised. If the exponent otherwise is too large or too small, the available memory could be exhausted resulting in undefined behavior.
9.5.5 Comparisons

int mag_equal(const mag_t x, const mag_t y)
Returns nonzero iff x and y have the same value.

int mag_cmp(const mag_t x, const mag_t y)
Returns negative, zero, or positive, depending on whether x is smaller, equal, or larger than y.

int mag_cmp_2exp_si(const mag_t x, slong y)
Returns negative, zero, or positive, depending on whether x is smaller, equal, or larger than $2^y$.

void mag_min(mag_t res, const mag_t x, const mag_t y)
Sets res respectively to the smaller or the larger of x and y.

9.5.6 Input and output

void mag_print(const mag_t x)
Prints x to standard output.

void mag_fprint(FILE *file, const mag_t x)
Prints x to the stream file.

char *mag_dump_str(const mag_t x)
Allocates a string and writes a binary representation of x to it that can be read by mag_load_str().
The returned string needs to be deallocated with flint_free.

int mag_load_str(mag_t x, const char *str)
Parses str into x. Returns a nonzero value if str is not formatted correctly.

int mag_dump_file(FILE *stream, const mag_t x)
Writes a binary representation of x to stream that can be read by mag_load_file(). Returns a nonzero value if the data could not be written.

int mag_load_file(mag_t x, FILE *stream)
Reads x from stream. Returns a nonzero value if the data is not formatted correctly or the read failed. Note that the data is assumed to be delimited by a whitespace or end-of-file, i.e., when writing multiple values with mag_dump_file() make sure to insert a whitespace to separate consecutive values.

9.5.7 Random generation

void mag_randtest(mag_t res, flint_rand_t state, slong expbits)
Sets res to a random finite value, with an exponent up to expbits bits large.

void mag_randtest_special(mag_t res, flint_rand_t state, slong expbits)
Like mag_randtest(), but also sometimes sets res to infinity.
9.5.8 Arithmetic

```c
void mag_add(mag_t res, const mag_t x, const mag_t y)
void mag_add_ui(mag_t res, const mag_t x, ulong y)
    Sets res to an upper bound for \(x + y\).
void mag_add_lower(mag_t res, const mag_t x, const mag_t y)
void mag_add_ui_lower(mag_t res, const mag_t x, ulong y)
    Sets res to a lower bound for \(x + y\).
void mag_add_2exp_fmpz(mag_t res, const mag_t x, const fmpz_t e)
    Sets res to an upper bound for \(x + 2^e\).
void mag_add_ui_2exp_si(mag_t res, const mag_t x, ulong y, slong e)
    Sets res to a lower bound for \(x + y2^e\).
void mag_sub(mag_t res, const mag_t x, const mag_t y)
    Sets res to an upper bound for \(\max(x - y, 0)\).
void mag_sub_lower(mag_t res, const mag_t x, const mag_t y)
    Sets res to a lower bound for \(\max(x - y, 0)\).
void mag_mul_2exp_si(mag_t res, const mag_t x, slong y)
void mag_mul_2exp_fmpz(mag_t res, const mag_t x, const fmpz_t y)
    Sets res to \(x \cdot 2^y\). This operation is exact.
void mag_mul(mag_t res, const mag_t x, const mag_t y)
void mag_mul_ui(mag_t res, const mag_t x, ulong y)
void mag_mul_fmpz(mag_t res, const mag_t x, const fmpz_t y)
    Sets res to an upper bound for \(xy\).
void mag_mul_lower(mag_t res, const mag_t x, const mag_t y)
void mag_mul_ui_lower(mag_t res, const mag_t x, ulong y)
void mag_mul_fmpz_lower(mag_t res, const mag_t x, const fmpz_t y)
    Sets res to a lower bound for \(xy\).
void mag_addmul(mag_t z, const mag_t x, const mag_t y)
    Sets \(z\) to an upper bound for \(x + xy\).
void mag_div(mag_t res, const mag_t x, const mag_t y)
void mag_div_ui(mag_t res, const mag_t x, ulong y)
void mag_div_fmpz(mag_t res, const mag_t x, const fmpz_t y)
    Sets res to an upper bound for \(x/y\).
void mag_div_lower(mag_t res, const mag_t x, const mag_t y)
    Sets res to a lower bound for \(x/y\).
void mag_inv(mag_t res, const mag_t x)
    Sets res to an upper bound for \(1/x\).
void mag_inv_lower(mag_t res, const mag_t x)
    Sets res to a lower bound for \(1/x\).
```
9.5.9 Fast, unsafe arithmetic

The following methods assume that all inputs are finite and that all exponents (in all inputs as well as the final result) fit as fmpz inline values. They also assume that the output variables do not have promoted exponents, as they will be overwritten directly (thus leaking memory).

```c
void mag_fast_init_set(mag_t x, const mag_t y)
Initialises x and sets it to the value of y.
```

```c
void mag_fast_zero(mag_t res)
Sets res to zero.
```

```c
int mag_fast_is_zero(const mag_t x)
Returns nonzero iff x is zero.
```

```c
void mag_fast_mul(mag_t res, const mag_t x, const mag_t y)
Sets res to an upper bound for \( x \times y \).
```

```c
void mag_fast_addmul(mag_t z, const mag_t x, const mag_t y)
Sets z to an upper bound for \( z + x \times y \).
```

```c
void mag_fast_add_2exp_si(mag_t res, const mag_t x, slong e)
Sets res to an upper bound for \( x + 2^e \).
```

```c
void mag_fast_mul_2exp_si(mag_t res, const mag_t x, slong e)
Sets res to an upper bound for \( x \times 2^e \).
```

9.5.10 Powers and logarithms

```c
void mag_pow_ui(mag_t res, const mag_t x, ulong e)
```

```c
void mag_pow_fmpz(mag_t res, const mag_t x, const fmpz_t e)
Sets res to an upper bound for \( x^e \).
```

```c
void mag_pow_ui_lower(mag_t res, const mag_t x, ulong e)
```

```c
void mag_pow_fmpz_lower(mag_t res, const mag_t x, const fmpz_t e)
Sets res to a lower bound for \( x^e \).
```

```c
void mag_sqrt(mag_t res, const mag_t x)
Sets res to an upper bound for \( \sqrt{x} \).
```

```c
void mag_sqrt_lower(mag_t res, const mag_t x)
Sets res to a lower bound for \( \sqrt{x} \).
```

```c
void mag_rsqrt(mag_t res, const mag_t x)
Sets res to an upper bound for \( 1/\sqrt{x} \).
```

```c
void mag_rsqrt_lower(mag_t res, const mag_t x)
Sets res to a lower bound for \( 1/\sqrt{x} \).
```

```c
void mag_hypot(mag_t res, const mag_t x, const mag_t y)
Sets res to an upper bound for \( \sqrt{x^2 + y^2} \).
```

```c
void mag_root(mag_t res, const mag_t x, ulong n)
Sets res to an upper bound for \( x^{1/n} \).
```

```c
void mag_log(mag_t res, const mag_t x)
Sets res to an upper bound for \log(\max(1, x)) \).
```
void **mag_log_lower**( **mag_t** res, const **mag_t** x)  
Sets res to a lower bound for \( \log(\max(1, x)) \).

void **mag_neg_log**( **mag_t** res, const **mag_t** x)  
Sets res to an upper bound for \(- \log(\min(1, x))\), i.e. an upper bound for \(|\log(x)|\) for \( x \leq 1 \).

void **mag_neg_log_lower**( **mag_t** res, const **mag_t** x)  
Sets res to a lower bound for \(- \log(\min(1, x))\), i.e. a lower bound for \(|\log(x)|\) for \( x \leq 1 \).

void **mag_log_ui**( **mag_t** res, ulong n)  
Sets res to an upper bound for \( \log(n) \).

void **mag_loglp**( **mag_t** res, const **mag_t** x)  
Sets res to an upper bound for \( \log(1 + x) \). The bound is computed accurately for small \( x \).

void **mag_exp**( **mag_t** res, const **mag_t** x)  
Sets res to an upper bound for \( \exp(x) \).

void **mag_exp_lower**( **mag_t** res, const **mag_t** x)  
Sets res to a lower bound for \( \exp(x) \).

void **mag_expinv**( **mag_t** res, const **mag_t** x)  
Sets res to an upper bound for \( \exp(-x) \).

void **mag_expinv_lower**( **mag_t** res, const **mag_t** x)  
Sets res to a lower bound for \( \exp(-x) \).

void **mag_expm1**( **mag_t** res, const **mag_t** x)  
Sets res to an upper bound for \( \exp(x) - 1 \). The bound is computed accurately for small \( x \).

void **mag_exp_tail**( **mag_t** res, const **mag_t** x, ulong N)  
Sets res to an upper bound for \( \sum_{k=N}^{\infty} x^k / k! \).

void **mag_binpow_uiui**( **mag_t** res, ulong m, ulong n)  
Sets res to an upper bound for \( (1 + 1/m)^n \).

void **mag_geom_series**( **mag_t** res, const **mag_t** x, ulong N)  
Sets res to an upper bound for \( \sum_{k=N}^{\infty} x^k \).

### 9.5.11 Special functions

void **mag_const_pi**( **mag_t** res)  
void **mag_const_pi_lower**( **mag_t** res)  
Sets res to an upper (respectively lower) bound for \( \pi \).

void **mag_atan**( **mag_t** res, const **mag_t** x)  
void **mag_atan_lower**( **mag_t** res, const **mag_t** x)  
Sets res to an upper (respectively lower) bound for \( \tan(x) \).

void **mag_cosh**( **mag_t** res, const **mag_t** x)  
void **mag_cosh_lower**( **mag_t** res, const **mag_t** x)  
Sets res to an upper or lower bound for \( \cosh(x) \) or \( \sinh(x) \).
void \texttt{mag\_fac\_ui}(mag\_t res, ulong n)
Sets \(res\) to an upper bound for \(n!\).

void \texttt{mag\_rfac\_ui}(mag\_t res, ulong n)
Sets \(res\) to an upper bound for \(1/n!\).

void \texttt{mag\_bin\_uiui}(mag\_t res, ulong n, ulong k)
Sets \(res\) to an upper bound for the binomial coefficient \(\binom{n}{k}\).

void \texttt{mag\_bernoulli\_div\_fac\_ui}(mag\_t res, ulong n)
Sets \(res\) to an upper bound for \(|B_n|/n!\) where \(B_n\) denotes a Bernoulli number.

void \texttt{mag\_polylog\_tail}(mag\_t res, const mag\_t z, slong s, ulong d, ulong N)
Sets \(res\) to an upper bound for
\[
\sum_{k=N}^{\infty} \frac{z^k \log^d(k)}{k^s}.
\]

The bounding strategy is described in \textit{Algorithms for polylogarithms}. Note: in applications where \(s\) in this formula may be real or complex, the user can simply substitute any convenient integer \(s'\) such that \(s' \leq \text{Re}(s)\).

void \texttt{mag\_hurwitz\_zeta\_uiui}(mag\_t res, ulong s, ulong a)
Sets \(res\) to an upper bound for \(\zeta(s, a) = \sum_{k=0}^{\infty} (k + a)^{-s}\). We use the formula
\[
\zeta(s, a) \leq \frac{1}{a^s} + \frac{1}{(s - 1)a^{s-1}}
\]
which is obtained by estimating the sum by an integral. If \(s \leq 1\) or \(a = 0\), the bound is infinite.

\section{9.6 \texttt{arf\_h} – arbitrary-precision floating-point numbers}

A variable of type \texttt{arf\_t} holds an arbitrary-precision binary floating-point number: that is, a rational number of the form \(x \cdot 2^y\) where \(x, y \in \mathbb{Z}\) and \(x\) is odd, or one of the special values zero, plus infinity, minus infinity, or NaN (not-a-number). There is currently no support for negative zero, unsigned infinity, or a NaN with a payload.

The exponent of a finite and nonzero floating-point number can be defined in different ways: for example, as the component \(y\) above, or as the unique integer \(e\) such that \(x \cdot 2^y = m \cdot 2^e\) where \(0.5 \leq |m| < 1\). The internal representation of an \texttt{arf\_t} stores the exponent in the latter format.

Except where otherwise noted, functions have the following semantics:

- Functions taking \texttt{prec} and \texttt{rnd} parameters at the end of the argument list and returning an \texttt{int} flag round the result in the output variable to \texttt{prec} bits in the direction specified by \texttt{rnd}. The return flag is 0 if the result is exact (not rounded) and 1 if the result is inexact (rounded). Correct rounding is guaranteed: the result is the floating-point number obtained by viewing the inputs as exact numbers, in principle carrying out the mathematical operation exactly, and rounding the resulting real number to the nearest representable floating-point number whose mantissa has at most the specified number of bits, in the specified direction of rounding. In particular, the error is at most 1ulp with directed rounding modes and 0.5ulp when rounding to nearest.

- Other functions perform the operation exactly.

Since exponents are bignums, overflow or underflow cannot occur.
9.6.1 Types, macros and constants

**type arf_struct**

**type arf_t**

An `arf_struct` contains four words: an `fmpz` exponent (`exp`), a `size` field tracking the number of limbs used (one bit of this field is also used for the sign of the number), and two more words. The last two words hold the value directly if there are at most two limbs, and otherwise contain one `alloc` field (tracking the total number of allocated limbs, not all of which might be used) and a pointer to the actual limbs. Thus, up to 128 bits on a 64-bit machine and 64 bits on a 32-bit machine, no space outside of the `arf_struct` is used.

An `arf_t` is defined as an array of length one of type `arf_struct`, permitting an `arf_t` to be passed by reference.

**type arf_rnd_t**

Specifies the rounding mode for the result of an approximate operation.

**ARF_RND_DOWN**

Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards zero.

**ARF_RND_UP**

Specifies that the result of an operation should be rounded to the nearest representable number in the direction away from zero.

**ARF_RND_FLOOR**

Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards minus infinity.

**ARF_RND_CEIL**

Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards plus infinity.

**ARF_RND_NEAR**

Specifies that the result of an operation should be rounded to the nearest representable number, rounding to even if there is a tie between two values.

**ARF_PREC_EXACT**

If passed as the precision parameter to a function, indicates that no rounding is to be performed.

**Warning:** use of this value is unsafe in general. It must only be passed as input under the following two conditions:

- The operation in question can inherently be viewed as an exact operation in \( \mathbb{Z}[\frac{1}{2}] \) for all possible inputs, provided that the precision is large enough. Examples include addition, multiplication, conversion from integer types to arbitrary-precision floating-point types, and evaluation of some integer-valued functions.
- The exact result of the operation will certainly fit in memory. Note that, for example, adding two numbers whose exponents are far apart can easily produce an exact result that is far too large to store in memory.

The typical use case is to work with small integer values, double precision constants, and the like. It is also useful when writing test code. If in doubt, simply try with some convenient high precision instead of using this special value, and check that the result is exact.
9.6.2 Memory management

void **arf_init**(const **arf_t** **x**)

Initializes the variable **x** for use. Its value is set to zero.

void **arf_clear**(const **arf_t** **x**)

Clears the variable **x**, freeing or recycling its allocated memory.

**slong arf_allocated_bytes**(const **arf_t** **x**)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add sizeof(arf_struct) to get the size of the object as a whole.

9.6.3 Special values

void **arf_zero**(const **arf_t** **res**)

void **arf_one**(const **arf_t** **res**)

void **arf_pos_inf**(const **arf_t** **res**)

void **arf_neg_inf**(const **arf_t** **res**)

void **arf_nan**(const **arf_t** **res**)

Sets **res** respectively to 0, 1, \(+\infty\), \(-\infty\), NaN.

**int arf_is_zero**(const **arf_t** **x**)

**int arf_is_one**(const **arf_t** **x**)

**int arf_is_pos_inf**(const **arf_t** **x**)

**int arf_is_neg_inf**(const **arf_t** **x**)

**int arf_is_nan**(const **arf_t** **x**)

Returns nonzero iff **x** respectively equals 0, 1, \(+\infty\), \(-\infty\), NaN.

**int arf_is_inf**(const **arf_t** **x**)

Returns nonzero iff **x** equals either \(+\infty\) or \(-\infty\).

**int arf_is_normal**(const **arf_t** **x**)

Returns nonzero iff **x** is a finite, nonzero floating-point value, i.e. not one of the special values 0, \(+\infty\), \(-\infty\), NaN.

**int arf_is_special**(const **arf_t** **x**)

Returns nonzero iff **x** is one of the special values 0, \(+\infty\), \(-\infty\), NaN, i.e. not a finite, nonzero floating-point value.

**int arf_is_finite**(const **arf_t** **x**)

Returns nonzero iff **x** is a finite floating-point value, i.e. not one of the values \(+\infty\), \(-\infty\), NaN. (Note that this is not equivalent to the negation of arf_is_inf(.))
9.6.4 Assignment, rounding and conversions

void \texttt{arf\_set}(arf\_t res, const arf\_t x)

void \texttt{arf\_set\_mpz}(arf\_t res, const mpz\_t x)

void \texttt{arf\_set\_fmpz}(arf\_t res, const fmpz\_t x)

void \texttt{arf\_set\_ui}(arf\_t res, ulong x)

void \texttt{arf\_set\_si}(arf\_t res, slong x)

void \texttt{arf\_set\_mpfr}(arf\_t res, const mpfr\_t x)

void \texttt{arf\_set\_d}(arf\_t res, double x)

Sets \textit{res} to the exact value of \textit{x}.

void \texttt{arf\_swap}(arf\_t x, arf\_t y)

Swaps \textit{x} and \textit{y} efficiently.

void \texttt{arf\_init\_set\_ui}(arf\_t res, ulong x)

Initializes \textit{res} and sets it to \textit{x} in a single operation.

int \texttt{arf\_set\_round}(arf\_t res, const arf\_t x, slong prec, arf\_rnd\_t rnd)

int \texttt{arf\_set\_round\_si}(arf\_t res, slong x, slong prec, arf\_rnd\_t rnd)

int \texttt{arf\_set\_round\_ui}(arf\_t res, ulong x, slong prec, arf\_rnd\_t rnd)

int \texttt{arf\_set\_round\_mpz}(arf\_t res, const mpz\_t x, slong prec, arf\_rnd\_t rnd)

int \texttt{arf\_set\_round\_fmpz}(arf\_t res, const fmpz\_t x, slong prec, arf\_rnd\_t rnd)

Sets \textit{res} to \textit{x}, rounded to \textit{prec} bits in the direction specified by \textit{rnd}.

void \texttt{arf\_set\_si\_2exp\_si}(arf\_t res, slong m, slong e)

void \texttt{arf\_set\_ui\_2exp\_si}(arf\_t res, ulong m, slong e)

void \texttt{arf\_set\_fmpz\_2exp}(arf\_t res, const fmpz\_t m, const fmpz\_t e)

Sets \textit{res} to \textit{m} \cdot 2^{\textit{e}}.

int \texttt{arf\_set\_round\_fmpz\_2exp}(arf\_t res, const fmpz\_t x, const fmpz\_t e, slong prec, arf\_rnd\_t rnd)

Sets \textit{res} to \textit{x} \cdot 2^{\textit{e}}, rounded to \textit{prec} bits in the direction specified by \textit{rnd}.

void \texttt{arf\_get\_fmpz\_2exp}(fmpz\_t m, fmpz\_t e, const arf\_t x)

Sets \textit{m} and \textit{e} to the unique integers such that \textit{x} = \textit{m} \cdot 2^{\textit{e}} and \textit{m} is odd, provided that \textit{x} is a nonzero finite fraction. If \textit{x} is zero, both \textit{m} and \textit{e} are set to zero. If \textit{x} is infinite or NaN, the result is undefined.

void \texttt{arf\_frexp}(arf\_t m, fmpz\_t e, const arf\_t x)

Writes \textit{x} as \textit{m} \cdot 2^{\textit{e}}, where 0.5 \leq |\textit{m}| < 1 if \textit{x} is a normal value. If \textit{x} is a special value, copies this to \textit{m} and sets \textit{e} to zero. Note: for the inverse operation (ldexp), use \texttt{arf\_mul\_2exp\_fmpz()}.

double \texttt{arf\_get\_d}(const arf\_t x, arf\_rnd\_t rnd)

Returns \textit{x} rounded to a double in the direction specified by \textit{rnd}. This method rounds correctly when overflowing or underflowing the double exponent range (this was not the case in an earlier version).
int \texttt{arf_get_mpfr}(mpfr\_t res, const \texttt{arf\_t} x, mpfr\_rnd\_t rnd)

Sets the MPFR variable \texttt{res} to the value of \texttt{x}. If the precision of \texttt{x} is too small to allow \texttt{res} to be represented exactly, it is rounded in the specified MPFR rounding mode. The return value (-1, 0 or 1) indicates the direction of rounding, following the convention of the MPFR library.

If \texttt{x} has an exponent too large or small to fit in the MPFR type, the result overflows to an infinity or underflows to a (signed) zero, and the corresponding MPFR exception flags are set.

int \texttt{arf_get_fmpz}(fmpz\_t res, const \texttt{arf\_t} x, \texttt{arf\_rnd\_t} rnd)

Sets \texttt{res} to \texttt{x} rounded to the nearest integer in the direction specified by \texttt{rnd}. If \texttt{rnd} is \texttt{ARF\_RND\_NEAR}, rounds to the nearest even integer in case of a tie. Returns inexact (beware: accordingly returns whether \texttt{x} is not an integer).

This method aborts if \texttt{x} is infinite or NaN, or if the exponent of \texttt{x} is so large that allocating memory for the result fails.

Warning: this method will allocate a huge amount of memory to store the result if the exponent of \texttt{x} is huge. Memory allocation could succeed even if the required space is far larger than the physical memory available on the machine, resulting in swapping. It is recommended to check that \texttt{x} is within a reasonable range before calling this method.

\texttt{slong \texttt{arf_get_si}(const \texttt{arf\_t} x, \texttt{arf\_rnd\_t} rnd)}

Returns \texttt{x} rounded to the nearest integer in the direction specified by \texttt{rnd}. If \texttt{rnd} is \texttt{ARF\_RND\_NEAR}, rounds to the nearest even integer in case of a tie. Aborts if \texttt{x} is infinite, NaN, or the value is too large to fit in a \texttt{slong}.

int \texttt{arf_get_fmpz\_fixed\_fmpz}(fmpz\_t res, const \texttt{arf\_t} x, const fmpz\_t e)

int \texttt{arf_get_fmpz\_fixed\_si}(fmpz\_t res, const \texttt{arf\_t} x, slong e)

Converts \texttt{x} to a mantissa with predetermined exponent, i.e. sets \texttt{res} to an integer \texttt{y} such that \( y \times 2^e \approx x \), truncating if necessary. Returns 0 if exact and 1 if truncation occurred.

The warnings for \texttt{arf_get_fmpz()} apply.

void \texttt{arf\_floor}(arf\_t res, const arf\_t x)

void \texttt{arf\_ceil}(arf\_t res, const arf\_t x)

Sets \texttt{res} to \lfloor \texttt{x} \rfloor \text{ and } \lceil \texttt{x} \rceil \text{ respectively. The result is always represented exactly, requiring no more bits to store than the input. To round the result to a floating-point number with a lower precision, call \texttt{arf\_set\_round()} afterwards.}

void \texttt{arf\_get\_fmpq}(fmpq\_t res, const arf\_t x)

Set \texttt{res} to the exact rational value of \texttt{x}. This method aborts if \texttt{x} is infinite or NaN, or if the exponent of \texttt{x} is so large that allocating memory for the result fails.

\textbf{9.6.5 Comparisons and bounds}

int \texttt{arf\_equal(const arf\_t x, const arf\_t y)}

int \texttt{arf\_equal\_si(const arf\_t x, slong y)}

int \texttt{arf\_equal\_ui(const arf\_t x, ulong y)}

int \texttt{arf\_equal\_d(const arf\_t x, double y)}

Returns nonzero iff \texttt{x} and \texttt{y} are exactly equal. NaN is not treated specially, i.e. NaN compares as equal to itself.

For comparison with a \texttt{double}, the values -0 and +0 are both treated as zero, and all NaN values are treated as identical.

int \texttt{arf\_cmp(const arf\_t x, const arf\_t y)}

int \texttt{arf\_cmp\_si(const arf\_t x, slong y)}
int arf_cmp_ui(const arf_t x, ulong y)

int arf_cmp_d(const arf_t x, double y)
  Returns negative, zero, or positive, depending on whether x is respectively smaller, equal, or greater compared to y. Comparison with NaN is undefined.

int arf_cmpabs(const arf_t x, const arf_t y)
int arf_cmpabs_ui(const arf_t x, ulong y)
int arf_cmpabs_d(const arf_t x, double y)

int arf_cmpabs_mag(const arf_t x, const mag_t y)
  Compares the absolute values of x and y.

int arf_cmp_2exp_si(const arf_t x, slong e)
int arf_cmpabs_2exp_si(const arf_t x, slong e)
  Compares x (respectively its absolute value) with \(2^e\).

int arf_sgn(const arf_t x)
  Returns \(-1\), 0 or \(+1\) according to the sign of x. The sign of NaN is undefined.

void arf_min(arf_t res, const arf_t a, const arf_t b)
void arf_max(arf_t res, const arf_t a, const arf_t b)
  Sets res respectively to the minimum and the maximum of a and b.

slong arf_bits(const arf_t x)
  Returns the number of bits needed to represent the absolute value of the mantissa of x, i.e. the minimum precision sufficient to represent x exactly. Returns 0 if x is a special value.

int arf_is_int(const arf_t x)
  Returns nonzero iff x is integer-valued.

int arf_is_int_2exp_si(const arf_t x, slong e)
  Returns nonzero iff x equals \(n2^e\) for some integer \(n\).

void arf_abs_bound_lt_2exp_fmpz(fmpz_t res, const arf_t x)
  Sets res to the smallest integer \(b\) such that \(|x| < 2^b\). If x is zero, infinity or NaN, the result is undefined.

void arf_abs_bound_le_2exp_fmpz(fmpz_t res, const arf_t x)
  Sets res to the smallest integer \(b\) such that \(|x| \leq 2^b\). If x is zero, infinity or NaN, the result is undefined.

slong arf_abs_bound_lt_2exp_si(const arf_t x)
  Returns the smallest integer \(b\) such that \(|x| < 2^b\), clamping the result to lie between -ARF_PREC_EXACT and ARF_PREC_EXACT inclusive. If x is zero, -ARF_PREC_EXACT is returned, and if x is infinity or NaN, ARF_PREC_EXACT is returned.

9.6.6 Magnitude functions

void arf_get_mag(mag_t res, const arf_t x)
  Sets res to an upper bound for the absolute value of x.

void arf_get_mag_lower(mag_t res, const arf_t x)
  Sets res to a lower bound for the absolute value of x.

void arf_set_mag(arf_t res, const mag_t x)
  Sets res to x. This operation is exact.
void \texttt{mag\_init\_set\_arf}(\textit{mag\_t} \texttt{res}, \textit{arf\_t} \texttt{x})
\begin{quote}
Initializes \texttt{res} and sets it to an upper bound for \texttt{x}.
\end{quote}

void \texttt{mag\_fast\_init\_set\_arf}(\textit{mag\_t} \texttt{res}, \textit{arf\_t} \texttt{x})
\begin{quote}
Initializes \texttt{res} and sets it to an upper bound for \texttt{x}. Assumes that the exponent of \texttt{res} is small (this function is unsafe).
\end{quote}

void \texttt{arf\_mag\_set\_ulp}(\textit{mag\_t} \texttt{res}, \textit{arf\_t} \texttt{x}, \textit{slong} \texttt{prec})
\begin{quote}
Sets \texttt{res} to the magnitude of the unit in the last place (ulp) of \texttt{x} at precision \texttt{prec}.
\end{quote}

void \texttt{arf\_mag\_add\_ulp}(\textit{mag\_t} \texttt{res}, \textit{mag\_t} \texttt{x}, \textit{arf\_t} \texttt{y}, \textit{slong} \texttt{prec})
\begin{quote}
Sets \texttt{res} to an upper bound for the sum of \texttt{x} and the magnitude of the unit in the last place (ulp) of \texttt{y} at precision \texttt{prec}.
\end{quote}

void \texttt{arf\_mag\_fast\_add\_ulp}(\textit{mag\_t} \texttt{res}, \textit{mag\_t} \texttt{x}, \textit{arf\_t} \texttt{y}, \textit{slong} \texttt{prec})
\begin{quote}
Sets \texttt{res} to an upper bound for the sum of \texttt{x} and the magnitude of the unit in the last place (ulp) of \texttt{y} at precision \texttt{prec}. Assumes that all exponents are small.
\end{quote}

\section*{9.6.7 Shallow assignment}

void \texttt{arf\_init\_set\_shallow}(\textit{arf\_t} \texttt{z}, \textit{arf\_t} \texttt{x})
\begin{quote}
\textbf{Initializes} \texttt{z} to a shallow copy of \texttt{x}. A shallow copy just involves copying struct data (no heap allocation is performed).

The target variable \texttt{z} may not be cleared or modified in any way (it can only be used as constant input to functions), and may not be used after \texttt{x} has been cleared. Moreover, after \texttt{x} has been assigned shallowly to \texttt{z}, no modification of \texttt{x} is permitted as \texttt{z} is in use.
\end{quote}

void \texttt{arf\_init\_neg\_shallow}(\textit{arf\_t} \texttt{z}, \textit{arf\_t} \texttt{x})
\begin{quote}
\textbf{Initializes} \texttt{z} shallowly to the negation of \texttt{x}.
\end{quote}

\section*{9.6.8 Random number generation}

void \texttt{arf\_randtest}(\textit{arf\_t} \texttt{res}, \textit{flint\_rand\_t} \texttt{state}, \textit{slong} \texttt{bits}, \textit{slong} \texttt{mag\_bits})
\begin{quote}
Generates a finite random number whose mantissa has precision at most \texttt{bits} and whose exponent has at most \texttt{mag\_bits} bits. The values are distributed non-uniformly: special bit patterns are generated with high probability in order to allow the test code to exercise corner cases.
\end{quote}

void \texttt{arf\_randtest\_not\_zero}(\textit{arf\_t} \texttt{res}, \textit{flint\_rand\_t} \texttt{state}, \textit{slong} \texttt{bits}, \textit{slong} \texttt{mag\_bits})
\begin{quote}
Identical to \texttt{arf\_randtest()}, except that zero is never produced as an output.
\end{quote}

void \texttt{arf\_randtest\_special}(\textit{arf\_t} \texttt{res}, \textit{flint\_rand\_t} \texttt{state}, \textit{slong} \texttt{bits}, \textit{slong} \texttt{mag\_bits})
\begin{quote}
Identical to \texttt{arf\_randtest()}, except that the output occasionally is set to an infinity or NaN.
\end{quote}

void \texttt{arf\_urandom}(\textit{arf\_t} \texttt{res}, \textit{flint\_rand\_t} \texttt{state}, \textit{slong} \texttt{bits}, \textit{arf\_rnd\_t} \texttt{rnd})
\begin{quote}
Sets \texttt{res} to a uniformly distributed random number in the interval \([0,1]\). The method uses rounding from integers to floats based on the rounding mode \texttt{rnd}.
9.6.9 Input and output

voidarf_debug(constarf_t x)
    Prints information about the internal representation of x.

voidarf_print(constarf_t x)
    Prints x as an integer mantissa and exponent.

voidarf_printd(constarf_t x, slong d)
    Prints x as a decimal floating-point number, rounding to d digits. Rounding is faithful (at most 1 ulp error).

char *arf_get_str(constarf_t x, slong d)
    Returns x as a decimal floating-point number, rounding to d digits. Rounding is faithful (at most 1 ulp error).

voidarf_fprint(FILE *file, constarf_t x)
    Prints x as an integer mantissa and exponent to the stream file.

voidarf_fprintd(FILE *file, constarf_t y, slong d)
    Prints x as a decimal floating-point number to the stream file, rounding to d digits. Rounding is faithful (at most 1 ulp error).

char *arf_dump_str(constarf_t x)
    Allocates a string and writes a binary representation of x to it that can be read by arf_load_str(). The returned string needs to be deallocated with flint_free.

intarf_load_str(arf_t x, const char *str)
    Parses str into x. Returns a nonzero value if str is not formatted correctly.

intarf_dump_file(FILE *stream, constarf_t x)
    Writes a binary representation of x to stream that can be read by arf_load_file(). Returns a nonzero value if the data could not be written.

intarf_load_file(arf_t x, FILE *stream)
    Reads x from stream. Returns a nonzero value if the data is not formatted correctly or the read failed. Note that the data is assumed to be delimited by a whitespace or end-of-file, i.e., when writing multiple values with arf_dump_file() make sure to insert a whitespace to separate consecutive values.

9.6.10 Addition and multiplication

voidarf_abs(arf_t res, constarf_t x)
    Sets res to the absolute value of x exactly.

voidarf_neg(arf_t res, constarf_t x)
    Sets res to −x exactly.

intarf_neg_round(arf_t res, constarf_t x, slong prec, arf_rnd_t rnd)
    Sets res to −x.

intarf_add(arf_t res, constarf_t x, constarf_t y, slong prec, arf_rnd_t rnd)

intarf_add_si(arf_t res, constarf_t x, slong y, slong prec, arf_rnd_t rnd)

intarf_add_ui(arf_t res, constarf_t x, ulong y, slong prec, arf_rnd_t rnd)

intarf_add_fmpz(arf_t res, constarf_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)
    Sets res to x + y.
int arf_add_fmpz_2exp(arf_t res, const arf_t x, const fmpz_t y, const fmpz_t e, slong prec, arf_rnd_t rnd)

Sets res to $x + y2^e$.

int arf_sub(arf_t res, const arf_t x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_sub_si(arf_t res, const arf_t x, slong y, slong prec, arf_rnd_t rnd)
int arf_sub_ui(arf_t res, const arf_t x, ulong y, slong prec, arf_rnd_t rnd)
int arf_sub_fmpz(arf_t res, const arf_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)

Sets res to $x - y$.

void arf_mul_2exp_si(arf_t res, const arf_t x, slong e)
void arf_mul_2exp_fmpz(arf_t res, const arf_t x, const fmpz_t e)

Sets res to $x2^e$ exactly.

int arf_mul(arf_t res, const arf_t x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_mul_ui(arf_t res, const arf_t x, ulong y, slong prec, arf_rnd_t rnd)
int arf_mul_si(arf_t res, const arf_t x, slong y, slong prec, arf_rnd_t rnd)
int arf_mul_mpz(arf_t res, const arf_t x, const mpz_t y, slong prec, arf_rnd_t rnd)
int arf_mul_fmpz(arf_t res, const arf_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)

Sets res to $x \cdot y$.

int arf_addmul(arf_t z, const arf_t x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_addmul_ui(arf_t z, const arf_t x, ulong y, slong prec, arf_rnd_t rnd)
int arf_addmul_si(arf_t z, const arf_t x, slong y, slong prec, arf_rnd_t rnd)
int arf_addmul_mpz(arf_t z, const arf_t x, const mpz_t y, slong prec, arf_rnd_t rnd)
int arf_addmul_fmpz(arf_t z, const arf_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)

Performs a fused multiply-add $z = z + x \cdot y$, updating $z$ in-place.

int arf_submul(arf_t z, const arf_t x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_submul_ui(arf_t z, const arf_t x, ulong y, slong prec, arf_rnd_t rnd)
int arf_submul_si(arf_t z, const arf_t x, slong y, slong prec, arf_rnd_t rnd)
int arf_submul_mpz(arf_t z, const arf_t x, const mpz_t y, slong prec, arf_rnd_t rnd)
int arf_submul_fmpz(arf_t z, const arf_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)

Performs a fused multiply-subtract $z = z - x \cdot y$, updating $z$ in-place.

int arf_fma(arf_t res, const arf_t x, const arf_t y, const arf_t z, slong prec, arf_rnd_t rnd)

Sets res to $x \cdot y + z$. This is equivalent to an addmul except that res and $z$ can be separate variables.

int arf_sosq(arf_t res, const arf_t x, const arf_t y, slong prec, arf_rnd_t rnd)

Sets res to $x^2 + y^2$, rounded to prec bits in the direction specified by rnd.
9.6.11 Summation

```c
int arf_sum(arf_t res, arf_srcptr terms, slong len, slong prec, arf_rnd_t rnd)
```

Sets `res` to the sum of the array `terms` of length `len`, rounded to `prec` bits in the direction specified by `rnd`. The sum is computed as if done without any intermediate rounding error, with only a single rounding applied to the final result. Unlike repeated calls to `arf_add()` with infinite precision, this function does not overflow if the magnitudes of the terms are far apart. Warning: this function is implemented naively, and the running time is quadratic with respect to `len` in the worst case.

9.6.12 Dot products

```c
void arf_approx_dot(arf_t res, const arf_t initial, int subtract, arf_srcptr x, slong xstep, arf_srcptr y, slong ystep, slong len, slong prec, arf_rnd_t rnd)
```

Computes an approximate dot product, with the same meaning of the parameters as `arb_dot()`. This operation is not correctly rounded: the final rounding is done in the direction `rnd` but intermediate roundings are implementation-defined.

9.6.13 Division

```c
int arf_div(arf_t res, const arf_t x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_div_ui(arf_t res, const arf_t x, ulong y, slong prec, arf_rnd_t rnd)
int arf_ui_div(arf_t res, ulong x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_div_si(arf_t res, const arf_t x, slong y, slong prec, arf_rnd_t rnd)
int arf_si_div(arf_t res, slong x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_div_fmpz(arf_t res, const arf_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)
int arf_fmpz_div(arf_t res, const fmpz_t x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_fmpz_div_fmpz(arf_t res, const fmpz_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)
```

Sets `res` to `x/y`, rounded to `prec` bits in the direction specified by `rnd`, returning nonzero iff the operation is inexact. The result is NaN if `y` is zero.

9.6.14 Square roots

```c
int arf_sqrt(arf_t res, const arf_t x, slong prec, arf_rnd_t rnd)
int arf_sqrt_ui(arf_t res, ulong x, slong prec, arf_rnd_t rnd)
int arf_sqrt_fmpz(arf_t res, const fmpz_t x, slong prec, arf_rnd_t rnd)
```

Sets `res` to \( \sqrt{x} \). The result is NaN if `x` is negative.

```c
int arf_rsqrt(arf_t res, const arf_t x, slong prec, arf_rnd_t rnd)
```

Sets `res` to \( 1/\sqrt{x} \). The result is NaN if `x` is negative, and \(+\infty\) if `x` is zero.

```c
int arf_root(arf_t res, const arf_t x, ulong k, slong prec, arf_rnd_t rnd)
```

Sets `res` to \( x^{1/k} \). The result is NaN if `x` is negative. Warning: this function is a wrapper around the MPFR root function. It gets slow and uses much memory for large `k`. Consider working with `arb_root_ui()` for large `k` instead of using this function directly.
9.6.15 Complex arithmetic

```c
int arf_complex_mul(arf_t e, arf_t f, const arf_t a, const arf_t b, const arf_t c, const arf_t d, slong prec, arf_rnd_t rnd)
int arf_complex_mul_fallback(arf_t e, arf_t f, const arf_t a, const arf_t b, const arf_t c, const arf_t d, slong prec, arf_rnd_t rnd)
```

Computes the complex product \( e + fi = (a + bi)(c + di) \), rounding both \( e \) and \( f \) correctly to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \). The first bit in the return code indicates inexactness of \( e \), and the second bit indicates inexactness of \( f \).

If any of the components \( a, b, c, d \) is zero, two real multiplications and no additions are done. This convention is used even if any other part contains an infinity or NaN, and the behavior with infinite/NaN input is defined accordingly.

The `fallback` version is implemented naively, for testing purposes. No squaring optimization is implemented.

```c
int arf_complex_sqr(arf_t e, arf_t f, const arf_t a, const arf_t b, slong prec, arf_rnd_t rnd)
```

Computes the complex square \( e + fi = (a + bi)^2 \). This function has identical semantics to `arf_complex_mul()` (with \( c = a, b = d \)), but is faster.

9.6.16 Low-level methods

```c
int _arf_get_integer_mpn(mp_ptr y, mp_srcptr xp, mp_size_t xn, slong exp)
int _arf_set_mpn_fixed(arf_t z, mp_srcptr xp, mp_size_t xn, mp_size_t fixn, int negative, slong prec, arf_rnd_t rnd)
int _arf_set_round_ui(arf_t z, ulong x, int sgnbit, slong prec, arf_rnd_t rnd)
int _arf_set_round_uiui(arf_t z, slong *fix, mp_limb_t hi, mp_limb_t lo, int sgnbit, slong prec, arf_rnd_t rnd)
int _arf_set_round_mpn(arf_t z, slong *exp_shift, mp_srcptr x, mp_size_t xn, int sgnbit, slong prec, arf_rnd_t rnd)
```

Given a floating-point number \( x \) represented by \( \text{xn} \) limbs at \( \text{xp} \) and an exponent \( \text{exp} \), writes the integer part of \( x \) to \( y \), returning whether the result is inexact. The correct number of limbs is written (no limbs are written if the integer part of \( x \) is zero). Assumes that \( \text{xp}[0] \) is nonzero and that the top bit of \( \text{xp}[\text{xn}-1] \) is set.

Sets \( z \) to the fixed-point number having \( \text{xn} \) total limbs and \( \text{fixn} \) fractional limbs, negated if \( \text{negative} \) is set, rounding \( z \) to \( \text{prec} \) bits in the direction \( \text{rnd} \) and returning whether the result is inexact. Both \( \text{xn} \) and \( \text{fixn} \) must be nonnegative and not so large that the bit shift would overflow an \( \text{slong} \), but otherwise no assumptions are made about the input.

Sets \( z \) to the integer \( x \), negated if \( \text{sgnb} \) is 1, rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \). There are no assumptions on \( x \).

Sets the mantissa of \( z \) to the two-limb mantissa given by \( hi \) and \( lo \), negated if \( \text{sgnbit} \) is 1, rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \). Requires that not both \( hi \) and \( lo \) are zero. Writes the exponent shift to \( \text{fix} \) without writing the exponent of \( z \) directly.

Sets the mantissa of \( z \) to the mantissa given by the \( \text{xn} \) limbs in \( x \), negated if \( \text{sgnbit} \) is 1, rounded to \( \text{prec} \) bits in the direction specified by \( \text{rnd} \). Returns the inexact flag. Requires that \( \text{xn} \) is positive and that the top limb of \( z \) is nonzero. If \( x \) has leading zero bits, writes the shift to \( \text{exp}_{\text{shift}} \). This method does not write the exponent of \( z \) directly. Requires that \( z \) does not point to the limbs of \( z \).
9.7 acf.h – complex floating-point numbers

9.7.1 Types, macros and constants

type acf_struct

type acf_t
   An acf_struct consists of a pair of arf_structs. An acf_t is defined as an array of length one of type acf_struct, permitting an acf_t to be passed by reference.

type acf_ptr
   Alias for acf_struct *, used for vectors of numbers.

type acf_srcptr
   Alias for const acf_struct *, used for vectors of numbers when passed as constant input to functions.

acf_realref(x)
   Macro returning a pointer to the real part of x as an arf_t.

acf_imagref(x)
   Macro returning a pointer to the imaginary part of x as an arf_t.

9.7.2 Memory management

void acf_init(acf_t x)
   Initializes the variable x for use, and sets its value to zero.

void acf_clear(acf_t x)
   Clears the variable x, freeing or recycling its allocated memory.

void acf_swap(acf_t z, acf_t x)
   Swaps z and x efficiently.

slong acf_allocated_bytes(const acf_t x)
   Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add sizeof(acf_struct) to get the size of the object as a whole.

9.7.3 Basic manipulation

arf_ptr acf_real_ptr(acf_t z)
arf_ptr acf_imag_ptr(acf_t z)
   Returns a pointer to the real or imaginary part of z.

void acf_set(acf_t z, const acf_t x)
   Sets z to the value x.

int acf_equal(const acf_t x, const acf_t y)
   Returns whether x and y are equal.
9.7.4 Arithmetic

int acf_add(acf_t res, const acf_t x, const acf_t y, slong prec, arf_rnd_t rnd)

int acf_sub(acf_t res, const acf_t x, const acf_t y, slong prec, arf_rnd_t rnd)

int acf_mul(acf_t res, const acf_t x, const acf_t y, slong prec, arf_rnd_t rnd)

Sets res to the sum, difference or product of x or y, correctly rounding the real and imaginary parts in direction rnd. The return flag has the least significant bit set if the real part is inexact, and the second least significant bit set if the imaginary part is inexact.

9.7.5 Approximate arithmetic

The following operations are not correctly rounded. The rnd parameter specifies the final direction of rounding, but intermediate roundings are implementation-defined.

void acf_approx_inv(acf_t res, const acf_t x, slong prec, arf_rnd_t rnd)

void acf_approx_div(acf_t res, const acf_t x, const acf_t y, slong prec, arf_rnd_t rnd)

void acf_approx_sqrt(acf_t res, const acf_t x, slong prec, arf_rnd_t rnd)

Computes an approximate inverse, quotient or square root.

void acf_approx_dot(acf_t res, const acf_t initial, int subtract, acf_srcptr x, slong xstep, acf_srcptr y, slong ystep, slong len, slong prec, arf_rnd_t rnd)

Computes an approximate dot product, with the same meaning of the parameters as arb_dot() .

9.8 arb.h – real numbers

An arb_t represents a ball over the real numbers, that is, an interval \([m \pm r] \equiv [m - r, m + r]\) where the midpoint \(m\) and the radius \(r\) are (extended) real numbers and \(r\) is nonnegative (possibly infinite). The result of an (approximate) operation done on arb_t variables is a ball which contains the result of the (mathematically exact) operation applied to any choice of points in the input balls. In general, the output ball is not the smallest possible.

The precision parameter passed to each function roughly indicates the precision to which calculations on the midpoint are carried out (operations on the radius are always done using a fixed, small precision.)

For arithmetic operations, the precision parameter currently simply specifies the precision of the corresponding arf_t operation. In the future, the arithmetic might be made faster by incorporating sloppy rounding (typically equivalent to a loss of 1-2 bits of effective working precision) when the result is known to be inexact (while still propagating errors rigorously, of course). Arithmetic operations done on exact input with exactly representable output are always guaranteed to produce exact output.

For more complex operations, the precision parameter indicates a minimum working precision (algorithms might allocate extra internal precision to attempt to produce an output accurate to the requested number of bits, especially when the required precision can be estimated easily, but this is not generally required).

If the precision is increased and the inputs either are exact or are computed with increased accuracy as well, the output should converge proportionally, absent any bugs. The general intended strategy for using ball arithmetic is to add a few guard bits, and then repeat the calculation as necessary with an exponentially increasing number of guard bits (Ziv’s strategy) until the result is exact enough for one’s purposes (typically the first attempt will be successful).

The following balls with an infinite or NaN component are permitted, and may be returned as output from functions.

- The ball \([+\infty \pm c]\), where \(c\) is finite, represents the point at positive infinity. Such a ball can always be replaced by \([+\infty \pm 0]\) while preserving mathematical correctness (this is currently not done automatically by the library).
The ball \([-\infty \pm c]\), where \(c\) is finite, represents the point at negative infinity. Such a ball can always be replaced by \([-\infty \pm 0]\) while preserving mathematical correctness (this is currently not done automatically by the library).

The ball \([c \pm \infty]\), where \(c\) is finite or infinite, represents the whole extended real line \([-\infty, +\infty]\). Such a ball can always be replaced by \([0 \pm \infty]\) while preserving mathematical correctness (this is currently not done automatically by the library). Note that there is no way to represent a half-infinite interval such as \([0, \infty]\).

The ball \([\text{NaN} \pm c]\), where \(c\) is finite or infinite, represents an indeterminate value (the value could be any extended real number, or it could represent a function being evaluated outside its domain of definition, for example where the result would be complex). Such an indeterminate ball can always be replaced by \([\text{NaN} \pm \infty]\) while preserving mathematical correctness (this is currently not done automatically by the library).

### 9.8.1 Types, macros and constants

type \texttt{arb\_struct}

type \texttt{arb\_t}

An \texttt{arb\_struct} consists of an \texttt{arf\_struct} (the midpoint) and a \texttt{mag\_struct} (the radius). An \texttt{arb\_t} is defined as an array of length one of type \texttt{arb\_struct}, permitting an \texttt{arb\_t} to be passed by reference.

type \texttt{arb\_ptr}

Alias for \texttt{arb\_struct \*:}, used for vectors of numbers.

type \texttt{arb\_srcptr}

Alias for \texttt{const arb\_struct \*:}, used for vectors of numbers when passed as constant input to functions.

\texttt{arb\_midref(x)}

Macro returning a pointer to the midpoint of \(x\) as an \texttt{arf\_t}.

\texttt{arb\_radref(x)}

Macro returning a pointer to the radius of \(x\) as a \texttt{mag\_t}.

### 9.8.2 Memory management

\texttt{void arb\_init(arb\_t x)}

Initializes the variable \(x\) for use. Its midpoint and radius are both set to zero.

\texttt{void arb\_clear(arb\_t x)}

Clears the variable \(x\), freeing or recycling its allocated memory.

\texttt{arb\_ptr _arb\_vec\_init(slong n)}

Returns a pointer to an array of \(n\) initialized \texttt{arb\_struct} entries.

\texttt{void _arb\_vec\_clear(arb\_ptr v, slong n)}

Clears an array of \(n\) initialized \texttt{arb\_struct} entries.

\texttt{void arb\_swap(arb\_t x, arb\_t y)}

Swaps \(x\) and \(y\) efficiently.

\texttt{slong arb\_allocated\_bytes(const arb\_t x)}

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add \texttt{sizeof(arb\_struct)} to get the size of the object as a whole.
slong _arb_vec_allocated_bytes(arb_srcptr vec, slong len)

Returns the total number of bytes allocated for this vector, i.e. the space taken up by the vector itself plus the sum of the internal heap allocation sizes for all its member elements.

double _arb_vec_estimate_allocated_bytes(slong len, slong prec)

Estimates the number of bytes that need to be allocated for a vector of len elements with prec bits of precision, including the space for internal limb data. This function returns a double to avoid overflow issues when both len and prec are large.

This is only an approximation of the physical memory that will be used by an actual vector. In practice, the space varies with the content of the numbers; for example, zeros and small integers require no internal heap allocation even if the precision is huge. The estimate assumes that exponents will not be bignums. The actual amount may also be higher or lower due to overhead in the memory allocator or overcommitment by the operating system.

### 9.8.3 Assignment and rounding

```c
void arb_set(arb_t y, const arb_t x)
void arb_set_arf(arb_t y, const arf_t x)
void arb_set_si(arb_t y, slong x)
void arb_set_ui(arb_t y, ulong x)
void arb_set_d(arb_t y, double x)
void arb_set_fmpz(arb_t y, const fmpz_t x)
void arb_set_round(arb_t y, const arb_t x, slong prec)
void arb_set_round_fmpz(arb_t y, const fmpz_t x, slong prec)
void arb_set_round_fmpz_2exp(arb_t y, const fmpz_t x, const fmpz_t e, slong prec)
void arb_set_fmpq(arb_t y, const fmpq_t x, slong prec)
int arb_set_str(arb_t res, const char *inp, slong prec)
```

Sets `y` to the value of `x` without rounding.

```c
void arb_set_fmpz_2exp(arb_t y, const fmpz_t x, const fmpz_t e)
void arb_set_round_fmpz_2exp(arb_t y, const fmpz_t x, const fmpz_t e, slong prec)
```

Sets `y` to \( x \cdot 2^e \).

Sets `y` to the value of `x`, rounded to `prec` bits in the direction towards zero.

Sets `y` to \( x \cdot 2^e \), rounded to `prec` bits in the direction towards zero.

Sets `y` to the rational number `x`, rounded to `prec` bits in the direction towards zero.

Sets `res` to the value specified by the human-readable string `inp`. The input may be a decimal floating-point literal, such as “25”, “0.001”, “7e+141” or “-3.14159e-1”, and may also consist of two such literals separated by the symbol “+/−” and optionally enclosed in brackets, e.g. “[3.25 +/− 0.0001]”, or simply “[+/− 10]” with an implicit zero midpoint. The output is rounded to `prec` bits, and if the binary-to-decimal conversion is inexact, the resulting error is added to the radius. The symbols “inf” and “nan” are recognized (a nan midpoint results in an indeterminate interval, with infinite radius).

Returns 0 if successful and nonzero if unsuccessful. If unsuccessful, the result is set to an indeterminate interval.

Returns a nice human-readable representation of `x`, with at most `n` digits of the midpoint printed.

With default flags, the output can be parsed back with `arb_set_str()`, and this is guaranteed to produce an interval containing the original interval `x`.

538 Chapter 9. Real and complex numbers
By default, the output is rounded so that the value given for the midpoint is correct up to 1 ulp (unit in the last decimal place).

If `ARB_STR_MORE` is added to `flags`, more (possibly incorrect) digits may be printed.

If `ARB_STR_NO_RADIUS` is added to `flags`, the radius is not included in the output. Unless `ARB_STR_MORE` is set, the output is rounded so that the midpoint is correct to 1 ulp. As a special case, if there are no significant digits after rounding, the result will be shown as `0e+n`, meaning that the result is between `-1e+n` and `1e+n` (following the contract that the output is correct to within one unit in the only shown digit).

By adding a multiple `m` of `ARB_STR_CONDENSE` to `flags`, strings of more than three times `m` consecutive digits are condensed, only printing the leading and trailing `m` digits along with brackets indicating the number of digits omitted (useful when computing values to extremely high precision).

### 9.8.4 Assignment of special values

```c
void arb_zero(arb_t x)
    Sets x to zero.

void arb_one(arb_t f)
    Sets x to the exact integer 1.

void arb_pos_inf(arb_t x)
    Sets x to positive infinity, with a zero radius.

void arb_neg_inf(arb_t x)
    Sets x to negative infinity, with a zero radius.

void arb_zero_pm_inf(arb_t x)
    Sets x to `[0 \pm \infty]`, representing the whole extended real line.

void arb_indeterminate(arb_t x)
    Sets x to `[NaN \pm \infty]`, representing an indeterminate result.

void arb_zero_pm_one(arb_t x)
    Sets x to the interval `[0 \pm 1]`.

void arb_unit_interval(arb_t x)
    Sets x to the interval `[0, 1]`.
```

### 9.8.5 Input and output

The `arb_print...` functions print to standard output, while `arb_fprint...` functions print to the stream `file`.

```c
void arb_print(const arb_t x)
void arb_fprint(FILE *file, const arb_t x)
    Prints the internal representation of x.

void arb_printd(const arb_t x, slong digits)
void arb_fprintd(FILE *file, const arb_t x, slong digits)
    Prints x in decimal. The printed value of the radius is not adjusted to compensate for the fact that the binary-to-decimal conversion of both the midpoint and the radius introduces additional error.

void arb_printn(const arb_t x, slong digits, ulong flags)
```
void arb_fprintf(FILE *file, const arb_t x, slong digits, ulong flags)
Prints a nice decimal representation of $x$. By default, the output shows the midpoint with a
 guaranteed error of at most one unit in the last decimal place. In addition, an explicit error bound
 is printed so that the displayed decimal interval is guaranteed to enclose $x$. See arb_get_str()
 for details.

char *arb_dump_str(const arb_t x)
Returns a serialized representation of $x$ as a null-terminated ASCII string that can be read by
arb_load_str(). The format consists of four hexadecimal integers representing the midpoint
mantissa, midpoint exponent, radius mantissa and radius exponent (with special values to indi-
cate zero, infinity and NaN values), separated by single spaces. The returned string needs to be
deallocated with flint_free.

int arb_load_str(arb_t x, const char *str)
Sets $x$ to the serialized representation given in $str$. Returns a nonzero value if $str$ is not formatted
correctly (see arb_dump_str()).

int arb_dump_file(FILE *stream, const arb_t x)
Writes a serialized ASCII representation of $x$ to $stream$ in a form that can be read by
arb_load_file(). Returns a nonzero value if the data could not be written.

int arb_load_file(arb_t x, FILE *stream)
Reads $x$ from a serialized ASCII representation in $stream$. Returns a nonzero value if the data is
not formatted correctly or the read failed. Note that the data is assumed to be delimited by a
whitespace or end-of-file, i.e., when writing multiple values with arb_dump_file() make sure to
insert a whitespace to separate consecutive values.

It is possible to serialize and deserialize a vector as follows (warning: without error handling):}

```c
fp = fopen("data.txt", "w");
for (i = 0; i < n; i++)
{
    arb_dump_file(fp, vec + i);
    fprintf(fp, "\n"); // or any whitespace character
}
fclose(fp);

fp = fopen("data.txt", "r");
for (i = 0; i < n; i++)
{
    arb_load_file(vec + i, fp);
}
fclose(fp);
```

### 9.8.6 Random number generation

void arb_randtest(arb_t x, flint_rand_t state, slong prec, slong mag_bits)
Generates a random ball. The midpoint and radius will both be finite.

void arb_randtest_exact(arb_t x, flint_rand_t state, slong prec, slong mag_bits)
Generates a random number with zero radius.

void arb_randtest_precise(arb_t x, flint_rand_t state, slong prec, slong mag_bits)
Generates a random number with radius around $2^{-\text{prec}}$ the magnitude of the midpoint.

void arb_randtest_positive(arb_t x, flint_rand_t state, slong prec, slong mag_bits)
Generates a random precise number which is guaranteed to be positive.
void `arb_randtest_wide`(arb_t x, flint_rand_t state, slong prec, slong mag_bits)
Generates a random number with midpoint and radius chosen independently, possibly giving a
very large interval.

void `arb_randtest_special`(arb_t x, flint_rand_t state, slong prec, slong mag_bits)
Generates a random interval, possibly having NaN or an infinity as the midpoint and possibly
having an infinite radius.

void `arb_get_rand_fmpq`(fmpq_t q, flint_rand_t state, const arb_t x, slong bits)
Sets q to a random rational number from the interval represented by x. A denominator is chosen
by multiplying the binary denominator of x by a random integer up to bits bits.
The outcome is undefined if the midpoint or radius of x is non-finite, or if the exponent of the
midpoint or radius is so large or small that representing the endpoints as exact rational numbers
would cause overflows.

void `arb_urandom`(arb_t x, flint_rand_t state, slong prec)
Sets x to a uniformly distributed random number in the interval [0, 1]. The method uses rounding
from integers to floats, hence the radius might not be 0.

9.8.7 Radius and interval operations

void `arb_get_mid_arb`(arb_t m, const arb_t x)
Sets m to the midpoint of x.

void `arb_get_rad_arb`(arb_t r, const arb_t x)
Sets r to the radius of x.

void `arb_add_error_arf`(arb_t x, const arf_t err)
void `arb_add_error_mag`(arb_t x, const mag_t err)
void `arb_add_error`(arb_t x, const arb_t err)
Adds the absolute value of err to the radius of x (the operation is done in-place).

void `arb_add_error_2exp_si`(arb_t x, slong e)
void `arb_add_error_2exp_fmpz`(arb_t x, const fmpz_t e)
Adds $2^e$ to the radius of x.

void `arb_union`(arb_t z, const arb_t x, const arb_t y, slong prec)
Sets z to a ball containing both x and y.

int `arb_intersection`(arb_t z, const arb_t x, const arb_t y, slong prec)
If x and y overlap according to `arb_overlaps()`, then z is set to a ball containing the intersection
of x and y and a nonzero value is returned. Otherwise zero is returned and the value of z is
undefined. If x or y contains NaN, the result is NaN.

void `arb_nonnegative_part`(arb_t res, const arb_t x)
Sets res to the intersection of x with $[0, \infty]$. If x is nonnegative, an exact copy is made. If x is finite
and contains negative numbers, an interval of the form $[r/2 \pm r/2]$ is produced, which certainly
contains no negative points. In the special case when x is strictly negative, res is set to zero.

void `arb_get_abs_ubound_arf`(arf_t u, const arb_t x, slong prec)
Sets u to the upper bound for the absolute value of x, rounded up to prec bits. If x contains NaN, the
result is NaN.

void `arb_get_abs_lbound_arf`(arf_t u, const arb_t x, slong prec)
Sets u to the lower bound for the absolute value of x, rounded down to prec bits. If x contains NaN, the
result is NaN.
void arb_get_ubound_arf(arf_t u, const arb_t x, slong prec)
    Sets u to the upper bound for the value of x, rounded up to prec bits. If x contains NaN, the result is NaN.

void arb_get_lbound_arf(arf_t u, const arb_t x, slong prec)
    Sets u to the lower bound for the value of x, rounded down to prec bits. If x contains NaN, the result is NaN.

void arb_get_mag(mag_t z, const arb_t x)
    Sets z to an upper bound for the absolute value of x. If x contains NaN, the result is positive infinity.

void arb_get_mag_lower(mag_t z, const arb_t x)
    Sets z to a lower bound for the absolute value of x. If x contains NaN, the result is zero.

void arb_get_mag_lower_nonnegative(mag_t z, const arb_t x)
    Sets z to a lower bound for the signed value of x, or zero if x overlaps with the negative half-axis. If x contains NaN, the result is zero.

void arb_set_interval_mag(arb_t x, const mag_t a, const mag_t b, slong prec)
void arb_set_interval_arf(arb_t x, const arf_t a, const arf_t b, slong prec)
void arb_set_interval_mpfr(arb_t x, const mpfr_t a, const mpfr_t b, slong prec)
    Sets x to a ball containing the interval \([a, b]\). We require that \(a \leq b\).

void arb_set_interval_neg_pos_mag(arb_t x, const mag_t a, const mag_t b, slong prec)
    Sets x to a ball containing the interval \([-a, b]\).

void arb_get_interval_arf(arf_t a, arf_t b, const arb_t x, slong prec)
void arb_get_interval_mpfr(mpfr_t a, mpfr_t b, const arb_t x)
    Constructs an interval \([a, b]\) containing the ball x. The MPFR version uses the precision of the output variables.

slong arb_rel_error_bits(const arb_t x)
    Returns the effective relative error of x measured in bits, defined as the difference between the position of the top bit in the radius and the top bit in the midpoint, plus one. The result is clamped between plus/minus ARF_PREC_EXACT.

slong arb_rel_accuracy_bits(const arb_t x)
    Returns the effective relative accuracy of x measured in bits, equal to the negative of the return value from arb_rel_error_bits().

slong arb_rel_one_accuracy_bits(const arb_t x)
    Given a ball with midpoint \(m\) and radius \(r\), returns an approximation of the relative accuracy of \(\max(1, |m|) \pm r\) measured in bits.
**slong arb_bits(const arb_t x)**

Returns the number of bits needed to represent the absolute value of the mantissa of the midpoint of \( x \), i.e. the minimum precision sufficient to represent \( x \) exactly. Returns 0 if the midpoint of \( x \) is a special value.

**void arb_trim(arb_t y, const arb_t x)**

Sets \( y \) to a trimmed copy of \( x \): rounds \( x \) to a number of bits equal to the accuracy of \( x \) (as indicated by its radius), plus a few guard bits. The resulting ball is guaranteed to contain \( x \), but is more economical if \( x \) has less than full accuracy.

**int arb_get_unique_fmpz(fmpz_t z, const arb_t x)**

If \( x \) contains a unique integer, sets \( z \) to that value and returns nonzero. Otherwise (if \( x \) represents no integers or more than one integer), returns zero.

This method aborts if there is a unique integer but that integer is so large that allocating memory for the result fails.

Warning: this method will allocate a huge amount of memory to store the result if there is a unique integer and that integer is huge. Memory allocation could succeed even if the required space is far larger than the physical memory available on the machine, resulting in swapping. It is recommended to check that the midpoint of \( x \) is within a reasonable range before calling this method.

**void arb_floor(arb_t y, const arb_t x, slong prec)**

**void arbceil(arb_t y, const arb_t x, slong prec)**

**void arb_trunc(arb_t y, const arb_t x, slong prec)**

**void arb_nint(arb_t y, const arb_t x, slong prec)**

Sets \( y \) to a ball containing respectively, \( \lfloor x \rfloor \) and \( \lceil x \rceil \), \( \text{trunc}(x) \), \( \text{nint}(x) \), with the midpoint of \( y \) rounded to at most \( \text{prec} \) bits.

**void arb_get_fmpz_mid_rad_10exp(fmpz_t mid, fmpz_t rad, fmpz_t exp, const arb_t x, slong n)**

Assuming that \( x \) is finite and not exactly zero, computes integers \( \text{mid} \), \( \text{rad} \), \( \text{exp} \) such that \( x \in [m - r, m + r] \times 10^e \) and such that the larger out of \( \text{mid} \) and \( \text{rad} \) has at least \( n \) digits plus a few guard digits. If \( x \) is infinite or exactly zero, the outputs are all set to zero.

**int arb_can_round_arf(const arb_t x, slong prec, arf_rnd_t rnd)**

**int arb_can_round_mpfr(const arb_t x, slong prec, mpfr_rnd_t rnd)**

Returns nonzero if rounding the midpoint of \( x \) to \( \text{prec} \) bits in the direction \( \text{rnd} \) is guaranteed to give the unique correctly rounded floating-point approximation for the real number represented by \( x \).

In other words, if this function returns nonzero, applying \( \text{arf_set_round()} \), or \( \text{arf_get_mpfr()} \), or \( \text{arf_get_d()} \) to the midpoint of \( x \) is guaranteed to return a correctly rounded \( \text{arf}_t \), \( \text{mpfr}_t \) (provided that \( \text{prec} \) is the precision of the output variable), or \( \text{double} \) (provided that \( \text{prec} \) is 53). Moreover, \( \text{arf_get_mpfr()} \) is guaranteed to return the correct ternary value according to MPFR semantics.

Note that the \( \text{mpfr} \) version of this function takes an MPFR rounding mode symbol as input, while the \( \text{arf} \) version takes an \( \text{arf} \) rounding mode symbol. Otherwise, the functions are identical.

This function may perform a fast, inexact test; that is, it may return zero in some cases even when correct rounding actually is possible.

To be conservative, zero is returned when \( x \) is non-finite, even if it is an “exact” infinity.
9.8.8 Comparisons

int arb_is_zero(const arb_t x)
Returns nonzero iff the midpoint and radius of x are both zero.

int arb_is_nonzero(const arb_t x)
Returns nonzero iff zero is not contained in the interval represented by x.

int arb_is_one(const arb_t f)
Returns nonzero iff x is exactly 1.

int arb_is_finite(const arb_t x)
Returns nonzero iff the midpoint and radius of x are both finite floating-point numbers, i.e. not infinities or NaN.

int arb_is_exact(const arb_t x)
Returns nonzero iff the radius of x is zero.

int arb_is_int(const arb_t x)
Returns nonzero iff x is an exact integer.

int arb_is_int_2exp_si(const arb_t x, slong e)
Returns nonzero iff x exactly equals \( n2^e \) for some integer n.

int arb_equal(const arb_t x, const arb_t y)
Returns nonzero iff x and y are equal as balls, i.e. have both the same midpoint and radius.

Note that this is not the same thing as testing whether both x and y certainly represent the same real number, unless either x or y is exact (and neither contains NaN). To test whether both operands might represent the same mathematical quantity, use arb_overlaps() or arb_contains(), depending on the circumstance.

int arb_equal_si(const arb_t x, slong y)
Returns nonzero iff x is equal to the integer y.

int arb_is_positive(const arb_t x)

int arb_is_nonnegative(const arb_t x)

int arb_is_negative(const arb_t x)

int arb_is_nonpositive(const arb_t x)
Returns nonzero iff all points p in the interval represented by x satisfy, respectively, \( p > 0, p \geq 0, p < 0, p \leq 0 \). If x contains NaN, returns zero.

int arb_overlaps(const arb_t x, const arb_t y)
Returns nonzero iff x and y have some point in common. If either x or y contains NaN, this function always returns nonzero (as a NaN could be anything, it could in particular contain any number that is included in the other operand).

int arb_contains_arf(const arb_t x, const arf_t y)

int arb_contains_fmpq(const arb_t x, const fmpq_t y)

int arb_contains_fmpz(const arb_t x, const fmpz_t y)

int arb_contains_si(const arb_t x, slong y)

int arb_contains_mpfr(const arb_t x, const mpfr_t y)
int arb_contains(const arb_t x, const arb_t y)

Returns nonzero iff the given number (or ball) y is contained in the interval represented by x.

If x contains NaN, this function always returns nonzero (as it could represent anything, and in particular could represent all the points included in y). If y contains NaN and x does not, it always returns zero.

int arb_contains_int(const arb_t x)

Returns nonzero iff the interval represented by x contains an integer.

int arb_contains_zero(const arb_t x)

int arb_contains_negative(const arb_t x)

int arb_contains_nonpositive(const arb_t x)

int arb_contains_positive(const arb_t x)

int arb_contains_nonnegative(const arb_t x)

Returns nonzero iff there is any point p in the interval represented by x satisfying, respectively, $p = 0$, $p < 0$, $p \leq 0$, $p > 0$, $p \geq 0$. If x contains NaN, returns nonzero.

int arb_contains_interior(const arb_t x, const arb_t y)

Tests if y is contained in the interior of x; that is, contained in x and not touching either endpoint.

int arb_eq(const arb_t x, const arb_t y)

int arb_ne(const arb_t x, const arb_t y)

int arb_lt(const arb_t x, const arb_t y)

int arb_le(const arb_t x, const arb_t y)

int arb_gt(const arb_t x, const arb_t y)

int arb_ge(const arb_t x, const arb_t y)

Respectively performs the comparison $x = y$, $x \neq y$, $x < y$, $x \leq y$, $x > y$, $x \geq y$ in a mathematically meaningful way. If the comparison $t (op) u$ holds for all $t \in x$ and all $u \in y$, returns 1. Otherwise, returns 0.

The balls x and y are viewed as subintervals of the extended real line. Note that balls that are formally different can compare as equal under this definition: for example, $[-\infty \pm 3] = [-\infty \pm 0]$. Also $[-\infty] \leq [\infty \pm \infty]$.

The output is always 0 if either input has NaN as midpoint.

9.8.9 Arithmetic

void arb_neg(arb_t y, const arb_t x)

void arb_neg_round(arb_t y, const arb_t x, slong prec)

Sets y to the negation of x.

void arb_abs(arb_t y, const arb_t x)

Sets y to the absolute value of x. No attempt is made to improve the interval represented by x if it contains zero.

void arb_nonnegative_abs(arb_t y, const arb_t x)

Sets y to the absolute value of x. If x is finite and it contains zero, sets y to some interval $[r \pm r]$ that contains the absolute value of x.
void arb_sgn(arb_t y, const arb_t x)
    Sets y to the sign function of x. The result is $[0 \pm 1]$ if x contains both zero and nonzero numbers.

int arb_sgn_nonzero(const arb_t x)
    Returns 1 if x is strictly positive, -1 if x is strictly negative, and 0 if x is zero or a ball containing zero so that its sign is not determined.

void arb_min(arb_t z, const arb_t x, const arb_t y, slong prec)
    Sets z respectively to the minimum and the maximum of x and y.

void arb_max(arb_t z, const arb_t x, const arb_t y, slong prec)
    Sets z = x + y, rounded to prec bits. The precision can be ARF_PREC_EXACT provided that the result fits in memory.

void arb_minmax(arb_t z1, arb_t z2, const arb_t x, const arb_t y, slong prec)
    Sets z1 and z2 respectively to the minimum and the maximum of x and y.

void arb_sub(arb_t z, const arb_t x, const arb_t y, slong prec)
void arb_sub_arf(arb_t z, const arb_t x, const arf_t y, slong prec)
void arb_sub_ui(arb_t z, const arb_t x, ulong y, slong prec)
void arb_sub_si(arb_t z, const arb_t x, slong y, slong prec)
void arb_add_fmpz(arb_t z, const arb_t x, const fmpz_t y, slong prec)
    Sets z = x + y, rounded to prec bits. The precision can be ARF_PREC_EXACT provided that the result fits in memory.

void arb_add_fmpz_2exp(arb_t z, const arb_t x, const fmpz_t m, const fmpz_t e, slong prec)
    Sets z = x + m \cdot 2^e, rounded to prec bits. The precision can be ARF_PREC_EXACT provided that the result fits in memory.

void arb_sub_fmpz(arb_t z, const arb_t x, const fmpz_t y, slong prec)
    Sets z = x - y, rounded to prec bits. The precision can be ARF_PREC_EXACT provided that the result fits in memory.

void arb_mul(arb_t z, const arb_t x, const arb_t y, slong prec)
void arb_mul_arf(arb_t z, const arb_t x, const arf_t y, slong prec)
void arb_mul_ui(arb_t z, const arb_t x, ulong y, slong prec)
void arb_mul_si(arb_t z, const arb_t x, slong y, slong prec)
void arb_mul_fmpz(arb_t z, const arb_t x, const fmpz_t y, slong prec)
    Sets z = x \cdot y, rounded to prec bits. The precision can be ARF_PREC_EXACT provided that the result fits in memory.

void arb_mul_2exp_fmpz(arb_t z, const arb_t x, const fmpz_t e)
    Sets y to x multiplied by $2^e$.  

void arb_addmul(arb_t z, const arb_t x, const arb_t y, slong prec)
void arb_addmul_arf(arb_t z, const arb_t x, const arf_t y, slong prec)
void arb_addmul_si(arb_t z, const arb_t x, slong y, slong prec)
void arb_addmul_ui(arb_t z, const arb_t x, ulong y, slong prec)
void arb_addmul_fmpz(arb_t z, const arb_t x, fmpz_t y, slong prec)

Sets $z = z + x \cdot y$, rounded to prec bits. The precision can be ARF_PREC_EXACT provided that the result fits in memory.

void arb_submul(arb_t z, const arb_t x, const arb_t y, slong prec)
void arb_submul_arf(arb_t z, const arb_t x, const arf_t y, slong prec)
void arb_submul_si(arb_t z, const arb_t x, slong y, slong prec)
void arb_submul_ui(arb_t z, const arb_t x, ulong y, slong prec)
void arb_submul_fmpz(arb_t z, const arb_t x, const fmpz_t y, slong prec)

Sets $z = z - x \cdot y$, rounded to prec bits. The precision can be ARF_PREC_EXACT provided that the result fits in memory.

void arb_fma(arb_t res, const arb_t x, const arb_t y, const arb_t z, slong prec)
void arb_fma_arf(arb_t res, const arb_t x, const arf_t y, const arb_t z, slong prec)
void arb_fma_si(arb_t res, const arb_t x, slong y, const arb_t z, slong prec)
void arb_fma_ui(arb_t res, const arb_t x, ulong y, const arb_t z, slong prec)
void arb_fma_fmpz(arb_t res, const arb_t x, const fmpz_t y, constarb_t z, slong prec)

Sets res to $x \cdot y + z$. This is equivalent to an addmul except that res and z can be separate variables.

void arb_inv(arb_t z, const arb_t x, slong prec)
Sets $z$ to $1/x$.

void arb_div(arb_t z, const arb_t x, const arb_t y, slong prec)
void arb_div_arf(arb_t z, const arb_t x, const arf_t y, slong prec)
void arb_div_si(arb_t z, const arb_t x, slong y, slong prec)
void arb_div_ui(arb_t z, const arb_t x, ulong y, slong prec)
void arb_div_fmpz(arb_t z, const arb_t x, const fmpz_t y, slong prec)

void arb_fmpz_div_fmpz(arb_t z, const fmpz_t x, const fmpz_t y, slong prec)

void arb_fma_div_fmpz(arb_t z, const fmpz_t x, const fmpz_t y, ulong n, slong prec)

Sets $z = x/(2^n - 1)$, rounded to prec bits.
9.8.10 Dot product

void arb_dot_precise(arb_t res, const arb_t s, int subtract, arb_srcptr x, slong xstep, arb_srcptr y, slong ystep, slong len, slong prec)
void arb_dot_simple(arb_t res, const arb_t s, int subtract, arb_srcptr x, slong xstep, arb_srcptr y, slong ystep, slong len, slong prec)
void arb_dot(arb_t res, const arb_t s, int subtract, arb_srcptr x, slong xstep, arb_srcptr y, slong ystep, slong len, slong prec)

Computes the dot product of the vectors $x$ and $y$, setting $res$ to $s + (-1)^{\text{subtract}} \sum_{i=0}^{len-1} x_i y_i$.

The initial term $s$ is optional and can be omitted by passing NULL (equivalently, $s = 0$). The parameter $\text{subtract}$ must be 0 or 1. The length $len$ is allowed to be negative, which is equivalent to a length of zero. The parameters $xstep$ or $ystep$ specify a step length for traversing subsequences of the vectors $x$ and $y$; either can be negative to step in the reverse direction starting from the initial pointer. Aliasing is allowed between $res$ and $s$ but not between $res$ and the entries of $x$ and $y$.

The default version determines the optimal precision for each term and performs all internal calculations using mpn arithmetic with minimal overhead. This is the preferred way to compute a dot product; it is generally much faster and more precise than a simple loop.

The simple version performs fused multiply-add operations in a simple loop. This can be used for testing purposes and is also used as a fallback by the default version when the exponents are out of range for the optimized code.

The precise version computes the dot product exactly up to the final rounding. This can be extremely slow and is only intended for testing.

void arb_approx_dot(arb_t res, const arb_t s, int subtract, arb_srcptr x, slong xstep, arb_srcptr y, slong ystep, slong len, slong prec)
Computes an approximate dot product without error bounds. The radii of the inputs are ignored (only the midpoints are read) and only the midpoint of the output is written.

void arb_dot_ui(arb_t res, const arb_t initial, int subtract, arb_srcptr x, slong xstep, const ulong *y, slong ystep, slong len, slong prec)
void arb_dot_si(arb_t res, const arb_t initial, int subtract, arb_srcptr x, slong xstep, const slong *y, slong ystep, slong len, slong prec)
void arb_dot_uiui(arb_t res, const arb_t initial, int subtract, arb_srcptr x, slong xstep, const ulong *y, slong ystep, slong len, slong prec)
void arb_dot_siui(arb_t res, const arb_t initial, int subtract, arb_srcptr x, slong xstep, const slong *y, slong ystep, slong len, slong prec)
void arb_dot_fmpz(arb_t res, const arb_t initial, int subtract, arb_srcptr x, slong xstep, const fmpz *y, slong ystep, slong len, slong prec)

Equivalent to arb_dot(), but with integers in the array $y$. The uiui and siui versions take an array of double-limb integers as input; the siui version assumes that these represent signed integers in two’s complement form.

9.8.11 Powers and roots

void arb_sqrt(arb_t z, const arb_t x, slong prec)
void arb_sqrt_arf(arb_t z, const arf_t x, slong prec)
void arb_sqrt_fmpz(arb_t z, const fmpz_t x, slong prec)
void arb_sqrt_ui(arb_t z, ulong x, slong prec)
Sets $z$ to the square root of $x$, rounded to prec bits.

If $x = m \pm \epsilon$ where $m \geq r \geq 0$, the propagated error is bounded by $\sqrt{m} - \sqrt{m - r} = \sqrt{m}(1 - \sqrt{1 - r/m}) \leq \sqrt{m}(r/m + (r/m)^2)/2$. 

548 Chapter 9. Real and complex numbers
if input interval is $[m - r, m + r]$ with $r \leq m$, the error is largest at $m - r$ where it satisfies

$$m^{1/k} - (m - r)^{1/k} = m^{1/k}[1 - (1 - r/m)^{1/k}]$$

$$= m^{1/k}[1 - \exp(\log(1 - r/m)/k)]$$

$$\leq m^{1/k} \min(1, -\log(1 - r/m)/k)$$

$$= m^{1/k} \min(1, \log(1 + r/(m - r))/k).$$

This is evaluated using \texttt{mag_log1p()}.

void \texttt{arb_root\_ui}(\texttt{arb\_t} z, \texttt{arb\_t} x, \texttt{ulong} k, \texttt{ulong} prec)

Sets $z$ to the $k$-th root of $x$, rounded to $\texttt{prec}$ bits. This function selects between different algorithms. For large $k$, it evaluates $\exp(\log(x)/k)$. For small $k$, it uses \texttt{arf\_root()} at the midpoint and computes a propagated error bound as follows: if input interval is $[m - r, m + r]$ with $r \leq m$, the error is largest at $m - r$ where it satisfies

$$m^{1/k} - (m - r)^{1/k} = m^{1/k}[1 - (1 - r/m)^{1/k}]$$

$$= m^{1/k}[1 - \exp(\log(1 - r/m)/k)]$$

$$\leq m^{1/k} \min(1, -\log(1 - r/m)/k)$$

$$= m^{1/k} \min(1, \log(1 + r/(m - r))/k).$$

This is evaluated using \texttt{mag_log1p()}.

void \texttt{arb\_root}(\texttt{arb\_t} z, \texttt{arb\_t} x, \texttt{ulong} k, \texttt{ulong} prec)

Alias for \texttt{arb\_root\_ui()}, provided for backwards compatibility.

void \texttt{arb\_sqr}(\texttt{arb\_t} y, \texttt{arb\_t} x, \texttt{slong} prec)

Sets $y$ to be the square of $x$.

void \texttt{arb\_pow\_fmpz\_binexp}(\texttt{arb\_t} y, \texttt{arb\_t} b, \texttt{fmpz\_t} e, \texttt{slong} prec)

void \texttt{arb\_pow\_fmpz}(\texttt{arb\_t} y, \texttt{arb\_t} b, \texttt{fmpz\_t} e, \texttt{slong} prec)

void \texttt{arb\_pow\_ui}(\texttt{arb\_t} y, \texttt{arb\_t} b, \texttt{ulong} e, \texttt{ulong} prec)

void \texttt{arb\_ui\_pow\_ui}(\texttt{arb\_t} y, \texttt{ulong} b, \texttt{ulong} e, \texttt{ulong} prec)

void \texttt{arb\_si\_pow\_ui}(\texttt{arb\_t} y, \texttt{slong} b, \texttt{ulong} e, \texttt{slong} prec)

Sets $y = b^e$ using binary exponentiation (with an initial division if $e < 0$). Provided that $b$ and $e$ are small enough and the exponent is positive, the exact power can be computed by setting the precision to \texttt{ARF\_PREC\_EXACT}.

Note that these functions can get slow if the exponent is extremely large (in such cases \texttt{arb\_pow()} may be superior).

void \texttt{arb\_pow\_fmpq}(\texttt{arb\_t} y, \texttt{arb\_t} x, \texttt{fmpq\_t} a, \texttt{slong} prec)

Sets $y = b^e$, computed as $y = (b^{1/q})^q$ if the denominator of $e = p/q$ is small, and generally as $y = \exp(e \log b)$.

Note that this function can get slow if the exponent is extremely large (in such cases \texttt{arb\_pow()} may be superior).

void \texttt{arb\_pow}(\texttt{arb\_t} z, \texttt{arb\_t} x, \texttt{arb\_t} y, \texttt{slong} prec)

Sets $z = x^y$, computed using binary exponentiation if $y$ is a small exact integer, as $z = (x^{1/2})^{2y}$ if $y$ is a small exact half-integer, and generally as $z = \exp(y \log x)$, except giving the obvious finite result if $x$ is $a \pm a$ and $y$ is positive.
9.8.12 Exponentials and logarithms

void \texttt{arb_log_ui}(arb t z, ulong x, slong prec)
void \texttt{arb_log_fmpz}(arb t z, const fmpz t x, slong prec)
void \texttt{arb_log_arf}(arb t z, const arf t x, slong prec)

void \texttt{arb_log}(arb t z, const arb t x, slong prec)

Sets \( z = \log(x) \).

At low to medium precision (up to about 4096 bits), \texttt{arb_log_arf()} uses table-based argument reduction and fast Taylor series evaluation via \texttt{arb_atan_taylor_rs()}. At high precision, it falls back to MPFR. The function \texttt{arb_log()} simply calls \texttt{arb_log_arf()} with the midpoint as input, and separately adds the propagated error.

void \texttt{arb_log_ui_from_prev}(arb t log k1, ulong k1, arb t log k0, ulong k0, slong prec)

Computes \( \log(k_1) \), given \( \log(k_0) \) where \( k_0 < k_1 \). At high precision, this function uses the formula \( \log(k_1) = \log(k_0) + 2 \text{atanh}(\frac{k_1 - k_0}{k_1 + k_0}) \), evaluating the inverse hyperbolic tangent using binary splitting (for best efficiency, \( k_0 \) should be large and \( k_1 - k_0 \) should be small). Otherwise, it ignores \( \log(k_0) \) and evaluates the logarithm the usual way.

void \texttt{arb_log1p}(arb t z, const arb t x, slong prec)

Sets \( z = \log(1 + x) \), computed accurately when \( x \approx 0 \).

void \texttt{arb_log_base_ui}(arb t res, const arb t x, ulong b, slong prec)

Sets \( \text{res} \) to \( \log_b(x) \). The result is computed exactly when possible.

void \texttt{arb_log_hypot}(arb t res, const arb t x, const arb t y, slong prec)

Sets \( \text{res} \) to \( \log(\sqrt{x^2 + y^2}) \).

void \texttt{arb_exp}(arb t z, const arb t x, slong prec)

Sets \( z = \exp(x) \). Error propagation is done using the following rule: assuming \( x = m \pm r \), the error is largest at \( m + r \), and we have \( \exp(m + r) - \exp(m) = \exp(m)(\exp(r) - 1) \leq r \exp(m + r) \).

void \texttt{arb_expml}(arb t z, const arb t x, slong prec)

Sets \( z = \exp(x) - 1 \), using a more accurate method when \( x \approx 0 \).

void \texttt{arb_exp_invexp}(arb t z, arb t w, const arb t x, slong prec)

Sets \( z = \exp(x) \) and \( w = \exp(-x) \). The second exponential is computed from the first using a division, but propagated error bounds are computed separately.

9.8.13 Trigonometric functions

void \texttt{arb_sin}(arb t s, const arb t x, slong prec)
void \texttt{arb_cos}(arb t c, const arb t x, slong prec)
void \texttt{arb_sin_cos}(arb t s, arb t c, const arb t x, slong prec)

Sets \( s = \sin(x), c = \cos(x) \).

void \texttt{arb_sin_pi}(arb t s, const arb t x, slong prec)
void \texttt{arb_cos_pi}(arb t c, const arb t x, slong prec)
void \texttt{arb_sin_cos_pi}(arb t s, arb t c, const arb t x, slong prec)

Sets \( s = \sin(\pi x), c = \cos(\pi x) \).

void \texttt{arb_tan}(arb t y, const arb t x, slong prec)

Sets \( y = \tan(x) = \sin(x)/\cos(y) \).
Sets \( y = \cot(x) = \cos(x) / \sin(y) \).

Sets \( s = \sin(\pi x), c = \cos(\pi x) \) where \( x \) is a rational number (whose numerator and denominator are assumed to be reduced). We first use trigonometric symmetries to reduce the argument to the octant \([0, 1/4]\). Then we either multiply by a numerical approximation of \( \pi \) and evaluate the trigonometric function the usual way, or we use algebraic methods, depending on which is estimated to be faster. Since the argument has been reduced to the first octant, the first of these two methods gives full accuracy even if the original argument is close to some root other the origin.

Sets \( z = \tan(\pi x) \).

Sets \( z = \csc(\pi x) = 1 / \sin(\pi x) \).

Sets \( z =\) sinc\((\pi x) = \sin(\pi x) / (\pi x)\).

At low to medium precision (up to about 4096 bits), \texttt{arb_atan_arf()} uses table-based argument reduction and fast Taylor series evaluation via \_\texttt{arb_atan_taylor_rs().} At high precision, it falls back to MPFR. The function \texttt{arb_atan()} simply calls \texttt{arb_atan_arf()} with the midpoint as input, and separately adds the propagated error.

Sets \( z = \) asin\((x) = \tan(x) / \sqrt{1 - x^2}\). If \( x \) is not contained in the domain \([-1, 1]\), the result is an indeterminate interval.

Sets \( z = \) acos\((x) = \pi/2 - \) asin\((x)\). If \( x \) is not contained in the domain \([-1, 1]\), the result is an indeterminate interval.
9.8.15 Hyperbolic functions

```c
void arb_sinh(arb_t s, const arb_t x, slong prec)
void arb_cosh(arb_t c, const arb_t x, slong prec)
void arb_sinh_cosh(arb_t s, arb_t c, const arb_t x, slong prec)
```

Sets \( s = \sinh(x) \), \( c = \cosh(x) \). If the midpoint of \( x \) is close to zero and the hyperbolic sine is to be computed, evaluates \((e^{2x} - 1)/(2e^x)\) via \texttt{arb_expm1()} to avoid loss of accuracy. Otherwise evaluates \((e^x \pm 1)/(2e^x)\).

```c
void arb_tanh(arb_t y, const arb_t x, slong prec)
```

Sets \( y = \tanh(x) = \sinh(x)/\cosh(x) \), evaluated via \texttt{arb_expm1()} as \( \tanh(x) = (e^{2x} - 1)/(e^{2x} + 1) \) if \(|x|\) is small, and as \( \tanh(\pm x) = 1 - 2e^{\mp 2x}/(1 + e^{\mp 2x}) \) if \(|x|\) is large.

```c
void arb_coth(arb_t y, const arb_t x, slong prec)
```

Sets \( y = \coth(x) = \cosh(x)/\sinh(x) \), evaluated using the same strategy as \texttt{arb_tanh()}.

```c
void arb_sech(arb_t res, const arb_t x, slong prec)
void arb_csch(arb_t res, const arb_t x, slong prec)
```

Computes \( \sech(x) = 1/\cosh(x) \).
Computes \( \csch(x) = 1/\sinh(x) \).

9.8.16 Inverse hyperbolic functions

```c
void arb_atanh(arb_t z, const arb_t x, slong prec)
void arb_asinh(arb_t z, const arb_t x, slong prec)
void arb_acosh(arb_t z, const arb_t x, slong prec)
```

Sets \( z = \tanh^{-1}(x) \).
Sets \( z = \sinh^{-1}(x) \).
Sets \( z = \cosh^{-1}(x) \). If \( x < 1 \), the result is an indeterminate interval.

9.8.17 Constants

The following functions cache the computed values to speed up repeated calls at the same or lower precision. For further implementation details, see \textit{Algorithms for mathematical constants}.

```c
void arb_const_pi(arb_t z, slong prec)
void arb_const_sqrt_pi(arb_t z, slong prec)
void arb_const_log_sqrt2pi(arb_t z, slong prec)
void arb_const_log2(arb_t z, slong prec)
void arb_const_log10(arb_t z, slong prec)
void arb_const_euler(arb_t z, slong prec)
```

Computes \( \pi \).
Computes \( \sqrt{\pi} \).
Computes \( \log \sqrt{2\pi} \).
Computes \( \log(2) \).
Computes \( \log(10) \).
Computes Euler’s constant \( \gamma = \lim_{k \to \infty} (H_k - \log k) \) where \( H_k = 1 + 1/2 + \ldots + 1/k \).
void \texttt{arb\_const\_catalan}(\texttt{arb\_t z}, \texttt{slong prec})

Computes Catalan’s constant \(C = \sum_{n=0}^{\infty} (-1)^n/(2n + 1)^2\).

void \texttt{arb\_const\_e}(\texttt{arb\_t z}, \texttt{slong prec})

Computes \(e = \exp(1)\).

void \texttt{arb\_const\_khinchin}(\texttt{arb\_t z}, \texttt{slong prec})

Computes Khinchin’s constant \(K_0\).

void \texttt{arb\_const\_glaisher}(\texttt{arb\_t z}, \texttt{slong prec})

Computes the Glaisher-Kinkelin constant \(A = \exp(1/12 - \zeta’(-1))\).

void \texttt{arb\_const\_apery}(\texttt{arb\_t z}, \texttt{slong prec})

Computes Apery’s constant \(\zeta(3)\).

void \texttt{arb\_const\_reciprocal\_fibonacci}(\texttt{arb\_t z}, \texttt{slong prec})

Computes the reciprocal Fibonacci constant \(\sum_{n=1}^{\infty} 1/F_n\).

9.8.18 Lambert W function

void \texttt{arb\_lambertw}(\texttt{arb\_t res}, \texttt{const arb\_t x}, \texttt{int flags}, \texttt{slong prec})

Computes the Lambert W function, which solves the equation \(we^w = x\).

The Lambert W function has infinitely many complex branches \(W_k(x)\), two of which are real on a part of the real line. The principal branch \(W_0(x)\) is selected by setting \texttt{flags} to 0, and the \(W_{-1}\) branch is selected by setting \texttt{flags} to 1. The principal branch is real-valued for \(x \geq -1/e\) (taking values in \([-1, +\infty)\)) and the \(W_{-1}\) branch is real-valued for \(-1/e \leq x < 0\) and takes values in \((-\infty, -1]\). Elsewhere, the Lambert W function is complex and \texttt{arb\_lambertw()} should be used.

The implementation first computes a floating-point approximation heuristically and then computes a rigorously certified enclosure around this approximation. Some asymptotic cases are handled specially. The algorithm used to compute the Lambert W function is described in [Joh2017b], which follows the main ideas in [CGHK1996].

9.8.19 Gamma function and factorials

void \texttt{arb\_rising\_ui}(\texttt{arb\_t z}, \texttt{const arb\_t x}, \texttt{ulong n}, \texttt{slong prec})

void \texttt{arb\_rising}(\texttt{arb\_t z}, \texttt{const arb\_t x}, \texttt{const arb\_t n}, \texttt{slong prec})

Computes the rising factorial \(z = x(x+1)(x+2)\cdots(x+n-1)\). These functions are aliases for \texttt{arb\_hypgeom\_rising\_ui()} and \texttt{arb\_hypgeom\_rising()}. 

void \texttt{arb\_rising\_fmpq\_ui}(\texttt{arb\_t z}, \texttt{const fmpq\_t x}, \texttt{ulong n}, \texttt{slong prec})

Computes the rising factorial \(z = x(x+1)(x+2)\cdots(x+n-1)\) using binary splitting. If the denominator or numerator of \(x\) is large compared to \texttt{prec}, it is more efficient to convert \(x\) to an approximation and use \texttt{arb\_rising\_ui()}. 

void \texttt{arb\_rising2\_ui}(\texttt{arb\_t u}, \texttt{arb\_t v}, \texttt{const arb\_t x}, \texttt{ulong n}, \texttt{slong prec})

Letting \(u(x) = x(x+1)(x+2)\cdots(x+n-1)\), simultaneously compute \(u(x)\) and \(v(x) = u’(x)\). This function is a wrapper of \texttt{arb\_hypgeom\_rising\_ui\_jet()}. 

void \texttt{arb\_fac\_ui}(\texttt{arb\_t z}, \texttt{ulong n}, \texttt{slong prec})

Computes the factorial \(z = n!\) via the gamma function. 

void \texttt{arb\_doublefac\_ui}(\texttt{arb\_t z}, \texttt{ulong n}, \texttt{slong prec})

Computes the double factorial \(z = n!!\) via the gamma function. 

void \texttt{arb\_bin\_ui}(\texttt{arb\_t z}, \texttt{const arb\_t n}, \texttt{ulong k}, \texttt{slong prec})
void arb_bin_uiui(arb_t z, ulong n, ulong k, slong prec)
Computes the binomial coefficient \( z = \binom{n}{k} \), via the rising factorial as \( \binom{n}{k} = (n-k+1)_k/k! \).

void arb_gamma(arb_t z, const arb_t x, ulong prec)
Computes the logarithmic gamma function \( z = \log \Gamma(x) \). The complex branch structure is assumed, so if \( x \leq 0 \), the result is an indeterminate interval. This function is an alias for arb_hypgeom_lgamma().

void arb_rgamma(arb_t z, const arb_t x, ulong prec)
Computes the reciprocal gamma function \( z = 1/\Gamma(x) \), avoiding division by zero at the poles of the gamma function. This function is an alias for arb_hypgeom_rgamma().

void arb_digamma(arb_t y, const arb_t x, slong prec)
Computes the digamma function \( z = \psi(x) = (\log \Gamma(x))' = \Gamma'(x)/\Gamma(x) \).

9.8.20 Zeta function

void arb_zeta_ui_vec_borwein(arb_ptr z, ulong start, slong num, ulong step, slong prec)
Evaluates \( \zeta(s) \) at num consecutive integers \( s \) beginning with start and proceeding in increments of step. Uses Borwein’s formula ([Bor2000], [GS2003]), implemented to support fast multi-evaluation (but also works well for a single \( s \)).

Requires start \( \geq 2 \). For efficiency, the largest \( s \) should be at most about as large as \( \text{prec} \). Arguments approaching \( \text{LONG\_MAX} \) will cause overflows. One should therefore only use this function for \( s \) up to about \( \text{prec} \), and then switch to the Euler product.

The algorithm for single \( s \) is basically identical to the one used in MPFR (see [MPFR2012] for a detailed description). In particular, we evaluate the sum backwards to avoid storing more than one \( d_k \) coefficient, and use integer arithmetic throughout since it is convenient and the terms turn out to be slightly larger than \( 2^{\text{prec}} \). The only numerical error in the main loop comes from the division by \( k^s \), which adds less than 1 unit of error per term. For fast multi-evaluation, we repeatedly divide by \( k^{\text{step}} \). Each division reduces the input error and adds at most 1 unit of additional rounding error, so by induction, the error per term is always smaller than 2 units.

void arb_zeta_ui_asym(arb_t x, ulong s, slong prec)
void arb_zeta_ui_euler_product(arb_t z, ulong s, slong prec)
Computes \( \zeta(s) \) using the Euler product. This is fast only if \( s \) is large compared to the precision. Both methods are trivial wrappers for _acb_dirichlet_euler_product_real_ui().

void arb_zeta_ui_beroulli(arb_t x, ulong s, slong prec)
Computes \( \zeta(s) \) for even \( s \) via the corresponding Bernoulli number.

void arb_zeta_ui_borwein_bsplit(arb_t x, ulong s, slong prec)
Computes \( \zeta(s) \) for arbitrary \( s \geq 2 \) using a binary splitting implementation of Borwein’s algorithm. This has quasilinear complexity with respect to the precision (assuming that \( s \) is fixed).

void arb_zeta_ui_vec(arb_ptr x, ulong start, slong num, slong prec)
void arb_zeta_ui_vec_even(arb_ptr x, ulong start, slong num, slong prec)
void arb_zeta_ui_vec_odd(arb_ptr x, ulong start, slong num, slong prec)
    Computes $\zeta(s)$ at num consecutive integers (respectively num even or num odd integers) beginning with $s = start \geq 2$, automatically choosing an appropriate algorithm.

void arb_zeta_ui(arb_t x, ulong s, slong prec)
    Computes $\zeta(s)$ for nonnegative integer $s \neq 1$, automatically choosing an appropriate algorithm. This function is intended for numerical evaluation of isolated zeta values; for multi-evaluation, the vector versions are more efficient.

void arb_zeta(arb_t z, const arb_t s, slong prec)
    Sets $z$ to the value of the Riemann zeta function $\zeta(s)$.
    For computing derivatives with respect to $s$, use arb_poly_zeta_series().

void arb_hurwitz_zeta(arb_t z, const arb_t s, const arb_t a, slong prec)
    Sets $z$ to the value of the Hurwitz zeta function $\zeta(s,a)$.
    For computing derivatives with respect to $s$, use arb_poly_zeta_series().

9.8.21 Bernoulli numbers and polynomials

void arb_bernoulli_ui(arb_t b, ulong n, slong prec)

void arb_bernoulli_fmpz(arb_t b, const fmpz_t n, slong prec)
    Sets $b$ to the numerical value of the Bernoulli number $B_n$ approximated to $prec$ bits.
    The internal precision is increased automatically to give an accurate result. Note that, with huge fmpz input, the output will have a huge exponent and evaluation will accordingly be slower.

    A single division from the exact fraction of $B_n$ is used if this value is in the global cache or the exact numerator roughly is larger than $prec$ bits. Otherwise, the Riemann zeta function is used (see arb_bernoulli_ui_zeta()).

    This function reads $B_n$ from the global cache if the number is already cached, but does not automatically extend the cache by itself.

void arb_bernoulli_ui_zeta(arb_t b, ulong n, slong prec)
    Sets $b$ to the numerical value of $B_n$ accurate to $prec$ bits, computed using the formula $B_{2n} = \frac{(-1)^n+1\cdot2^{2n}}{2\pi^n}\zeta(2n)$.

    To avoid potential infinite recursion, we explicitly call the Euler product implementation of the zeta function. This method will only give high accuracy if the precision is small enough compared to $n$ for the Euler product to converge rapidly.

void arb_bernoulli_poly_ui(arb_t res, ulong n, const arb_t x, slong prec)
    Sets res to the value of the Bernoulli polynomial $B_n(x)$.
    Warning: this function is only fast if either $n$ or $x$ is a small integer.

    This function reads Bernoulli numbers from the global cache if they are already cached, but does not automatically extend the cache by itself.

void arb_power_sum_vec(arb_ptr res, const arb_t a, const arb_t b, slong len, slong prec)
    For $n$ from 0 to len - 1, sets entry $n$ in the output vector res to
    \[ S_n(a,b) = \frac{1}{n+1} (B_{n+1}(b) - B_{n+1}(a)) \]

    where $B_n(x)$ is a Bernoulli polynomial. If $a$ and $b$ are integers and $b \geq a$, this is equivalent to
    \[ S_n(a,b) = \sum_{k=a}^{b-1} k^n. \]

    The computation uses the generating function for Bernoulli polynomials.
9.8.22 Polylogarithms

void \texttt{arb\_polylog}(\texttt{arb\_t w}, \texttt{const arb\_t s}, \texttt{const arb\_t z}, \texttt{slong prec})

void \texttt{arb\_polylog\_si}(\texttt{arb\_t w}, \texttt{slong s}, \texttt{const arb\_t z}, \texttt{slong prec})

Sets \(w\) to the polylogarithm \(\text{Li}_s(z)\).

9.8.23 Other special functions

void \texttt{arb\_fib\_fmpz}(\texttt{arb\_t z}, \texttt{const fmpz\_t n}, \texttt{slong prec})

void \texttt{arb\_fib\_ui}(\texttt{arb\_t z}, \texttt{ulong n}, \texttt{slong prec})

Computes the Fibonacci number \(F_n\) using binary squaring.

void \texttt{arb\_agm}(\texttt{arb\_t z}, \texttt{const arb\_t x}, \texttt{const arb\_t y}, \texttt{slong prec})

Sets \(z\) to the arithmetic-geometric mean of \(x\) and \(y\).

void \texttt{arb\_chebyshev\_t\_ui}(\texttt{arb\_t a}, \texttt{ulong n}, \texttt{const arb\_t x}, \texttt{slong prec})

void \texttt{arb\_chebyshev\_u\_ui}(\texttt{arb\_t a}, \texttt{ulong n}, \texttt{const arb\_t x}, \texttt{slong prec})

Evaluates the Chebyshev polynomial of the first kind \(a = T_n(x)\) or the Chebyshev polynomial of the second kind \(a = U_n(x)\).

void \texttt{arb\_chebyshev\_t\_2\_ui}(\texttt{arb\_t a}, \texttt{arb\_t b}, \texttt{ulong n}, \texttt{const arb\_t x}, \texttt{slong prec})

void \texttt{arb\_chebyshev\_u\_2\_ui}(\texttt{arb\_t a}, \texttt{arb\_t b}, \texttt{ulong n}, \texttt{const arb\_t x}, \texttt{slong prec})

Simultaneously evaluates \(a = T_n(x), b = T_{n-1}(x)\) or \(a = U_n(x), b = U_{n-1}(x)\). Aliasing between \(a\), \(b\) and \(x\) is not permitted.

void \texttt{arb\_bell\_sum\_bsplit}(\texttt{arb\_t res}, \texttt{const fmpz\_t n}, \texttt{const fmpz\_t a}, \texttt{const fmpz\_t b}, \texttt{const fmpz\_t mmag}, \texttt{slong prec})

void \texttt{arb\_bell\_sum\_taylor}(\texttt{arb\_t res}, \texttt{const fmpz\_t n}, \texttt{const fmpz\_t a}, \texttt{const fmpz\_t b}, \texttt{const fmpz\_t mmag}, \texttt{slong prec})

Helper functions for Bell numbers, evaluating the sum \(\sum_{k=a}^{b-1} k^n/k!\). If \(mmag\) is non-NULL, it may be used to indicate that the target error tolerance should be \(2^{mmag-\text{prec}}\).

void \texttt{arb\_bell\_fmpz}(\texttt{arb\_t res}, \texttt{const fmpz\_t n}, \texttt{slong prec})

void \texttt{arb\_bell\_ui}(\texttt{arb\_t res}, \texttt{ulong n}, \texttt{slong prec})

Sets \(res\) to the Bell number \(B_n\). If the number is too large to fit exactly in \(\text{prec}\) bits, a numerical approximation is computed efficiently.

The algorithm to compute Bell numbers, including error analysis, is described in detail in [Joh2015].

void \texttt{arb\_euler\_number\_fmpz}(\texttt{arb\_t res}, \texttt{const fmpz\_t n}, \texttt{slong prec})

void \texttt{arb\_euler\_number\_ui}(\texttt{arb\_t res}, \texttt{ulong n}, \texttt{slong prec})

Sets \(res\) to the Euler number \(E_n\), which is defined by the exponential generating function \(1/\cosh(x)\).

The result will be exact if \(E_n\) is exactly representable at the requested precision.

void \texttt{arb\_fmpz\_euler\_number\_ui\_multi\_mod}(\texttt{fmpz\_t res}, \texttt{ulong n}, \texttt{double alpha})

void \texttt{arb\_fmpz\_euler\_number\_ui}(\texttt{fmpz\_t res}, \texttt{ulong n})

Computes the Euler number \(E_n\) as an exact integer. The default algorithm uses a table lookup, the Dirichlet beta function or a hybrid modular algorithm depending on the size of \(n\). The \texttt{multi\_mod} algorithm accepts a tuning parameter \(alpha\) which can be set to a negative value to use defaults.

void \texttt{arb\_partitions\_fmpz}(\texttt{arb\_t res}, \texttt{const fmpz\_t n}, \texttt{slong prec})
void _arb_partitions_ui(arb_t res, ulong n, slong prec)
    Sets res to the partition function \( p(n) \). When \( n \) is large and \( \log_2 p(n) \) is more than twice \( \text{prec} \), the leading term in the Hardy-Ramanujan asymptotic series is used together with an error bound. Otherwise, the exact value is computed and rounded.

void _arb_primorial_nth_ui(arb_t res, ulong n, slong prec)
    Sets res to the \( n \)th primorial, defined as the product of the first \( n \) prime numbers. The running time is quasilinear in \( n \).

void _arb_primorial_ui(arb_t res, ulong n, slong prec)
    Sets res to the primorial defined as the product of the positive integers up to and including \( n \). The running time is quasilinear in \( n \).

9.8.24 Internals for computing elementary functions

void _arb_atan_taylor_naive(mp_ptr y, mp_limb_t *error, mp_ptr x, mp_size_t xn, ulong N, int alternating)
void _arb_atan_taylor_rs(mp_ptr y, mp_limb_t *error, mp_ptr x, mp_size_t xn, ulong N, int alternating)

Computes an approximation of \( y = \sum_{k=0}^{N-1} x^{2k+1}/(2k+1) \) (if \( \text{alternating} \) is 0) or \( y = \sum_{k=0}^{N-1} (-1)^k x^{2k+1}/(2k+1) \) (if \( \text{alternating} \) is 1). Used internally for computing arctangents and logarithms. The \textit{naive} version uses the forward recurrence, and the \textit{rs} version uses a division-avoiding rectangular splitting scheme.

Requires \( N \leq 255, 0 \leq x \leq 1/16 \), and \( xN \) positive. The input \( x \) and output \( y \) are fixed-point numbers with \( xN \) fractional limbs. A bound for the ulp error is written to \textit{error}.

void _arb_exp_taylor_naive(mp_ptr y, mp_limb_t *error, mp_ptr x, mp_size_t xn, ulong N)
void _arb_exp_taylor_rs(mp_ptr y, mp_limb_t *error, mp_ptr x, mp_size_t xn, ulong N)

Computes an approximation of \( y = \sum_{k=0}^{N-1} x^k/k! \). Used internally for computing exponentials. The \textit{naive} version uses the forward recurrence, and the \textit{rs} version uses a division-avoiding rectangular splitting scheme.

Requires \( N \leq 287, 0 \leq x \leq 1/16 \), and \( xN \) positive. The input \( x \) is a fixed-point number with \( xN \) fractional limbs, and the output \( y \) is a fixed-point number with \( xN \) fractional limbs plus one extra limb for the integer part of the result.

A bound for the ulp error is written to \textit{error}.

void _arb_sin_cos_taylor_naive(mp_ptr ysin, mp_ptr ycos, mp_limb_t *error, mp_ptr x, mp_size_t xn, ulong N)
void _arb_sin_cos_taylor_rs(mp_ptr ysin, mp_ptr ycos, mp_limb_t *error, mp_ptr x, mp_size_t xn, ulong N, int sinonly, int alternating)

Computes approximations of \( y_s = \sum_{k=0}^{N-1} (-1)^k x^{2k+1}/(2k+1)! \) and \( y_c = \sum_{k=0}^{N-1} (-1)^k x^{2k}/(2k)! \). Used internally for computing sines and cosines. The \textit{naive} version uses the forward recurrence, and the \textit{rs} version uses a division-avoiding rectangular splitting scheme.

Requires \( N \leq 143, 0 \leq x \leq 1/16 \), and \( xN \) positive. The input \( x \) and outputs \( y_s, y_c \) are fixed-point numbers with \( xN \) fractional limbs. A bound for the ulp error is written to \textit{error}.

If \textit{sinonly} is 1, only the sine is computed; if \textit{sinonly} is 0 both the sine and cosine are computed. To compute sin and cos, alternating should be 1. If alternating is 0, the hyperbolic sine is computed (this is currently only intended to be used together with \textit{sinonly}).

int _arb_get_mpn_fixed_mod_log2(mp_ptr w, fmpz_t q, mp_limb_t *error, const arf_t x, mp_size_t wn)
Attempts to write \( w = x - q \log(2) \) with \( 0 \leq w < \log(2) \), where \( w \) is a fixed-point number with \( \text{wm} \) limbs and ulp error \( \text{error} \). Returns success.

```c
int _arb_get_mpn_fixed_mod_pi4(mp_ptr w, fmpz_t q, int *octant, mp_limb_t *error, const arf_t x, mp_size_t wn)
```

Attempts to write \( w = |x| - q\pi/4 \) with \( 0 \leq w < \pi/4 \), where \( w \) is a fixed-point number with \( \text{wm} \) limbs and ulp error \( \text{error} \). Returns success.

The value of \( q \mod 8 \) is written to \( \text{octant} \). The output variable \( q \) can be NULL, in which case the full value of \( q \) is not stored.

```c
slong _arb_exp_taylor_bound(slong mag, slong prec)
```

Returns \( n \) such that \( \sum_{k=n}^{\infty} x^k/k! \leq 2^{-\text{prec}} \), assuming \( |x| \leq 2^{\text{mag}} \leq 1/4 \).

```c
void _arb_exp_sum_bs_powtab(fmpz_t T, fmpz_t Q, flint_bitcnt_t *Qexp, const fmpz_t x, flint_bitcnt_t r, slong N)
```

Computes \( T \), \( Q \) and \( Q_{\text{exp}} \) such that \( T/(Q2^{Q_{\text{exp}}}) = \sum_{k=1}^{N} (x/2^{r})^k/k! \) using binary splitting. Note that the sum is taken to \( N \) inclusive and omits the constant term.

The \( \text{powtab} \) version precomputes a table of powers of \( x \), resulting in slightly higher memory usage but better speed. For best efficiency, \( N \) should have many trailing zero bits.

```c
void _arb_exp_arf_bb(const arf_t x, slong prec, int minus_one)
```

Computes the exponential function using a generic version of the rectangular splitting strategy, intended for intermediate precision.

```c
void _arb_atan_sum_bs_powtab(fmpz_t T, fmpz_t Q, flint_bitcnt_t *Qexp, const fmpz_t x, flint_bitcnt_t r, slong N)
```

Computes \( T \), \( Q \) and \( Q_{\text{exp}} \) such that \( T/(Q2^{Q_{\text{exp}}}) = \sum_{k=1}^{N} (-1)^k(x/2^r)^{2k}/(2k+1) \) using binary splitting. Note that the sum is taken to \( N \) inclusive, omits the linear term, and requires a final multiplication by \( (x/2^r) \) to give the true series for atan.

The \( \text{powtab} \) version precomputes a table of powers of \( x \), resulting in slightly higher memory usage but better speed. For best efficiency, \( N \) should have many trailing zero bits.

```c
void _arb_atan_arf_bb(const arf_t x, slong prec)
```

Computes the arctangent of \( x \). Initially, the argument-halving formula
\[
\text{atan}(x) = 2\text{atan} \left( \frac{x}{1 + \sqrt{1 + x^2}} \right)
\]
is applied up to 8 times to get a small argument. Then a version of the bit-burst algorithm is used.
The functional equation
\[
\arctan(x) = \arctan\left(\frac{p}{q}\right) + \arctan\left(\frac{wx}{px + q}\right), \quad w = \frac{qx - p}{px + q}, \quad p = \lfloor qx \rfloor
\]
is applied repeatedly instead of integrating a differential equation for the arctangent, as this appears
to be more efficient.

void \texttt{arb_atan_frac-bsplit}(\texttt{arb_t s}, \texttt{const fmpz_t p}, \texttt{const fmpz_t q}, \texttt{int hyperbolic}, \texttt{slong prec})
Computes the arctangent of \(p/q\), optionally the hyperbolic arctangent, using direct series summation
with binary splitting.

void \texttt{arb_sin_cos_arf_generic}(\texttt{arb_t s}, \texttt{arb_t c}, \texttt{const arf_t x}, \texttt{slong prec})
Computes the sine and cosine of \(x\) using a generic strategy. This function gets called internally
by the main \texttt{sin} and \texttt{cos} functions when the precision for argument reduction or series evaluation
based on lookup tables is exhausted.

This function first performs a cheap test to see if \(|x| < \pi/2 - \varepsilon\). If the test fails, it uses \(\pi\) to reduce
the argument to the first octant, and then evaluates the \texttt{sin} and \texttt{cos} functions recursively (this call
cannot result in infinite recursion).

If no argument reduction is needed, this function uses a generic version of the rectangular splitting
algorithm if the precision is not too high, and otherwise invokes the asymptotically fast bit-burst
algorithm.

void \texttt{arb_sin_cos_arf_bb}(\texttt{arb_t s}, \texttt{arb_t c}, \texttt{const arf_t x}, \texttt{slong prec})
Computes the sine and cosine of \(x\) using the bit-burst algorithm. It is required that \(|x| < \pi/2\) (this
is not checked).

void \texttt{arb_sin_cos_wide}(\texttt{arb_t s}, \texttt{arb_t c}, \texttt{const arb_t x}, \texttt{slong prec})
Computes an accurate enclosure (with both endpoints optimal to within about \(2^{-30}\) as afforded
by the radius format) of the range of \texttt{sin} and \texttt{cosine} on a given wide interval. The computation
is done by evaluating the \texttt{sin} and \texttt{cos} at the interval endpoints and determining whether peaks
of -1 or 1 occur between the endpoints. The interval is then converted back to a ball.

The internal computations are done with doubles, using a simple floating-point algorithm to ap-
proximate the \texttt{sin} and \texttt{cos} functions. It is easy to see that the cumulative errors in this algorithm add up
to less than \(2^{-30}\), with the dominant source of error being a single approximate reduction by \(\pi/2\).
This reduction is done safely using doubles up to a magnitude of about \(2^{20}\). For larger arguments,
a slower reduction using \texttt{arb_t} arithmetic is done as a preprocessing step.

void \texttt{arb_sin_cos_generic}(\texttt{arb_t s}, \texttt{arb_t c}, \texttt{const arb_t x}, \texttt{slong prec})
Computes the \texttt{sin} and \texttt{cos} of \(x\) by taking care of various special cases and computing the
propagated error before calling \texttt{arb_sin_cos_arf_generic()}. This is used as a fallback inside
\texttt{arb_sin_cos()} to take care of all cases without a fast path in that function.

void \texttt{arb_log_primes_vec-bsplit}(\texttt{arb_ptr res}, \texttt{slong n}, \texttt{slong prec})
Sets \texttt{res} to a vector containing the natural logarithms of the first \(n\) prime numbers, computed using
binary splitting applied to simultaneous Machine-type formulas. This function is not optimized for
large \(n\) or small \texttt{prec}.

\texttt{ARB_LOG_PRIME_CACHE_NUM}
Number of logarithms of small prime numbers to cache automatically.

\texttt{ARB_LOG_REDUCTION_DEFAULT_MAX_PREC}
Maximum precision to cache logarithms of small prime numbers automatically.

void \texttt{_arb_log_p_ensure_cached}(\texttt{slong prec})
Ensure that the internal cache of logarithms of small prime numbers has entries to at least \texttt{prec}
bits.
void **arb_exp_arf_log_reduction** *(arb_t res, const arf_t x, slong prec, int minus_one)*
Computes the exponential function using log reduction.

void **arb_exp_arf_generic** *(arb_t z, const arf_t x, slong prec, int minus_one)*
Computes the exponential function using an automatic choice between rectangular splitting and
the bit-burst algorithm, without precomputation.

void **arb_exp_arf** *(arb_t z, const arf_t x, slong prec, int minus_one, slong maglim)*
Computes the exponential function using an automatic choice between all implemented algorithms.

void **arb_log_newton** *(arb_t res, const arb_t x, slong prec)*
Computes the logarithm using Newton iteration.

void **arb_log_arf_newton** *(arb_t res, const arf_t x, slong prec)*
Computes the logarithm using Newton iteration.

**ARB_ATAN_GAUSS_PRIME_CACHE_NUM**
Number of primitive arctangents to cache automatically.

void **arb_atan_gauss_primes_vec_bsplit** *(arb_ptr res, slong n, slong prec)*
Sets res to the primitive angles corresponding to the first n nonreal Gaussian primes (ignoring
symmetries), computed using binary splitting applied to simultaneous Machine-type formulas. This
function is not optimized for large n or small prec.

void **_arb_atan_gauss_p_ensure_cached** *(slong prec)*

void **arb_sin_cos_arf_atan_reduction** *(arb_t res1, arb_t res2, const arf_t x, slong prec)*
Computes sin and/or cos using reduction by primitive angles.

void **arb_atan_newton** *(arb_t res, const arb_t x, slong prec)*
void **arb_atan_arf_newton** *(arb_t res, const arf_t x, slong prec)*
Computes the arctangent using Newton iteration.

### 9.8.25 Vector functions

void **_arb_vec_zero** *(arb_ptr vec, slong n)*
Sets all entries in vec to zero.

int **_arb_vec_is_zero** *(arb_srcptr vec, slong len)*
Returns nonzero iff all entries in x are zero.

int **_arb_vec_is_finite** *(arb_srcptr x, slong len)*
Returns nonzero iff all entries in x certainly are finite.

int **_arb_vec_equal** *(arb_srcptr vec1, arb_srcptr vec2, slong len)*
Returns nonzero iff vec1 and vec2 are equal in the sense of **arb_equal()**, i.e. have both the same
midpoint and radius elementwise.

int **_arb_vec_overlaps** *(arb_srcptr vec1, arb_srcptr vec2, slong len)*
Returns nonzero iff vec1 overlaps vec2 elementwise.

int **_arb_vec_contains** *(arb_srcptr vec1, arb_srcptr vec2, slong len)*
Returns nonzero iff vec1 contains vec2 elementwise.

void **_arb_vec_set** *(arb_ptr res, arb_srcptr vec, slong len)*
Sets res to a copy of vec.

void **_arb_vec_set_round** *(arb_ptr res, arb_srcptr vec, slong len, slong prec)*
Sets res to a copy of vec, rounding each entry to prec bits.

void **_arb_vec_swap** *(arb_ptr vec1, arb_ptr vec2, slong len)*
Swaps the entries of vec1 and vec2.
void _arb_vec_neg(arb_ptr B, arb_srcptr A, slong n)
void _arb_vec_sub(arb_ptr C, arb_srcptr A, arb_srcptr B, slong n, slong prec)
void _arb_vec_add(arb_ptr C, arb_srcptr A, arb_srcptr B, slong n, slong prec)
void _arb_vec_scalar_mul(arb_ptr res, arb_srcptr vec, slong len, const arb_t c, slong prec)
void _arb_vec_scalar_div(arb_ptr res, arb_srcptr vec, slong len, const arb_t c, slong prec)
void _arb_vec_scalar_mul_fmpz(arb_ptr res, arb_srcptr vec, slong len, const fmpz_t c, slong prec)
void _arb_vec_scalar_mul_2exp_si(arb_ptr res, arb_srcptr src, slong len, slong c)
void _arb_vec_addmul(arb_ptr res, arb_srcptr vec, slong len, const arb_t c, slong prec)

void _arb_vec_get_mag(mag_t bound, arb_srcptr vec, slong len)
Sets bound to an upper bound for the entries in vec.

slong _arb_vec_bits(arb_srcptr x, slong len)
Returns the maximum of arb_bits() for all entries in vec.

void _arb_vec_set_powers(arb_ptr xs, const arb_t x, slong len, slong prec)
Sets xs to the powers 1, x, x^2, ..., x^{len-1}.

void _arb_vec_add_error_arf_vec(arb_ptr res, arf_srcptr err, slong len)
void _arb_vec_add_error_mag_vec(arb_ptr res, mag_srcptr err, slong len)
Adds the magnitude of each entry in err to the radius of the corresponding entry in res.

void _arb_vec_indeterminate(arb_ptr vec, slong len)
Applies arb_indeterminate() elementwise.

void _arb_vec_trim(arb_ptr res, arb_srcptr vec, slong len)
Applies arb_trim() elementwise.

int _arb_vec_get_unique_fmpz_vec(fmpz *res, arb_srcptr vec, slong len)
Calls arb_get_unique_fmpz() elementwise and returns nonzero if all entries can be rounded uniquely to integers. If any entry in vec cannot be rounded uniquely to an integer, returns zero.

void _arb_vec_printn(arb_srcptr vec, slong len, slong digits, ulong flags)
void _arb_vec_printd(arb_srcptr vec, slong len, slong ndigits)
Prints vec in decimal using arb_printn() or arb_printd() on each entry.

9.9 acb.h – complex numbers

An acb_t represents a complex number with error bounds. An acb_t consists of a pair of real number balls of type arb_struct, representing the real and imaginary part with separate error bounds.

An acb_t thus represents a rectangle \([m_1 - r_1, m_1 + r_1] + [m_2 - r_2, m_2 + r_2]i\) in the complex plane. This is used instead of a disk or square representation (consisting of a complex floating-point midpoint with a single radius), since it allows implementing many operations more conveniently by splitting into ball operations on the real and imaginary parts. It also allows tracking when complex numbers have an exact (for example exactly zero) real part and an inexact imaginary part, or vice versa.

The interface for the acb_t type is slightly less developed than that for the arb_t type. In many cases, the user can easily perform missing operations by directly manipulating the real and imaginary parts.
9.9.1 Types, macros and constants

type acb_struct

type acb_t

An acb_struct consists of a pair of arb_structs. An acb_t is defined as an array of length one of type acb_struct, permitting an acb_t to be passed by reference.

type acb_ptr

Alias for acb_struct *, used for vectors of numbers.

type acb_srcptr

Alias for const acb_struct *, used for vectors of numbers when passed as constant input to functions.

acb_realref(x)

Macro returning a pointer to the real part of x as an arb_t.

acb_imagref(x)

Macro returning a pointer to the imaginary part of x as an arb_t.

9.9.2 Memory management

void acb_init(acb_t x)

Initializes the variable x for use, and sets its value to zero.

void acb_clear(acb_t x)

Clears the variable x, freeing or recycling its allocated memory.

acb_ptr _acb_vec_init(slong n)

Returns a pointer to an array of n initialized acb_structs.

void _acb_vec_clear(acb_ptr v, slong n)

Clears an array of n initialized acb_structs.

slong acb_allocated_bytes(const acb_t x)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add sizeof(acb_struct) to get the size of the object as a whole.

slong _acb_vec_allocated_bytes(acb_srcptr vec, slong len)

Returns the total number of bytes allocated for this vector, i.e. the space taken up by the vector itself plus the sum of the internal heap allocation sizes for all its member elements.

double _acb_vec_estimate_allocated_bytes(slong len, slong prec)

Estimates the number of bytes that need to be allocated for a vector of len elements with prec bits of precision, including the space for internal limb data. See comments for _arb_vec_estimate_allocated_bytes().

9.9.3 Basic manipulation

void acb_zero(acb_t z)

void acb_one(acb_t z)

void acb_onei(acb_t z)

Sets z respectively to 0, 1, i = \sqrt{-1}.

void acb_set(acb_t z, const acb_t x)
void \texttt{acb_set_ui(acb\_t z, ulong x)}

void \texttt{acb_set_si(acb\_t z, slong x)}

void \texttt{acb_set_d(acb\_t z, double x)}

void \texttt{acb_set_fmpz(acb\_t z, const fmpz\_t x)}

void \texttt{acb_set_arb(acb\_t z, const arb\_t c)}

Sets \( z \) to the value of \( x \).

void \texttt{acb_set_si_si(acb\_t z, slong x, slong y)}

void \texttt{acb_set_d_d(acb\_t z, double x, double y)}

void \texttt{acb_set_fmpz_fmpz(acb\_t z, const fmpz\_t x, const fmpz\_t y)}

void \texttt{acb_set_arb_arb(acb\_t z, const arb\_t x, const arb\_t y)}

Sets the real and imaginary part of \( z \) to the values \( x \) and \( y \) respectively.

void \texttt{acb_set_fmpq(acb\_t z, const fmpq\_t x, slong prec)}

void \texttt{acb_set_round(acb\_t z, const acb\_t x, slong prec)}

void \texttt{acb_set_round_fmpz(acb\_t z, const fmpz\_t x, slong prec)}

void \texttt{acb_set_round_arb(acb\_t z, const arb\_t x, slong prec)}

Sets \( z \) to \( x \), rounded to \( prec \) bits.

void \texttt{acb_swap(acb\_t z, acb\_t x)}

Swaps \( z \) and \( x \) efficiently.

void \texttt{acb_add_error_arf(acb\_t x, const arf\_t err)}

void \texttt{acb_add_error_mag(acb\_t x, const mag\_t err)}

void \texttt{acb_add_error_arb(acb\_t x, const arb\_t err)}

Adds \( err \) to the error bounds of both the real and imaginary parts of \( x \), modifying \( x \) in-place.

void \texttt{acb_get_mid(acb\_t m, const acb\_t x)}

Sets \( m \) to the midpoint of \( x \).

\textbf{9.9.4 Input and output}

The \texttt{acb\_print}... functions print to standard output, while \texttt{acb\_fprint}... functions print to the stream file.

void \texttt{acb\_print(const acb\_t x)}

void \texttt{acb\_fprint(FILE *file, const acb\_t x)}

Prints the internal representation of \( x \).

void \texttt{acb\_printd(const acb\_t x, slong digits)}

void \texttt{acb\_fprintd(FILE *file, const acb\_t x, slong digits)}

Prints \( x \) in decimal. The printed value of the radius is not adjusted to compensate for the fact that the binary-to-decimal conversion of both the midpoint and the radius introduces additional error.

void \texttt{acb\_printn(const acb\_t x, slong digits, ulong flags)}
void \texttt{acb_fprintn}(FILE *\texttt{file}, \texttt{const acb_t x, slong digits, ulong flags})

Prints a nice decimal representation of \(x\), using the format of \texttt{arb_get_str()} (or the corresponding \texttt{arb_printn()}) for the real and imaginary parts.

By default, the output shows the midpoint of both the real and imaginary parts with a guaranteed error of at most one unit in the last decimal place. In addition, explicit error bounds are printed so that the displayed decimal interval is guaranteed to enclose \(x\).

Any flags understood by \texttt{arb_get_str()} can be passed via \texttt{flags} to control the format of the real and imaginary parts.

\section*{9.9.5 Random number generation}

void \texttt{acb_randtest}(\texttt{acb_t z, flint_rand_t state, slong prec, slong mag_bits})

Generates a random complex number by generating separate random real and imaginary parts.

void \texttt{acb_randtest_special}(\texttt{acb_t z, flint_rand_t state, slong prec, slong mag_bits})

Generates a random complex number by generating separate random real and imaginary parts. Also generates NaNs and infinities.

void \texttt{acb_randtest_precise}(\texttt{acb_t z, flint_rand_t state, slong prec, slong mag_bits})

Generates a random complex number with precise real and imaginary parts.

void \texttt{acb_randtest_param}(\texttt{acb_t z, flint_rand_t state, slong prec, slong mag_bits})

Generates a random complex number, with very high probability of generating integers and half-integers.

void \texttt{acb_urandom}(\texttt{acb_t z, flint_rand_t state, slong prec})

Generates a random complex number with precise real and imaginary parts, uniformly chosen in the unit disk.

\section*{9.9.6 Precision and comparisons}

int \texttt{acb_is_zero}(\texttt{const acb_t z})

Returns nonzero iff \(z\) is zero.

int \texttt{acb_is_one}(\texttt{const acb_t z})

Returns nonzero iff \(z\) is exactly 1.

int \texttt{acb_is_finite}(\texttt{const acb_t z})

Returns nonzero iff \(z\) certainly is finite.

int \texttt{acb_isExact}(\texttt{const acb_t z})

Returns nonzero iff \(z\) is exact.

int \texttt{acb_is_int}(\texttt{const acb_t z})

Returns nonzero iff \(z\) is an exact integer.

int \texttt{acb_is_int_2exp_si}(\texttt{const acb_t x, slong e})

Returns nonzero iff \(z\) exactly equals \(n2^e\) for some integer \(n\).

int \texttt{acb_equal}(\texttt{const acb_t x, const acb_t y})

Returns nonzero iff \(x\) and \(y\) are identical as sets, i.e. if the real and imaginary parts are equal as balls.

Note that this is not the same thing as testing whether both \(x\) and \(y\) certainly represent the same complex number, unless either \(x\) or \(y\) is exact (and neither contains NaN). To test whether both operands might represent the same mathematical quantity, use \texttt{acb_overlaps()} or \texttt{acb_contains()}, depending on the circumstance.
int `acb_equal_si(const acb_t x, slong y)`

Returns nonzero iff \( x \) is equal to the integer \( y \).

int `acb_eq(const acb_t x, const acb_t y)`

Returns nonzero iff \( x \) and \( y \) are certainly equal, as determined by testing that \( \text{arb_eq()} \) holds for both the real and imaginary parts.

int `acb_ne(const acb_t x, const acb_t y)`

Returns nonzero iff \( x \) and \( y \) are certainly not equal, as determined by testing that \( \text{arb_ne()} \) holds for either the real or imaginary parts.

int `acb_overlaps(const acb_t x, const acb_t y)`

Returns nonzero iff \( x \) and \( y \) have some point in common.

void `acb_union(acb_t z, const acb_t x, const acb_t y, slong prec)`

Sets \( z \) to a complex interval containing both \( x \) and \( y \).

void `acb_get_abs_ubound_arf(arf_t u, const acb_t z, slong prec)`

Sets \( u \) to an upper bound for the absolute value of \( z \), computed using a working precision of \( prec \) bits.

void `acb_get_abs_lbound_arf(arf_t u, const acb_t z, slong prec)`

Sets \( u \) to a lower bound for the absolute value of \( z \), computed using a working precision of \( prec \) bits.

void `acb_get_rad_ubound_arf(arf_t u, const acb_t z, slong prec)`

Sets \( u \) to an upper bound for the error radius of \( z \) (the value is currently not computed tightly).

void `acb_get_mag(mag_t u, const acb_t x)`

Sets \( u \) to an upper bound for the absolute value of \( x \).

void `acb_get_mag_lower(mag_t u, const acb_t x)`

Sets \( u \) to a lower bound for the absolute value of \( x \).

int `acb_contains_fmpq(const acb_t x, const fmpq_t y)`

int `acb_contains_fmpz(const acb_t x, const fmpz_t y)`

int `acb_contains(const acb_t x, const acb_t y)`

Returns nonzero iff \( y \) is contained in \( x \).

int `acb_contains_zero(const acb_t x)`

Returns nonzero iff zero is contained in \( x \).

int `acb_contains_int(const acb_t x)`

Returns nonzero iff the complex interval represented by \( x \) contains an integer.

int `acb_contains_interior(const acb_t x, const acb_t y)`

Tests if \( y \) is contained in the interior of \( x \). This predicate always evaluates to false if \( x \) and \( y \) are both real-valued, since an imaginary part of 0 is not considered contained in the interior of the point interval 0. More generally, the same problem occurs for intervals with an exact real or imaginary part. Such intervals must be handled specially by the user where a different interpretation is intended.

```c
slong acb_rel_error_bits(const acb_t x)
```

Returns the effective relative error of \( x \) measured in bits. This is computed as if calling \( \text{arb_rel_error_bits()} \) on the real ball whose midpoint is the larger out of the real and imaginary midpoints of \( x \), and whose radius is the larger out of the real and imaginary radiiuses of \( x \).

```c
slong acb_rel_accuracy_bits(const acb_t x)
```

Returns the effective relative accuracy of \( x \) measured in bits, equal to the negative of the return value from \( \text{acb_rel_error_bits()} \).
\textbf{slong \texttt{acb\_rel\_one\_accuracy\_bits}(const \texttt{acb\_t x})}

Given a ball with midpoint \(m\) and radius \(r\), returns an approximation of the relative accuracy of 
\[\max(1, |m|) \pm r\] measured in bits.

\textbf{slong \texttt{acb\_bits}(const \texttt{acb\_t x})}

Returns the maximum of \textit{arb\_bits} applied to the real and imaginary parts of \(x\), i.e. the minimum
precision sufficient to represent \(x\) exactly.

\textbf{void \texttt{acb\_indeterminate}(\texttt{acb\_t x})}

Sets \(x\) to \([\text{NaN} \pm \infty] + [\text{NaN} \pm \infty]i\), representing an indeterminate result.

\textbf{void \texttt{acb\_trim}(\texttt{acb\_t y, const \texttt{acb\_t x}})}

Sets \(y\) to a a copy of \(x\) with both the real and imaginary parts trimmed (see \textit{arb\_trim()}).

\textbf{int \texttt{acb\_is\_real}(const \texttt{acb\_t x})}

Returns nonzero iff the imaginary part of \(x\) is zero. It does not test whether the real part of \(x\) also
is finite.

\textbf{int \texttt{acb\_get\_unique\_fmpz}(\texttt{fmpz\_t z, const \texttt{acb\_t x}})}

If \(x\) contains a unique integer, sets \(z\) to that value and returns nonzero. Otherwise (if \(x\) represents
no integers or more than one integer), returns zero.

\section*{9.9.7 Complex parts}

\textbf{void \texttt{acb\_get\_real}(\texttt{arb\_t re, const \texttt{acb\_t z}})}

Sets \(re\) to the real part of \(z\).

\textbf{void \texttt{acb\_get\_imag}(\texttt{arb\_t im, const \texttt{acb\_t z}})}

Sets \(im\) to the imaginary part of \(z\).

\textbf{void \texttt{acb\_arg}(\texttt{arb\_t r, const \texttt{acb\_t z, slong prec}})}

Sets \(r\) to a real interval containing the complex argument (phase) of \(z\). We define the complex
argument have a discontinuity on \((-\infty, 0]\), with the special value \(\text{arg}(0) = 0\), and
\(\text{arg}(a + 0i) = \pi\) for \(a < 0\). Equivalently, if \(z = a + bi\), the argument is given by \(\text{atan2}(b, a)\) (see \textit{arb\_atan2()}).

\textbf{void \texttt{acb\_abs}(\texttt{arb\_t r, const \texttt{acb\_t z, slong prec}})}

Sets \(r\) to the absolute value of \(z\).

\textbf{void \texttt{acb\_sgn}(\texttt{acb\_t r, const \texttt{acb\_t z, slong prec}})}

Sets \(r\) to the complex sign of \(z\), defined as 0 if \(z\) is exactly zero and the projection onto the unit
circle \(z/|z| = \exp(i \text{arg}(z))\) otherwise.

\textbf{void \texttt{acb\_csgn}(\texttt{arb\_t r, const \texttt{acb\_t z}})}

Sets \(r\) to the extension of the real sign function taking the value 1 for \(z\) strictly in the right half
plane, -1 for \(z\) strictly in the left half plane, and the sign of the imaginary part when \(z\) is on the
imaginary axis. Equivalently, \(\text{csgn}(z) = z/\sqrt{z^2}\) except that the value is 0 when \(z\) is exactly zero.

\section*{9.9.8 Arithmetic}

\textbf{void \texttt{acb\_neg}(\texttt{acb\_t z, const \texttt{acb\_t x}})}

\textbf{void \texttt{acb\_neg\_round}(\texttt{acb\_t z, const \texttt{acb\_t x, slong prec}})}

Sets \(z\) to the negation of \(x\).

\textbf{void \texttt{acb\_conj}(\texttt{acb\_t z, const \texttt{acb\_t x}})}

Sets \(z\) to the complex conjugate of \(x\).

\textbf{void \texttt{acb\_add\_ui}(\texttt{acb\_t z, const \texttt{acb\_t x, ulong y, slong prec}})}
void \texttt{acb_add_si}(acb\_t z, const acb\_t x, slong y, slong prec)

void \texttt{acb_add_fmpz}(acb\_t z, const acb\_t x, const fmpz\_t y, slong prec)

void \texttt{acb_add_arb}(acb\_t z, const acb\_t x, const arb\_t y, slong prec)

void \texttt{acb_add}(acb\_t z, const acb\_t x, const acb\_t y, slong prec)

Sets \(z\) to the sum of \(x\) and \(y\).

void \texttt{acb_sub_ui}(acb\_t z, const acb\_t x, ulong y, slong prec)

void \texttt{acb_sub_si}(acb\_t z, const acb\_t x, slong y, slong prec)

void \texttt{acb_sub_fmpz}(acb\_t z, const acb\_t x, const fmpz\_t y, slong prec)

void \texttt{acb_sub_arb}(acb\_t z, const acb\_t x, const arb\_t y, slong prec)

void \texttt{acb_sub}(acb\_t z, const acb\_t x, const acb\_t y, slong prec)

Sets \(z\) to the difference of \(x\) and \(y\).

void \texttt{acb_mul_onei}(acb\_t z, const acb\_t x)

Sets \(z\) to \(x\) multiplied by the imaginary unit.

void \texttt{acb_div_onei}(acb\_t z, const acb\_t x)

Sets \(z\) to \(x\) divided by the imaginary unit.

void \texttt{acb_mul_i_pow_si}(acb\_t z, const acb\_t x, slong k)

Sets \(z\) to \(x\) multiplied by \(i^k\), where \(i\) denotes the imaginary unit.

void \texttt{acb_mul_ui}(acb\_t z, const acb\_t x, ulong y, slong prec)

void \texttt{acb_mul_si}(acb\_t z, const acb\_t x, slong y, slong prec)

void \texttt{acb_mul_fmpz}(acb\_t z, const acb\_t x, const fmpz\_t y, slong prec)

void \texttt{acb_mul_arb}(acb\_t z, const acb\_t x, const arb\_t y, slong prec)

Sets \(z\) to the product of \(x\) and \(y\).

void \texttt{acb_mul}(acb\_t z, const acb\_t x, const acb\_t y, slong prec)

Sets \(z\) to the product of \(x\) and \(y\). If at least one part of \(x\) or \(y\) is zero, the operations is reduced to two real multiplications. If \(x\) and \(y\) are the same pointers, they are assumed to represent the same mathematical quantity and the squaring formula is used.

void \texttt{acb_mul_2exp_si}(acb\_t z, const acb\_t x, slong e)

void \texttt{acb_mul_2exp_fmpz}(acb\_t z, const acb\_t x, const fmpz\_t e)

Sets \(z\) to \(x\) multiplied by \(2^e\), without rounding.

void \texttt{acb_sqr}(acb\_t z, const acb\_t x, slong prec)

Sets \(z\) to \(x\) squared.

void \texttt{acb_cube}(acb\_t z, const acb\_t x, slong prec)

Sets \(z\) to \(x\) cubed, computed efficiently using two real squarings, two real multiplications, and scalar operations.

void \texttt{acb_addmul}(acb\_t z, const acb\_t x, const acb\_t y, slong prec)

void \texttt{acb_addmul_ui}(acb\_t z, const acb\_t x, ulong y, slong prec)

void \texttt{acb_addmul_si}(acb\_t z, const acb\_t x, slong y, slong prec)

void \texttt{acb_addmul_fmpz}(acb\_t z, const acb\_t x, const fmpz\_t y, slong prec)
void \texttt{acb_addmul}\_arb\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{arb} \_t \ y, \texttt{slong} \ \texttt{prec})\)

Sets \( z \) to \( z \) plus the product of \( x \) and \( y \).

void \texttt{acb_submul}\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{acb} \_t \ y, \texttt{slong} \ \texttt{prec})\)

void \texttt{acb_submul}\_ui\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{ulong} \ y, \texttt{slong} \ \texttt{prec})\)

void \texttt{acb_submul}\_si\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{slong} \ y, \texttt{slong} \ \texttt{prec})\)

void \texttt{acb_submul}\_fmpz\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{fmpz} \_t \ y, \texttt{slong} \ \texttt{prec})\)

void \texttt{acb_submul}\_arb\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{arb} \_t \ y, \texttt{slong} \ \texttt{prec})\)

Sets \( z \) to \( z \) minus the product of \( x \) and \( y \).

void \texttt{acb_inv}\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{slong} \ \texttt{prec})\)

Sets \( z \) to the multiplicative inverse of \( x \).

void \texttt{acb_div}\_ui\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{ulong} \ y, \texttt{slong} \ \texttt{prec})\)

void \texttt{acb_div}\_si\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{slong} \ y, \texttt{slong} \ \texttt{prec})\)

void \texttt{acb_div}\_fmpz\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{fmpz} \_t \ y, \texttt{slong} \ \texttt{prec})\)

void \texttt{acb_div}\_arb\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{arb} \_t \ y, \texttt{slong} \ \texttt{prec})\)

void \texttt{acb_div}\((\texttt{acb} \_t \ z, \texttt{acb} \_t \ x, \texttt{acb} \_t \ y, \texttt{slong} \ \texttt{prec})\)

Sets \( z \) to the quotient of \( x \) and \( y \).

\textit{9.9.9 Dot product}

void \texttt{acb_dot}\_precise\((\texttt{acb} \_t \ \texttt{res}, \texttt{acb} \_t \ s, \texttt{int} \ \texttt{subtract}, \texttt{acb} \_\texttt{srcptr} \ x, \texttt{slong} \ \texttt{xstep}, \texttt{acb} \_\texttt{srcptr} \ y, \texttt{slong} \ \texttt{ystep}, \texttt{slong} \ \texttt{len}, \texttt{slong} \ \texttt{prec})\)

Computes the dot product of the vectors \( x \) and \( y \), setting \( \texttt{res} \) to \( s + (-1)^{\texttt{subtract}} \sum_{i=0}^{\texttt{len}-1} x_{i}y_{i} \).

The initial term \( s \) is optional and can be omitted by passing \texttt{NULL} (equivalently, \( s = 0 \)). The parameter \texttt{subtract} must be 0 or 1. The length \( \texttt{len} \) is allowed to be negative, which is equivalent to a length of zero. The parameters \texttt{xstep} or \texttt{ystep} specify a step length for traversing subsequences of the vectors \( x \) and \( y \); either can be negative to step in the reverse direction starting from the initial pointer. Aliasing is allowed between \( \texttt{res} \) and \( s \) but not between \( \texttt{res} \) and the entries of \( x \) and \( y \).

The default version determines the optimal precision for each term and performs all internal calculations using mpn arithmetic with minimal overhead. This is the preferred way to compute a dot product; it is generally much faster and more precise than a simple loop.

The \texttt{simple} version performs fused multiply-add operations in a simple loop. This can be used for testing purposes and is also used as a fallback by the default version when the exponents are out of range for the optimized code.

The \texttt{precise} version computes the dot product exactly up to the final rounding. This can be extremely slow and is only intended for testing.

void \texttt{acb_approx_dot}\((\texttt{acb} \_t \ \texttt{res}, \texttt{acb} \_t \ s, \texttt{int} \ \texttt{subtract}, \texttt{acb} \_\texttt{srcptr} \ x, \texttt{slong} \ \texttt{xstep}, \texttt{acb} \_\texttt{srcptr} \ y, \texttt{slong} \ \texttt{ystep}, \texttt{slong} \ \texttt{len}, \texttt{slong} \ \texttt{prec})\)

Computes an approximate dot product \textit{without error bounds}. The radii of the inputs are ignored (only the midpoints are read) and only the midpoint of the output is written.

void \texttt{acb_dot}\_ui\((\texttt{acb} \_t \ \texttt{res}, \texttt{acb} \_t \ \texttt{initial}, \texttt{int} \ \texttt{subtract}, \texttt{acb} \_\texttt{srcptr} \ x, \texttt{slong} \ \texttt{xstep}, \texttt{const} \texttt{ulong} \ *y, \texttt{slong} \ \texttt{ystep}, \texttt{slong} \ \texttt{len}, \texttt{slong} \ \texttt{prec})\)
void acb_dot_si(acb_t res, const acb_t initial, int subtract, acb_srcptr x, slong xstep, const slong *y, slong ystep, slong len, slong prec)
void acb_dot_uui(acb_t res, const acb_t initial, int subtract, acb_srcptr x, slong xstep, const ulong *y, slong ystep, slong len, slong prec)
void acb_dot_siui(acb_t res, const acb_t initial, int subtract, acb_srcptr x, slong xstep, const ulong *y, slong ystep, slong len, slong prec)
void acb_dot_fmpz(acb_t res, const acb_t initial, int subtract, acb_srcptr x, slong xstep, const fmpz *y, slong ystep, slong len, slong prec)

Equivalent to acb_dot(), but with integers in the array y. The uui and siui versions take an array of double-limb integers as input; the siui version assumes that these represent signed integers in two's complement form.

9.9.10 Mathematical constants

void acb_const_pi(acb_t y, slong prec)
Sets y to the constant 𝜋.

9.9.11 Powers and roots

void acb_sqrt(acb_t r, const acb_t z, slong prec)
Sets r to the square root of z. If either the real or imaginary part is exactly zero, only a single real square root is needed. Generally, we use the formula $\sqrt{a + bi} = u/2 + i(b/u, u = \sqrt{|a + bi| + a}$, requiring two real square root extractions.

void acb_sqrt analytic(acb_t r, const acb_t z, int analytic, slong prec)
Computes the square root. If analytic is set, gives a NaN-containing result if z touches the branch cut.

void acb_rsqrt(acb_t r, const acb_t z, slong prec)
Sets r to the reciprocal square root of z. If either the real or imaginary part is exactly zero, only a single real reciprocal square root is needed. Generally, we use the formula $1/\sqrt{a + bi} = ((a + r) - bi)/v, r = |a + bi|, v = \sqrt{r(a + bi + r)^2}$, requiring one real square root and one real reciprocal square root.

void acb_rsqrt analytic(acb_t r, const acb_t z, int analytic, slong prec)
Computes the reciprocal square root. If analytic is set, gives a NaN-containing result if z touches the branch cut.

void acb_sqrts(acb_t y1, acb_t y2, const acb_t x, slong prec)
Sets y1 and y2 to the two square roots of x, without any precision loss due to branch cuts. The order in which the square roots appear is not specified.

void acb_quadratic_roots_fmpz(acb_t r1, acb_t r2, const fmpz_t a, const fmpz_t b, const fmpz_t c, slong prec)
Sets r1 and r2 to the roots of the quadratic polynomial $ax^2 + bx + c$. Requires that a is nonzero. This function is implemented so that both roots are computed accurately even when direct use of the quadratic formula would lose accuracy.

void acb_root_ui(acb_t r, const acb_t z, ulong k, slong prec)
Sets r to the principal k-th root of z.

void acb_pow_fmpz(acb_t y, const acb_t b, const fmpz_t e, slong prec)
void acb_pow_ui(acb_t y, const acb_t b, ulong e, slong prec)
void acb_pow_si(acb_t y, const acb_t b, slong e, slong prec)

Sets $y = b^e$ using binary exponentiation (with an initial division if $e < 0$). Note that these functions can get slow if the exponent is extremely large (in such cases `acb_pow()` may be superior).

void acb_pow_arb(acb_t z, const acb_t x, const arb_t y, slong prec)

Sets $z = x^y$, computed using binary exponentiation if $y$ if a small exact integer, as $z = (x^{1/2})^{2y}$ if $y$ is a small exact half-integer, and generally as $z = \exp(y \log x)$.

void acb_pow_analytic(acb_t r, const acb_t x, const acb_t y, int analytic, slong prec)

Computes the power $x^y$. If `analytic` is set, gives a NaN-containing result if $x$ touches the branch cut (unless $y$ is an integer).

void acb_unit_root(acb_t res, ulong order, slong prec)

Sets `res` to $\exp\left(\frac{2\pi i}{\text{order}}\right)$ to precision `prec`.

9.9.12 Exponentials and logarithms

void acb_exp(acb_t y, const acb_t z, slong prec)

Sets $y$ to the exponential function of $z$, computed as $\exp(a + bi) = \exp(a)(\cos(b) + \sin(b)i)$.

void acb_exp_pi_i(acb_t y, const acb_t z, slong prec)

Sets $y$ to $\exp(\pi iz)$.

void acb_exp_invexp(acb_t s, acb_t t, const acb_t z, slong prec)

Sets $s = \exp(z)$ and $t = \exp(-z)$.

void acb_expml(acb_t res, const acb_t z, slong prec)

Sets `res` to $\exp(z) - 1$, using a more accurate method when $z \approx 0$.

void acb_log(acb_t y, const acb_t z, slong prec)

Sets $y$ to the principal branch of the natural logarithm of $z$, computed as $\log(a + bi) = \frac{1}{2}\log(a^2 + b^2) + i \arg(a + bi)$.

void acb_log_analytic(acb_t r, const acb_t z, int analytic, slong prec)

Computes the natural logarithm. If `analytic` is set, gives a NaN-containing result if $z$ touches the branch cut.

void acb_log1p(acb_t z, const acb_t x, slong prec)

Sets $z = \log(1 + x)$, computed accurately when $x \approx 0$.

9.9.13 Trigonometric functions

void acb_sin(acb_t s, const acb_t x, slong prec)

void acb_cos(acb_t c, const acb_t x, slong prec)

void acb_sin_cos(acb_t s, acb_t c, const acb_t x, slong prec)

Sets $s = \sin(z)$, $c = \cos(z)$, evaluated as $\sin(a + bi) = \sin(a) \cosh(b) + i \cos(a) \sinh(b)$, $\cos(a + bi) = \cos(a) \cosh(b) - i \sin(a) \sinh(b)$.

void acb_tan(acb_t s, const acb_t x, slong prec)

Sets $s = \tan(z) = \sin(z)/\cos(z)$. For large imaginary parts, the function is evaluated in a numerically stable way as $\pm i$ plus a decreasing exponential factor.

void acb_cot(acb_t s, const acb_t x, slong prec)

Sets $s = \cot(z) = \cos(z)/\sin(z)$. For large imaginary parts, the function is evaluated in a numerically stable way as $\pm i$ plus a decreasing exponential factor.
void \texttt{acb\_sin\_pi}(\texttt{acb\_t} s, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

void \texttt{acb\_cos\_pi}(\texttt{acb\_t} s, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

void \texttt{acb\_sin\_cos\_pi}(\texttt{acb\_t} s, \texttt{acb\_t} c, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Sets \( s = \sin(\pi z) \), \( c = \cos(\pi z) \), evaluating the trigonometric factors of the real and imaginary part accurately via \texttt{arb\_sin\_cos\_pi()}.

void \texttt{acb\_tan\_pi}(\texttt{acb\_t} s, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Sets \( s = \tan(\pi z) \). Uses the same algorithm as \texttt{acb\_tan()} , but evaluates the sine and cosine accurately via \texttt{arb\_sin\_cos\_pi()}.

void \texttt{acb\_cot\_pi}(\texttt{acb\_t} s, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Sets \( s = \cot(\pi z) \). Uses the same algorithm as \texttt{acb\_cot()} , but evaluates the sine and cosine accurately via \texttt{arb\_sin\_cos\_pi()}.

void \texttt{acb\_sec}(\texttt{acb\_t} res, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Computes \( \sec(z) = 1/\cos(z) \).

void \texttt{acb\_csc}(\texttt{acb\_t} res, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Computes \( \csc(x) = 1/\sin(z) \).

void \texttt{acb\_csc\_pi}(\texttt{acb\_t} res, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Computes \( \csc(\pi x) = 1/\sin(\pi z) \). Evaluates the sine accurately via \texttt{acb\_sin\_pi()}.

void \texttt{acb\_sinc}(\texttt{acb\_t} s, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Sets \( s = \text{sinc}(z) = \sin(z)/z \).

void \texttt{acb\_sinc\_pi}(\texttt{acb\_t} s, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Sets \( s = \text{sinc}(\pi x) = \sin(\pi z)/\pi z \).

\section*{9.9.14 Inverse trigonometric functions}

void \texttt{acb\_asin}(\texttt{acb\_t} res, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Sets \( \text{res} \) to \( \text{asin}(z) = -i \log(i z + \sqrt{1 - z^2}) \).

void \texttt{acb\_acos}(\texttt{acb\_t} res, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Sets \( \text{res} \) to \( \text{acos}(z) = \frac{1}{2} \pi - \text{asin}(z) \).

void \texttt{acb\_atan}(\texttt{acb\_t} res, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Sets \( \text{res} \) to \( \text{atan}(z) = \frac{1}{2}i(\log(1 - iz) - \log(1 + iz)) \).

\section*{9.9.15 Hyperbolic functions}

void \texttt{acb\_sinh}(\texttt{acb\_t} s, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

void \texttt{acb\_cosh}(\texttt{acb\_t} c, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

void \texttt{acb\_sinh\_cosh}(\texttt{acb\_t} s, \texttt{acb\_t} c, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

void \texttt{acb\_tanh}(\texttt{acb\_t} s, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

void \texttt{acb\_coth}(\texttt{acb\_t} s, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Respectively computes \( \sinh(z) = -i \sinh(iz) \), \( \cosh(z) = \cosh(iz) \), \( \tanh(z) = -i \tanh(iz) \), \( \coth(z) = i \cot(iz) \).

void \texttt{acb\_sech}(\texttt{acb\_t} res, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Computes \( \text{sech}(z) = 1/\cosh(z) \).

void \texttt{acb\_csch}(\texttt{acb\_t} res, \texttt{const} \texttt{acb\_t} z, \texttt{slong} prec)

Computes \( \text{csch}(z) = 1/\sinh(z) \).
9.9.16 Inverse hyperbolic functions

\begin{verbatim}
void acb_asinh(acb_t res, const acb_t z, slong prec)
Sets res to \( \text{asinh}(z) = -i \text{asin}(iz) \).

void acb_acosh(acb_t res, const acb_t z, slong prec)
Sets res to \( \text{acosh}(z) = \log(z + \sqrt{z^2 - 1}) \).

void acb_atanh(acb_t res, const acb_t z, slong prec)
Sets res to \( \text{atanh}(z) = -i \text{atan}(iz) \).
\end{verbatim}

9.9.17 Lambert W function

\begin{verbatim}
void acb_lambertw_asym(acb_t res, const acb_t z, const fmpz_t k, slong L, slong M, slong prec)
Sets res to the Lambert W function \( W_k(z) \) computed using \( L \) and \( M \) terms in the bivariate series giving the asymptotic expansion at zero or infinity. This algorithm is valid everywhere, but the error bound is only finite when \( |\log(z)| \) is sufficiently large.

int acb_lambertw_check_branch(const acb_t w, const fmpz_t k, slong prec)
Tests if \( w \) definitely lies in the image of the branch \( W_k(z) \). This function is used internally to verify that a computed approximation of the Lambert W function lies on the intended branch. Note that this will necessarily evaluate to false for points exactly on (or overlapping) the branch cuts, where a different algorithm has to be used.

void acb_lambertw_bound_deriv(mag_t res, const acb_t z, const acb_t ez1, const fmpz_t k)
Sets res to an upper bound for \( |W_k'(z)| \). The input \( ez1 \) should contain the precomputed value of \( e^{z+1} \).

Along the real line, the directional derivative of \( W_k(z) \) is understood to be taken. As a result, the user must handle the branch cut discontinuity separately when using this function to bound perturbations in the value of \( W_k(z) \).

void acb_lambertw(acb_t res, const acb_t z, const fmpz_t k, int flags, slong prec)
Sets res to the Lambert W function \( W_k(z) \) where the index \( k \) selects the branch (with \( k = 0 \) giving the principal branch). The placement of branch cuts follows [CGHJK1996].

If flags is nonzero, nonstandard branch cuts are used.

If flags is set to ACB_LAMBERTW_LEFT, computes \( W_{\text{left}}(z) \) which corresponds to \( W_k(z) \) in the upper half plane and \( W_{k+1}(z) \) in the lower half plane, connected continuously to the left of the branch points. In other words, the branch cut on \((-\infty, 0)\) is rotated counterclockwise to \((0, +\infty)\). (For \( k = -1 \) and \( k = 0 \), there is also a branch cut on \((-1/e, 0)\), continuous from below instead of from above to maintain counterclockwise continuity.)

If flags is set to ACB_LAMBERTW_MIDDLE, computes \( W_{\text{middle}}(z) \) which corresponds to \( W_{-1}(z) \) in the upper half plane and \( W_1(z) \) in the lower half plane, connected continuously through \((-1/e, 0)\) with branch cuts on \((-\infty, -1/e)\) and \((0, +\infty)\). \( W_{\text{middle}}(z) \) extends the real analytic function \( W_{-1}(x) \) defined on \((-1/e, 0)\) to a complex analytic function, whereas the standard branch \( W_{-1}(z) \) has a branch cut along the real segment.

The algorithm used to compute the Lambert W function is described in [Joh2017b].
\end{verbatim}
9.9.18 Rising factorials

```c
void acb_rising_ui(acb_t z, const acb_t x, ulong n, slong prec)
void acb_rising(acb_t z, const acb_t x, const acb_t n, slong prec)
```

Computes the rising factorial \( z = x(x + 1)(x + 2) \cdots (x + n - 1) \). These functions are aliases for
```c
acb_hypgeom_rising_ui() and acb_hypgeom_rising().
```

```c
void acb_rising_ui_get_mag(mag_t bound, const acb_t x, ulong n)
```

Computes an upper bound for the absolute value of the rising factorial \( z = x(x + 1)(x + 2) \cdots (x + n - 1) \). Not currently optimized for large \( n \).

9.9.19 Gamma function

```c
void acb_gamma(acb_t y, const acb_t x, slong prec)
void acb_rgamma(acb_t y, const acb_t x, slong prec)
void acb_lgamma(acb_t y, const acb_t x, slong prec)
```

Computes the gamma function \( y = \Gamma(x) \). This is an alias for \( acb_hypgeom_gamma() \).

Computes the reciprocal gamma function \( y = 1/\Gamma(x) \), avoiding division by zero at the poles of the gamma function. This is an alias for \( acb_hypgeom_rgamma() \).

Computes the logarithmic gamma function \( y = \log \Gamma(x) \). This is an alias for \( acb_hypgeom_lgamma() \).

The branch cut of the logarithmic gamma function is placed on the negative half-axis, which means that \( \log \Gamma(z) + \log z = \log \Gamma(z+1) \) holds for all \( z \), whereas \( \log \Gamma(z) \neq \log(\Gamma(z)) \) in general. In the left half plane, the reflection formula with correct branch structure is evaluated via \( acb_log_sin_pi() \).

```c
void acb_digamma(acb_t y, const acb_t x, slong prec)
void acb_log_sin_pi(acb_t res, const acb_t z, slong prec)
```

Computes the digamma function \( y = \psi(x) = (\log \Gamma(x))' = \Gamma'(x)/\Gamma(x) \).

Computes the logarithmic sine function defined by
\[
S(z) = \log(\pi) - \log(z) + \log(1 - z)
\]
which is equal to
\[
S(z) = \int_{1/2}^{z} \pi \cot(\pi t) dt
\]
where the path of integration goes through the upper half plane if \( 0 < \arg(z) \leq \pi \) and through the lower half plane if \( -\pi < \arg(z) \leq 0 \). Equivalently,
\[
S(z) = \log(\sin(\pi(z - n))) \mp n\pi i, \quad n = |\Re(z)|
\]
where the negative sign is taken if \( 0 < \arg(z) \leq \pi \) and the positive sign is taken otherwise (if the interval \( \arg(z) \) does not certainly satisfy either condition, the union of both cases is computed). After subtracting \( n \), we have \( 0 \leq \Re(z) < 1 \). In this strip, we use \( S(z) = \log(\sin(\pi(z))) \) if the imaginary part of \( z \) is small. Otherwise, we use \( S(z) = i\pi(z - 1/2) + \log((1 + e^{-2\pi z})/2) \) in the lower half-plane and the conjugated expression in the upper half-plane to avoid exponent overflow.

The function is evaluated at the midpoint and the propagated error is computed from \( S'(z) \) to get a continuous change when \( z \) is non-real and \( n \) spans more than one possible integer value.
void **acb_polygamma(acb_t res, const acb_t s, const acb_t z, slong prec)
Sets res to the value of the generalized polygamma function \( \psi(s, z) \).

If \( s \) is a nonnegative order, this is simply the \( s \)-order derivative of the digamma function. If \( s = 0 \), this function simply calls the digamma function internally. For integers \( s \geq 1 \), it calls the Hurwitz zeta function. Note that for small integers \( s \geq 1 \), it can be faster to use \( \text{acb_poly_digamma_series()} \) and read off the coefficients.

The generalization to other values of \( s \) is due to Espinosa and Moll [EM2004]:

\[
\psi(s, z) = \frac{\zeta'(s+1, z) + (\gamma + \psi(-s))\zeta(s+1, z)}{\Gamma(-s)}
\]

void **acb_barnes_g(acb_t res, const acb_t z, slong prec)
void **acb_log_barnes_g(acb_t res, const acb_t z, slong prec)

Computes Barnes \( G \)-function or the logarithmic Barnes \( G \)-function, respectively. The logarithmic version has branch cuts on the negative real axis and is continuous elsewhere in the complex plane, in analogy with the logarithmic gamma function. The functional equation

\[
\log G(z + 1) = \log \Gamma(z) + \log G(z).
\]

holds for all \( z \).

For small integers, we directly use the recurrence relation \( G(z + 1) = \Gamma(z)G(z) \) together with the initial value \( G(1) = 1 \). For general \( z \), we use the formula

\[
\log G(z) = (z - 1) \log \Gamma(z) - \zeta'(-1, z) + \zeta'(-1).
\]

9.9.20 Zeta function

void **acb_zeta(acb_t z, const acb_t s, slong prec)
Sets \( z \) to the value of the Riemann zeta function \( \zeta(s) \). Note: for computing derivatives with respect to \( s \), use \( \text{acb_poly_zeta_series()} \) or related methods.

This is a wrapper of \( \text{acb_dirichlet_zeta()} \).

void **acb_hurwitz_zeta(acb_t z, const acb_t s, const acb_t a, slong prec)
Sets \( z \) to the value of the Hurwitz zeta function \( \zeta(s, a) \). Note: for computing derivatives with respect to \( s \), use \( \text{acb_poly_zeta_series()} \) or related methods.

This is a wrapper of \( \text{acb_dirichlet_hurwitz()} \).

void **acb_bernoulli_poly_ui(acb_t res, ulong n, const acb_t x, slong prec)
Sets \( res \) to the value of the Bernoulli polynomial \( B_n(x) \).

Warning: this function is only fast if either \( n \) or \( x \) is a small integer.

This function reads Bernoulli numbers from the global cache if they are already cached, but does not automatically extend the cache by itself.

9.9.21 Polylogarithms

void **acb_polylog(acb_t w, const acb_t s, const acb_t z, slong prec)
void **acb_polylog_si(acb_t w, slong s, const acb_t z, slong prec)
Sets \( w \) to the polylogarithm \( \text{Li}_s(z) \).
9.9.22 Arithmetic-geometric mean

See *Algorithms for the arithmetic-geometric mean* for implementation details.

```c
void acb_agm1(acb_t m, const acb_t z, slong prec)
```

Sets $m$ to the arithmetic-geometric mean $M(z) = \text{agm}(1,z)$, defined such that the function is continuous in the complex plane except for a branch cut along the negative half axis (where it is continuous from above). This corresponds to always choosing an “optimal” branch for the square root in the arithmetic-geometric mean iteration.

```c
void acb_agm1_cpx(acb_ptr m, const acb_t z, slong len, slong prec)
```

Sets the coefficients in the array $m$ to the power series expansion of the arithmetic-geometric mean at the point $z$ truncated to length $len$, i.e. $M(z + x) \in \mathbb{C}[x]$.

```c
void acb_agm(acb_t m, const acb_t x, const acb_t y, slong prec)
```

Sets $m$ to the arithmetic-geometric mean of $x$ and $y$. The square roots in the AGM iteration are chosen so as to form the “optimal” AGM sequence. This gives a well-defined function of $x$ and $y$ except when $x/y$ is a negative real number, in which case there are two optimal AGM sequences. In that case, an arbitrary but consistent choice is made (if a decision cannot be made due to inexact arithmetic, the union of both choices is returned).

9.9.23 Other special functions

```c
void acb_chebyshev_t_ui(acb_t a, ulong n, const acb_t x, slong prec)
```

```c
void acb_chebyshev_u_ui(acb_t a, ulong n, const acb_t x, slong prec)
```

Evaluates the Chebyshev polynomial of the first kind $a = T_n(x)$ or the Chebyshev polynomial of the second kind $a = U_n(x)$.

```c
void acb_chebyshev_t2_ui(acb_t a, acb_t b, ulong n, const acb_t x, slong prec)
```

```c
void acb_chebyshev_u2_ui(acb_t a, acb_t b, ulong n, const acb_t x, slong prec)
```

Simultaneously evaluates $a = T_n(x), b = T_{n-1}(x)$ or $a = U_n(x), b = U_{n-1}(x)$. Aliasing between $a$, $b$ and $x$ is not permitted.

9.9.24 Piecewise real functions

The following methods extend common piecewise real functions to piecewise complex analytic functions, useful together with the *acb_calc.h* module. If *analytic* is set, evaluation on a discontinuity or non-analytic point gives a NaN result.

```c
void acb_real_abs(acb_t res, const acb_t z, int analytic, slong prec)
```

The absolute value is extended to $+z$ in the right half plane and $-z$ in the left half plane, with a discontinuity on the vertical line $\text{Re}(z) = 0$.

```c
void acb_real_sgn(acb_t res, const acb_t z, int analytic, slong prec)
```

The sign function is extended to $+1$ in the right half plane and $-1$ in the left half plane, with a discontinuity on the vertical line $\text{Re}(z) = 0$. If *analytic* is not set, this is effectively the same function as *acb_csgn()*.

```c
void acb_real_heaviside(acb_t res, const acb_t z, int analytic, slong prec)
```

The Heaviside step function (or unit step function) is extended to $+1$ in the right half plane and $-1$ in the left half plane, with a discontinuity on the vertical line $\text{Re}(z) = 0$.

```c
void acb_real_floor(acb_t res, const acb_t z, int analytic, slong prec)
```

The floor function is extended to a piecewise constant function equal to $n$ in the strips with real part $(n,n+1)$, with discontinuities on the vertical lines $\text{Re}(z) = n$. 

---

9.9. acb.h – complex numbers
The ceiling function is extended to a piecewise constant function equal to \( n + 1 \) in the strips with real part \((n, n+1)\), with discontinuities on the vertical lines \(\text{Re}(z) = n\).

The real function \(\max(x, y)\) is extended to a piecewise analytic function of two variables by returning \(x\) when \(\text{Re}(x) \geq \text{Re}(y)\) and returning \(y\) when \(\text{Re}(x) < \text{Re}(y)\), with discontinuities where \(\text{Re}(x) = \text{Re}(y)\).

The real function \(\min(x, y)\) is extended to a piecewise analytic function of two variables by returning \(x\) when \(\text{Re}(x) \leq \text{Re}(y)\) and returning \(y\) when \(\text{Re}(x) > \text{Re}(y)\), with discontinuities where \(\text{Re}(x) = \text{Re}(y)\).

Extends the real square root function on \([0, +\infty)\) to the usual complex square root on the cut plane. Like \(\text{arb_sqrtpos()}\), only the nonnegative part of \(z\) is considered if \(z\) is purely real and \(\text{analytic}\) is not set. This is useful for integrating \(\sqrt{f(x)}\) where it is known that \(f(x) \geq 0\): unlike \(\text{acb_sqrt_analytic()}\), no spurious imaginary terms \([\pm \epsilon]i\) are created when the balls computed for \(f(x)\) straddle zero.

### 9.9.25 Vector functions

Sets all entries in \(\text{vec}\) to zero.

Returns nonzero iff all entries in \(x\) are zero.

Returns nonzero iff all entries in \(x\) have zero imaginary part.

Returns nonzero iff all entries in \(x\) certainly are finite.

Returns true iff \(\text{vec1}\) overlaps \(\text{vec2}\) elementwise.

Returns true iff \(\text{vec1}\) contains \(\text{vec2}\) elementwise.

Sets \(\text{res}\) to a copy of \(\text{vec}\).

Sets \(\text{res}\) to a copy of \(\text{vec}\), rounding each entry to \(\text{prec}\) bits.

Swaps the entries of \(\text{vec1}\) and \(\text{vec2}\).

Sets each entry of \(\text{re}\) (resp. \(\text{im}\)) to the real (resp. imaginary) part of the corresponding entry of \(\text{vec}\).
void _acb_vec_set_real_imag(acb_ptr vec, arb_srcptr re, arb_srcptr im, slong len)
   Sets vec to the vector with real part re and imaginary part im.
void _acb_vec_neg(acb_ptr res, _acb_srcptr vec, slong len)
void _acb_vec_add(acb_ptr res, _acb_srcptr vec1, _acb_srcptr vec2, slong len, slong prec)
void _acb_vec_sub(acb_ptr res, _acb_srcptr vec1, _acb_srcptr vec2, slong len, slong prec)
void _acb_vec_scalar_submul(acb_ptr res, _acb_srcptr vec, slong len, const _acb_t c, slong prec)
void _acb_vec_scalar_addmul(acb_ptr res, _acb_srcptr vec, slong len, const _acb_t c, slong prec)
void _acb_vec_scalar_mul(acb_ptr res, _acb_srcptr vec, slong len, const _acb_t c, slong prec)
void _acb_vec_scalar_mul_ui(acb_ptr res, _acb_srcptr vec, ulong c, slong prec)
void _acb_vec_scalar_mul_onei(acb_ptr res, _acb_srcptr vec, slong len, slong c)
void _acb_vec_scalar_mul_2exp_si(acb_ptr res, _acb_srcptr vec, slong len, const fmpz_t c, slong prec)
   Performs the respective scalar operation elementwise.
void _acb_vec_scalar_mul_fmpz(acb_ptr res, _acb_srcptr vec, const fmpz_t c, slong prec)
void _acb_vec_scalar_mul_arb(acb_ptr res, _acb_srcptr vec, slong len, const arb_t c, slong prec)
void _acb_vec_scalar_div(acb_ptr res, _acb_srcptr vec, slong len, const _acb_t c, slong prec)
void _acb_vec_scalar_div_ui(acb_ptr res, _acb_srcptr vec, ulong c, slong prec)
void _acb_vec_scalar_div_fmpz(acb_ptr res, _acb_srcptr vec, const fmpz_t c, slong prec)
void _acb_vec_scalar_div_arb(acb_ptr res, _acb_srcptr vec, slong len, const arb_t c, slong prec)
void _acb_vec_scalar_div_onei(acb_ptr res, _acb_srcptr vec, slong len, slong c)
void _acb_vec_scalar_div_2exp_si(acb_ptr res, _acb_srcptr vec, slong len, const fmpz_t c, slong prec)
   In order to avoid precision loss, this function does not simply compute powers of a primitive root.
void _acb_vec_scalar_div_fmpz(acb_ptr res, _acb_srcptr vec, const fmpz_t c, slong prec)

slong _acb_vec_bits(_acb_srcptr vec, slong len)
   Returns the maximum of arb_bits() for all entries in vec.
void _acb_vec_set_powers(_acb_ptr xs, const _acb_t x, slong len, slong prec)
   Sets xs to the powers 1, x, x^2, ..., x^{len-1}.
void _acb_vec_unit_roots(_acb_ptr z, slong order, slong len, slong prec)
   Sets z to the powers 1, z, z^2, ..., z^{len-1} where z = \exp(\frac{2\pi}{order}) to precision prec. order can be taken negative.

void _acb_vec_add_error_arf_vec(_acb_ptr res, _arf_srcptr err, slong len)
void _acb_vec_add_error_mag_vec(_acb_ptr res, mag_srcptr err, slong len)
   Adds the magnitude of each entry in err to the radius of the corresponding entry in res.
void _acb_vec_indeterminate(_acb_ptr vec, slong len)
   Applies _acb_indeterminate() elementwise.
void _acb_vec_trim(_acb_ptr res, _acb_srcptr vec, slong len)
   Applies _acb_trim() elementwise.
int _acb_vec_get_unique_fmpz_vec(fmpz *res, _acb_srcptr vec, slong len)
   Calls _acb_get_unique_fmpz() elementwise and returns nonzero if all entries can be rounded uniquely to integers. If any entry in vec cannot be rounded uniquely to an integer, returns zero.
void _acb_vec_sort_pretty(acb_ptr vec, slong len)
Sorts the vector of complex numbers based on the real and imaginary parts. This is intended to reveal structure when printing a set of complex numbers, not to apply an order relation in a rigorous way.

void _acb_vec_printd(acb_srcptr vec, slong len, slong digits)
void _acb_vec_printn(acb_srcptr vec, slong len, slong digits, ulong flags)
Prints vec in decimal using acb_printd() or acb_printn() on each entry.

9.10 arb_poly.h – polynomials over the real numbers

An arb_poly_t represents a polynomial over the real numbers, implemented as an array of coefficients of type arb_struct.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

9.10.1 Types, macros and constants

type arb_poly_struct

type arb_poly_t

Contains a pointer to an array of coefficients (coeffs), the used length (length), and the allocated size of the array (alloc).

An arb_poly_t is defined as an array of length one of type arb_poly_struct, permitting an arb_poly_t to be passed by reference.

9.10.2 Memory management

void arb_poly_init(arb_poly_t poly)
Initializes the polynomial for use, setting it to the zero polynomial.

void arb_poly_clear(arb_poly_t poly)
Clears the polynomial, deallocating all coefficients and the coefficient array.

void arb_poly_fit_length(arb_poly_t poly, slong len)
Makes sure that the coefficient array of the polynomial contains at least len initialized coefficients.

void _arb_poly_set_length(arb_poly_t poly, slong len)
Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void _arb_poly_normalise(arb_poly_t poly)
Strips any trailing coefficients which are identical to zero.

ulong arb_poly_allocated_bytes(const arb_poly_t x)
Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add sizeof(arb_poly_struct) to get the size of the object as a whole.
9.10.3 Basic manipulation

*slong* `arb_poly_length(const arb_poly_t poly)`

Returns the length of `poly`, i.e. zero if `poly` is identically zero, and otherwise one more than the index of the highest term that is not identically zero.

*slong* `arb_poly_degree(const arb_poly_t poly)`

Returns the degree of `poly`, defined as one less than its length. Note that if one or several leading coefficients are balls containing zero, this value can be larger than the true degree of the exact polynomial represented by `poly`, so the return value of this function is effectively an upper bound.

*int* `arb_poly_is_zero(const arb_poly_t poly)`

Returns 1 if `poly` is exactly the polynomial 0, 1 or `x` respectively. Returns 0 otherwise.

*int* `arb_poly_is_one(const arb_poly_t poly)`

Sets `poly` to the constant 0 respectively 1.

*void* `arb_poly_zero(arb_poly_t poly)`

*void* `arb_poly_one(arb_poly_t poly)`

Sets `dest` to a copy of `src`.

*void* `arb_poly_set_round(arb_poly_t dest, const arb_poly_t src, slong prec)`

Sets `dest` to a copy of `src`, rounded to `prec` bits.

*void* `arb_poly_set_trunc(arb_poly_t dest, const arb_poly_t src, slong n)`

Sets `dest` to a copy of `src`, truncated to length `n` and rounded to `prec` bits.

*void* `arb_poly_set_coeff_si(arb_poly_t poly, slong n, slong c)`

Sets the coefficient with index `n` in `poly` to the value `c`. We require that `n` is nonnegative.

*void* `arb_poly_get_coeff_arb(arb_t v, const arb_poly_t poly, slong n)`

Sets `v` to the value of the coefficient with index `n` in `poly`. We require that `n` is nonnegative.

*void* `arb_poly_get_coeff_ptr(poly, n)`

Given `n ≥ 0`, returns a pointer to coefficient `n` of `poly`, or NULL if `n` exceeds the length of `poly`.

*void* `_arb_poly_shift_right(arb_ptr res, arb_srcptr poly, slong len, slong n)`

Sets `res` to `poly` divided by `x^n`, throwing away the lower coefficients. We require that `n` is nonnegative.

*void* `_arb_poly_shift_left(arb_ptr res, arb_srcptr poly, slong len, slong n)`

Sets `res` to `poly` multiplied by `x^n`. We require that `n` is nonnegative.

*void* `arb_poly_truncate(arb_poly_t poly, slong n)`

Truncates `poly` to have length at most `n`, i.e. degree strictly smaller than `n`. We require that `n` is nonnegative.

*slong* `arb_poly_valuation(const arb_poly_t poly)`

Returns the degree of the lowest term that is not exactly zero in `poly`. Returns -1 if `poly` is the zero polynomial.

9.10. `arb_poly.h` — polynomials over the real numbers
9.10.4 Conversions

```c
void arb_poly_set_fmpz_poly(arb_poly_t poly, const fmpz_poly_t src, slong prec)
void arb_poly_set_fmpq_poly(arb_poly_t poly, const fmpq_poly_t src, slong prec)
void arb_poly_set_si(arb_poly_t poly, slong src)
    Sets poly to src, rounding the coefficients to prec bits.
```

9.10.5 Input and output

```c
void arb_poly_printd(const arb_poly_t poly, slong digits)
    Prints the polynomial as an array of coefficients, printing each coefficient using arb_printd.
void arb_poly_fprintd(FILE *file, const arb_poly_t poly, slong digits)
    Prints the polynomial as an array of coefficients to the stream file, printing each coefficient using arb_fprintd.
```

9.10.6 Random generation

```c
void arb_poly_randtest(arb_poly_t poly, flint_rand_t state, slong len, slong prec, slong mag_bits)
    Creates a random polynomial with length at most len.
```

9.10.7 Comparisons

```c
int arb_poly_contains(const arb_poly_t poly1, const arb_poly_t poly2)
int arb_poly_contains_fmpz_poly(const arb_poly_t poly1, const fmpz_poly_t poly2)
int arb_poly_contains_fmpq_poly(const arb_poly_t poly1, const fmpq_poly_t poly2)
    Returns nonzero iff poly1 contains poly2.
int arb_poly_equal(const arb_poly_t A, const arb_poly_t B)
    Returns nonzero iff A and B are equal as polynomial balls, i.e. all coefficients have equal midpoint and radius.
int _arb_poly_overlaps(arb_srcptr poly1, slong len1, arb_srcptr poly2, slong len2)
int arb_poly_overlaps(const arb_poly_t poly1, const arb_poly_t poly2)
    Returns nonzero iff poly1 overlaps with poly2. The underscore function requires that len1 is at least as large as len2.
int arb_poly_get_unique_fmpz_poly(fmpz_poly_t z, const arb_poly_t x)
    If x contains a unique integer polynomial, sets z to that value and returns nonzero. Otherwise (if x represents no integers or more than one integer), returns zero, possibly partially modifying z.
```
9.10.8 Bounds

void _arb_poly_majorant(arb_ptr res, arb_srcptr poly, slong len, slong prec)

void arb_poly_majorant(arb_poly_t res, const arb_poly_t poly, slong prec)

Sets res to an exact real polynomial whose coefficients are upper bounds for the absolute values of the coefficients in poly, rounded to prec bits.

9.10.9 Arithmetic

void _arb_poly_add(arb_ptr C, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)

Sets \(\{C, \max(lenA, \text{lenB})\}\) to the sum of \(\{A, \text{lenA}\}\) and \(\{B, \text{lenB}\}\). Allows aliasing of the input and output operands.

void arb_poly_add(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong prec)

void _arb_poly_add_si(arb_poly_t C, const arb_poly_t A, slong B, slong prec)

Sets C to the sum of A and B.

void _arb_poly_sub(arb_ptr C, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)

Sets \(\{C, \max(lenA, \text{lenB})\}\) to the difference of \(\{A, \text{lenA}\}\) and \(\{B, \text{lenB}\}\). Allows aliasing of the input and output operands.

void arb_poly_sub(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong prec)

void arb_poly_add_series(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong len, slong prec)

Sets C to the sum of A and B, truncated to length len.

void arb_poly_sub_series(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong len, slong prec)

Sets C to the difference of A and B, truncated to length len.

void arb_poly_neg(arb_poly_t C, const arb_poly_t A)

Sets C to the negation of A.

void arb_poly_scalar_mul_2exp_si(arb_poly_t C, const arb_poly_t A, slong c)

Sets C to A multiplied by \(2^c\).

void arb_poly_scalar_mul(arb_poly_t C, const arb_poly_t A, const arb_t c, slong prec)

Sets C to A multiplied by c.

void arb_poly_scalar_div(arb_poly_t C, const arb_poly_t A, const arb_t c, slong prec)

Sets C to A divided by c.

void _arb_poly_mullow_classical(arb_ptr C, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong n, slong prec)

void _arb_poly_mullow_block(arb_ptr C, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong n, slong prec)

void _arb_poly_mullow(arb_ptr C, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong n, slong prec)

Sets \(\{C, n\}\) to the product of \(\{A, \text{lenA}\}\) and \(\{B, \text{lenB}\}\), truncated to length n. The output is not allowed to be aliased with either of the inputs. We require lenA ≥ lenB > 0, n > 0, lenA + lenB − 1 ≥ n.

The classical version uses a plain loop. This has good numerical stability but gets slow for large n.
The block version decomposes the product into several subproducts which are computed exactly over the integers.

It first attempts to find an integer \( c \) such that \( A(2^c x) \) and \( B(2^c x) \) have slowly varying coefficients, to reduce the number of blocks.

The scaling factor \( c \) is chosen in a quick, heuristic way by picking the first and last nonzero terms in each polynomial. If the indices in \( A \) are \( a_2, a_1 \) and the log-2 magnitudes are \( e_2, e_1 \), and the indices in \( B \) are \( b_2, b_1 \) with corresponding magnitudes \( f_2, f_1 \), then we compute \( c \) as the weighted arithmetic mean of the slopes, rounded to the nearest integer:

\[
    c = \left\lfloor \frac{(e_2 - e_1) + (f_2 + f_1)}{(a_2 - a_1) + (b_2 - b_1)} + \frac{1}{2} \right\rfloor.
\]

This strategy is used because it is simple. It is not optimal in all cases, but will typically give good performance when multiplying two power series with a similar decay rate.

The default algorithm chooses the classical algorithm for short polynomials and the block algorithm for long polynomials.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

```c
void arb_poly_mullow_classical(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong n, slong prec)

void arb_poly_mullow_ztrunc(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong n, slong prec)

void arb_poly_mullow_block(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong n, slong prec)

void arb_poly_mullow(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong n, slong prec)

void _arb_poly_mul(arb_ptr C, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)

void arb_poly_mul(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong prec)

void arb_poly_inv_series(arb_ptr Q, arb_srcptr A, slong Alen, slong len, slong prec)

void arb_poly_inv_series(arb_poly_t Q, const arb_poly_t A, slong n, slong prec)

void _arb_poly_div_series(arb_ptr Q, arb_srcptr A, slong Alen, arb_srcptr B, slong Blen, slong n, slong prec)

void arb_poly_div_series(arb_poly_t Q, const arb_poly_t A, const arb_poly_t B, slong n, slong prec)
```

Sets \( C \) to the product of \( A \) and \( B \), truncated to length \( n \). If the same variable is passed for \( A \) and \( B \), sets \( C \) to the square of \( A \) truncated to length \( n \).

Sets \( \{ C, \lfloor \log_2 \text{lenA} \rfloor \} \) to the product of \( \{ A, \text{lenA} \} \) and \( \{ B, \text{lenB} \} \). The output is not allowed to be aliased with either of the inputs. We require \( \text{lenA} \geq \text{lenB} > 0 \). This function is implemented as a simple wrapper for \_arb_poly_mullow().

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

Sets \( C \) to the product of \( A \) and \( B \). If the same variable is passed for \( A \) and \( B \), sets \( C \) to the square of \( A \).

Sets \( \{ Q, \text{len} \} \) to the power series inverse of \( \{ A, \text{Alen} \} \). Uses Newton iteration.

Sets \( Q \) to the power series inverse of \( A \), truncated to length \( n \).

Sets \( \{ Q, n \} \) to the power series quotient of \( \{ A, \text{Alen} \} \) by \( \{ B, \text{Blen} \} \). Uses Newton iteration followed by multiplication.

Sets \( Q \) to the power series quotient \( A \) divided by \( B \), truncated to length \( n \).
void _arb_poly_div(arb_ptr Q, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)
void _arb_poly_rem(arb_ptr R, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)
void _arb_poly_divrem(arb_ptr Q, arb_ptr R, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)
int arb_poly_divrem(arb_poly_t Q, arb_poly_t R, const arb_poly_t A, const arb_poly_t B, slong prec)

Performs polynomial division with remainder, computing a quotient \( Q \) and a remainder \( R \) such that \( A = BQ + R \). The implementation reverses the inputs and performs power series division.

If the leading coefficient of \( B \) contains zero (or if \( B \) is identically zero), returns 0 indicating failure without modifying the outputs. Otherwise returns nonzero.

void _arb_poly_div_root(arb_ptr Q, arb_t R, arb_srcptr A, slong len, const arb_t c, slong prec)

Divides \( A \) by the polynomial \( x - c \), computing the quotient \( Q \) as well as the remainder \( R = f(c) \).

9.10.10 Composition

void _arb_poly_taylor_shift(arb_ptr g, const arb_t c, slong n, slong prec)
void arb_poly_taylor_shift(arb_poly_t g, const arb_poly_t f, const arb_t c, slong prec)

Sets \( g \) to the Taylor shift \( f(x + c) \). The underscore methods act in-place on \( g = f \) which has length \( n \).

void _arb_poly_compose(arb_ptr res, arb_srcptr poly1, slong len1, arb_srcptr poly2, slong len2, slong prec)
void arb_poly_compose(arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, slong n, slong prec)

Sets \( res \) to the composition \( h(x) = f(g(x)) \) where \( f \) is given by \( poly1 \) and \( g \) is given by \( poly2 \). The underscore method does not support aliasing of the output with either input polynomial.

void _arb_poly_compose_series(arb_ptr res, arb_srcptr poly1, slong len1, arb_srcptr poly2, slong len2, slong n, slong prec)
void arb_poly_compose_series(arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, slong n, slong prec)

Sets \( res \) to the power series composition \( h(x) = f(g(x)) \) truncated to order \( O(x^n) \) where \( f \) is given by \( poly1 \) and \( g \) is given by \( poly2 \). Wraps _gr_poly_compose_series() which chooses automatically between various algorithms.

We require that the constant term in \( g(x) \) is exactly zero. The underscore method does not support aliasing of the output with either input polynomial.

void _arb_poly_revert_series(arb_ptr h, arb_srcptr f, slong flen, slong n, slong prec)
void arb_poly_revert_series(arb_poly_t h, const arb_poly_t f, slong n, slong prec)

Sets \( h \) to the power series reversion of \( f \), i.e., the expansion of the compositional inverse function \( f^{-1}(x) \), truncated to order \( O(x^n) \). Wraps _gr_poly_revert_series() which chooses automatically between various algorithms.

We require that the constant term in \( f \) is exactly zero and that the linear term is nonzero. The underscore methods assume that \( flen \) is at least 2, and do not support aliasing.

9.10. arb_poly.h – polynomials over the real numbers
9.10.11 Evaluation

```c
void _arb_poly_evaluate_horner(arb_t y, arb_srcptr f, slong len, const arb_t x, slong prec)
void arb_poly_evaluate_horner(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
void _arb_poly_evaluate_rectangular(arb_t y, arb_srcptr f, slong len, const arb_t x, slong prec)
void arb_poly_evaluate_rectangular(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
void _arb_poly_evaluate(arb_t y, arb_srcptr f, slong len, const arb_t x, slong prec)
void arb_poly_evaluate(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
Sets \( y = f(x) \), evaluated respectively using Horner’s rule, rectangular splitting, and an automatic algorithm choice.

void _arb_poly_evaluate_acb_horner(acb_t y, arb_srcptr f, slong len, const acb_t x, slong prec)
void arb_poly_evaluate_acb_horner(acb_t y, const arb_poly_t f, const acb_t x, slong prec)
void _arb_poly_evaluate_acb_rectangular(acb_t y, arb_srcptr f, slong len, const acb_t x, slong prec)
void arb_poly_evaluate_acb_rectangular(acb_t y, const arb_poly_t f, const acb_t x, slong prec)
void _arb_poly_evaluate_acb(arb_t y, arb_srcptr f, slong len, const acb_t x, slong prec)
void arb_poly_evaluate_acb(arb_t y, const arb_poly_t f, const acb_t x, slong prec)
Sets \( y = f(x) \) where \( x \) is a complex number, evaluating the polynomial respectively using Horner’s rule, rectangular splitting, and an automatic algorithm choice.
```

When Horner’s rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

```c
void _arb_poly_evaluate2_horner(arb_t y, arb_t z, arb_srcptr f, slong len, const arb_t x, slong prec)
void arb_poly_evaluate2_horner(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
void _arb_poly_evaluate2_rectangular(arb_t y, arb_t z, arb_srcptr f, slong len, const arb_t x, slong prec)
void arb_poly_evaluate2_rectangular(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
void _arb_poly_evaluate2(arb_t y, arb_t z, arb_srcptr f, slong len, const arb_t x, slong prec)
void arb_poly_evaluate2(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
Sets \( y = f(x) \), \( z = f'(x) \), evaluated respectively using Horner’s rule, rectangular splitting, and an automatic algorithm choice.
```

When Horner’s rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.
void \texttt{arb\_poly\_evaluate2\_acb\_rectangular}(acb\_t y, acb\_t z, const arb\_poly\_t f, const acb\_t x, slong prec)

void \texttt{arb\_poly\_evaluate2\_acb}(acb\_t y, acb\_t z, const arb\_poly\_t f, const acb\_t x, slong len, const acb\_t x, slong prec)

void \texttt{arb\_poly\_evaluate2\_acb}(acb\_t y, acb\_t z, const arb\_poly\_t f, const acb\_t x, slong prec)

Sets \( y = f(x), z = f'(x) \), evaluated respectively using Horner’s rule, rectangular splitting, and an automatic algorithm choice.

\subsection*{9.10.12 Product trees}

void \texttt{arb\_poly\_product\_roots\_complex}(arb\_ptr poly, arb\_srcptr r, slong rn, acb\_srcptr c, slong cn, slong prec)

Generates the polynomial
\[
\prod_{i=0}^{\text{rn}-1} (x - r_i) \prod_{i=0}^{\text{cn}-1} (x - c_i)(x - \overline{c_i})
\]
having \( \text{rn} \) real roots given by the array \( r \) and having \( 2\text{cn} \) complex roots in conjugate pairs given by the length-\( \text{cn} \) array \( c \). Either \( \text{rn} \) or \( \text{cn} \) or both may be zero.

Note that only one representative from each complex conjugate pair is supplied (unless a pair is supposed to be repeated with higher multiplicity). To construct a polynomial from complex roots where the conjugate pairs have not been distinguished, use \texttt{acb\_poly\_product\_roots()} instead.

\begin{verbatim}
arb\_ptr * arb\_poly\_tree\_alloc\_{\text{len}}
\end{verbatim}

Returns an initialized data structured capable of representing a remainder tree (product tree) of \( \text{len} \) roots.

void \texttt{arb\_poly\_tree\_free}(arb\_ptr *tree, slong len)

Deallocates a tree structure as allocated using \texttt{arb\_poly\_tree\_alloc}.

void \texttt{arb\_poly\_tree\_build}(arb\_ptr *tree, arb\_srcptr roots, slong len, slong prec)

Constructs a product tree from a given array of \( \text{len} \) roots. The tree structure must be pre-allocated to the specified length using \texttt{arb\_poly\_tree\_alloc()}.

\subsection*{9.10.13 Multipoint evaluation}

void \texttt{arb\_poly\_evaluate\_vec\_iter}(arb\_ptr ys, arb\_srcptr poly, slong plen, arb\_srcptr xs, slong n, slong prec)

void \texttt{arb\_poly\_evaluate\_vec\_iter}(arb\_ptr ys, const arb\_poly\_t poly, arb\_srcptr xs, slong n, slong prec)

Evaluates the polynomial simultaneously at \( n \) given points, calling \texttt{arb\_poly\_evaluate()} repeatedly.

void \texttt{arb\_poly\_evaluate\_vec\_fast\_precomp}(arb\_ptr vs, arb\_srcptr poly, slong plen, arb\_ptr *tree, slong len, slong prec)
void _arb_poly_evaluate_vec_fast(arb_ptr ys, arb_srcptr poly, slong plen, arb_srcptr xs, slong n, slong prec)

void arb_poly_evaluate_vec_fast(arb_ptr ys, const arb_poly_t poly, arb_srcptr xs, slong n, slong prec)

Evaluates the polynomial simultaneously at $n$ given points, using fast multipoint evaluation.

### 9.10.14 Interpolation

void _arb_poly_interpolate_newton(arb_ptr poly, arb_srcptr xs, arb_srcptr ys, slong n, slong prec)

void arb_poly_interpolate_newton(arb_poly_t poly, arb_srcptr xs, arb_srcptr ys, slong n, slong prec)

Recover the unique polynomial of length at most $n$ that interpolates the given $x$ and $y$ values.

This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

void _arb_poly_interpolate_barycentric(arb_ptr poly, arb_srcptr xs, arb_srcptr ys, slong n, slong prec)

void arb_poly_interpolate_barycentric(arb_poly_t poly, arb_srcptr xs, arb_srcptr ys, slong n, slong prec)

Recover the unique polynomial of length at most $n$ that interpolates the given $x$ and $y$ values.

This implementation uses the barycentric form of Lagrange interpolation.

void _arb_poly_interpolation_weights(arb_ptr w, arb_ptr *tree, slong len, slong prec)

void _arb_poly_interpolate_fast_precomp(arb_ptr poly, arb_srcptr ys, arb_ptr *tree, arb_srcptr weights, slong len, slong prec)

void _arb_poly_interpolate_fast(arb_ptr poly, arb_srcptr xs, arb_srcptr ys, slong n, slong prec)

void _arb_poly_interpolate_fast(arb_poly_t poly, arb_srcptr xs, arb_srcptr ys, slong n, slong prec)

Recover the unique polynomial of length at most $n$ that interpolates the given $x$ and $y$ values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the $x$ values and a vector of interpolation weights as additional inputs.

### 9.10.15 Differentiation

void _arb_poly_derivative(arb_ptr res, arb_srcptr poly, slong len, slong prec)

Sets $\{res, len - 1\}$ to the derivative of $\{poly, len\}$. Allows aliasing of the input and output.

void arb_poly_derivative(arb_poly_t res, const arb_poly_t poly, slong prec)

Sets $res$ to the derivative of $poly$.

void _arb_poly_nth_derivative(arb_ptr res, arb_srcptr poly, ulong n, slong len, slong prec)

Sets $\{res, len - n\}$ to the $n$th derivative of $\{poly, len\}$. Does nothing if $len <= n$. Allows aliasing of the input and output.

void arb_poly_nth_derivative(arb_poly_t res, const arb_poly_t poly, ulong n, slong prec)

Sets $res$ to the $n$th derivative of $poly$.

void _arb_poly_integral(arb_ptr res, arb_srcptr poly, slong len, slong prec)

Sets $\{res, len\}$ to the integral of $\{poly, len - 1\}$. Allows aliasing of the input and output.

void arb_poly_integral(arb_poly_t res, const arb_poly_t poly, slong prec)

Sets $res$ to the integral of $poly$.
Computes the Borel transform of the input polynomial, mapping \( \sum_k a_k x^k \) to \( \sum_k \frac{a_k}{k!} x^k \). The underscore method allows aliasing.

Computes the inverse Borel transform of the input polynomial, mapping \( \sum_k a_k x^k \) to \( \sum_k a_k k! x^k \). The underscore method allows aliasing.

Computes the binomial transform of the input polynomial, truncating the output to length \( len \). The binomial transform maps the coefficients \( a_k \) in the input polynomial to the coefficients \( b_k \) in the output polynomial via \( b_n = \sum_{k=0}^{n} (-1)^k \binom{n}{k} a_k \). The binomial transform is equivalent to the power series composition \( f(x) \rightarrow (1 - x)^{-1} f(x/(x - 1)) \), and is its own inverse.

The basecase version evaluates coefficients one by one from the definition, generating the binomial coefficients by a recurrence relation.

The convolution version uses the identity \( (e^x B(f(x))) \) where \( T \) denotes the binomial transform operator and \( B \) denotes the Borel transform operator. This only costs a single polynomial multiplication, plus some scalar operations.

The default version automatically chooses an algorithm.

Both methods allow aliasing.

Computes the Graeffe transform of input polynomial.

The Graeffe transform \( G \) of a polynomial \( P \) is defined through the equation \( G(x^2) = \pm P(x)P(-x) \). The sign is given by \( (-1)^d \), where \( d = \deg(P) \). The Graeffe transform has the property that its roots are exactly the squares of the roots of \( P \).

The underscore method assumes that \( a \) and \( b \) are initialized, \( a \) is of length \( len \), and \( b \) is of length at least \( len \). Both methods allow aliasing.
9.10.17 Powers and elementary functions

void _arb_poly_pow_ui_trunc_binexp(arb_ptr res, arb_srcptr f, slong flen, ulong exp, slong len, slong prec)

Sets \(\{res, len\}\) to \(\{f, flen\}\) raised to the power \(exp\), truncated to length \(len\). Requires that \(len\) is no longer than the length of the power as computed without truncation (i.e. no zero-padding is performed). Does not support aliasing of the input and output, and requires that \(flen\) and \(len\) are positive. Uses binary exponentiation.

void arb_poly_pow_ui_trunc_binexp(arb_poly_t res, const arb_poly_t poly, ulong exp, slong len, slong prec)

Sets \(res\) to \(poly\) raised to the power \(exp\), truncated to length \(len\). Uses binary exponentiation.

void _arb_poly_ui(arb_ptr res, arb_srcptr f, slong flen, ulong exp, slong prec)

Sets \(res\) to \(\{f, flen\}\) raised to the power \(exp\). Does not support aliasing of the input and output, and requires that \(flen\) is positive.

void arb_poly_pow_ui(arb_poly_t res, const arb_poly_t poly, ulong exp, slong prec)

Sets \(res\) to \(poly\) raised to the power \(exp\).

void _arb_poly_pow_series(arb_ptr h, arb_srcptr f, slong flen, arb_srcptr g, slong glen, slong len, slong prec)

Sets \(\{h, len\}\) to the power series \(f(x)^g(x) = \exp(g(x) \log f(x))\) truncated to length \(len\). This function detects special cases such as \(g\) being an exact small integer or \(\pm 1/2\), and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that \(flen\) and \(glen\) do not exceed \(len\).

void arb_poly_pow_series(arb_poly_t h, const arb_poly_t f, const arb_poly_t g, slong len, slong prec)

Sets \(h\) to the power series \(f(x)^g(x) = \exp(g(x) \log f(x))\) truncated to length \(len\). This function detects special cases such as \(g\) being an exact small integer or \(\pm 1/2\), and computes such powers more efficiently.

void _arb_poly_pow_arb_series(arb_ptr h, arb_srcptr f, slong flen, const arb_t g, slong len, slong prec)

Sets \(\{h, len\}\) to the power series \(f(x)^g = \exp(g \log f(x))\) truncated to length \(len\). This function detects special cases such as \(g\) being an exact small integer or \(\pm 1/2\), and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that \(flen\) does not exceed \(len\).

void arb_poly_pow_arb_series(arb_poly_t h, const arb_poly_t f, const arb_t g, slong len, slong prec)

Sets \(h\) to the power series \(f(x)^g = \exp(g \log f(x))\) truncated to length \(len\).

void _arb_poly_sqrt_series(arb_ptr g, arb_srcptr h, slong hlen, slong n, slong prec)

void arb_poly_sqrt_series(arb_poly_t g, const arb_poly_t h, slong n, slong prec)

Sets \(g\) to the power series square root of \(h\), truncated to length \(n\). Uses division-free Newton iteration for the reciprocal square root, followed by a multiplication. The underscore method does not support aliasing of the input and output arrays. It requires that \(hlen\) and \(n\) are greater than zero.

void _arb_poly_rsqrt_series(arb_ptr g, arb_srcptr h, slong hlen, slong n, slong prec)

void arb_poly_rsqrt_series(arb_poly_t g, const arb_poly_t h, slong n, slong prec)

Sets \(g\) to the reciprocal power series square root of \(h\), truncated to length \(n\). Uses division-free Newton iteration.
The underscore method does not support aliasing of the input and output arrays. It requires that \( hlen \) and \( n \) are greater than zero.

```c
void _arb_poly_log_series( arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec )
```

`void arb_poly_log_series( arb_poly_t res, const arb_poly_t f, slong n, slong prec )`

Sets \( res \) to the power series logarithm of \( f \), truncated to length \( n \). Uses the formula \( \log(f(x)) = \int f'(x)/f(x)dx \), adding the logarithm of the constant term in \( f \) as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that \( flen \) and \( n \) are greater than zero.

```c
void _arb_poly_log1p_series( arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec )
```

`void arb_poly_log1p_series( arb_poly_t res, const arb_poly_t f, slong n, slong prec )`

Computes the power series logarithm of \( 1 + f \), with better accuracy when the constant term of \( f \) is small.

The underscore method supports aliasing of the input and output arrays. They require that \( flen \) and \( n \) are greater than zero.

```c
void _arb_poly_atan_series( arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec )
```

`void arb_poly_atan_series( arb_poly_t res, const arb_poly_t f, slong n, slong prec )`

Sets \( res \) respectively to the power series inverse tangent, inverse sine and inverse cosine of \( f \), truncated to length \( n \).

Uses the formulas

\[
\begin{align*}
\tan^{-1}(f(x)) &= \int f'(x)/(1 + f(x)^2)dx, \\
\sin^{-1}(f(x)) &= \int f'(x)/(1 - f(x)^2)^{1/2}dx, \\
\cos^{-1}(f(x)) &= -\int f'(x)/(1 - f(x)^2)^{1/2}dx,
\end{align*}
\]

adding the inverse function of the constant term in \( f \) as the constant of integration.

The underscore method supports aliasing of the input and output arrays. They require that \( flen \) and \( n \) are greater than zero.

```c
void _arb_poly_exp_series_basecase( arb_ptr f, arb_srcptr h, slong hlen, slong n, slong prec )
```

`void arb_poly_exp_series_basecase( arb_poly_t f, const arb_poly_t h, slong n, slong prec )`

Saves \( f \) to the power series exponential of \( h \), truncated to length \( n \).

The basecase version uses a simple recurrence for the coefficients, requiring \( O(nm) \) operations where \( m \) is the length of \( h \).

The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is \( O(M(n)) \). Redundant operations in the Newton iteration are avoided by using the scheme described in [HZ2004].

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

```c
void _arb_poly_sin_cos_series( arb_ptr s, arb_ptr c, arb_srcptr h, slong hlen, slong n, slong prec )
```

### 9.10. `arb_poly.h` – polynomials over the real numbers

589
void \texttt{arb\_poly\_sin\_cos\_series}(\texttt{arb\_poly\_t} s, \texttt{arb\_poly\_t} c, const \texttt{arb\_poly\_t} h, \texttt{slong} n, \texttt{slong} prec)

Sets \(s\) and \(c\) to the power series sine and cosine of \(h\), computed simultaneously. The underscore method supports aliasing and requires the lengths to be nonzero.

void \_\texttt{arb\_poly\_sin\_series}(\texttt{arb\_ptr} s, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_cos\_series}(\texttt{arb\_ptr} c, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} n, \texttt{slong} prec)

Respectively evaluates the power series sine or cosine. These functions simply wrap \texttt{arb\_poly\_sin\_cos\_series()}. The underscore methods support aliasing and require the lengths to be nonzero.

void \_\texttt{arb\_poly\_tan\_series}(\texttt{arb\_ptr} g, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} len, \texttt{slong} prec)

Sets \(g\) to the power series tangent of \(h\).

For small \(n\) takes the quotient of the sine and cosine as computed using the basecase algorithm. For large \(n\), uses Newton iteration to invert the inverse tangent series. The complexity is \(O(M(n))\).

The underscore version does not support aliasing, and requires the lengths to be nonzero.

void \_\texttt{arb\_poly\_sin\_cos\_pi\_series}(\texttt{arb\_ptr} s, \texttt{arb\_ptr} c, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_sin\_pi\_series}(\texttt{arb\_ptr} s, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_cos\_pi\_series}(\texttt{arb\_ptr} c, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_cot\_pi\_series}(\texttt{arb\_ptr} c, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} n, \texttt{slong} prec)

Compute the respective trigonometric functions of the input multiplied by \(\pi\).

void \_\texttt{arb\_poly\_sinh\_cosh\_series\_basecase}(\texttt{arb\_ptr} s, \texttt{arb\_ptr} c, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_sinh\_cosh\_series\_basecase}(\texttt{arb\_poly\_t} s, \texttt{arb\_poly\_t} c, const \texttt{arb\_poly\_t} h, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_sinh\_cosh\_series\_exponential}(\texttt{arb\_ptr} s, \texttt{arb\_ptr} c, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_sinh\_cosh\_series\_exponential}(\texttt{arb\_poly\_t} s, \texttt{arb\_poly\_t} c, const \texttt{arb\_poly\_t} h, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_sinh\_cosh\_series}(\texttt{arb\_ptr} s, \texttt{arb\_ptr} c, \texttt{arb\_srcptr} h, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_sinh\_cosh\_series}(\texttt{arb\_poly\_t} s, \texttt{arb\_poly\_t} c, const \texttt{arb\_poly\_t} h, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_sinh\_series}(\texttt{arb\_ptr} s, \texttt{arb\_srcptr} h, \texttt{slong} hlen, \texttt{slong} n, \texttt{slong} prec)

void \_\texttt{arb\_poly\_sinh\_series}(\texttt{arb\_poly\_t} s, \texttt{arb\_poly\_t} c, const \texttt{arb\_poly\_t} h, \texttt{slong} n, \texttt{slong} prec)
void \texttt{arb\_poly\_sinh\_series}(\texttt{arb\_poly\_t} \ s, \texttt{const arb\_poly\_t} \ h, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{_arb\_poly\_cosh\_series}(\texttt{arb\_ptr} \ c, \texttt{arb\_srcptr} \ h, \texttt{slong} \ hlen, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{arb\_poly\_cosh\_series}(\texttt{arb\_poly\_t} \ c, \texttt{const arb\_poly\_t} \ h, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

Sets \( s \) and \( c \) respectively to the hyperbolic sine and cosine of the power series \( h \), truncated to length \( n \).

The implementations mirror those for sine and cosine, except that the exponential version computes both functions using the exponential function instead of the hyperbolic tangent.

void \texttt{_arb\_poly\_sinc\_series}(\texttt{arb\_ptr} \ s, \texttt{arb\_srcptr} \ h, \texttt{slong} \ hlen, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{arb\_poly\_sinc\_series}(\texttt{arb\_poly\_t} \ s, \texttt{const arb\_poly\_t} \ h, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

Sets \( c \) to the sinc function of the power series \( h \), truncated to length \( n \).

void \texttt{_arb\_poly\_sinc\_pi\_series}(\texttt{arb\_ptr} \ s, \texttt{arb\_srcptr} \ h, \texttt{slong} \ hlen, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{arb\_poly\_sinc\_pi\_series}(\texttt{arb\_poly\_t} \ s, \texttt{const arb\_poly\_t} \ h, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

Compute the sinc function of the input multiplied by \( \pi \).

9.10.18 Lambert W function

void \texttt{_arb\_poly\_lambertw\_series}(\texttt{arb\_ptr} \ res, \texttt{arb\_srcptr} \ z, \texttt{slong} \ zlen, \texttt{int} flags, \texttt{slong} \ len, \texttt{slong} \ \texttt{prec})

void \texttt{arb\_poly\_lambertw\_series}(\texttt{arb\_poly\_t} \ res, \texttt{const arb\_poly\_t} \ z, \texttt{int} flags, \texttt{slong} \ len, \texttt{slong} \ \texttt{prec})

Sets \( res \) to the Lambert W function of the power series \( z \). If \texttt{flags} is 0, the principal branch is computed; if \texttt{flags} is 1, the second real branch \( W_1(z) \) is computed. The underscore method allows aliasing, but assumes that the lengths are nonzero.

9.10.19 Gamma function and factorials

void \texttt{_arb\_poly\_gamma\_series}(\texttt{arb\_ptr} \ res, \texttt{arb\_srcptr} \ h, \texttt{slong} \ hlen, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{arb\_poly\_gamma\_series}(\texttt{arb\_poly\_t} \ res, \texttt{const arb\_poly\_t} \ h, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{_arb\_poly\_rgamma\_series}(\texttt{arb\_ptr} \ res, \texttt{arb\_srcptr} \ h, \texttt{slong} \ hlen, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{arb\_poly\_rgamma\_series}(\texttt{arb\_poly\_t} \ res, \texttt{const arb\_poly\_t} \ h, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{_arb\_poly\_lgamma\_series}(\texttt{arb\_ptr} \ res, \texttt{arb\_srcptr} \ h, \texttt{slong} \ hlen, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{arb\_poly\_lgamma\_series}(\texttt{arb\_poly\_t} \ res, \texttt{const arb\_poly\_t} \ h, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{_arb\_poly\_digamma\_series}(\texttt{arb\_ptr} \ res, \texttt{arb\_srcptr} \ h, \texttt{slong} \ hlen, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

void \texttt{arb\_poly\_digamma\_series}(\texttt{arb\_poly\_t} \ res, \texttt{const arb\_poly\_t} \ h, \texttt{slong} \ n, \texttt{slong} \ \texttt{prec})

Sets \( res \) to the series expansion of \( \Gamma(h(x)), 1/\Gamma(h(x)), \text{ or } \log \Gamma(h(x)), \psi(h(x)) \), truncated to length \( n \).

These functions first generate the Taylor series at the constant term of \( h \), and then call \texttt{_arb\_poly\_compose\_series()}\. The Taylor coefficients are generated using the Riemann zeta function if the constant term of \( h \) is a small integer, and with Stirling’s series otherwise.

The underscore methods support aliasing of the input and output arrays, and require that \texttt{hlen} and \texttt{n} are greater than zero.

void \texttt{_arb\_poly\_rising\_ui\_series}(\texttt{arb\_ptr} \ res, \texttt{arb\_srcptr} \ f, \texttt{slong} \ flen, \texttt{ulong} \ r, \texttt{slong} \ \texttt{trunc}, \texttt{slong} \ \texttt{prec})
void **arb_poly_rising_ui_series**(arb_poly_t res, const arb_poly_t f, ulong r, slong trunc, slong prec)

Sets res to the rising factorial $(f)(f + 1)(f + 2) \cdots (f + r - 1)$, truncated to length trunc. The underscore method assumes that flen, r and trunc are at least 1, and does not support aliasing. Uses binary splitting.

### 9.10.20 Zeta function

void **arb_poly_zeta_series**(arb_poly_t res, const arb_poly_t s, const arb_t a, int deflate, slong n, slong prec)

Sets res to the Hurwitz zeta function $\zeta(s, a)$ where $s$ a power series and $a$ is a constant, truncated to length $n$. To evaluate the usual Riemann zeta function, set $a = 1$.

If $\text{deflate}$ is nonzero, evaluates $\zeta(s, a) + 1/(1 - s)$, which is well-defined as a limit when the constant term of $s$ is 1. In particular, expanding $\zeta(s, a) + 1/(1 - s)$ with $s = 1 + x$ gives the Stieltjes constants

$$
\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a) x^k.
$$

If $a = 1$, this implementation uses the reflection formula if the midpoint of the constant term of $s$ is negative.

void **_arb_poly_riemann_siegel_theta_series**(arb_ptr res, arb_srcptr h, slong hlen, slong n, slong prec)

void **arb_poly_riemann_siegel_theta_series**(arb_poly_t res, const arb_poly_t h, slong n, slong prec)

Sets res to the series expansion of the Riemann-Siegel theta function

$$
\theta(h) = \arg \left( \Gamma \left( \frac{2ih + 1}{4} \right) \right) - \log \frac{\pi}{2} h
$$

where the argument of the gamma function is chosen continuously as the imaginary part of the log gamma function.

The underscore method does not support aliasing of the input and output arrays, and requires that the lengths are greater than zero.

void **_arb_poly_riemann_siegel_z_series**(arb_ptr res, arb_srcptr h, slong hlen, slong n, slong prec)

void **arb_poly_riemann_siegel_z_series**(arb_poly_t res, const arb_poly_t h, slong n, slong prec)

Sets res to the series expansion of the Riemann-Siegel Z-function

$$
Z(h) = e^{i \theta(h)} \zeta(1/2 + ih).
$$

The zeros of the Z-function on the real line precisely correspond to the imaginary parts of the zeros of the Riemann zeta function on the critical line.

The underscore method supports aliasing of the input and output arrays, and requires that the lengths are greater than zero.
9.10.21 Root-finding

void _arb_poly_root_bound_fujiwara(mag_t bound, arb_srcptr poly, slong len)

void arb_poly_root_bound_fujiwara(mag_t bound, arb_poly_t poly)

Sets bound to an upper bound for the magnitude of all the complex roots of poly. Uses Fujiwara’s bound

\[ 2 \max \left\{ \frac{|a_{n-1}|}{a_n}, \frac{|a_{n-2}|}{a_n}^{1/2}, \ldots, \frac{|a_1|}{a_n}^{1/(n-1)}, \frac{|a_0|}{2a_n}^{1/n} \right\} \]

where \( a_0, \ldots, a_n \) are the coefficients of poly.

void _arb_poly_newton_convergence_factor(arf_t convergence_factor, arb_srcptr poly, slong len, const arb_t convergence_interval, slong prec)

Given an interval \( I \) specified by convergence_interval, evaluates a bound for \( C = \sup_{t, u \in I} \frac{1}{2} \left( \frac{f''(t)}{|f'(u)|} \right) \), where \( f \) is the polynomial defined by the coefficients \( \{ \text{poly}, \text{len} \} \). The bound is obtained by evaluating \( f'(I) \) and \( f''(I) \) directly. If \( f \) has large coefficients, \( I \) must be extremely precise in order to get a finite factor.

int _arb_poly_newton_step(arb_t xnew, arb_srcptr poly, slong len, const arb_t x, const arb_t convergence_interval, const arf_t convergence_factor, slong prec)

Performs a single step with Newton’s method.

The input consists of the polynomial \( f \) specified by the coefficients \( \{ \text{poly}, \text{len} \} \), an interval \( x = [m - r, m + r] \) known to contain a single root of \( f \), an interval \( I \) (convergence_interval) containing \( x \) with an associated bound (convergence_factor) for \( C = \sup_{t, u \in I} \frac{1}{2} \left( \frac{f''(t)}{|f'(u)|} \right) \), and a working precision \( \text{prec} \).

The Newton update consists of setting \( x' = [m' - r', m' + r'] \) where \( m' = m - f(m)/f'(m) \) and \( r' = Cr^2 \). The expression \( m - f(m)/f'(m) \) is evaluated using ball arithmetic at a working precision of \( \text{prec} \) bits, and the rounding error during this evaluation is accounted for in the output. We now check that \( x' \in I \) and \( m' < m \). If both conditions are satisfied, we set \( xnew \) to \( x' \) and return nonzero. If either condition fails, we set \( xnew \) to \( x \) and return zero, indicating that no progress was made.

void _arb_poly_newton_refine_root(arb_t r, arb_srcptr poly, slong len, const arb_t start, const arb_t convergence_interval, const arf_t convergence_factor, slong eval_extra_prec, slong prec)

Refines a precise estimate of a polynomial root to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for _arb_poly_newton_step, except for the precision parameters: \( \text{prec} \) is the target accuracy and \( \text{eval_extra_prec} \) is the estimated number of guard bits that need to be added to evaluate the polynomial accurately close to the root (typically, if the polynomial has large coefficients of alternating signs, this needs to be approximately the bit size of the coefficients).

9.10.22 Other special polynomials

void _arb_poly_swinerton_dyer_ui(arb_ptr poly, ulong n, slong trunc, slong prec)

void arb_poly_swinerton_dyer_ui(arb_poly_t poly, ulong n, slong prec)

Computes the Swinerton-Dyer polynomial \( S_n \), which has degree \( 2^n \) and is the rational minimal polynomial of the sum of the square roots of the first \( n \) prime numbers.

If \( \text{prec} \) is set to zero, a precision is chosen automatically such that arb_poly_get_unique_fmpz_poly() should be successful. Otherwise a working precision of \( \text{prec} \) bits is used.

The underscore version accepts an additional \( \text{trunc} \) parameter. Even when computing a truncated polynomial, the array \( \text{poly} \) must have room for \( 2^n + 1 \) coefficients, used as temporary space.
9.11 acb_poly.h – polynomials over the complex numbers

An `acb_poly_t` represents a polynomial over the complex numbers, implemented as an array of coefficients of type `acb_struct`.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

9.11.1 Types, macros and constants

type `acb_poly_struct`

type `acb_poly_t`

Contains a pointer to an array of coefficients (coeffs), the used length (length), and the allocated size of the array (alloc).

An `acb_poly_t` is defined as an array of length one of type `acb_poly_struct`, permitting an `acb_poly_t` to be passed by reference.

9.11.2 Memory management

void `acb_poly_init`(acb_poly_t poly)

Initializes the polynomial for use, setting it to the zero polynomial.

void `acb_poly_clear`(acb_poly_t poly)

Clears the polynomial, deallocating all coefficients and the coefficient array.

void `acb_poly_fit_length`(acb_poly_t poly, slong len)

Makes sure that the coefficient array of the polynomial contains at least `len` initialized coefficients.

void `acb_poly_set_length`(acb_poly_t poly, slong len)

Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void `acb_poly_normalise`(acb_poly_t poly)

Strips any trailing coefficients which are identical to zero.

void `acb_poly_swap`(acb_poly_t poly1, acb_poly_t poly2)

Swaps `poly1` and `poly2` efficiently.

slong `acb_poly_allocated_bytes`(const acb_poly_t x)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(acb_poly_struct)` to get the size of the object as a whole.

9.11.3 Basic properties and manipulation

slong `acb_poly_length`(const acb_poly_t poly)

Returns the length of `poly`, i.e. zero if `poly` is identically zero, and otherwise one more than the index of the highest term that is not identically zero.

slong `acb_poly_degree`(const acb_poly_t poly)

Returns the degree of `poly`, defined as one less than its length. Note that if one or several leading coefficients are balls containing zero, this value can be larger than the true degree of the exact polynomial represented by `poly`, so the return value of this function is effectively an upper bound.
int acb_poly_is_zero(const acb_poly_t poly)
int acb_poly_is_one(const acb_poly_t poly)
int acb_poly_is_x(const acb_poly_t poly)

Returns 1 if \( poly \) is exactly the polynomial 0, 1 or \( x \) respectively. Returns 0 otherwise.

doacb_poly_zero(acb_poly_t poly)

Sets \( poly \) to the zero polynomial.

doacb_poly_one(acb_poly_t poly)

Sets \( poly \) to the constant polynomial 1.

doacb_poly_set(acb_poly_t dest, const acb_poly_t src)

Sets \( dest \) to a copy of \( src \).

doacb_poly_set_round(acb_poly_t dest, const acb_poly_t src, slong prec)

Sets \( dest \) to a copy of \( src \), rounded to \( prec \) bits.

doacb_poly_set_trunc(acb_poly_t dest, const acb_poly_t src, slong n)

doacb_poly_set_trunc_round(acb_poly_t dest, const acb_poly_t src, slong n, slong prec)

Sets \( dest \) to a copy of \( src \), truncated to length \( n \) and rounded to \( prec \) bits.

void acb_poly_set_coeff_si(acb_poly_t poly, slong n, slong c)

Sets the coefficient with index \( n \) in \( poly \) to the value \( c \). We require that \( n \) is nonnegative.

void acb_poly_set_coeff_acb(acb_poly_t poly, slong n, const acb_t c)

Sets \( c \) to the value of the coefficient with index \( n \) in \( poly \). We require that \( n \) is nonnegative.

acb_poly_get_coeff_ptr(const acb_poly_t poly, slong n)

Given \( n \geq 0 \), returns a pointer to coefficient \( n \) of \( poly \), or NULL if \( n \) exceeds the length of \( poly \).

void _acb_poly_shift_right(acb_ptr res, acb_srcptr poly, slong len, slong n)

Sets \( res \) to \( poly \) divided by \( x^n \), throwing away the lower coefficients. We require that \( n \) is nonnegative.

void _acb_poly_shift_left(acb_ptr res, acb_srcptr poly, slong len, slong n)

Sets \( res \) to \( poly \) multiplied by \( x^n \). We require that \( n \) is nonnegative.

void acb_poly_truncate(acb_poly_t poly, slong n)

Truncates \( poly \) to have length at most \( n \), i.e. degree strictly smaller than \( n \). We require that \( n \) is nonnegative.

slong acb_poly_valuation(const acb_poly_t poly)

Returns the degree of the lowest term that is not exactly zero in \( poly \). Returns -1 if \( poly \) is the zero polynomial.
9.11.4 Input and output

void `acb_poly_printd`(const `acb_poly_t` poly, `slong` digits)
  Prints the polynomial as an array of coefficients, printing each coefficient using `acb_printd`.

void `acb_poly_fprintd`(FILE *file, const `acb_poly_t` poly, `slong` digits)
  Prints the polynomial as an array of coefficients to the stream file, printing each coefficient using `acb_fprintd`.

9.11.5 Random generation

void `acb_poly_randtest`(acb_poly_t poly, flint_rand_t state, `slong` len, `slong` prec, `slong` mag_bits)
  Creates a random polynomial with length at most len.

9.11.6 Comparisons

int `acb_poly_equal`(const `acb_poly_t` A, const `acb_poly_t` B)
  Returns nonzero iff A and B are identical as interval polynomials.

int `acb_poly_contains`(const `acb_poly_t` poly1, const `acb_poly_t` poly2)

int `acb_poly_contains_fmpz_poly`(const `acb_poly_t` poly1, const fmpz_poly_t poly2)

int `acb_poly_contains_fmpq_poly`(const `acb_poly_t` poly1, const fmpq_poly_t poly2)
  Returns nonzero iff poly2 is contained in poly1.

int _acb_poly_overlaps(acb_srcptr poly1, `slong` len1, acb_srcptr poly2, `slong` len2)

int `acb_poly_overlaps`(const `acb_poly_t` poly1, const `acb_poly_t` poly2)
  Returns nonzero iff poly1 overlaps with poly2. The underscore function requires that len1 is at least as large as len2.

int `acb_poly_get_unique_fmpz_poly`(fmpz_poly_t z, const `acb_poly_t` x)
  If x contains a unique integer polynomial, sets z to that value and returns nonzero. Otherwise (if x represents no integers or more than one integer), returns zero, possibly partially modifying z.

int `acb_poly_is_real`(const `acb_poly_t` poly)
  Returns nonzero iff all coefficients in poly have zero imaginary part.

9.11.7 Conversions

void `acb_poly_set_fmpz_poly`(acb_poly_t poly, const fmpz_poly_t re, `slong` prec)

void `acb_poly_set2_fmpz_poly`(acb_poly_t poly, const fmpz_poly_t re, const fmpz_poly_t im, `slong` prec)

void `acb_poly_set_arb_poly`(acb_poly_t poly, const arb_poly_t re)

void `acb_poly_set2_arb_poly`(acb_poly_t poly, const arb_poly_t re, const arb_poly_t im)

void `acb_poly_set_fmpq_poly`(acb_poly_t poly, const fmpq_poly_t re, `slong` prec)

void `acb_poly_set2_fmpq_poly`(acb_poly_t poly, const fmpq_poly_t re, const fmpq_poly_t im, `slong` prec)
  Sets poly to the given real part re plus the imaginary part im, both rounded to prec bits.

void `acb_poly_set_acb`(acb_poly_t poly, const acb_t src)
void acb_poly_set_si(acb_poly_t poly, slong src)
    Sets poly to src.

9.11.8 Bounds

void _acb_poly_majorant(arb_ptr res, arb_srcptr poly, slong len, slong prec)

void acb_poly_majorant(arb_poly_t res, const acb_poly_t poly, slong prec)
    Sets res to an exact real polynomial whose coefficients are upper bounds for the absolute values of
    the coefficients in poly, rounded to prec bits.

9.11.9 Arithmetic

void _acb_poly_add(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)
    Sets \(\{C, \max(lenA, lenB)\}\) to the sum of \(\{A, lenA\}\) and \(\{B, lenB\}\). Allows aliasing of the input
    and output operands.

void acb_poly_add(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong prec)

void acb_poly_add_si(acb_poly_t C, const acb_poly_t A, slong B, slong prec)
    Sets C to the sum of A and B.

void _acb_poly_sub(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)
    Sets \(\{C, \max(lenA, lenB)\}\) to the difference of \(\{A, lenA\}\) and \(\{B, lenB\}\). Allows aliasing of the input
    and output operands.

void acb_poly_sub(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong prec)

void acb_poly_add_series(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong len, slong prec)
    Sets C to the sum of A and B, truncated to length len.

void acb_poly_sub_series(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong len, slong prec)
    Sets C to the difference of A and B, truncated to length len.

void acb_poly_neg(acb_poly_t C, const acb_poly_t A)
    Sets C to the negation of A.

void acb_poly_scalar_mul_2exp_si(acb_poly_t C, const acb_poly_t A, slong c)
    Sets C to A multiplied by \(2^c\).

void acb_poly_scalar_mul(acb_poly_t C, const acb_poly_t A, const acb_t c, slong prec)
    Sets C to A multiplied by c.

void acb_poly_scalar_div(acb_poly_t C, const acb_poly_t A, const acb_t c, slong prec)
    Sets C to A divided by c.

void _acb_poly_mullow_classical(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB,
                                slong n, slong prec)

void _acb_poly_mullow_transpose(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB,
                                slong n, slong prec)

void _acb_poly_mullow_transpose_gauss(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB,
                                      slong n, slong prec)
void _acb_poly_mullow(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong n, slong prec)

Sets \( \{C, n\} \) to the product of \( \{A, \text{len}A\} \) and \( \{B, \text{len}B\} \), truncated to length \( n \). The output is not allowed to be aliased with either of the inputs. We require \( \text{len}A \geq \text{len}B > 0, n > 0, \text{len}A + \text{len}B - 1 \geq n \).

The **classical** version uses a plain loop.

The **transpose** version evaluates the product using four real polynomial multiplications (via \_arb_poly_mullow()\).

The **transpose_gauss** version evaluates the product using three real polynomial multiplications. This is almost always faster than **transpose**, but has worse numerical stability when the coefficients vary in magnitude.

The default function _acb_poly_mullow() automatically switches between classical and transpose multiplication.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void acb_poly_mullow_classical(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong n, slong prec)

void acb_poly_mullow_transpose(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong n, slong prec)

void acb_poly_mullow_transpose_gauss(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong n, slong prec)

void acb_poly_mullow(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong n, slong prec)

Sets \( C \) to the product of \( A \) and \( B \), truncated to length \( n \). If the same variable is passed for \( A \) and \( B \), sets \( C \) to the square of \( A \) truncated to length \( n \).

void _acb_poly_mul(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)

Sets \( \{C, \text{len}A + \text{len}B - 1\} \) to the product of \( \{A, \text{len}A\} \) and \( \{B, \text{len}B\} \). The output is not allowed to be aliased with either of the inputs. We require \( \text{len}A \geq \text{len}B > 0 \). This function is implemented as a simple wrapper for _acb_poly_mullow()\).

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

void _acb_poly_div_series(acb_ptr Q, acb_ptr A, slong lenA, acb_ptr B, slong lenB, slong prec)

Sets \( Q \) to the power series quotient of \( A \) divided by \( B \), truncated to length \( n \).
void _acb_poly_rem(acb_ptr R, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)

void _acb_poly_divrem(acb_ptr Q, acb_ptr R, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)

int acb_poly_divrem(acb_poly_t Q, acb_poly_t R, const acb_poly_t A, const acb_poly_t B, slong prec)

Performs polynomial division with remainder, computing a quotient $Q$ and a remainder $R$ such that $A = BQ + R$. The implementation reverses the inputs and performs power series division.

If the leading coefficient of $B$ contains zero (or if $B$ is identically zero), returns 0 indicating failure without modifying the outputs. Otherwise returns nonzero.

void _acb_poly_div_root(acb_ptr Q, acb_t R, acb_srcptr A, slong len, const acb_t c, slong prec)

Divides $A$ by the polynomial $x - c$, computing the quotient $Q$ as well as the remainder $R = f(c)$.

### 9.11.10 Composition

void _acb_poly_taylor_shift(acb_ptr g, const acb_t c, slong n, slong prec)

void acb_poly_taylor_shift(acb_poly_t g, const acb_poly_t f, const acb_t c, slong prec)

Sets $g$ to the Taylor shift $f(x+c)$. The underscore methods act in-place on $g = f$ which has length $n$.

void _acb_poly_compose(acb_ptr res, acb_srcptr poly1, slong len1, acb_srcptr poly2, slong len2, slong prec)

void acb_poly_compose(acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, slong prec)

Sets res to the composition $h(x) = f(g(x))$ where $f$ is given by poly1 and $g$ is given by poly2. The underscore method does not support aliasing of the output with either input polynomial.

void _acb_poly_compose_series(acb_ptr res, acb_srcptr poly1, slong len1, acb_srcptr poly2, slong len2, slong n, slong prec)

void acb_poly_compose_series(acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, slong n, slong prec)

Sets res to the power series composition $h(x) = f(g(x))$ truncated to order $O(x^n)$ where $f$ is given by poly1 and $g$ is given by poly2. Wraps _gr_poly_compose_series() which chooses automatically between various algorithms.

We require that the constant term in $g(x)$ is exactly zero. The underscore method does not support aliasing of the output with either input polynomial.

void _acb_poly_revert_series(acb_ptr h, acb_srcptr f, slongflen, slong n, slong prec)

void acb_poly_revert_series(acb_poly_t h, const acb_poly_t f, slong n, slong prec)

Sets $h$ to the power series reversion of $f$, i.e. the expansion of the compositional inverse function $f^{-1}(x)$, truncated to order $O(x^n)$. Wraps _gr_poly_revert_series() which chooses automatically between various algorithms.

We require that the constant term in $f$ is exactly zero and that the linear term is nonzero. The underscore method assumes that $flen$ is at least 2, and do not support aliasing.
9.11.11 Evaluation

void _acb_poly_evaluate_horner(acb_t y, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate_horner(acb_t y, const acb_poly_t f, const acb_t x, slong prec)
void _acb_poly_evaluate_rectangular(acb_t y, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate_rectangular(acb_t y, const acb_poly_t f, const acb_t x, slong prec)
void _acb_poly_evaluate(acb_t y, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate(acb_t y, const acb_poly_t f, const acb_t x, slong prec)

Sets $y = f(x)$, evaluated respectively using Horner’s rule, rectangular splitting, and an automatic algorithm choice.

void _acb_poly_evaluate2_horner(acb_t y, acb_t z, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate2_horner(acb_t y, acb_t z, const acb_poly_t f, const acb_t x, slong prec)
void _acb_poly_evaluate2_rectangular(acb_t y, acb_t z, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate2_rectangular(acb_t y, acb_t z, const acb_poly_t f, const acb_t x, slong prec)
void _acb_poly_evaluate2(acb_t y, acb_t z, acb_srcptr f, slong len, const acb_t x, slong prec)
void acb_poly_evaluate2(acb_t y, acb_t z, const acb_poly_t f, const acb_t x, slong prec)

Sets $y = f(x)$, $z = f'(x)$, evaluated respectively using Horner’s rule, rectangular splitting, and an automatic algorithm choice.

When Horner’s rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

9.11.12 Product trees

void _acb_poly_product_roots(acb_ptr poly, acb_srcptr xs, slong n, slong prec)
void acb_poly_product_roots(acb_poly_t poly, acb_srcptr xs, slong n, slong prec)

Generates the polynomial $(x - x_0)(x - x_1)\cdots(x - x_{n-1})$.

acb_ptr *acb_poly_tree_alloc(slong len)

Returns an initialized data structured capable of representing a remainder tree (product tree) of len roots.

void _acb_poly_tree_free(acb_ptr *tree, slong len)
Deallocates a tree structure as allocated using _acb_poly_tree_alloc.

void _acb_poly_tree_build(acb_ptr *tree, acb_srcptr roots, slong len, slong prec)
Constructs a product tree from a given array of len roots. The tree structure must be pre-allocated to the specified length using _acb_poly_tree_alloc().
9.11.13 Multipoint evaluation

```c
void _acb_poly_evaluate_vec_iter(acb_ptr ys, acb_srcptr poly, slong plen, acb_srcptr xs, slong n, slong prec)
```

Evaluates the polynomial simultaneously at \( n \) given points, calling \_acb_poly_evaluate\) repeatedly.

```c
void acb_poly_evaluate_vec_iter(acb_ptr ys, const acb_poly_t poly, acb_srcptr xs, slong n, slong prec)
```

9.11.14 Interpolation

```c
void _acb_poly_interpolate_newton(acb_ptr poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

Recover the unique polynomial of length at most \( n \) that interpolates the given \( x \) and \( y \) values. This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

```c
void acb_poly_interpolate_newton(acb_poly_t poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

```c
void _acb_poly_interpolate_barycentric(acb_ptr poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

Recover the unique polynomial of length at most \( n \) that interpolates the given \( x \) and \( y \) values. This implementation uses the barycentric form of Lagrange interpolation.

```c
void acb_poly_interpolate_barycentric(acb_poly_t poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

```c
void _acb_poly_interpolation_weights(acb_ptr w, acb_ptr *tree, slong len, slong prec)
```

```c
void _acb_poly_interpolate_fast_precomp(acb_ptr poly, acb_ptr *tree, slong len, slong prec)
```

```c
void _acb_poly_interpolate_fast(acb_ptr ys, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

Evaluates the polynomial simultaneously at \( n \) given points, using fast multipoint evaluation.

```c
void acb_poly_interpolate_fast(acb_poly_t poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

Recover the unique polynomial of length at most \( n \) that interpolates the given \( x \) and \( y \) values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the \( x \) values and a vector of interpolation weights as additional inputs.
9.11.15 Differentiation

void _acb_poly_derivative(acb_ptr res, acb_srcptr poly, slong len, slong prec)
Sets \{res, len - 1\} to the derivative of \{poly, len\}. Allows aliasing of the input and output.

void acb_poly_derivative(acb_poly_t res, const acb_poly_t poly, slong prec)
Sets res to the derivative of poly.

void _acb_poly_nth_derivative(acb_ptr res, acb_srcptr poly, ulong n, slong len, slong prec)
Sets \{res, len - n\} to the nth derivative of \{poly, len\}. Does nothing if len <= n. Allows aliasing of the input and output.

void acb_poly_nth_derivative(acb_poly_t res, const acb_poly_t poly, ulong n, slong prec)
Sets res to the nth derivative of poly.

void _acb_poly_integral(acb_ptr res, acb_srcptr poly, slong len, slong prec)
Sets \{res, len\} to the integral of \{poly, len - 1\}. Allows aliasing of the input and output.

void acb_poly_integral(acb_poly_t res, const acb_poly_t poly, slong prec)
Sets res to the integral of poly.

9.11.16 Transforms

void _acb_poly_borel_transform(acb_ptr res, acb_srcptr poly, slong len, slong prec)
void acb_poly_borel_transform(acb_poly_t res, const acb_poly_t poly, slong prec)
Computes the Borel transform of the input polynomial, mapping \( \sum_k a_k x^k \) to \( \sum_k (a_k/k!)x^k \). The underscore method allows aliasing.

void _acb_poly_inv_borel_transform(acb_ptr res, acb_srcptr poly, slong len, slong prec)
void acb_poly_inv_borel_transform(acb_poly_t res, const acb_poly_t poly, slong prec)
Computes the inverse Borel transform of the input polynomial, mapping \( \sum_k a_k x^k \) to \( \sum_k a_k k!x^k \). The underscore method allows aliasing.

void _acb_poly_binomial_transform_basecase(acb_ptr b, acb_srcptr a, slong alen, slong len, slong prec)
void acb_poly_binomial_transform_basecase(acb_poly_t b, const acb_poly_t a, slong len, slong prec)
Computes the binomial transform of the input polynomial, truncating the output to length len. See arb_poly_binomial_transform() for details.

void _acb_poly_binomial_transform_convolution(acb_ptr b, acb_srcptr a, slong alen, slong len, slong prec)
void acb_poly_binomial_transform_convolution(acb_poly_t b, const acb_poly_t a, slong len, slong prec)
Computes the binomial transform of the input polynomial, truncating the output to length len. See arb_poly_binomial_transform() for details.

The underscore methods do not support aliasing, and assume that the lengths are nonzero.

void _acb_poly_graeffe_transform(acb_ptr b, acb_srcptr a, slong len, slong prec)
void \texttt{acb\_poly\_graeffe\_transform}(\texttt{acb\_poly\_t} b, \texttt{const acb\_poly\_t} a, \texttt{slong} \texttt{prec})

Computes the Graeffe transform of input polynomial, which is of length \textit{len}. See \texttt{arb\_poly\_graeffe\_transform()} for details.

The underscore method assumes that \( a \) and \( b \) are initialized, \( a \) is of length \textit{len}, and \( b \) is of length at least \textit{len}. Both methods allow aliasing.

\section*{9.11.17 Elementary functions}

void \texttt{_acb\_poly\_pow\_ui\_trunc\_binexp}(\texttt{acb\_ptr} \texttt{res}, \texttt{acb\_srcptr} \texttt{f}, \texttt{slong} \texttt{flen}, \texttt{ulong} \texttt{exp}, \texttt{slong} \texttt{len}, \texttt{ulong} \texttt{prec})

Sets \( \{\text{res}, \text{len}\} \) to \( \{f, \text{flen}\} \) raised to the power \textit{exp}, truncated to length \textit{len}. Requires that \textit{len} is no longer than the length of the power as computed without truncation (i.e. no zero-padding is performed). Does not support aliasing of the input and output, and requires that \textit{flen} and \textit{len} are positive. Uses binary exponentiation.

void \texttt{acb\_poly\_pow\_ui\_trunc\_binexp}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{const acb\_poly\_t} \texttt{poly}, \texttt{ulong} \texttt{exp}, \texttt{slong} \texttt{len}, \texttt{ulong} \texttt{prec})

Sets \texttt{res} to \texttt{poly} raised to the power \textit{exp}, truncated to length \textit{len}. Uses binary exponentiation.

void \texttt{acb\_poly\_pow\_ui}(\texttt{acb\_ptr} \texttt{res}, \texttt{acb\_srcptr} \texttt{f}, \texttt{slong} \texttt{flen}, \texttt{ulong} \texttt{exp}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to \( \{f, \text{flen}\} \) raised to the power \textit{exp}. Does not support aliasing of the input and output, and requires that \textit{flen} is positive.

void \texttt{acb\_poly\_pow\_ui}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{const acb\_poly\_t} \texttt{poly}, \texttt{ulong} \texttt{exp}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to \texttt{poly} raised to the power \textit{exp}.

void \texttt{_acb\_poly\_pow\_series}(\texttt{acb\_ptr} \texttt{h}, \texttt{acb\_srcptr} \texttt{f}, \texttt{slong} \texttt{flen}, \texttt{acb\_srcptr} \texttt{g}, \texttt{slong} \texttt{glen}, \texttt{slong} \texttt{len}, \texttt{slong} \texttt{prec})

Sets \( \{h, \text{len}\} \) to the power series \( f(x)^g = \exp(g \log f(x)) \) truncated to length \textit{len}. This function detects special cases such as \( g \) being an exact small integer or \( \pm 1/2 \), and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that \textit{flen} and \textit{glen} do not exceed \textit{len}.

void \texttt{acb\_poly\_pow\_series}(\texttt{acb\_poly\_t} \texttt{h}, \texttt{const acb\_poly\_t} \texttt{f}, \texttt{const acb\_poly\_t} \texttt{g}, \texttt{slong} \texttt{len}, \texttt{slong} \texttt{prec})

Sets \texttt{h} to the power series \( f(x)^g = \exp(g \log f(x)) \) truncated to length \textit{len}. This function detects special cases such as \( g \) being an exact small integer or \( \pm 1/2 \), and computes such powers more efficiently.

void \texttt{acb\_poly\_pow\_acb\_series}(\texttt{acb\_ptr} \texttt{h}, \texttt{acb\_srcptr} \texttt{f}, \texttt{slong} \texttt{flen}, \texttt{const acb\_t} \texttt{g}, \texttt{slong} \texttt{len}, \texttt{slong} \texttt{prec})

Sets \( \{h, \text{len}\} \) to the power series \( f(x)^g = \exp(g \log f(x)) \) truncated to length \textit{len}. This function detects special cases such as \( g \) being an exact small integer or \( \pm 1/2 \), and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that \textit{flen} does not exceed \textit{len}.

void \texttt{acb\_poly\_pow\_acb\_series}(\texttt{acb\_poly\_t} \texttt{h}, \texttt{const acb\_poly\_t} \texttt{f}, \texttt{const acb\_t} \texttt{g}, \texttt{slong} \texttt{len}, \texttt{slong} \texttt{prec})

Sets \texttt{h} to the power series \( f(x)^g = \exp(g \log f(x)) \) truncated to length \textit{len}.

void \texttt{acb\_poly\_sqrt\_series}(\texttt{acb\_ptr} \texttt{g}, \texttt{acb\_srcptr} \texttt{h}, \texttt{slong} \texttt{hlen}, \texttt{slong} \texttt{n}, \texttt{slong} \texttt{prec})

Sets \texttt{g} to the power series square root of \texttt{h}, truncated to length \texttt{n}. Uses division-free Newton iteration for the reciprocal square root, followed by a multiplication.

The underscore method does not support aliasing of the input and output arrays. It requires that \textit{hlen} and \textit{n} are greater than zero.
Flint Documentation, Release 3.1.0-dev

604 Chapter 9. Real and complex numbers

void \_acb\_poly\_rsqrt\_series(acb\_ptr g, acb\_srcptr h, slong hlen, slong n, slong prec)

void acb\_poly\_rsqrt\_series(acb\_poly\_t g, const acb\_poly\_t h, slong n, slong prec)

Sets g to the reciprocal power series square root of h, truncated to length n. Uses division-free Newton iteration.

The underscore method does not support aliasing of the input and output arrays. It requires that hlen and n are greater than zero.

void \_acb\_poly\_log\_series(acb\_ptr res, acb\_srcptr f, slong flen, slong n, slong prec)

void acb\_poly\_log\_series(acb\_poly\_t res, const acb\_poly\_t f, slong n, slong prec)

Sets res to the power series logarithm of f, truncated to length n. Uses the formula \(\log(f(x)) = \int f'(x)/f(x)dx\), adding the logarithm of the constant term in f as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that flen and n are greater than zero.

void \_acb\_poly\_log1p\_series(acb\_ptr res, acb\_srcptr f, slong flen, slong n, slong prec)

void acb\_poly\_log1p\_series(acb\_poly\_t res, const acb\_poly\_t f, slong n, slong prec)

Computes the power series log(1 + f), with better accuracy when the constant term of f is small.

void \_acb\_poly\_atan\_series(acb\_ptr res, acb\_srcptr f, slong flen, slong n, slong prec)

void acb\_poly\_atan\_series(acb\_poly\_t res, const acb\_poly\_t f, slong n, slong prec)

Sets res the power series inverse tangent of f, truncated to length n.

Uses the formula

\[\tan^{-1}(f(x)) = \int f'(x)/(1 + f(x)^2)dx,\]

adding the function of the constant term in f as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that flen and n are greater than zero.

void \_acb\_poly\_exp\_series\_basecase(acb\_ptr f, acb\_srcptr h, slong hlen, slong n, slong prec)

void acb\_poly\_exp\_series\_basecase(acb\_poly\_t f, const acb\_poly\_t h, slong n, slong prec)

void \_acb\_poly\_exp\_series(acb\_ptr f, acb\_srcptr h, slong hlen, slong n, slong prec)

void acb\_poly\_exp\_series(acb\_poly\_t f, const acb\_poly\_t h, slong n, slong prec)

Sets f to the power series exponential of h, truncated to length n.

The basecase version uses a simple recurrence for the coefficients, requiring \(O(nm)\) operations where m is the length of h.

The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is \(O(M(n))\). Redundant operations in the Newton iteration are avoided by using the scheme described in [HZZ2004].

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

void \_acb\_poly\_exp\_pi\_i\_series(acb\_ptr f, acb\_srcptr h, slong hlen, slong n, slong prec)

void acb\_poly\_exp\_pi\_i\_series(acb\_poly\_t f, const acb\_poly\_t h, slong n, slong prec)

Sets f to the power series \(\exp(\pi i h)\) truncated to length n. The underscore method supports aliasing and allows the input to be shorter than the output, but requires the lengths to be nonzero.

void \_acb\_poly\_sin\_cos\_series(acb\_ptr s, acb\_ptr c, acb\_srcptr h, slong hlen, slong n, slong prec)
void acb_poly_sin_cos_series(acb_poly_t s, acb_poly_t c, const acb_poly_t h, slong n, slong prec)
    Sets s and c to the power series sine and cosine of h, computed simultaneously. The underscore method supports aliasing and requires the lengths to be nonzero.

void _acb_poly_sin_series(acb_ptr s, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_sin_series(acb_poly_t s, const acb_poly_t h, slong n, slong prec)

void _acb_poly_cos_series(acb_ptr c, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_cos_series(acb_poly_t c, const acb_poly_t h, slong n, slong prec)
    Respectively evaluates the power series sine or cosine. These functions simply wrap _acb_poly_sin_cos_series(). The underscore methods support aliasing and require the lengths to be nonzero.

void _acb_poly_tan_series(acb_ptr g, acb_srcptr h, slong hlen, slong len, slong prec)

void acb_poly_tan_series(acb_poly_t g, const acb_poly_t h, slong n, slong prec)
    Sets g to the power series tangent of h.
    For small n takes the quotient of the sine and cosine as computed using the basecase algorithm.
    For large n, uses Newton iteration to invert the inverse tangent series.
    The complexity is $O(M(n))$.
    The underscore version does not support aliasing, and requires the lengths to be nonzero.

void _acb_poly_sin_cos_pi_series(acb_ptr s, acb_ptr c, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_sin_cos_pi_series(acb_poly_t s, acb_poly_t c, const acb_poly_t h, slong n, slong prec)

void _acb_poly_sin_pi_series(acb_ptr s, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_sin_pi_series(acb_poly_t s, const acb_poly_t h, slong n, slong prec)

void _acb_poly_cos_pi_series(acb_ptr c, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_cos_pi_series(acb_poly_t c, const acb_poly_t h, slong n, slong prec)

void _acb_poly_cot_pi_series(acb_ptr c, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_cot_pi_series(acb_poly_t c, const acb_poly_t h, slong n, slong prec)
    Compute the respective trigonometric functions of the input multiplied by $\pi$.

void _acb_poly_sinh_cosh_series_basecase(acb_ptr s, acb_ptr c, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_sinh_cosh_series_basecase(acb_poly_t s, acb_poly_t c, const acb_poly_t h, slong n, slong prec)

void _acb_poly_sinh_cosh_series_exponential(acb_ptr s, acb_ptr c, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_sinh_cosh_series_exponential(acb_poly_t s, acb_poly_t c, const acb_poly_t h, slong n, slong prec)

void _acb_poly_sinh_cosh_series(acb_ptr s, acb_ptr c, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_sinh_cosh_series(acb_poly_t s, acb_poly_t c, const acb_poly_t h, slong n, slong prec)

void _acb_poly_sinh_series(acb_ptr s, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_sinh_series(acb_poly_t s, const acb_poly_t h, slong n, slong prec)
void acb_poly_sinh_series(acb_poly_t s, const acb_poly_t h, slong n, slong prec)

void _acb_poly_cosh_series(acb_ptr c, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_cosh_series(acb_poly_t c, const acb_poly_t h, slong n, slong prec)

Sets \( s \) and \( c \) respectively to the hyperbolic sine and cosine of the power series \( h \), truncated to length \( n \).

The implementations mirror those for sine and cosine, except that the exponential version computes both functions using the exponential function instead of the hyperbolic tangent.

void _acb_poly_sinc_series(acb_ptr s, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_sinc_series(acb_poly_t s, const acb_poly_t h, slong n, slong prec)

Sets \( s \) to the sinc function of the power series \( h \), truncated to length \( n \).

9.11.18 Lambert W function

void _acb_poly_lambertw_series(acb_ptr res, acb_srcptr z, slong zlen, const fmpz_t k, int flags, slong len, slong prec)

void acb_poly_lambertw_series(acb_poly_t res, const acb_poly_t z, const fmpz_t k, int flags, slong len, slong prec)

Sets \( res \) to branch \( k \) of the Lambert W function of the power series \( z \). The argument \( flags \) is reserved for future use. The underscore method allows aliasing, but assumes that the lengths are nonzero.

9.11.19 Gamma function

void _acb_poly_gamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_gamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)

void _acb_poly_rgamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_rgamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)

void _acb_poly_lgamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_lgamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)

void _acb_poly_digamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)

void acb_poly_digamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)

Sets \( res \) to the series expansion of \( \Gamma(h(x)), 1/\Gamma(h(x)), \text{ or } \log \Gamma(h(x)), \psi(h(x)) \), truncated to length \( n \).

These functions first generate the Taylor series at the constant term of \( h \), and then call _acb_poly_compose_series(). The Taylor coefficients are generated using Stirling’s series.

The underscore methods support aliasing of the input and output arrays, and require that \( hlen \) and \( n \) are greater than zero.

void _acb_poly_rising_ui_series(acb_ptr res, acb_srcptr f, slong flen, ulong r, slong trunc, slong prec)

void acb_poly_rising_ui_series(acb_poly_t res, const acb_poly_t f, slong r, slong trunc, slong prec)

Sets \( res \) to the rising factorial \( f(1)(f + 1)(f + 2) \cdots (f + r - 1) \), truncated to length \( trunc \). The underscore method assumes that \( flen \), \( r \) and \( trunc \) are at least 1, and does not support aliasing. Uses binary splitting.
9.11.20 Power sums

```c
void _acb_poly_powsum_series_naive(acb_ptr z, const acb_t s, const acb_t a, const acb_t q, slong n, slong len, slong prec)
```

Computes

\[ z = S(s, a, n) = \sum_{k=0}^{n-1} q^k (k + a)^{n+t} \]

as a power series in \( t \) truncated to length \( \text{len} \). This function evaluates the sum naively term by term. The threaded version splits the computation over the number of threads returned by `flint_get_num_threads()`.

```c
void _acb_poly_powsum_series_naive_threaded(acb_ptr z, const acb_t s, const acb_t a, const acb_t q, slong n, slong len, slong prec)
```

```c
void _acb_poly_powsum_one_series_sieved(acb_ptr z, const acb_t s, slong n, slong len, slong prec)
```

Computes

\[ z = S(s, 1, n) \sum_{k=1}^{n} \frac{1}{k^{s+t}} \]

as a power series in \( t \) truncated to length \( \text{len} \). This function stores a table of powers that have already been calculated, computing \((ij)^r = s^r t^r\) whenever \( k = ij \) is composite. As a further optimization, it groups all even \( k \) and evaluates the sum as a polynomial in \( 2^{-(s+t)} \). This scheme requires about \( n/\log n \) powers, \( n/2 \) multiplications, and temporary storage of \( n/6 \) power series. Due to the extra power series multiplications, it is only faster than the naive algorithm when \( \text{len} \) is small.

9.11.21 Zeta function

```c
void _acb_poly_zeta_em_choose_param(mag_t bound, ulong *N, ulong *M, const acb_t s, const acb_t a, slong d, slong target, slong prec)
```

Chooses \( N \) and \( M \) for Euler-Maclaurin summation of the Hurwitz zeta function, using a default algorithm.

```c
void _acb_poly_zeta_em_bound1(mag_t bound, const acb_t s, const acb_t a, slong N, slong M, slong d, slong wp)
```

```c
void _acb_poly_zeta_em_bound(arb_ptr vec, const acb_t s, const acb_t a, ulong N, ulong M, slong d, slong wp)
```

Compute bounds for Euler-Maclaurin evaluation of the Hurwitz zeta function or its power series, using the formulas in [Joh2013].

```c
void _acb_poly_zeta_em_tail_naive(acb_ptr z, const acb_t s, const acb_t Na, acb_srcptr Nasx, slong M, slong len, slong prec)
```

```c
void _acb_poly_zeta_em_tail_bsplit(acb_ptr z, const acb_t s, const acb_t Na, acb_srcptr Nasx, slong M, slong len, slong prec)
```

Evaluates the tail in the Euler-Maclaurin sum for the Hurwitz zeta function, respectively using the naive recurrence and binary splitting.

```c
void _acb_poly_zeta_em_sum(acb_ptr z, const acb_t s, const acb_t a, int deflate, ulong N, ulong M, slong d, slong prec)
```

Evaluates the truncated Euler-Maclaurin sum of order \( N, M \) for the length-\( d \) truncated Taylor series of the Hurwitz zeta function \( \zeta(s, a) \) at \( s \), using a working precision of \( \text{prec} \) bits. With \( a = 1 \), this gives the usual Riemann zeta function.

If \( \text{deflate} \) is nonzero, \( \zeta(s, a) - 1/(s-1) \) is evaluated (which permits series expansion at \( s = 1 \)).
void _acb_poly_zeta_cpx_series(acb_ptr z, const acb_t s, const acb_t a, int deflate, slong d, slong prec)

Computes the series expansion of $\zeta(s+x,a)$ (or $\zeta(s+x,a) - 1/(s+x-1)$ if deflate is nonzero) to order $d$.

This function wraps \texttt{acb_poly_zeta_em_sum()}, automatically choosing default values for $N, M$ using \texttt{acb_poly_zeta_em_choose_param()} to target an absolute truncation error of $2^{-\text{prec}}$.

void _acb_poly_zeta_series(acb_ptr res, acb_srcptr h, slong hlen, const acb_t a, int deflate, slong len, slong prec)

void _acb_poly_zeta_series(acb_poly_t res, const acb_poly_t f, const acb_t a, int deflate, slong n, slong prec)

Sets \text{res} to the Hurwitz zeta function $\zeta(s,a)$ where $s$ a power series and $a$ is a constant, truncated to length $n$. To evaluate the usual Riemann zeta function, set $a = 1$.

If \text{deflate} is nonzero, evaluates $\zeta(s,a) + 1/(1-s)$, which is well-defined as a limit when the constant term of $s$ is 1. In particular, expanding $\zeta(s,a) + 1/(1-s)$ with $s = 1+x$ gives the Stieltjes constants

$$\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a)x^k.$$

If $a = 1$, this implementation uses the reflection formula if the midpoint of the constant term of $s$ is negative.

### 9.11.22 Other special functions

void _acb_poly_polylog_cpx_small(acb_ptr w, const acb_t s, const acb_t z, slong len, slong prec)

void _acb_poly_polylog_cpx_zeta(acb_ptr w, const acb_t s, const acb_t z, slong len, slong prec)

void _acb_poly_polylog_cpx(acb_ptr w, const acb_t s, const acb_t z, slong len, slong prec)

Sets \text{w} to the Taylor series with respect to $x$ of the polylogarithm $\text{Li}_{s+1}(z)$, where $s$ and $z$ are given complex constants. The output is computed to length \text{len} which must be positive. Aliasing between \text{w} and \text{s} or \text{z} is not permitted.

The \text{small} version uses the standard power series expansion with respect to $z$, convergent when $|z| < 1$. The \text{zeta} version evaluates the polylogarithm as a sum of two Hurwitz zeta functions. The default version automatically delegates to the \text{small} version when $z$ is close to zero, and the \text{zeta} version otherwise. For further details, see \textit{Algorithms for polylogarithms}.

void _acb_poly_polylog_series(acb_ptr w, acb_srcptr s, slong slen, const acb_t z, slong len, slong prec)

void acb_poly_polylog_series(acb_poly_t w, const acb_poly_t s, const acb_t z, slong len, slong prec)

Sets \text{w} to the polylogarithm $\text{Li}_s(z)$ where $s$ is a given power series, truncating the output to length \text{len}. The underscore method requires all lengths to be positive and supports aliasing between all inputs and outputs.

void _acb_poly_erf_series(acb_ptr res, acb_srcptr z, slong zlen, slong n, slong prec)

void acb_poly_erf_series(acb_poly_t res, const acb_poly_t z, slong n, slong prec)

Sets \text{res} to the error function of the power series $z$, truncated to length $n$. These methods are provided for backwards compatibility. See \texttt{acb_hypgeom_erf_series()}, \texttt{acb_hypgeom_erfc_series()}, \texttt{acb_hypgeom_erfi_series()}.

void _acb_poly_agm1_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)
void acb_poly_agm1_series(acb_poly_t res, const acb_poly_t z, slong n, slong prec)

Sets res to the arithmetic-geometric mean of 1 and the power series z, truncated to length n.

See the acb_elliptic.h module for power series of elliptic functions. The following wrappers are available for backwards compatibility.

void _acb_poly_elliptic_k_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)

void acb_poly_elliptic_k_series(acb_poly_t res, const acb_poly_t z, slong n, slong prec)

void _acb_poly_elliptic_p_series(acb_ptr res, acb_srcptr z, slong zlen, const acb_t tau, slong len, slong prec)

void acb_poly_elliptic_p_series(acb_poly_t res, const acb_poly_t z, const acb_t tau, slong n, slong prec)

9.11.23 Root-finding

void _acb_poly_root_bound_fujiwara(mag_t bound, acb_srcptr poly, slong len)

void acb_poly_root_bound_fujiwara(mag_t bound, acb_poly_t poly)

Sets bound to an upper bound for the magnitude of all the complex roots of poly. Uses Fujiwara’s bound

\[
2 \max \left\{ \left| \frac{a_{n-1}}{a_n} \right|, \left| \frac{a_{n-2}}{a_n} \right|^{1/2}, \ldots, \left| \frac{a_1}{a_n} \right|^{1/(n-1)}, \left| \frac{a_0}{2a_n} \right|^{1/n} \right\}
\]

where \(a_0, \ldots, a_n\) are the coefficients of poly.

void _acb_poly_root_inclusion(acb_t r, const acb_t m, acb_srcptr poly, acb_srcptr polyder, slong len, slong prec)

Given any complex number \(m\), and a nonconstant polynomial \(f\) and its derivative \(f'\), sets \(r\) to a complex interval centered on \(m\) that is guaranteed to contain at least one root of \(f\). Such an interval is obtained by taking a ball of radius \(|f(m)/f'(m)|n\) where \(n\) is the degree of \(f\). Proof: assume that the distance to the nearest root exceeds \(r = |f(m)/f'(m)|n\). Then

\[|f'(m)| = \sum \frac{1}{m - \zeta_i} \leq \sum \frac{1}{|m - \zeta_i|} < \frac{n}{r} = \frac{|f'(m)|}{|f(m)|}\]

which is a contradiction (see [Koh2010]).

slong _acb_poly_validate_roots(acb_ptr roots, acb_srcptr poly, slong len, slong prec)

Given a list of approximate roots of the input polynomial, this function sets a rigorous bounding interval for each root, and determines which roots are isolated from all the other roots. It then rearranges the list of roots so that the isolated roots are at the front of the list, and returns the count of isolated roots.

If the return value equals the degree of the polynomial, then all roots have been found. If the return value is smaller, all the remaining output intervals are guaranteed to contain roots, but it is possible that not all of the polynomial’s roots are contained among them.

void _acb_poly_refine_roots_durand_kerner(acb_ptr roots, acb_srcptr poly, slong len, slong prec)

Refines the given roots simultaneously using a single iteration of the Durand-Kerner method. The radius of each root is set to an approximation of the correction, giving a rough estimate of its error (not a rigorous bound).

slong _acb_poly_find_roots(acb_ptr roots, acb_srcptr poly, acb_srcptr initial, slong len, slong maxiter, slong prec)
Flint Documentation, Release 3.1.0-dev

```
slong acb_poly_find_roots(acb_ptr roots, const acb_poly_t poly, acb_srcptr initial, slong maxiter, slong prec)

Attempts to compute all the roots of the given nonzero polynomial poly using a working precision of prec bits. If n denotes the degree of poly, the function writes n approximate roots with rigorous error bounds to the preallocated array roots, and returns the number of roots that are isolated.

If the return value equals the degree of the polynomial, then all roots have been found. If the return value is smaller, all the output intervals are guaranteed to contain roots, but it is possible that not all of the polynomial’s roots are contained among them.

The roots are computed numerically by performing several steps with the Durand-Kerner method and terminating if the estimated accuracy of the roots approaches the working precision or if the number of steps exceeds maxiter, which can be set to zero in order to use a default value. Finally, the approximate roots are validated rigorously.

Initial values for the iteration can be provided as the array initial. If initial is set to NULL, default values (0.4 + 0.9i)k are used.

The polynomial is assumed to be squarefree. If there are repeated roots, the iteration is likely to find them (with low numerical accuracy), but the error bounds will not converge as the precision increases.

int _acb_poly_validate_real_roots(acb_srcptr roots, acb_srcptr poly, slong len, slong prec)
```

```
int acb_poly_validate_real_roots(acb_srcptr roots, const acb_poly_t poly, slong prec)

Given a strictly real polynomial poly (of length len) and isolating intervals for all its complex roots, determines if all the real roots are separated from the non-real roots. If this function returns nonzero, every root enclosure that touches the real axis (as tested by applying arb_contains_zero() to the imaginary part) corresponds to a real root (its imaginary part can be set to zero), and every other root enclosure corresponds to a non-real root (with known sign for the imaginary part).

If this function returns zero, then the signs of the imaginary parts are not known for certain, based on the accuracy of the inputs and the working precision prec.
```

### 9.12 arb_fmpz_poly.h – extra methods for integer polynomials

This module provides methods for FLINT polynomials with integer and rational coefficients (fmpz_poly_t) and (fmpq_poly_t) requiring use of Arb real or complex numbers.

Some methods output real or complex numbers while others use real and complex numbers internally to produce an exact result. This module also contains some useful helper functions not specifically related to real and complex numbers.

Note that methods that combine Arb polynomials and FLINT polynomials are found in the respective Arb polynomial modules, such as arb_poly_set_fmpz_poly() and arb_poly_get_unique_fmpz_poly().

#### 9.12.1 Evaluation

```
void _arb_fmpz_poly_evaluate_arb_horner(arb_t res, const fmpz *poly, slong len, const arb_t x, slong prec)
```

```
void arb_fmpz_poly_evaluate_arb_horner(arb_t res, const fmpz_poly_t poly, const arb_t x, slong prec)
```

```
void _arb_fmpz_poly_evaluate_arb_rectangular(arb_t res, const fmpz *poly, slong len, const arb_t x, slong prec)
```

610 Chapter 9. Real and complex numbers
void \texttt{arb\_fmpz\_poly\_evaluate\_arb\_rectangular}(\texttt{arb\_t res}, \texttt{const fmpz\_poly\_t poly}, \texttt{const arb\_t x}, slong \texttt{prec})

void \_\texttt{arb\_fmpz\_poly\_evaluate\_arb}(\texttt{arb\_t res}, \texttt{const fmpz\* poly}, slong \texttt{len}, \texttt{const arb\_t x}, slong \texttt{prec})

void \texttt{arb\_fmpz\_poly\_evaluate\_arb}(\texttt{arb\_t res}, \texttt{const fmpz\_poly\_t poly}, \texttt{const arb\_t x}, slong \texttt{prec})

void \_\texttt{arb\_fmpz\_poly\_evaluate\_acb\_horner}(\texttt{acb\_t res}, \texttt{const fmpz\* poly}, slong \texttt{len}, \texttt{const acb\_t x}, slong \texttt{prec})

void \texttt{arb\_fmpz\_poly\_evaluate\_acb\_horner}(\texttt{acb\_t res}, \texttt{const fmpz\_poly\_t poly}, \texttt{const acb\_t x}, slong \texttt{prec})

void \_\texttt{arb\_fmpz\_poly\_evaluate\_acb\_rectangular}(\texttt{acb\_t res}, \texttt{const fmpz\* poly}, slong \texttt{len}, \texttt{const acb\_t x}, slong \texttt{prec})

void \texttt{arb\_fmpz\_poly\_evaluate\_acb\_rectangular}(\texttt{acb\_t res}, \texttt{const fmpz\_poly\_t poly}, \texttt{const acb\_t x}, slong \texttt{prec})

void \_\texttt{arb\_fmpz\_poly\_evaluate\_acb}(\texttt{acb\_t res}, \texttt{const fmpz\* poly}, slong \texttt{len}, \texttt{const acb\_t x}, slong \texttt{prec})

void \texttt{arb\_fmpz\_poly\_evaluate\_acb}(\texttt{acb\_t res}, \texttt{const fmpz\_poly\_t poly}, \texttt{const acb\_t x}, slong \texttt{prec})

Evaluates \texttt{poly} (given by a polynomial object or an array with \texttt{len} coefficients) at the given real or complex number, respectively using Horner’s rule, rectangular splitting, or a default algorithm choice.

### 9.12.2 Utility methods

\texttt{ulong arb\_fmpz\_poly\_deflation(const fmpz\_poly\_t poly)}

Finds the maximal exponent by which \texttt{poly} can be deflated.

void \texttt{arb\_fmpz\_poly\_deflate(fmpz\_poly\_t res, const fmpz\_poly\_t poly, ulong deflation)}

Sets \texttt{res} to a copy of \texttt{poly} deflated by the exponent \texttt{deflation}.

### 9.12.3 Polynomial roots

void \texttt{arb\_fmpz\_poly\_complex\_roots(acb\_ptr roots, const fmpz\_poly\_t poly, int flags, slong prec)}

Writes to \texttt{roots} all the real and complex roots of the polynomial \texttt{poly}, computed to at least \texttt{prec} accurate bits. The root enclosures are guaranteed to be disjoint, so that all roots are isolated.

The real roots are written first in ascending order (with the imaginary parts set exactly to zero). The following nonreal roots are written in arbitrary order, but with conjugate pairs grouped together (the root in the upper plane leading the root in the lower plane).

The input polynomial must be squarefree. For a general polynomial, compute the squarefree part \(f/gcd(f,f')\) or do a full squarefree factorization to obtain the multiplicities of the roots:

```c
fmpz\_poly\_factor\_t fac;
fmpz\_poly\_factor\_init(fac);
fmpz\_poly\_factor\_squarefree(fac, poly);

for (i = 0; i < fac->num; i++)
{
    deg = fmpz\_poly\_degree(fac->p + i);
    flint\_printf("\%wd roots of multiplicity \%wd\n", deg, fac->exp[i]);
    roots = _acb\_vec\_init(deg);
    arb\_fmpz\_poly\_complex\_roots(roots, fac->p + i, 0, prec);
    _acb\_vec\_clear(roots, deg);
}
```

(continues on next page)
All roots are refined to a relative accuracy of at least \( \text{prec} \) bits. The output values will generally have higher actual precision, depending on the precision needed for isolation and the precision used internally by the algorithm.

This implementation should be adequate for general use, but it is not currently competitive with state-of-the-art isolation methods for finding real roots alone.

The following flags are supported:

- \textit{ARB_FMPZ\_POLY\_ROOTS\_VERBOSE}

### 9.12.4 Special polynomials

Note: see also the methods available in FLINT (e.g. for cyclotomic polynomials).

```c
void arb_fmpz_poly_gauss_period_minpoly(fmpz_poly_t res, ulong q, ulong n)
```

Sets \( \text{res} \) to the minimal polynomial of the Gaussian periods \( \sum_{a \in H} \zeta^a \) where \( \zeta = \exp(2\pi i / q) \) and \( H \) are the cosets of the subgroups of order \( d = (q - 1)/n \) of \((\mathbb{Z}/q\mathbb{Z})^\times\). The resulting polynomial has degree \( n \). When \( d = 1 \), the result is the cyclotomic polynomial \( \Phi_q \).

The implementation assumes that \( q \) is prime, and that \( n \) is a divisor of \( q - 1 \) such that \( n \) is coprime with \( d \). If any condition is not met, \( \text{res} \) is set to the zero polynomial.

This method provides a fast (in practice) way to construct finite field extensions of prescribed degree. If \( q \) satisfies the conditions stated above and \( (q - 1)/f \) additionally is coprime with \( n \), where \( f \) is the multiplicative order of \( p \mod q \), then the Gaussian period minimal polynomial is irreducible over \( \text{GF}(p) \) [CP2005].

### 9.13 acb\_dft.h – Discrete Fourier transform

Warning: the interfaces in this module are experimental and may change without notice.

All functions support aliasing.

Let \( G \) be a finite abelian group, and \( \chi \) a character of \( G \). For any map \( f : G \to \mathbb{C} \), the discrete fourier transform \( \hat{f} : G \to \mathbb{C} \) is defined by

\[
\hat{f}(\chi) = \sum_{x \in G} \overline{\chi(x)} f(x)
\]

Note that by the inversion formula

\[
\hat{\hat{f}}(\chi) = \#G \times f(\chi^{-1})
\]

it is straightforward to recover \( f \) from its DFT \( \hat{f} \).
9.13.1 Main DFT functions

If $G = \mathbb{Z}/n\mathbb{Z}$, we compute the DFT according to the usual convention

$$w_x = \sum_{y \mod n} v_y e^{-2\pi i x y / n}$$

```c
void acb_dft(acb_ptr w, acb_srcptr v, slong n, slong prec)
    Set $w$ to the DFT of $v$ of length $n$, using an automatic choice of algorithm.

void acb_dft_inverse(acb_ptr w, acb_srcptr v, slong n, slong prec)
    Compute the inverse DFT of $v$ into $w$.
```

If several computations are to be done on the same group, the FFT scheme should be reused.

```c
type acb_dft_pre_struct

Stores a fast DFT scheme on $\mathbb{Z}/n\mathbb{Z}$ as a recursive decomposition into simpler DFT with some tables of roots of unity.

An `acb_dft_pre_t` is defined as an array of `acb_dft_pre_struct` of length 1, permitting it to be passed by reference.

void acb_dft_precomp_init(acb_dft_pre_t pre, slong len, slong prec)
    Initializes the fast DFT scheme of length $len$, using an automatic choice of algorithms depending on the factorization of $len$.

    The length $len$ is stored as `pre->n`.

void acb_dft_precomp_clear(acb_dft_pre_t pre)
    Clears `pre`.

void acb_dft_precomp(acb_ptr w, acb_srcptr v, const acb_dft_pre_t pre, slong prec)
    Computes the DFT of the sequence $v$ into $w$ by applying the precomputed scheme `pre`. Both $v$ and $w$ must have length `len`.

void acb_dft_inverse_precomp(acb_ptr w, acb_srcptr v, const acb_dft_pre_t pre, slong prec)
    Compute the inverse DFT of $v$ into $w$.
```

9.13.2 DFT on products

A finite abelian group is isomorphic to a product of cyclic components

$$G = \bigoplus_{i=1}^{r} \mathbb{Z}/n_i\mathbb{Z}$$

Characters are product of component characters and the DFT reads

$$\hat{f}(x_1, \ldots, x_r) = \sum_{y_1, \ldots, y_r} f(y_1, \ldots, y_r)e^{-2\pi i \sum \frac{x_i y_i}{n_i}}$$

We assume that $f$ is given by a vector of length $\prod n_i$ corresponding to a lexicographic ordering of the values $y_1, \ldots, y_r$, and the computation returns the same indexing for values of $\hat{f}$.

```c
void acb_dirichlet_dft_prod(acb_ptr w, acb_srcptr v, slong *cyc, slong num, slong prec)
    Computes the DFT on the group product of `num` cyclic components of sizes `cyc`. Assume the entries of $v$ are indexed according to lexicographic ordering of the cyclic components.

type acb_dft_prod_struct
```
type `acb_dft_prod_t`
Stores a fast DFT scheme on a product of cyclic groups.

An `acb_dft_prod_t` is defined as an array of `acb_dft_prod_struct` of length 1, permitting it to be passed by reference.

```c
void acb_dft_prod_init(acb_dft_prod_t t, slong *cyc, slong num, slong prec)
```
Stores in `t` a DFT scheme for the product of `num` cyclic components whose sizes are given in the array `cyc`.

```c
void acb_dft_prod_clear(acb_dft_prod_t t)
```
Clears `t`.

```c
void acb_dirichlet_dft_prod_precomp(acb_ptr w, acb_srcptr v, const acb_dft_prod_t prod, slong prec)
```
Sets `w` to the DFT of `v`. Assume the entries are lexicographically ordered according to the product of cyclic groups initialized in `t`.

### 9.13.3 Convolution

For functions \( f \) and \( g \) on \( G \) we consider the convolution

\[
(f \ast g)(x) = \sum_{y \in G} f(x - y)g(y)
\]

```c
void acb_dft_convol_naive(acb_ptr w, acb_srcptr f, acb_srcptr g, slong len, slong prec)
void acb_dft_convol_rad2(acb_ptr w, acb_srcptr f, acb_srcptr g, slong len, slong prec)
void acb_dft_convol(acb_ptr w, acb_srcptr f, acb_srcptr g, slong len, slong prec)
```
Sets `w` to the convolution of `f` and `g` of length `len`.

The `naive` version simply uses the definition.

The `rad2` version embeds the sequence into a power of 2 length and uses the formula

\[
\hat{f} \ast \hat{g}(\chi) = \hat{f}(\chi)\hat{g}(\chi)
\]

to compute it using three radix 2 FFT.

The default version uses radix 2 FFT unless `len` is a product of small primes where a non padded FFT is faster.

### 9.13.4 FFT algorithms

Fast Fourier transform techniques allow to compute efficiently all values \( \hat{f}(\chi) \) by reusing common computations.

Specifically, if \( H \triangleleft G \) is a subgroup of size \( M \) and index \( [G : H] = m \), then writing \( f_x(h) = f(xh) \) the translate of \( f \) by representatives \( x \) of \( G/H \), one has a decomposition

\[
\hat{f}(\chi) = \sum_{x \in G/H} \chi(x)\hat{f}(\chi_H)
\]

so that the DFT on \( G \) can be computed using \( m \) DFT on \( H \) (of appropriate translates of \( f \)), then \( M \) DFT on \( G/H \), one for each restriction \( \chi_H \).

This decomposition can be done recursively.
Naive algorithm

```c
void acb_dft_naive(acb_ptr w, acb_srcptr v, slong n, slong prec)
```
Computes the DFT of \(v\) into \(w\), where \(v\) and \(w\) have size \(n\), using the naive \(O(n^2)\) algorithm.

```c
type acb_dft_naive_struct

type acb_dft_naive_t
```

```c
void acb_dft_naive_init(acb_dft_naive_t t, slong len, slong prec)
```
Stores a table of roots of unity in \(t\). The length \(len\) is stored as \(t->n\).

```c
void acb_dft_naive_clear(acb_dft_naive_t t)
```

```c
void acb_dft_naive_precomp(acb_ptr w, acb_srcptr v, const acb_dft_naive_t t, slong prec)
```
Sets \(w\) to the DFT of \(v\) of size \(t->n\), using the naive algorithm data \(t\).

CRT decomposition

```c
void acb_dft_crt(acb_ptr w, acb_srcptr v, slong n, slong prec)
```
Computes the DFT of \(v\) into \(w\), where \(v\) and \(w\) have size \(len\), using CRT to express \(\mathbb{Z}/n\mathbb{Z}\) as a product of cyclic groups.

```c
type acb_dft_crt_struct

type acb_dft_crt_t
```

```c
void acb_dft_crt_init(acb_dft_crt_t t, slong len, slong prec)
```

```c
void acb_dft_crt_clear(acb_dft_crt_t t)
```
Initialize a CRT decomposition of \(\mathbb{Z}/n\mathbb{Z}\) as a direct product of cyclic groups. The length \(len\) is stored as \(t->n\).

```c
void acb_dft_crt_precomp(acb_ptr w, acb_srcptr v, const acb_dft_crt_t t, slong prec)
```
Sets \(w\) to the DFT of \(v\) of size \(t->n\), using the CRT decomposition scheme \(t\).

Cooley-Tukey decomposition

```c
void acb_dft_cyc(acb_ptr w, acb_srcptr v, slong n, slong prec)
```
Computes the DFT of \(v\) into \(w\), where \(v\) and \(w\) have size \(n\), using each prime factor of \(m\) of \(n\) to decompose with the subgroup \(H = m\mathbb{Z}/n\mathbb{Z}\).

```c
type acb_dft_cyc_struct

type acb_dft_cyc_t
```

```c
void acb_dft_cyc_init(acb_dft_cyc_t t, slong len, slong prec)
```

```c
void acb_dft_cyc_clear(acb_dft_cyc_t t)
```
Initialize a decomposition of \(\mathbb{Z}/n\mathbb{Z}\) into cyclic subgroups. The length \(len\) is stored as \(t->n\).

```c
void acb_dft_cyc_precomp(acb_ptr w, acb_srcptr v, const acb_dft_cyc_t t, slong prec)
```
Sets \(w\) to the DFT of \(v\) of size \(t->n\), using the cyclic decomposition scheme \(t\).
Radix 2 decomposition

void **acb_dft_rad2**\( (acb\_ptr \ w, \ acb\_srcptr \ v, \ int \ e, \ slong \ prec) \)

Computes the DFT of \( v \) into \( w \), where \( v \) and \( w \) have size \( 2^e \), using a radix 2 FFT.

void **acb_dft_inverse_rad2**\( (acb\_ptr \ w, \ acb\_srcptr \ v, \ int \ e, \ slong \ prec) \)

Computes the inverse DFT of \( v \) into \( w \), where \( v \) and \( w \) have size \( 2^e \), using a radix 2 FFT.

type **acb_dft_rad2\_struct**

type **acb_dft_rad2\_t**

void **acb_dft_rad2\_init**\( (acb\_dft\_rad2\_t \ t, \ int \ e, \ slong \ prec) \)

Initialize and clear a radix 2 FFT of size \( 2^e \), stored as \( t->n \).

void **acb_dft_rad2\_precomp**\( (acb\_ptr \ w, \ acb\_srcptr \ v, \ const \ acb\_dft\_rad2\_t \ t, \ slong \ prec) \)

Sets \( w \) to the DFT of \( v \) of size \( t->n \), using the precomputed radix 2 scheme \( t \).

Bluestein transform

void **acb_dft_bluestein**\( (acb\_ptr \ w, \ acb\_srcptr \ v, \ slong \ n, \ slong \ prec) \)

Computes the DFT of \( v \) into \( w \), where \( v \) and \( w \) have size \( n \), by conversion to a radix 2 one using Bluestein’s convolution trick.

type **acb_dft_bluestein\_struct**

type **acb_dft_bluestein\_t**

Stores a Bluestein scheme for some length \( n \) : that is a \( acb\_dft\_rad2\_t \) of size \( 2^e \geq 2n - 1 \) and a size \( n \) array of convolution factors.

void **acb_dft_bluestein\_init**\( (acb\_dft\_bluestein\_t \ t, \ slong \ len, \ slong \ prec) \)

Initialize and clear a Bluestein scheme to compute DFT of size \( len \).

void **acb_dft_bluestein\_precomp**\( (acb\_ptr \ w, \ acb\_srcptr \ v, \ const \ acb\_dft\_bluestein\_t \ t, \ slong \ prec) \)

Sets \( w \) to the DFT of \( v \) of size \( t->n \), using the precomputed Bluestein scheme \( t \).

9.14 arb_mat.h – matrices over the real numbers

An **arb_mat\_t** represents a dense matrix over the real numbers, implemented as an array of entries of type **arb\_struct**. The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

**Note:** Methods prefixed with **arb_mat\_approx** treat all input entries as floating-point numbers (ignoring the radii of the balls) and compute floating-point output (balls with zero radius) representing approximate solutions \textit{without error bounds}. All other methods compute rigorous error bounds. The **approx** methods are typically useful for computing initial values or preconditioners for rigorous solvers. Some users may also find **approx** methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.
9.14.1 Types, macros and constants

**type arb_mat_struct**

**type arb_mat_t**

Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An `arb_mat_t` is defined as an array of length one of type `arb_mat_struct`, permitting an `arb_mat_t` to be passed by reference.

**arb_mat_entry(mat, i, j)**

Macro giving a pointer to the entry at row `i` and column `j`.

**arb_mat_nrows(mat)**

Returns the number of rows of the matrix.

**arb_mat_ncols(mat)**

Returns the number of columns of the matrix.

9.14.2 Memory management

**void arb_mat_init(arb_mat_t mat, slong r, slong c)**

Initializes the matrix, setting it to the zero matrix with `r` rows and `c` columns.

**void arb_mat_clear(arb_mat_t mat)**

Clears the matrix, deallocating all entries.

**slong arb_mat_allocated_bytes(const arb_mat_t x)**

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(arb_mat_struct)` to get the size of the object as a whole.

**void arb_mat_window_init(arb_mat_t window, const arb_mat_t mat, slong r1, slong c1, slong r2, slong c2)**

Initializes `window` to a window matrix into the submatrix of `mat` starting at the corner at row `r1` and column `c1` (inclusive) and ending at row `r2` and column `c2` (exclusive).

**void arb_mat_window_clear(arb_mat_t window)**

Frees the window matrix.

9.14.3 Conversions

**void arb_mat_set(arb_mat_t dest, const arb_mat_t src)**

**void arb_mat_set_fmpz_mat(arb_mat_t dest, const fmpz_mat_t src)**

**void arb_mat_set_round_fmpz_mat(arb_mat_t dest, const fmpz_mat_t src, slong prec)**

**void arb_mat_set_fmpq_mat(arb_mat_t dest, const fmpq_mat_t src, slong prec)**

Sets `dest` to `src`. The operands must have identical dimensions.
9.14.4 Random generation

void `arb_mat_randtest` (arb_mat_t mat, flint_rand_t state, slong prec, slong mag_bits)
    Sets mat to a random matrix with up to prec bits of precision and with exponents of width up to mag_bits.

void `arb_mat_randtest_cho` (arb_mat_t mat, flint_rand_t state, slong prec, slong mag_bits)
    Sets mat to a random lower-triangular matrix with precise entries and positive diagonal entries.
    Requires that mat is square.

void `arb_mat_randtest_spd` (arb_mat_t mat, flint_rand_t state, slong prec, slong mag_bits)
    Sets mat to a random symmetric positive definite matrix, obtained as a product $L L^T$ where $L$ is a random Cholesky matrix. Requires that mat is square.

9.14.5 Input and output

void `arb_mat_printd` (const arb_mat_t mat, slong digits)
    Prints each entry in the matrix with the specified number of decimal digits.

void `arb_mat_fprintd` (FILE *file, const arb_mat_t mat, slong digits)
    Prints each entry in the matrix with the specified number of decimal digits to the stream file.

9.14.6 Comparisons

Predicate methods return 1 if the property certainly holds and 0 otherwise.

int `arb_mat_equal` (const arb_mat_t mat1, const arb_mat_t mat2)
    Returns whether the matrices have the same dimensions and identical intervals as entries.

int `arb_mat_overlaps` (const arb_mat_t mat1, const arb_mat_t mat2)
    Returns whether the matrices have the same dimensions and each entry in mat1 overlaps with the corresponding entry in mat2.

int `arb_mat_contains` (const arb_mat_t mat1, const arb_mat_t mat2)

int `arb_mat_contains_fmpz_mat` (const arb_mat_t mat1, const fmpz_mat_t mat2)

int `arb_mat_contains_fmpq_mat` (const arb_mat_t mat1, const fmpq_mat_t mat2)
    Returns whether the matrices have the same dimensions and each entry in mat2 is contained in the corresponding entry in mat1.

int `arb_mat_eq` (const arb_mat_t mat1, const arb_mat_t mat2)
    Returns whether mat1 and mat2 certainly represent the same matrix.

int `arb_mat_ne` (const arb_mat_t mat1, const arb_mat_t mat2)
    Returns whether mat1 and mat2 certainly do not represent the same matrix.

int `arb_mat_is_empty` (const arb_mat_t mat)
    Returns whether the number of rows or the number of columns in mat is zero.

int `arb_mat_is_square` (const arb_mat_t mat)
    Returns whether the number of rows is equal to the number of columns in mat.

int `arb_mat_is_exact` (const arb_mat_t mat)
    Returns whether all entries in mat have zero radius.

int `arb_mat_is_zero` (const arb_mat_t mat)
    Returns whether all entries in mat are exactly zero.
int `arb_mat_is_finite(const arb_mat_t mat)`

Returns whether all entries in `mat` are finite.

int `arb_mat_is_triu(const arb_mat_t mat)`

Returns whether `mat` is upper triangular; that is, all entries below the main diagonal are exactly zero.

int `arb_mat_is_tril(const arb_mat_t mat)`

Returns whether `mat` is lower triangular; that is, all entries above the main diagonal are exactly zero.

int `arb_mat_is_diag(const arb_mat_t mat)`

Returns whether `mat` is a diagonal matrix; that is, all entries off the main diagonal are exactly zero.

### 9.14.7 Special matrices

void `arb_mat_zero(arb_mat_t mat)`

Sets all entries in `mat` to zero.

void `arb_mat_one(arb_mat_t mat)`

Sets the entries on the main diagonal to ones, and all other entries to zero.

void `arb_mat_ones(arb_mat_t mat)`

Sets all entries in the matrix to ones.

void `arb_mat_indeterminate(arb_mat_t mat)`

Sets all entries in the matrix to indeterminate (NaN).

void `arb_mat_hilbert(arb_mat_t mat, slong prec)`

Sets `mat` to the Hilbert matrix, which has entries $A_{j,k} = \frac{1}{j + k + 1}$.

void `arb_mat_pascal(arb_mat_t mat, int triangular, slong prec)`

Sets `mat` to a Pascal matrix, whose entries are binomial coefficients. If `triangular` is 0, constructs a full symmetric matrix with the rows of Pascal’s triangle as successive antidiagonals. If `triangular` is 1, constructs the upper triangular matrix with the rows of Pascal’s triangle as columns, and if `triangular` is -1, constructs the lower triangular matrix with the rows of Pascal’s triangle as rows.

The entries are computed using recurrence relations. When the dimensions get large, some precision loss is possible; in that case, the user may wish to create the matrix at slightly higher precision and then round it to the final precision.

void `arb_mat_stirling(arb_mat_t mat, int kind, slong prec)`

Sets `mat` to a Stirling matrix, whose entries are Stirling numbers. If `kind` is 0, the entries are set to the unsigned Stirling numbers of the first kind. If `kind` is 1, the entries are set to the signed Stirling numbers of the first kind. If `kind` is 2, the entries are set to the Stirling numbers of the second kind.

The entries are computed using recurrence relations. When the dimensions get large, some precision loss is possible; in that case, the user may wish to create the matrix at slightly higher precision and then round it to the final precision.

void `arb_mat_dct(arb_mat_t mat, int type, slong prec)`

Sets `mat` to the DCT (discrete cosine transform) matrix of order $n$ where $n$ is the smallest dimension of `mat` (if `mat` is not square, the matrix is extended periodically along the larger dimension). There are many different conventions for defining DCT matrices; here, we use the normalized “DCT-II” transform matrix

$$A_{j,k} = \sqrt{\frac{2}{n}} \cos \left( \frac{\pi j}{n} \left( k + \frac{1}{2} \right) \right)$$
which satisfies $A^{-1} = A^T$. The type parameter is currently ignored and should be set to 0. In the future, it might be used to select a different convention.

### 9.14.8 Transpose

```c
void arb_mat_transpose(arb_mat_t dest, const arb_mat_t src)
```
Sets `dest` to the exact transpose `src`. The operands must have compatible dimensions. Aliasing is allowed.

### 9.14.9 Norms

```c
void arb_mat_bound_inf_norm(mag_t b, const arb_mat_t A)
void arb_mat_frobenius_norm(arb_t res, const arb_mat_t A, slong prec)
void arb_mat_bound_frobenius_norm(mag_t res, const arb_mat_t A)
```
Sets `b` to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of `A`. Sets `res` to the Frobenius norm (i.e. the square root of the sum of squares of entries) of `A`. Sets `res` to an upper bound for the Frobenius norm of `A`.

### 9.14.10 Arithmetic

```c
void arb_mat_neg(arb_mat_t dest, const arb_mat_t src)
void arb_mat_add(arb_mat_t res, const arb_mat_t mat1, const arb_mat_t mat2, slong prec)
void arb_mat_sub(arb_mat_t res, const arb_mat_t mat1, const arb_mat_t mat2, slong prec)
void arb_mat_mul_classical(arb_mat_t C, const arb_mat_t A, const arb_mat_t B, slong prec)
void arb_mat_mul_threaded(arb_mat_t C, const arb_mat_t A, const arb_mat_t B, slong prec)
void arb_mat_mul_block(arb_mat_t C, const arb_mat_t A, const arb_mat_t B, slong prec)
void arb_mat_mul(arb_mat_t res, const arb_mat_t mat1, const arb_mat_t mat2, slong prec)
void arb_mat_sqr_classical(arb_mat_t B, const arb_mat_t A, slong prec)
void arb_mat_sqr(arb_mat_t res, const arb_mat_t mat, slong prec)
```
Sets `dest` to the exact negation of `src`. The operands must have the same dimensions. Sets `res` to the sum of `mat1` and `mat2`. The operands must have the same dimensions. Sets `res` to the difference of `mat1` and `mat2`. The operands must have the same dimensions. Set `C` to the entrywise product of `A` and `B`. The operands must have the same dimensions. Sets `res` to the matrix square of `mat`. The operands must both be square with the same dimensions.

The classical version performs matrix multiplication in the trivial way.

The block version decomposes the input matrices into one or several blocks of uniformly scaled matrices and multiplies large blocks via `fmpz_mat_mul`. It also invokes 
```
_arb_mat_addmul_rad_mag_fast()
```
for the radius matrix multiplications.

The threaded version performs classical multiplication but splits the computation over the number of threads returned by `flint_get_num_threads()`.

The default version chooses an algorithm automatically.

Sets `C` to the entrywise product of `A` and `B`. The operands must have the same dimensions.

Sets `res` to the matrix square of `mat`. The operands must both be square with the same dimensions.
void arb_mat_pow_ui(arb_mat_t res, const arb_mat_t mat, ulong exp, slong prec)

Sets res to mat raised to the power exp. Requires that mat is a square matrix.

void _arb_mat_addmul_rad_mag_fast(arb_mat_t C, mag_srcptr A, mag_srcptr B, slong ar, slong ac, slong bc)

Helper function for matrix multiplication. Adds to the radii of C the matrix product of the matrices represented by A and B, where A is a linear array of coefficients in row-major order and B is a linear array of coefficients in column-major order. This function assumes that all exponents are small and is unsafe for general use.

void arb_mat_approx_mul(arb_mat_t res, const arb_mat_t mat1, const arb_mat_t mat2, slong prec)

Approximate matrix multiplication. The input radii are ignored and the output matrix is set to an approximate floating-point result. The radii in the output matrix will not necessarily be zeroed.

9.14.11 Scalar arithmetic

void arb_mat_scalar_mul_2exp_si(arb_mat_t B, const arb_mat_t A, slong c)

Sets B to A multiplied by $2^c$.

void arb_mat_scalar_addmul_si(arb_mat_t B, const arb_mat_t A, slong c, slong prec)

void arb_mat_scalar_addmul_fmpz(arb_mat_t B, const arb_mat_t A, const fmpz_t c, slong prec)

void arb_mat_scalar_addmul_arb(arb_mat_t B, const arb_mat_t A, const arb_t c, slong prec)

Sets B to $B + A \times c$.

void arb_mat_scalar_mul_si(arb_mat_t B, const arb_mat_t A, slong c, slong prec)

void arb_mat_scalar_mul_fmpz(arb_mat_t B, const arb_mat_t A, const fmpz_t c, slong prec)

void arb_mat_scalar_mul_arb(arb_mat_t B, const arb_mat_t A, const arb_t c, slong prec)

Sets B to $A \times c$.

void arb_mat_scalar_div_si(arb_mat_t B, const arb_mat_t A, slong c, slong prec)

void arb_mat_scalar_div_fmpz(arb_mat_t B, const arb_mat_t A, const fmpz_t c, slong prec)

void arb_mat_scalar_div_arb(arb_mat_t B, const arb_mat_t A, const arb_t c, slong prec)

Sets B to $A/c$.

9.14.12 Vector arithmetic

void _arb_mat_vector_mul_row(arb_ptr res, arb_srcptr v, const arb_mat_t A, slong prec)

void _arb_mat_vector_mul_col(arb_ptr res, const arb_mat_t A, arb_srcptr v, slong prec)

void arb_mat_vector_mul_row(arb_ptr res, arb_srcptr v, const arb_mat_t A, slong prec)

void arb_mat_vector_mul_col(arb_ptr res, const arb_mat_t A, arb_srcptr v, slong prec)

Sets res to the product $vA$, (resp. $Av$), where res and v are seen as row (resp. column) vectors. The lengths of the vectors must match the dimensions of A.

The underscore methods do not allow aliasing between res and v.
9.14.13 Gaussian elimination and solving

```c
int arb_mat_lu_classical(slong *perm, arb_mat_t LU, const arb_mat_t A, slong prec)
int arb_mat_lu_recursive(slong *perm, arb_mat_t LU, const arb_mat_t A, slong prec)
intarb_mat_lu(slong *perm, arb_mat_t LU, const arb_mat_t A, slong prec)
```

Given an \( n \times n \) matrix \( A \), computes an LU decomposition \( PLU = A \) using Gaussian elimination with partial pivoting. The input and output matrices can be the same, performing the decomposition in-place.

Entry \( i \) in the permutation vector \( \text{perm} \) is set to the row index in the input matrix corresponding to row \( i \) in the output matrix.

The algorithm succeeds and returns nonzero if it can find \( n \) invertible (i.e., not containing zero) pivot entries. This guarantees that the matrix is invertible.

The algorithm fails and returns zero, leaving the entries in \( P \) and \( LU \) undefined, if it cannot find \( n \) invertible pivot elements. In this case, either the matrix is singular, the input matrix was computed to insufficient precision, or the LU decomposition was attempted at insufficient precision.

The *classical* version uses Gaussian elimination directly while the *recursive* version performs the computation in a block recursive way to benefit from fast matrix multiplication. The default version chooses an algorithm automatically.

```c
void arb_mat_solve_tril_classical(arb_mat_t X, const arb_mat_t L, const arb_mat_t B, int unit, slong prec)
void arb_mat_solve_tril_recursive(arb_mat_t X, const arb_mat_t L, const arb_mat_t B, int unit, slong prec)
void arb_mat_solve_tril(arb_mat_t X, const arb_mat_t L, const arb_mat_t B, int unit, slong prec)
void arb_mat_solve_triu_classical(arb_mat_t X, const arb_mat_t U, const arb_mat_t B, int unit, slong prec)
void arb_mat_solve_triu_recursive(arb_mat_t X, const arb_mat_t U, const arb_mat_t B, int unit, slong prec)
void arb_mat_solve_triu(arb_mat_t X, const arb_mat_t U, const arb_mat_t B, int unit, slong prec)
```

Solves the lower triangular system \( LX = B \) or the upper triangular system \( UX = B \), respectively. If \( \text{unit} \) is set, the main diagonal of \( L \) or \( U \) is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The *classical* versions perform the computations iteratively while the *recursive* versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

```c
void arb_mat_solve_lu_precomp(arb_mat_t X, const slong *perm, const arb_mat_t LU, const arb_mat_t B, slong prec)
```

Solves \( AX = B \) given the precomputed nonsingular LU decomposition \( A = PLU \). The matrices \( X \) and \( B \) are allowed to be aliased with each other, but \( X \) is not allowed to be aliased with \( LU \).

```c
int arb_mat_solve(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, slong prec)
int arb_mat_solve_lu(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, slong prec)
```
int arb_mat_solve_precond(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, slong prec)
Solves $AX = B$ where $A$ is a nonsingular $n \times n$ matrix and $X$ and $B$ are $n \times m$ matrices.

If $m > 0$ and $A$ cannot be inverted numerically (indicating either that $A$ is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that $A$ is invertible and that the exact solution matrix is contained in the output.

Three algorithms are provided:

- The $lu$ version performs LU decomposition directly in ball arithmetic. This is fast, but the bounds typically blow up exponentially with $n$, even if the system is well-conditioned. This algorithm is usually the best choice at very high precision.
- The $precond$ version computes an approximate inverse to precondition the system [HS1967]. This is usually several times slower than direct LU decomposition, but the bounds do not blow up with $n$ if the system is well-conditioned. This algorithm is usually the best choice for large systems at low to moderate precision.
- The default version selects between $lu$ and $precomp$ automatically.

The automatic choice should be reasonable most of the time, but users may benefit from trying either $lu$ or $precond$ in specific applications. For example, the $lu$ solver often performs better for ill-conditioned systems where use of very high precision is unavoidable.

int arb_mat_solve_preapprox(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, const arb_mat_t R, const arb_mat_t T, slong prec)
Solves $AX = B$ where $A$ is a nonsingular $n \times n$ matrix and $X$ and $B$ are $n \times m$ matrices, given an approximation $R$ of the matrix inverse of $A$, and given the approximation $T$ of the solution $X$.

If $m > 0$ and $A$ cannot be inverted numerically (indicating either that $A$ is singular or that the precision is insufficient, or that $R$ is not a close enough approximation of the inverse of $A$), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that $A$ is invertible and that the exact solution matrix is contained in the output.

int arb_mat_inv(arb_mat_t X, const arb_mat_t A, slong prec)
Sets $X = A^{-1}$ where $A$ is a square matrix, computed by solving the system $AX = I$.

If $A$ cannot be inverted numerically (indicating either that $A$ is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the matrix is invertible and that the exact inverse is contained in the output.

void arb_mat_det_lu(arb_t det, const arb_mat_t A, slong prec)
void arb_mat_det_precond(arb_t det, const arb_mat_t A, slong prec)
void arb_mat_det(arb_t det, const arb_mat_t A, slong prec)
Sets $det$ to the determinant of the matrix $A$.

The $lu$ version uses Gaussian elimination with partial pivoting. If at some point an invertible pivot element cannot be found, the elimination is stopped and the magnitude of the determinant of the remaining submatrix is bounded using Hadamard’s inequality.

The $precond$ version computes an approximate LU factorization of $A$ and multiplies by the inverse $L$ and $U$ matrices as preconditioners to obtain a matrix close to the identity matrix [Rum2010]. An enclosure for this determinant is computed using Gershgorin circles. This is about four times slower than direct Gaussian elimination, but much more numerically stable.

The default version automatically selects between the $lu$ and $precond$ versions and additionally handles small or triangular matrices by direct formulas.

void arb_mat_approx_solve_triu(arb_mat_t X, const arb_mat_t U, const arb_mat_t B, int unit, slong prec)
These methods perform approximate solving without any error control. The radii in the input matrices are ignored, the computations are done numerically with floating-point arithmetic (using ordinary Gaussian elimination and triangular solving, accelerated through the use of block recursive strategies for large matrices), and the output matrices are set to the approximate floating-point results with zeroed error bounds.

Approximate solutions are useful for computing preconditioning matrices for certified solutions. Some users may also find these methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.

### 9.14.14 Cholesky decomposition and solving

void \( \text{arb\_mat\_approx\_solve\_tril} ( \text{arb\_mat\_t} X, \text{const\ arb\_mat\_t} L, \text{const\ arb\_mat\_t} B, \text{int\ unit, slong\ prec} ) \)

int \( \text{arb\_mat\_approx\_lu} ( \text{slong} *P, \text{arb\_mat\_t} LU, \text{const\ arb\_mat\_t} A, \text{slong\ prec} ) \)

void \( \text{arb\_mat\_approx\_solve\_lu\_precomp} ( \text{arb\_mat\_t} X, \text{const\ slong} *\text{perm}, \text{const\ arb\_mat\_t} A, \text{const\ arb\_mat\_t} B, \text{slong\ prec} ) \)

int \( \text{arb\_mat\_approx\_solve} ( \text{arb\_mat\_t} X, \text{const\ arb\_mat\_t} A, \text{const\ arb\_mat\_t} B, \text{slong\ prec} ) \)

int \( \text{arb\_mat\_approx\_inv} ( \text{arb\_mat\_t} X, \text{const\ arb\_mat\_t} A, \text{slong\ prec} ) \)

These methods perform approximate solving without any error control. The radii in the input matrices are ignored, the computations are done numerically with floating-point arithmetic (using ordinary Gaussian elimination and triangular solving, accelerated through the use of block recursive strategies for large matrices), and the output matrices are set to the approximate floating-point results with zeroed error bounds.

Approximate solutions are useful for computing preconditioning matrices for certified solutions. Some users may also find these methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.

### 9.14.14 Cholesky decomposition and solving

int \( \text{arb\_mat\_cholesky\_banachiewicz} ( \text{arb\_mat\_t} A, \text{slong\ prec} ) \)

int \( \text{arb\_mat\_cho} ( \text{arb\_mat\_t} L, \text{const\ arb\_mat\_t} A, \text{slong\ prec} ) \)

Computes the Cholesky decomposition of \( A \), returning nonzero if the symmetric matrix defined by the lower triangular part of \( A \) is certainly positive definite.

If a nonzero value is returned, then \( L \) is set to the lower triangular matrix such that \( A = L \cdot L^T \).

If zero is returned, then either the matrix is not symmetric positive definite, the input matrix was computed to insufficient precision, or the decomposition was attempted at insufficient precision.

The underscore method computes \( L \) from \( A \) in-place, leaving the strict upper triangular region undefined.

void \( \text{arb\_mat\_solve\_cho\_precomp} ( \text{arb\_mat\_t} X, \text{const\ arb\_mat\_t} L, \text{const\ arb\_mat\_t} B, \text{slong\ prec} ) \)

Solves \( AX = B \) given the precomputed Cholesky decomposition \( A = LL^T \). The matrices \( X \) and \( B \) are allowed to be aliased with each other, but \( X \) is not allowed to be aliased with \( L \).

int \( \text{arb\_mat\_spd\_solve} ( \text{arb\_mat\_t} X, \text{const\ arb\_mat\_t} A, \text{const\ arb\_mat\_t} B, \text{slong\ prec} ) \)

Solves \( AX = B \) where \( A \) is a symmetric positive definite matrix and \( X \) and \( B \) are \( n \times m \) matrices, using Cholesky decomposition.

If \( m > 0 \) and \( A \) cannot be factored using Cholesky decomposition (indicating either that \( A \) is not symmetric positive definite or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the symmetric matrix defined through the lower triangular part of \( A \) is invertible and that the exact solution matrix is contained in the output.

void \( \text{arb\_mat\_inv\_cho\_precomp} ( \text{arb\_mat\_t} X, \text{const\ arb\_mat\_t} L, \text{slong\ prec} ) \)

Sets \( X = A^{-1} \) where \( A \) is a symmetric positive definite matrix whose Cholesky decomposition \( L \) has been computed with \( \text{arb\_mat\_cho}() \). The inverse is calculated using the method of [Kri2013] which is more efficient than solving \( AX = I \) with \( \text{arb\_mat\_solve\_cho\_precomp}() \).

int \( \text{arb\_mat\_spd\_inv} ( \text{arb\_mat\_t} X, \text{const\ arb\_mat\_t} A, \text{slong\ prec} ) \)

Sets \( X = A^{-1} \) where \( A \) is a symmetric positive definite matrix. It is calculated using the method of [Kri2013] which computes fewer intermediate results than solving \( AX = I \) with \( \text{arb\_mat\_spd\_solve}() \).
If \( A \) cannot be factored using Cholesky decomposition (indicating either that \( A \) is not symmetric positive definite or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the symmetric matrix defined through the lower triangular part of \( A \) is invertible and that the exact inverse is contained in the output.

```c
int _arb_mat_ldl_inplace(arb_mat_t A, slong prec)
```

```c
int _arb_mat_ldl_golub_and_van_loan(arb_mat_t A, slong prec)
```

```c
int arb_mat_ldl(arb_mat_t res, const arb_mat_t A, slong prec)
```

Computes the \( LDL^T \) decomposition of \( A \), returning nonzero iff the symmetric matrix defined by the lower triangular part of \( A \) is certainly positive definite.

If a nonzero value is returned, then \( res \) is set to a lower triangular matrix that encodes the \( LDL^T \) decomposition of \( A \). In particular, \( L \) is a lower triangular matrix with ones on its diagonal and whose strictly lower triangular region is the same as that of \( res \). \( D \) is a diagonal matrix with the same diagonal as that of \( res \).

If zero is returned, then either the matrix is not symmetric positive definite, the input matrix was computed to insufficient precision, or the decomposition was attempted at insufficient precision.

The underscore methods compute \( res \) from \( A \) in-place, leaving the strict upper triangular region undefined. The default method uses algorithm 4.1.2 from [GVL1996].

```c
void arb_mat_solve_ldl_precomp(arb_mat_t X, const arb_mat_t L, const arb_mat_t B, slong prec)
```

Solves \( AX = B \) given the precomputed \( A = LDL^T \) decomposition encoded by \( L \). The matrices \( X \) and \( B \) are allowed to be aliased with each other, but \( X \) is not allowed to be aliased with \( L \).

```c
void arb_mat_inv_ldl_precomp(arb_mat_t X, const arb_mat_t L, slong prec)
```

Sets \( X = A^{-1} \) where \( A \) is a symmetric positive definite matrix whose \( LDL^T \) decomposition encoded by \( L \) has been computed with \( arb_mat_ldl() \). The inverse is calculated using the method of [Kri2013] which is more efficient than solving \( AX = I \) with \( arb_mat_solve_ldl_precomp() \).

### 9.14.15 Characteristic polynomial and companion matrix

```c
void _arb_mat_charpoly(arb_ptr poly, const arb_mat_t mat, slong prec)
```

```c
void arb_mat_charpoly(arb_poly_t poly, const arb_mat_t mat, slong prec)
```

Sets \( poly \) to the characteristic polynomial of \( mat \) which must be a square matrix. If the matrix has \( n \) rows, the underscore method requires space for \( n + 1 \) output coefficients. Employs a division-free algorithm using \( O(n^4) \) operations.

```c
void _arb_mat_companion(arb_mat_t mat, arb_srcptr poly, slong prec)
```

```c
void arb_mat_companion(arb_mat_t mat, const arb_poly_t poly, slong prec)
```

Sets the \( n \) by \( n \) matrix \( mat \) to the companion matrix of the polynomial \( poly \) which must have degree \( n \). The underscore method reads \( n + 1 \) input coefficients.
9.14.16 Special functions

void \texttt{arb\_mat\_exp\_taylor\_sum}(arb\_mat\_t S, const arb\_mat\_t A, slong N, slong prec)

Sets \( S \) to the truncated exponential Taylor series
\[ S = \sum_{k=0}^{N-1} A^k / k! \]. Uses rectangular splitting to compute the sum using \( O(\sqrt{N}) \) matrix multiplications. The recurrence relation for factorials is used to get scalars that are small integers instead of full factorials. As in [Joh2014b], all divisions are postponed to the end by computing partial factorials of length \( O(\sqrt{N}) \). The scalars could be reduced by doing more divisions, but this appears to be slower in most cases.

void \texttt{arb\_mat\_exp}(arb\_mat\_t B, const arb\_mat\_t A, slong prec)

Sets \( B \) to the exponential of the matrix \( A \), defined by the Taylor series
\[ \exp(A) = \sum_{k=0}^{\infty} A^k / k! \].

The function is evaluated as \( \exp(A/2^r)^2 \), where \( r \) is chosen to give rapid convergence.

The elementwise error when truncating the Taylor series after \( N \) terms is bounded by the error in the infinity norm, for which we have
\[
\left\| \exp(2^{-r}A) - \sum_{k=0}^{N-1} \frac{(2^{-r}A)^k}{k!} \right\|_\infty \leq \sum_{k=N}^{\infty} \frac{(2^{-r}A)^k}{k!}.
\]

We bound the sum on the right using \texttt{mag\_exp\_tail()} . Truncation error is not added to entries whose values are determined by the sparsity structure of \( A \).

void \texttt{arb\_mat\_trace}(arb\_t trace, const arb\_mat\_t mat, slong prec)

Sets \( \text{trace} \) to the trace of the matrix, i.e. the sum of entries on the main diagonal of \( \text{mat} \). The matrix is required to be square.

void \texttt{_arb\_mat\_diag\_prod}(arb\_t res, const arb\_mat\_t mat, slong a, slong b, slong prec)

void \texttt{arb\_mat\_diag\_prod}(arb\_t res, const arb\_mat\_t mat, slong prec)

Sets \( \text{res} \) to the product of the entries on the main diagonal of \( \text{mat} \). The underscore method computes the product of the entries between index \( a \) inclusive and \( b \) exclusive (the indices must be in range).

9.14.17 Sparsity structure

void \texttt{arb\_mat\_entrywise\_is\_zero}(fmpz\_mat\_t dest, const arb\_mat\_t src)

Sets each entry of \( \text{dest} \) to indicate whether the corresponding entry of \( \text{src} \) is certainly zero. If the entry of \( \text{src} \) at row \( i \) and column \( j \) is zero according to \texttt{arb\_is\_zero()} then the entry of \( \text{dest} \) at that row and column is set to one, otherwise that entry of \( \text{dest} \) is set to zero.

void \texttt{arb\_mat\_entrywise\_not\_is\_zero}(fmpz\_mat\_t dest, const arb\_mat\_t src)

Sets each entry of \( \text{dest} \) to indicate whether the corresponding entry of \( \text{src} \) is not certainly zero. This the complement of \texttt{arb\_mat\_entrywise\_is\_zero()}. 

slong \texttt{arb\_mat\_count\_is\_zero}(const arb\_mat\_t mat)

Returns the number of entries of \( \text{mat} \) that are certainly zero according to \texttt{arb\_is\_zero()}. 

slong \texttt{arb\_mat\_count\_not\_is\_zero}(const arb\_mat\_t mat)

Returns the number of entries of \( \text{mat} \) that are not certainly zero.
9.14.18 Component and error operations

void arb_mat_get_mid(arb_mat_t B, const arb_mat_t A)
    Sets the entries of B to the exact midpoints of the entries of A.

void arb_mat_add_error_mag(arb_mat_t mat, const mag_t err)
    Adds err in-place to the radii of the entries of mat.

9.14.19 Eigenvalues and eigenvectors

To compute eigenvalues and eigenvectors, one can convert to an acb_mat_t and use the functions in acb_mat.h: Eigenvalues and eigenvectors. In the future dedicated methods for real matrices will be added here.

9.14.20 LLL reduction

int arb_mat_spd_get_fmpz_mat(fmpz_mat_t B, const arb_mat_t A, slong prec)
    Attempts to set B to a symmetric and positive definite matrix obtained by rounding the midpoints of entries of \(2^{\text{prec}} \cdot A\) to integers. Returns 1 on success. Returns 0 and leaves B undefined if A is not symmetric or the result of rounding is not a positive definite matrix. The warnings of \texttt{arf_get_fmpz()} apply.

void arb_mat_spd_lll_reduce(fmpz_mat_t U, const arb_mat_t A, slong prec)
    Given a symmetric positive definite matrix A, sets U to an invertible matrix such that \(U^T A U\) is close to being LLL-reduced. If \texttt{arb_mat_spd_get_fmpz_mat()} succeeds at the chosen precision, we call \texttt{fmpz_lll()}, and otherwise set U to the identity matrix. The warnings of \texttt{arf_get_fmpz()} apply.

int arb_mat_spd_is_lll_reduced(const arb_mat_t A, slong tol_exp, slong prec)
    Given a symmetric positive definite matrix A, returns nonzero iff A is certainly LLL-reduced with a tolerance of \(\varepsilon = 2^{\text{tol}\_\text{exp}}\), meaning that it satisfies the inequalities \(|\mu_{j,k}| \leq \eta + \varepsilon\) and \((\delta - \varepsilon)||b_{k-1}^*||^2 \leq ||b_k^*||^2 + \mu_{k,k-1}^2||b_{k-1}^*||^2\) (with the usual notation) for the default parameters \(\eta = 0.51, \delta = 0.99\).

9.15 acb_mat.h – matrices over the complex numbers

An \texttt{acb_mat_t} represents a dense matrix over the complex numbers, implemented as an array of entries of type \texttt{acb_struct}. The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

\textbf{Note:} Methods prefixed with \texttt{acb_mat_approx} treat all input entries as floating-point numbers (ignoring the radii of the balls) and compute floating-point output (balls with zero radius) representing approximate solutions \textit{without error bounds}. All other methods compute rigorous error bounds. The \texttt{approx} methods are typically useful for computing initial values or preconditioners for rigorous solvers. Some users may also find \texttt{approx} methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.
### 9.15.1 Types, macros and constants

**type acb_mat_struct**

**type acb_mat_t**

Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An `acb_mat_t` is defined as an array of length one of type `acb_mat_struct`, permitting an `acb_mat_t` to be passed by reference.

**acb_mat_entry(mat, i, j)**

Macro giving a pointer to the entry at row `i` and column `j`.

**acb_mat_nrows(mat)**

Returns the number of rows of the matrix.

**acb_mat_ncols(mat)**

Returns the number of columns of the matrix.

### 9.15.2 Memory management

**void acb_mat_init(acb_mat_t mat, slong r, slong c)**

Initializes the matrix, setting it to the zero matrix with `r` rows and `c` columns.

**void acb_mat_clear(acb_mat_t mat)**

Clears the matrix, deallocating all entries.

**slong acb_mat_allocated_bytes(const acb_mat_t x)**

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(acb_mat_struct)` to get the size of the object as a whole.

**void acb_mat_window_init(acb_mat_t window, const acb_mat_t mat, slong r1, slong c1, slong r2, slong c2)**

Initializes `window` to a window matrix into the submatrix of `mat` starting at the corner at row `r1` and column `c1` (inclusive) and ending at row `r2` and column `c2` (exclusive).

**void acb_mat_window_clear(acb_mat_t window)**

Frees the window matrix.

### 9.15.3 Conversions

**void acb_mat_set(acb_mat_t dest, const acb_mat_t src)**

**void acb_mat_set_fmpz_mat(acb_mat_t dest, const fmpz_mat_t src)**

**void acb_mat_set_round_fmpz_mat(acb_mat_t dest, const fmpz_mat_t src, slong prec)**

**void acb_mat_set_fmpq_mat(acb_mat_t dest, const fmpq_mat_t src, slong prec)**

**void acb_mat_set_arb_mat(acb_mat_t dest, const arb_mat_t src)**

**void acb_mat_set_round_arb_mat(acb_mat_t dest, const arb_mat_t src, slong prec)**

Sets `dest` to `src`. The operands must have identical dimensions.

**void acb_mat_get_real(arb_mat_t re, const arb_mat_t mat)**

**void acb_mat_get_imag(arb_mat_t im, const arb_mat_t mat)**

Sets `re` or `im` to the real or imaginary part of `mat`, respectively. The operands must have identical dimensions.
void \texttt{acb\_mat\_set\_real\_imag}(\texttt{acb\_mat\_t mat, const arb\_mat\_t re, const arb\_mat\_t im})

Sets \texttt{mat} to the complex matrix with real and imaginary parts \texttt{re}, \texttt{im}. The operands must have identical dimensions.

\subsection*{9.15.4 Random generation}

void \texttt{acb\_mat\_randtest}(\texttt{acb\_mat\_t mat, flint\_rand\_t state, slong prec, slong mag\_bits})

Sets \texttt{mat} to a random matrix with up to \texttt{prec} bits of precision and with exponents of width up to \texttt{mag\_bits}.

void \texttt{acb\_mat\_randtest\_eig}(\texttt{acb\_mat\_t mat, flint\_rand\_t state, acb\_srcptr E, slong prec})

Sets \texttt{mat} to a random matrix with the prescribed eigenvalues supplied as the vector \texttt{E}. The output matrix is required to be square. We generate a random unitary matrix via a matrix exponential, and then evaluate an inverse Schur decomposition.

\subsection*{9.15.5 Input and output}

void \texttt{acb\_mat\_printd}(\texttt{const acb\_mat\_t mat, slong digits})

Prints each entry in the matrix with the specified number of decimal digits.

void \texttt{acb\_mat\_fprintd}(\texttt{FILE \*file, const acb\_mat\_t mat, slong digits})

Prints each entry in the matrix with the specified number of decimal digits to the stream \texttt{file}.

\subsection*{9.15.6 Comparisons}

Predicate methods return 1 if the property certainly holds and 0 otherwise.

int \texttt{acb\_mat\_equal}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})

Returns whether the matrices have the same dimensions and identical intervals as entries.

int \texttt{acb\_mat\_overlaps}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})

Returns whether the matrices have the same dimensions and each entry in \texttt{mat1} overlaps with the corresponding entry in \texttt{mat2}.

int \texttt{acb\_mat\_contains}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})

int \texttt{acb\_mat\_contains\_fmpz\_mat}(\texttt{const acb\_mat\_t mat1, const fmpz\_mat\_t mat2})

int \texttt{acb\_mat\_contains\_fmpq\_mat}(\texttt{const acb\_mat\_t mat1, const fmpq\_mat\_t mat2})

Returns whether the matrices have the same dimensions and each entry in \texttt{mat2} is contained in the corresponding entry in \texttt{mat1}.

int \texttt{acb\_mat\_eq}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})

Returns whether \texttt{mat1} and \texttt{mat2} certainly represent the same matrix.

int \texttt{acb\_mat\_ne}(\texttt{const acb\_mat\_t mat1, const acb\_mat\_t mat2})

Returns whether \texttt{mat1} and \texttt{mat2} certainly do not represent the same matrix.

int \texttt{acb\_mat\_is\_real}(\texttt{const acb\_mat\_t mat})

Returns whether all entries in \texttt{mat} have zero imaginary part.

int \texttt{acb\_mat\_is\_empty}(\texttt{const acb\_mat\_t mat})

Returns whether the number of rows or the number of columns in \texttt{mat} is zero.

int \texttt{acb\_mat\_is\_square}(\texttt{const acb\_mat\_t mat})

Returns whether the number of rows is equal to the number of columns in \texttt{mat}.
int acb_mat_is_exact(const acb_mat_t mat)
Returns whether all entries in mat have zero radius.

int acb_mat_is_zero(const acb_mat_t mat)
Returns whether all entries in mat are exactly zero.

int acb_mat_is_finite(const acb_mat_t mat)
Returns whether all entries in mat are finite.

int acb_mat_is_triu(const acb_mat_t mat)
Returns whether mat is upper triangular; that is, all entries below the main diagonal are exactly zero.

int acb_mat_is_tril(const acb_mat_t mat)
Returns whether mat is lower triangular; that is, all entries above the main diagonal are exactly zero.

int acb_mat_is_diag(const acb_mat_t mat)
Returns whether mat is a diagonal matrix; that is, all entries off the main diagonal are exactly zero.

### 9.15.7 Special matrices

void acb_mat_zero(acb_mat_t mat)
Sets all entries in mat to zero.

void acb_mat_one(acb_mat_t mat)
Sets the entries on the main diagonal to ones, and all other entries to zero.

void acb_mat_ones(acb_mat_t mat)
Sets all entries in the matrix to ones.

void acb_mat_onei(acb_mat_t mat)
Sets the entries of the main diagonal to $i = \sqrt{-1}$ and all other entries to zero.

void acb_mat_indeterminate(acb_mat_t mat)
Sets all entries in the matrix to indeterminate (NaN).

void acb_mat_dft(acb_mat_t mat, int type, slong prec)
Sets mat to the DFT (discrete Fourier transform) matrix of order $n$ where $n$ is the smallest dimension of mat (if mat is not square, the matrix is extended periodically along the larger dimension). Here, we use the normalized DFT matrix

$$A_{j,k} = \frac{\omega^{j k}}{\sqrt{n}}, \quad \omega = e^{-2\pi i/n}.$$  

The type parameter is currently ignored and should be set to 0. In the future, it might be used to select a different convention.

### 9.15.8 Transpose

void acb_mat_transpose(acb_mat_t dest, const acb_mat_t src)
Sets dest to the exact transpose src. The operands must have compatible dimensions. Aliasing is allowed.

void acb_mat_conjugate_transpose(acb_mat_t dest, const acb_mat_t src)
Sets dest to the conjugate transpose of src. The operands must have compatible dimensions. Aliasing is allowed.

void acb_mat_conjugate(acb_mat_t dest, const acb_mat_t src)
Sets dest to the elementwise complex conjugate of src.
9.15.9 Norms

```c
void acb_mat_bound_inf_norm(mag_t b, const acb_mat_t A)
    Sets b to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of A.

void acb_mat_frobenius_norm(arb_t res, const acb_mat_t A, slong prec)
    Sets res to the Frobenius norm (i.e. the square root of the sum of squares of entries) of A.

void acb_mat_bound_frobenius_norm(mag_t res, const acb_mat_t A)
    Sets res to an upper bound for the Frobenius norm of A.
```

9.15.10 Arithmetic

```c
void acb_mat_neg(acb_mat_t dest, const acb_mat_t src)
    Sets dest to the exact negation of src. The operands must have the same dimensions.

void acb_mat_add(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)
    Sets res to the sum of mat1 and mat2. The operands must have the same dimensions.

void acb_mat_sub(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)
    Sets res to the difference of mat1 and mat2. The operands must have the same dimensions.

void acb_mat_mul_classical(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)

void acb_mat_mul_threaded(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)

void acb_mat_mul_reorder(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)

void acb_mat_mul(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)
    Sets res to the matrix product of mat1 and mat2. The operands must have compatible dimensions for matrix multiplication.

    The classical version performs matrix multiplication in the trivial way.

    The threaded version performs classical multiplication but splits the computation over the number of threads returned by flint_get_num_threads().

    The reorder version reorders the data and performs one to four real matrix multiplications via arb_mat_mul() .

    The default version chooses an algorithm automatically.

void acb_mat_mul_entrywise(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)
    Sets res to the entrywise product of mat1 and mat2. The operands must have the same dimensions.

void acb_mat_sqr_classical(acb_mat_t res, const acb_mat_t mat, slong prec)

void acb_mat_sqr(acb_mat_t res, const acb_mat_t mat, slong prec)
    Sets res to the matrix square of mat. The operands must both be square with the same dimensions.

void acb_mat_pow_ui(acb_mat_t res, const acb_mat_t mat, ulong exp, slong prec)
    Sets res to mat raised to the power exp. Requires that mat is a square matrix.

void acb_mat_approx_mul(acb_mat_t res, const acb_mat_t mat1, const acb_mat_t mat2, slong prec)
```
Approximate matrix multiplication. The input radii are ignored and the output matrix is set to an approximate floating-point result. For performance reasons, the radii in the output matrix will not necessarily be written (zeroed), but will remain zero if they are already zeroed in res before calling this function.

9.15.11 Scalar arithmetic

```c
void acb_mat_scalar_mul_2exp_si(acb_mat_t B, const acb_mat_t A, slong c)
  Sets B to A multiplied by \(2^c\).

void acb_mat_scalar_addmul_si(acb_mat_t B, const acb_mat_t A, slong c, slong prec)
void acb_mat_scalar_addmul_fmpz(acb_mat_t B, const acb_mat_t A, const fmpz_t c, slong prec)
void acb_mat_scalar_addmul_arb(acb_mat_t B, const acb_mat_t A, const arb_t c, slong prec)
void acb_mat_scalar_addmul_acb(acb_mat_t B, const acb_mat_t A, const acb_t c, slong prec)
  Sets B to \(B + A \times c\).

void acb_mat_scalar_mul_si(acb_mat_t B, const acb_mat_t A, slong c, slong prec)
void acb_mat_scalar_mul_fmpz(acb_mat_t B, const acb_mat_t A, const fmpz_t c, slong prec)
void acb_mat_scalar_mul_arb(acb_mat_t B, const acb_mat_t A, const arb_t c, slong prec)
void acb_mat_scalar_mul_acb(acb_mat_t B, const acb_mat_t A, const acb_t c, slong prec)
  Sets B to \(A \times c\).

void acb_mat_scalar_div_si(acb_mat_t B, const acb_mat_t A, slong c, slong prec)
void acb_mat_scalar_div_fmpz(acb_mat_t B, const acb_mat_t A, const fmpz_t c, slong prec)
void acb_mat_scalar_div_arb(acb_mat_t B, const acb_mat_t A, const arb_t c, slong prec)
void acb_mat_scalar_div_acb(acb_mat_t B, const acb_mat_t A, const acb_t c, slong prec)
  Sets B to \(A/c\).
```

9.15.12 Vector arithmetic

```c
void _acb_mat_vector_mul_row(acb_ptr res, acb_srcptr v, const acb_mat_t A, slong prec)
void _acb_mat_vector_mul_col(acb_ptr res, const acb_mat_t A, acb_srcptr v, slong prec)
void acb_mat_vector_mul_row(acb_ptr res, acb_srcptr v, const acb_mat_t A, slong prec)
void acb_mat_vector_mul_col(acb_ptr res, const acb_mat_t A, acb_srcptr v, slong prec)
  Sets res to the product \(vA\), (resp. \(Ae\)), where res and v are seen as row (resp. column) vectors. The lengths of the vectors must match the dimensions of A.
  The underscore methods do not allow aliasing between res and v.
```
9.15.13 Gaussian elimination and solving

```c
int acb_mat_lu_classical(slong *perm, acb_mat_t LU, const acb_mat_t A, slong prec)
int acb_mat_lu_recursive(slong *perm, acb_mat_t LU, const acb_mat_t A, slong prec)
int acb_mat_lu(slong *perm, acb_mat_t LU, const acb_mat_t A, slong prec)
```

Given an \( n \times n \) matrix \( A \), computes an LU decomposition \( PLU = A \) using Gaussian elimination with partial pivoting. The input and output matrices can be the same, performing the decomposition in-place.

Entry \( i \) in the permutation vector \( \text{perm} \) is set to the row index in the input matrix corresponding to row \( i \) in the output matrix.

The algorithm succeeds and returns nonzero if it can find \( n \) invertible (i.e. not containing zero) pivot entries. This guarantees that the matrix is invertible.

The algorithm fails and returns zero, leaving the entries in \( P \) and \( LU \) undefined, if it cannot find \( n \) invertible pivot elements. In this case, either the matrix is singular, the input matrix was computed to insufficient precision, or the LU decomposition was attempted at insufficient precision.

The classical version uses Gaussian elimination directly while the recursive version performs the computation in a block recursive way to benefit from fast matrix multiplication. The default version chooses an algorithm automatically.

```c
void acb_mat_solve_tril_classical(acb_mat_t X, const acb_mat_t L, const acb_mat_t B, int unit, slong prec)
void acb_mat_solve_tril_recursive(acb_mat_t X, const acb_mat_t L, const acb_mat_t B, int unit, slong prec)
void acb_mat_solve_tril(acb_mat_t X, const acb_mat_t L, const acb_mat_t B, int unit, slong prec)
void acb_mat_solve_triu_classical(acb_mat_t X, const acb_mat_t U, const acb_mat_t B, int unit, slong prec)
void acb_mat_solve_triu_recursive(acb_mat_t X, const acb_mat_t U, const acb_mat_t B, int unit, slong prec)
void acb_mat_solve_triu(acb_mat_t X, const acb_mat_t U, const acb_mat_t B, int unit, slong prec)
```

Solves the lower triangular system \( LX = B \) or the upper triangular system \( UX = B \), respectively. If \( \text{unit} \) is set, the main diagonal of \( L \) or \( U \) is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The classical versions perform the computations iteratively while the recursive versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

```c
void acb_mat_solve_lu_precomp(acb_mat_t X, const slong *perm, const acb_mat_t LU, const acb_mat_t B, slong prec)
```

Solves \( AX = B \) given the precomputed nonsingular LU decomposition \( A = PLU \). The matrices \( X \) and \( B \) are allowed to be aliased with each other, but \( X \) is not allowed to be aliased with \( LU \).

```c
int acb_mat_solve(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, slong prec)
int acb_mat_solve_lu(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, slong prec)
```
int acb_mat_solve_precond(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, slong prec)
Solves $AX = B$ where $A$ is a nonsingular $n \times n$ matrix and $X$ and $B$ are $n \times m$ matrices.

If $m > 0$ and $A$ cannot be inverted numerically (indicating either that $A$ is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that $A$ is invertible and that the exact solution matrix is contained in the output.

Three algorithms are provided:

- The $lu$ version performs LU decomposition directly in ball arithmetic. This is fast, but the bounds typically blow up exponentially with $n$, even if the system is well-conditioned. This algorithm is usually the best choice at very high precision.
- The $precond$ version computes an approximate inverse to precondition the system. This is usually several times slower than direct LU decomposition, but the bounds do not blow up with $n$ if the system is well-conditioned. This algorithm is usually the best choice for large systems at low to moderate precision.
- The default version selects between $lu$ and $precomp$ automatically.

The automatic choice should be reasonable most of the time, but users may benefit from trying either $lu$ or $precond$ in specific applications. For example, the $lu$ solver often performs better for ill-conditioned systems where use of very high precision is unavoidable.

int acb_mat_inv(acb_mat_t X, const acb_mat_t A, slong prec)
Sets $X = A^{-1}$ where $A$ is a square matrix, computed by solving the system $AX = I$.

If $A$ cannot be inverted numerically (indicating either that $A$ is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the matrix is invertible and that the exact inverse is contained in the output.

void acb_mat_det_lu(acb_t det, const acb_mat_t A, slong prec)
void acb_mat_det_precond(acb_t det, const acb_mat_t A, slong prec)
void acb_mat_det(acb_t det, const acb_mat_t A, slong prec)

Sets $det$ to the determinant of the matrix $A$.

The $lu$ version uses Gaussian elimination with partial pivoting. If at some point an invertible pivot element cannot be found, the elimination is stopped and the magnitude of the determinant of the remaining submatrix is bounded using Hadamard's inequality.

The $precond$ version computes an approximate LU factorization of $A$ and multiplies by the inverse $L$ and $U$ matrices as preconditioners to obtain a matrix close to the identity matrix [Run2010]. An enclosure for this determinant is computed using Gershgorin circles. This is about four times slower than direct Gaussian elimination, but much more numerically stable.

The default version automatically selects between the $lu$ and $precond$ versions and additionally handles small or triangular matrices by direct formulas.

void acb_mat_approx_solve_triu(acb_mat_t X, const acb_mat_t U, const acb_mat_t B, int unit, slong prec)
void acb_mat_approx_solve_tril(acb_mat_t X, const acb_mat_t L, const acb_mat_t B, int unit, slong prec)

int acb_mat_approx_lu(slong *P, acb_mat_t LU, const acb_mat_t A, slong prec)
void acb_mat_approx_solve_lu_precomp(acb_mat_t X, const slong *perm, const acb_mat_t A, const acb_mat_t B, slong prec)
int acb_mat_approx_solve(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, slong prec)
int acb_mat_approx_inv(acb_mat_t X, const acb_mat_t A, slong prec)

These methods perform approximate solving without any error control. The radii in the input matrices are ignored, the computations are done numerically with floating-point arithmetic (using ordinary Gaussian elimination and triangular solving, accelerated through the use of block recursive strategies for large matrices), and the output matrices are set to the approximate floating-point results with zeroed error bounds.

9.15.14 Characteristic polynomial and companion matrix

void _acb_mat_charpoly(acb_ptr poly, const acb_mat_t mat, slong prec)

void acb_mat_charpoly(acb_poly_t poly, const acb_mat_t mat, slong prec)

Sets poly to the characteristic polynomial of mat which must be a square matrix. If the matrix has \( n \) rows, the underscore method requires space for \( n + 1 \) output coefficients. Employs a division-free algorithm using \( O(n^4) \) operations.

void _acb_mat_companion(acb_mat_t mat, acb_srcptr poly, slong prec)

void acb_mat_companion(acb_mat_t mat, const acb_poly_t poly, slong prec)

Sets the \( n \) by \( n \) matrix mat to the companion matrix of the polynomial poly which must have degree \( n \). The underscore method reads \( n + 1 \) input coefficients.

9.15.15 Special functions

void acb_mat_exp_taylor_sum(acb_mat_t S, const acb_mat_t A, slong N, slong prec)

Sets S to the truncated exponential Taylor series \( S = \sum_{k=0}^{N-1} \frac{A^k}{k!} \). See arb_mat_exp_taylor_sum() for implementation notes.

void acb_mat_exp(acb_mat_t B, const acb_mat_t A, slong prec)

Sets B to the exponential of the matrix A, defined by the Taylor series

\[
\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.
\]

The function is evaluated as \( \exp(A/2^r)^2 \), where \( r \) is chosen to give rapid convergence of the Taylor series. Error bounds are computed as for arb_mat_exp().

void acb_mat_trace(acb_t trace, const acb_mat_t mat, slong prec)

Sets trace to the trace of the matrix, i.e. the sum of entries on the main diagonal of mat. The matrix is required to be square.

void _acb_mat_diag_prod(acb_t res, const acb_mat_t mat, slong a, slong b, slong prec)

void acb_mat_diag_prod(acb_t res, const acb_mat_t mat, slong prec)

Sets res to the product of the entries on the main diagonal of mat. The underscore method computes the product of the entries between index \( a \) inclusive and \( b \) exclusive (the indices must be in range).
9.15.16 Component and error operations

void acb_mat_get_mid(acb_mat_t B, const acb_mat_t A)
Sets the entries of B to the exact midpoints of the entries of A.

void acb_mat_add_error_mag(acb_mat_t mat, const mag_t err)
Adds err in-place to the radii of the entries of mat.

9.15.17 Eigenvalues and eigenvectors

The functions in this section are experimental. There are classes of matrices where the algorithms fail to converge even as prec is increased, or for which the error bounds are much worse than necessary. In some cases, it can help to manually precondition the matrix A by applying a similarity transformation $T^{-1}AT$.

- If $A$ is badly scaled, take $T$ to be a matrix such that the entries of $T^{-1}AT$ are more uniform (this is known as balancing).
- Simply taking $T$ to be a random invertible matrix can help if an algorithm fails to converge despite $A$ being well-scaled. (This can be the case when dealing with multiple eigenvalues.)

int acb_mat_approx_eig_qr(acb_ptr E, acb_mat_t L, acb_mat_t R, const acb_mat_t A, const mag_t tol, slong maxiter, slong prec)
Computes floating-point approximations of all the n eigenvalues (and optionally eigenvectors) of the given n by n matrix $A$. The approximations of the eigenvalues are written to the vector $E$, in no particular order. If $L$ is not NULL, approximations of the corresponding left eigenvectors are written to the rows of $L$. If $R$ is not NULL, approximations of the corresponding right eigenvectors are written to the columns of $R$.

The parameters tol and maxiter can be used to control the target numerical error and the maximum number of iterations allowed before giving up. Passing NULL and 0 respectively results in default values being used.

Uses the implicitly shifted QR algorithm with reduction to Hessenberg form. No guarantees are made about the accuracy of the output. A nonzero return value indicates that the QR iteration converged numerically, but this is only a heuristic termination test and does not imply any statement whatsoever about error bounds. The output may also be accurate even if this function returns zero.

void acb_mat_eig_global_enclosure(mag_t eps, const acb_mat_t A, acb_srcptr E, const acb_mat_t R, slong prec)
Given an n by n matrix $A$, a length-n vector $E$ containing approximations of the eigenvalues of $A$, and an n by n matrix $R$ containing approximations of the corresponding right eigenvectors, computes a rigorous bound $\varepsilon$ such that every eigenvalue $\lambda$ of $A$ satisfies $|\lambda - \hat{\lambda}_k| \leq \varepsilon$ for some $\hat{\lambda}_k$ in $E$. In other words, the union of the balls $B_k = \{z : |z - \hat{\lambda}_k| \leq \varepsilon\}$ is guaranteed to be an enclosure of all eigenvalues of $A$.

Note that there is no guarantee that each ball $B_k$ can be identified with a single eigenvalue: it is possible that some balls contain several eigenvalues while other balls contain no eigenvalues. In other words, this method is not powerful enough to compute isolating balls for the individual eigenvalues (or even for clusters of eigenvalues other than the whole spectrum). Nevertheless, in practice the balls $B_k$ will represent eigenvalues one-to-one with high probability if the given approximations are good.

The output can be used to certify that all eigenvalues of $A$ lie in some region of the complex plane (such as a specific half-plane, strip, disk, or annulus) without the need to certify the individual eigenvalues. The output is easily converted into lower or upper bounds for the absolute values or real or imaginary parts of the spectrum, and with high probability these bounds will be tight. Using acb_add_error_mag() and acb_union(), the output can also be converted to a single
void acb_mat_eig_enclosure_rump(acb_t lambda, acb_mat_t J, acb_mat_t R, const acb_mat_t A,
        const acb_t lambda_approx, const acb_mat_t R_approx, slong prec)

Given an \( n \) by \( n \) matrix \( A \) and an approximate eigenvalue-eigenvector pair \( \lambda_{\text{approx}} \) and \( R_{\text{approx}} \) (where \( R_{\text{approx}} \) is an \( n \) by 1 matrix), computes an enclosure \( \lambda \) guaranteed to contain at least one of the eigenvalues of \( A \), along with an enclosure \( R \) for a corresponding right eigenvector.

More generally, this function can handle clustered (or repeated) eigenvalues. If \( R_{\text{approx}} \) is an \( n \) by \( k \) matrix containing approximate eigenvectors for a presumed cluster of \( k \) eigenvalues near \( \lambda_{\text{approx}} \), this function computes an enclosure \( \lambda \) guaranteed to contain at least \( k \) eigenvalues of \( A \) along with a matrix \( R \) guaranteed to contain a basis for the \( k \)-dimensional invariant subspace associated with these eigenvalues. Note that for multiple eigenvalues, determining the individual eigenvectors is an ill-posed problem; describing an enclosure of the invariant subspace is the best we can hope for.

For \( k = 1 \), it is guaranteed that \( AR - R\lambda \) contains the zero matrix. For \( k > 2 \), this cannot generally be guaranteed (in particular, \( A \) might not diagonalizable). In this case, we can still compute an approximately diagonal \( k \) by \( k \) interval matrix \( J \approx \lambda I \) such that \( AR - RJ \) is guaranteed to contain the zero matrix. This matrix has the property that the Jordan canonical form of (any exact matrix contained in) \( A \) has a \( k \) by \( k \) submatrix equal to the Jordan canonical form of (some exact matrix contained in) \( J \). The output \( J \) is optional (the user can pass \( \text{NULL} \) to omit it).

The algorithm follows section 13.4 in [Rum2010], corresponding to the \texttt{verifyeig()} routine in INT-LAB. The initial approximations can, for instance, be computed using \texttt{acb_mat_approx_eig_qr()}. No assumptions are made about the structure of \( A \) or the quality of the given approximations.

int acb_mat_eig_simple_rump(acb_ptr E, acb_mat_t L, acb_mat_t R, const acb_mat_t A,
        acb_srcptr E_approx, const acb_mat_t R_approx, slong prec)

int acb_mat_eig_simple_vdhoeven_mourrain(acb_ptr E, acb_mat_t L, acb_mat_t R, const
        acb_mat_t A, acb_srcptr E_approx, const acb_mat_t R_approx, slong prec)

int acb_mat_eig_simple(acb_ptr E, acb_mat_t L, acb_mat_t R, const acb_mat_t A, acb_srcptr
        E_approx, const acb_mat_t R_approx, slong prec)

Computes all the eigenvalues (and optionally corresponding eigenvectors) of the given \( n \) by \( n \) matrix \( A \).

Attempts to prove that \( A \) has \( n \) simple (isolated) eigenvalues, returning 1 if successful and 0 otherwise. On success, isolating complex intervals for the eigenvalues are written to the vector \( E \), in no particular order. If \( L \) is not \( \text{NULL} \), enclosures of the corresponding left eigenvectors are written to the rows of \( L \). If \( R \) is not \( \text{NULL} \), enclosures of the corresponding right eigenvectors are written to the columns of \( R \).

The left eigenvectors are normalized so that \( L = R^{-1} \). This produces a diagonalization \( LAR = D \) where \( D \) is the diagonal matrix with the entries in \( E \) on the diagonal.

The user supplies approximations \( E_{\text{approx}} \) and \( R_{\text{approx}} \) of the eigenvalues and the right eigenvectors. The initial approximations can, for instance, be computed using \texttt{acb_mat_approx_eig_qr()}. No assumptions are made about the structure of \( A \) or the quality of the given approximations.

Two algorithms are implemented:
The *rump* version calls `acb_mat_eig_enclosure_rump()` repeatedly to certify eigenvalue-eigenvector pairs one by one. The iteration is stopped to return non-success if a new eigenvalue overlaps with previously computed one. Finally, \( L \) is computed by a matrix inversion. This has complexity \( O(n^4) \).

The *vdhoeven_mourrain* version uses the algorithm in [HM2017] to certify all eigenvalues and eigenvectors in one step. This has complexity \( O(n^3) \).

The default version currently uses *vdhoeven_mourrain*.

By design, these functions terminate instead of attempting to compute eigenvalue clusters if some eigenvalues cannot be isolated. To compute all eigenvalues of a matrix allowing for overlap, `acb_mat_eig_multiple_rump()` may be used as a fallback, or `acb_mat_eig_multiple()` may be used in the first place.

```c
int acb_mat_eig_multiple_rump(acb_ptr E, const acb_mat_t A, acb_srcptr E_approx, const acb_mat_t R_approx, slong prec)

int acb_mat_eig_multiple(acb_ptr E, const acb_mat_t A, acb_srcptr E_approx, const acb_mat_t R_approx, slong prec)
```

Computes all the eigenvalues of the given \( n \times n \) matrix \( A \). On success, the output vector \( E \) contains \( n \) complex intervals, each representing one eigenvalue of \( A \) with the correct multiplicities in case of overlap. The output intervals are either disjoint or identical, and identical intervals are guaranteed to be grouped consecutively. Each complete run of \( k \) identical intervals thus represents a cluster of exactly \( k \) eigenvalues which could not be separated from each other at the current precision, but which could be isolated from the other \( n - k \) eigenvalues of the matrix.

The user supplies approximations \( E\_approx \) and \( R\_approx \) of the eigenvalues and the right eigenvectors. The initial approximations can, for instance, be computed using `acb_mat_approx_eig_qr()`. No assumptions are made about the structure of \( A \) or the quality of the given approximations.

The *rump* algorithm groups approximate eigenvalues that are close and calls `acb_mat_eig_enclosure_rump()` repeatedly to validate each cluster. The complexity is \( O(mn^4) \) for \( m \) clusters.

The default version, as currently implemented, first attempts to call `acb_mat_eig_simple_vdhoeven_mourrain()` hoping that the eigenvalues are actually simple. It then uses the *rump* algorithm as a fallback.

### 9.16 acb_hypgeom.h – hypergeometric functions of complex variables

The generalized hypergeometric function is formally defined by

\[
_pF_q(a_1, \ldots, a_p; b_1, \ldots, b_q; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \ldots (a_p)_k}{(b_1)_k \ldots (b_q)_k} \frac{z^k}{k!}.
\]

It can be interpreted using analytic continuation or regularization when the sum does not converge. In a looser sense, we understand “hypergeometric functions” to be linear combinations of generalized hypergeometric functions with prefactors that are products of exponentials, powers, and gamma functions.
9.16.1 Rising factorials

void acb_hypgeom_rising_ui_forward(acb_t res, const acb_t x, ulong n, slong prec)
void acb_hypgeom_rising_ui_bs(acb_t res, const acb_t x, ulong n, slong prec)
void acb_hypgeom_rising_ui_rs(acb_t res, const acb_t x, ulong n, ulong m, slong prec)
void acb_hypgeom_rising_ui_rec(acb_t res, const acb_t x, ulong n, slong prec)
void acb_hypgeom_rising_ui_powsum(acb_t res, const acb_t x, const acb_t n, slong prec)

Computes the rising factorial \((x)_n\).

The forward version uses the forward recurrence. The bs version uses binary splitting. The rs version uses rectangular splitting. It takes an extra tuning parameter \(m\) which can be set to zero to choose automatically. The rec version chooses an algorithm automatically, avoiding use of the gamma function (so that it can be used in the computation of the gamma function). The default versions (rising_ui and rising_ui) choose an algorithm automatically and may additionally fall back on the gamma function.

void acb_hypgeom_rising_ui_jet_powsum(acb_ptr res, const acb_t x, ulong n, slong len, slong prec)
void acb_hypgeom_rising_ui_jet_bs(acb_ptr res, const acb_t x, ulong n, slong len, slong prec)
void acb_hypgeom_rising_ui_jet_rs(acb_ptr res, const acb_t x, ulong n, ulong m, slong len, slong prec)
void acb_hypgeom_rising_ui_jet(acb_ptr res, const acb_t x, ulong n, slong len, slong prec)

Computes the jet of the rising factorial \((x)_n\), truncated to length \(len\). In other words, constructs the polynomial \((X + x)_n \in \mathbb{R}[X]\), truncated if \(len < n + 1\) (and zero-extended if \(len > n + 1\)).

The powsum version computes the sequence of powers of \(x\) and forms integral linear combinations of these. The bs version uses binary splitting. The rs version uses rectangular splitting. It takes an extra tuning parameter \(m\) which can be set to zero to choose automatically. The default version chooses an algorithm automatically.

void acb_hypgeom_log_rising_ui(acb_ptr res, const acb_t x, ulong n, slong prec)

Computes the log-rising factorial \(\log((x)_n) = \sum_{k=0}^{n-1} \log(x + k)\).

This first computes the ordinary rising factorial and then determines the branch correction \(2\pi i m\) with respect to the principal logarithm. The correction is computed using Hare’s algorithm in floating-point arithmetic if this is safe; otherwise, a direct computation of \(\sum_{k=0}^{n-1} \arg(x + k)\) is used as a fallback.

void acb_hypgeom_log_rising_ui_jet(acb_ptr res, const acb_t x, ulong n, slong len, slong prec)

Computes the jet of the log-rising factorial \(\log((x)_n)\), truncated to length \(len\).

9.16.2 Gamma function

void acb_hypgeom_gamma_stirling_sum_horner(acb_t s, const acb_t z, slong N, slong prec)
void acb_hypgeom_gamma_stirling_sum_improved(acb_t s, const acb_t z, slong N, slong K, slong prec)

Sets \(res\) to the final sum in the Stirling series for the gamma function truncated before the term with index \(N\), i.e. computes \(\sum_{n=1}^{N} \frac{B_{2n}}{(2n-1)!2^{2n-1}}\). The horner version uses Horner scheme with gradual precision adjustments. The improved version uses rectangular splitting for the low-index terms and reexpands the high-index terms as hypergeometric polynomials, using a splitting parameter \(K\) (which can be set to 0 to use a default value).

void acb_hypgeom_gamma_stirling(acb_t s, const acb_t x, int reciprocal, slong prec)

Sets \(res\) to the gamma function of \(x\) computed using the Stirling series together with argument reduction. If \(reciprocal\) is set, the reciprocal gamma function is computed instead.
int acb_hypgeom_gamma_taylor(acb_t res, const acb_t x, int reciprocal, slong prec)

Attempts to compute the gamma function of x using Taylor series together with argument reduction. This is only supported if x and prec are both small enough. If successful, returns 1; otherwise, does nothing and returns 0. If reciprocal is set, the reciprocal gamma function is computed instead.

void acb_hypgeom_gamma(acb_t res, const acb_t x, slong prec)

Sets res to the gamma function of x computed using a default algorithm choice.

void acb_hypgeom_rgamma(acb_t res, const acb_t x, slong prec)

Sets res to the reciprocal gamma function of x computed using a default algorithm choice.

void acb_hypgeom_lgamma(acb_t res, const acb_t x, slong prec)

Sets res to the principal branch of the log-gamma function of x computed using a default algorithm choice.

9.16.3 Convergent series

In this section, we define

\[ T(k) = \prod_{i=0}^{p-1} (a_i)_{k} \prod_{i=0}^{q-1} (b_i)_{k} \]

and

\[ pFq(a_0, \ldots, a_{p-1}; b_1 \ldots b_{q-1}; z) = p+1Fq(a_0, \ldots, a_{p-1}, 1; b_1 \ldots b_{q-1}; z) = \sum_{k=0}^{\infty} T(k) \]

For the conventional generalized hypergeometric function \( pFq \), compute \( pFq+1 \) with the explicit parameter \( b_q = 1 \), or remove a 1 from the \( a_i \) parameters if there is one.

void acb_hypgeom_pfq_bound_factor(mag_t C, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, slong n)

Computes a factor \( C \) such that \( |\sum_{k=n}^{\infty} T(k)| \leq C |T(n)| \). See Convergent series. As currently implemented, the bound becomes infinite when \( n \) is too small, even if the series converges.

slong acb_hypgeom_pfq_choose_n(acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, slong prec)

Heuristically attempts to choose a number of terms \( n \) to sum of a hypergeometric series at a working precision of \( prec \) bits.

Uses double precision arithmetic internally. As currently implemented, it can fail to produce a good result if the parameters are extremely large or extremely close to nonpositive integers.

Numerical cancellation is assumed to be significant, so truncation is done when the current term is \( prec \) bits smaller than the largest encountered term.

This function will also attempt to pick a reasonable truncation point for divergent series.

void acb_hypgeom_pfq_sum_forward(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, slong n, slong prec)

void acb_hypgeom_pfq_sum_rs(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, slong n, slong prec)

void acb_hypgeom_pfq_sum_bs(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, slong n, slong prec)

void acb_hypgeom_pfq_sum_fme(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, slong n, slong prec)
void acb_hypgeom_pfq_sum(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, slong n, slong prec)

Computes $s = \sum_{k=0}^{n-1} T(k)$ and $t = T(n)$. Does not allow aliasing between input and output variables. We require $n \geq 0$.

The forward version computes the sum using forward recurrence.

The bs version computes the sum using binary splitting.

The rs version computes the sum in reverse order using rectangular splitting. It only computes a magnitude bound for the value of $t$.

The fme version uses fast multipoint evaluation.

The default version automatically chooses an algorithm depending on the inputs.

void acb_hypgeom_pfq_sum_bs_invz(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t w, slong n, slong prec)

Like acb_hypgeom_pfq_sum(), but taking advantage of $w = 1/z$ possibly having few bits.

void acb_hypgeom_pfq_sum_invz(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, const acb_t w, slong n, slong prec)

Like acb_hypgeom_pfq_sum(), but taking advantage of $w = 1/z$ possibly having few bits.

void acb_hypgeom_pfq_direct(acb_t res, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, slong n, slong prec)

Computes $p_{f_q}(z) = \sum_{k=0}^{\infty} T(k) = \sum_{k=0}^{n-1} T(k) + \varepsilon$

directly from the defining series, including a rigorous bound for the truncation error $\varepsilon$ in the output.

If $n < 0$, this function chooses a number of terms automatically using acb_hypgeom_pfq_choose_n().

void acb_hypgeom_pfq_series_sum_forward(acb_poly_t s, acb_poly_t t, const acb_poly_struct *a, slong p, const acb_poly_struct *b, slong q, const acb_poly_t z, int regularized, slong n, slong len, slong prec)

void acb_hypgeom_pfq_series_sum_bs(acb_poly_t s, acb_poly_t t, const acb_poly_struct *a, slong p, const acb_poly_struct *b, slong q, const acb_poly_t z, int regularized, slong n, slong len, slong prec)

void acb_hypgeom_pfq_series_sum_rs(acb_poly_t s, acb_poly_t t, const acb_poly_struct *a, slong p, const acb_poly_struct *b, slong q, const acb_poly_t z, int regularized, slong n, slong len, slong prec)

void acb_hypgeom_pfq_series_sum(acb_poly_t s, acb_poly_t t, const acb_poly_struct *a, slong p, const acb_poly_struct *b, slong q, const acb_poly_t z, int regularized, slong n, slong len, slong prec)

Computes $s = \sum_{k=0}^{n-1} T(k)$ and $t = T(n)$ given parameters and argument that are power series.

Does not allow aliasing between input and output variables. We require $n \geq 0$ and that len is positive.

If regularized is set, the regularized sum is computed, avoiding division by zero at the poles of the gamma function.

The forward, bs, rs and default versions use forward recurrence, binary splitting, rectangular splitting, and an automatic algorithm choice.
Computes \( p f_q(z) \) directly using the defining series, given parameters and argument that are power series. The result is a power series of length \( len \). We require that \( len \) is positive.

An error bound is computed automatically as a function of the number of terms \( n \). If \( n < 0 \), the number of terms is chosen automatically.

If \( \text{regularized} \) is set, the regularized hypergeometric function is computed instead.

### 9.16.4 Asymptotic series

\( U(a, b, z) \) is the confluent hypergeometric function of the second kind with the principal branch cut, and \( U^* = z^a U(a, b, z) \). For details about how error bounds are computed, see Asymptotic series for the confluent hypergeometric function.

```c
void acb_hypgeom_u_asym(const acb_t a, const acb_t b, const acb_t z, slong n, slong prec)
```

Sets \( res \) to \( U^*(a, b, z) \) computed using \( n \) terms of the asymptotic series, with a rigorous bound for the error included in the output. We require \( n \geq 0 \).

```c
int acb_hypgeom_u_use_asym(const acb_t z, slong prec)
```

Heuristically determines whether the asymptotic series can be used to evaluate \( U(a, b, z) \) to \( prec \) accurate bits (assuming that \( a \) and \( b \) are small).

### 9.16.5 Generalized hypergeometric function

```c
void acb_hypgeom_pfq(acb_t res, const acb_srcptr a, slong p, const acb_srcptr b, slong q, const acb_t z, int regularized, slong prec)
```

Computes the generalized hypergeometric function \( p F_q(z) \), or the regularized version if \( \text{regularized} \) is set.

This function automatically delegates to a specialized implementation when the order \( (p, q) \) is one of \( (0,0), (1,0), (0,1), (1,1), (2,1) \). Otherwise, it falls back to direct summation.

While this is a top-level function meant to take care of special cases automatically, it does not generally perform the optimization of deleting parameters that appear in both \( a \) and \( b \). This can be done ahead of time by the user in applications where duplicate parameters are likely to occur.

### 9.16.6 Confluent hypergeometric functions

```c
void acb_hypgeom_u_1f1_series(acb_poly_t res, const acb_poly_t a, const acb_poly_t b, const acb_poly_t z, slong len, slong prec)
```

Computes \( U(a, b, z) \) as a power series truncated to length \( len \), given \( a, b, z \in \mathbb{C}[[x]] \). If \( b[0] \in \mathbb{Z} \), it computes one extra derivative and removes the singularity (it is then assumed that \( b[1] \neq 0 \)). As currently implemented, the output is indeterminate if \( b \) is nonexact and contains an integer.

```c
void acb_hypgeom_u_1f1(acb_t res, const acb_t a, const acb_t b, const acb_t z, slong prec)
```

Computes \( U(a, b, z) \) as a sum of two convergent hypergeometric series. If \( b \in \mathbb{Z} \), it computes the limit value via \( \text{acb_hypgeom_u_1f1_series()} \). As currently implemented, the output is indeterminate if \( b \) is nonexact and contains an integer.

```c
void acb_hypgeom_u(acb_t res, const acb_t a, const acb_t b, const acb_t z, slong prec)
```

Computes \( U(a, b, z) \) using an automatic algorithm choice. The function \( \text{acb_hypgeom_u_asym()} \) is used if \( a \) or \( a - b + 1 \) is a nonpositive integer (in which case the asymptotic series terminates), or if \( z \) is sufficiently large. Otherwise \( \text{acb_hypgeom_u_1f1()} \) is used.
void acb_hypgeom_m_asym(acb_t res, const acb_t a, const acb_t b, const acb_t z, int regularized, slong prec)

void acb_hypgeom_m_1f1(acb_t res, const acb_t a, const acb_t b, const acb_t z, int regularized, slong prec)

void acb_hypgeom_m(acb_t res, const acb_t a, const acb_t b, const acb_t z, int regularized, slong prec)

Computes the confluent hypergeometric function $M(a,b,z) = \frac{1}{\Gamma(b)} \Gamma(a,b,z)$ if regularized is set.

void acb_hypgeom_1f1(acb_t res, const acb_t a, const acb_t b, const acb_t z, int regularized, slong prec)

Alias for acb_hypgeom_m() .

void acb_hypgeom_0f1_asym(acb_t res, const acb_t a, const acb_t z, int regularized, slong prec)

void acb_hypgeom_0f1_direct(acb_t res, const acb_t a, const acb_t z, int regularized, slong prec)

void acb_hypgeom_0f1(acb_t res, const acb_t a, const acb_t z, int regularized, slong prec)

Computes the confluent hypergeometric function $\frac{e^{-z}z}{\Gamma(a)}$ if regularized is set, using asymptotic expansions, direct summation, or an automatic algorithm choice. The asymp version uses the asymptotic expansions of Bessel functions, together with the connection formulas

$$
\frac{e^{-z}z}{\Gamma(a)} = (-z)^{(1-a)/2}J_{a-1}(2\sqrt{-z}) = z^{(1-a)/2}U_{a-1}(2\sqrt{-z}).
$$

The Bessel-$J$ function is used in the left half-plane and the Bessel-$I$ function is used in the right half-plane, to avoid loss of accuracy due to evaluating the square root on the branch cut.

### 9.16.7 Error functions and Fresnel integrals

void acb_hypgeom_erf_propagated_error(mag_t re, mag_t im, const acb_t z)

Sets re and im to upper bounds for the error in the real and imaginary part resulting from approximating the error function of $z$ by the error function evaluated at the midpoint of $z$. Uses the first derivative.

void acb_hypgeom_1f1a(acb_t res, const acb_t z, slong prec)

void acb_hypgeom_1f1b(acb_t res, const acb_t z, slong prec)

void acb_hypgeom_erf_asym(acb_t res, const acb_t z, int complementary, slong prec, slong prec2)

Computes the error function respectively using

$$
erf(z) = \frac{2z}{\sqrt{\pi}} F_1(\frac{1}{2}, \frac{3}{2}, -z^2)
$$

$$
erf(z) = 2\frac{2z}{\sqrt{\pi}} F_1(1, \frac{3}{2}, z^2)
$$

$$
erf(z) = \frac{z}{\sqrt{z^2}} \left(1 - \frac{e^{-z^2}}{\sqrt{\pi}} U(\frac{1}{2}, \frac{1}{2}, z^2)\right) = \frac{z}{\sqrt{z^2}} - \frac{e^{-z^2}}{2\sqrt{\pi}} U^*(\frac{1}{2}, \frac{1}{2}, z^2).
$$

The asymp version takes a second precision to use for the $U$ term. It also takes an extra flag complementary, computing the complementary error function if set.

void acb_hypgeom_erf(acb_t res, const acb_t z, slong prec)

Computes the error function using an automatic algorithm choice. If $z$ is too small to use the asymptotic expansion, a working precision sufficient to circumvent cancellation in the hypergeometric series is determined automatically, and a bound for the propagated error is computed with acb_hypgeom_erf_propagated_error().
void _acb_hypgeom_erf_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)

void acb_hypgeom_erf_series(acb_poly_t res, const acb_poly_t z, slong len, slong prec)
  Computes the error function of the power series z, truncated to length len.

void acb_hypgeom_erfc(acb_t res, const acb_t z, slong prec)
  Computes the complementary error function erfc(z) = 1 - erf(z). This function avoids catastrophic cancellation for large positive z.

void _acb_hypgeom_erfc_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)

void acb_hypgeom_erfc_series(acb_poly_t res, const acb_poly_t z, slong len, slong prec)
  Computes the complementary error function of the power series z, truncated to length len.

void _acb_hypgeom_erfi_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)

void acb_hypgeom_erfi_series(acb_poly_t res, const acb_poly_t z, slong len, slong prec)
  Computes the imaginary error function erfi(z) = -i erf(iz). This is a trivial wrapper of acb_hypgeom_erf().

void _acb_hypgeom_fresnel_series(acb_ptr res1, acb_ptr res2, z, slong zlen, int normalized, slong len, slong prec)

void acb_hypgeom_fresnel_series(acb_poly_t res1, acb_poly_t res2, const acb_poly_t z, slong len, slong prec)
  Sets res1 to the Fresnel sine integral $S(z)$ and res2 to the Fresnel cosine integral $C(z)$. Optionally, just a single function can be computed by passing NULL as the other output variable. The definition $S(z) = \int_0^z \sin(t^2)dt$ is used if normalized is 0, and $S(z) = \int_0^z \sin(\frac{1}{2}\pi t^2)dt$ is used if normalized is 1 (the latter is the Abramowitz & Stegun convention). $C(z)$ is defined analogously.

void _acb_hypgeom_fresnel_series(acb_ptr res1, acb_ptr res2, acb_srcptr z, slong zlen, int normalized, slong len, slong prec)

void acb_hypgeom_fresnel_series(acb_poly_t res1, acb_poly_t res2, const acb_poly_t z, int normalized, slong len, slong prec)
  Sets res1 to the Fresnel sine integral and res2 to the Fresnel cosine integral of the power series z, truncated to length len. Optionally, just a single function can be computed by passing NULL as the other output variable.

### 9.16.8 Bessel functions

void acb_hypgeom_bessel_j_asymp(acb_t res, const acb_t nu, const acb_t z, slong prec)

Computes the Bessel function of the first kind via `acb_hypgeom_u_asymp()`. For all complex $\nu, z$, we have

$$J_\nu(z) = \frac{z^\nu}{2^\nu \pi^\nu \Gamma(\nu + 1)} {}_1 F_1 \left( \nu + \frac{1}{2}, 2\nu + 1, 2iz \right) = A_+ B_+ + A_- B_-$$

where

$$A_\pm = z^\nu (\frac{1}{2})^{\frac{1}{2} - \nu} (\mp iz) \frac{1}{2 + \nu} (2\pi)^{-1/2} = (\pm iz)^{-1/2 - \nu} z^{-\nu} (2\pi)^{-1/2}$$

$$B_\pm = e^{\pm iz} U^\nu(\nu + \frac{1}{2}, 2\nu + 1, \mp 2iz).$$

Nicer representations of the factors $A_\pm$ can be given depending conditionally on the parameters. If $\nu + \frac{1}{2} = n \in \mathbb{Z}$, we have $A_\pm = (\pm i)^n (2\pi z)^{-1/2}$. And if $\text{Re}(z) > 0$, we have $A_\pm = \exp(\mp i(2\nu + 1/4\pi)/(2\pi z))^{-1/2}.$

void acb_hypgeom_bessel_j_0f1(acb_t res, const acb_t nu, const acb_t z, slong prec)

Computes the Bessel function of the first kind from

$$J_\nu(z) = \frac{1}{\Gamma(\nu + 1)} \left(\frac{z}{2}\right)^\nu {}_0 F_1 \left( \nu + 1, -\frac{z^2}{4} \right).$$
void \texttt{acb_hypgeom\_bessel\_j}(\texttt{acb\_t res}, const \texttt{acb\_t nu}, const \texttt{acb\_t z}, \texttt{slong prec})

Computes the Bessel function of the first kind \( J_\nu(z) \) using an automatic algorithm choice.

void \texttt{acb_hypgeom\_bessel\_y}(\texttt{acb\_t res}, const \texttt{acb\_t nu}, const \texttt{acb\_t z}, \texttt{slong prec})

Computes the Bessel function of the second kind \( Y_\nu(z) \) from the formula

\[
Y_\nu(z) = \frac{\cos(\nu \pi) J_\nu(z) - J_{-\nu}(z)}{\sin(\nu \pi)}
\]

unless \( \nu = n \) is an integer in which case the limit value

\[
Y_n(z) = -\frac{2}{\pi} \left( e^{iK_n(iz)} + [\log(i) - \log(z)] J_n(z) \right)
\]

is computed. As currently implemented, the output is indeterminate if \( \nu \) is nonexact and contains an integer.

void \texttt{acb_hypgeom\_bessel\_jy}(\texttt{acb\_t res1}, \texttt{acb\_t res2}, const \texttt{acb\_t nu}, const \texttt{acb\_t z}, \texttt{slong prec})

Sets \texttt{res1} to \( J_\nu(z) \) and \texttt{res2} to \( Y_\nu(z) \), computed simultaneously. From these values, the user can easily construct the Bessel functions of the third kind (Hankel functions) \( H^{(1)}_\nu(z), H^{(2)}_\nu(z) = J_\nu(z) \pm iY_\nu(z) \).

### 9.16.9 Modified Bessel functions

void \texttt{acb_hypgeom\_bessel\_i\_asymp}(\texttt{acb\_t res}, const \texttt{acb\_t nu}, const \texttt{acb\_t z}, \texttt{int scaled}, \texttt{slong prec})

void \texttt{acb_hypgeom\_bessel\_i\_0f1}(\texttt{acb\_t res}, const \texttt{acb\_t nu}, const \texttt{acb\_t z}, \texttt{int scaled}, \texttt{slong prec})

void \texttt{acb_hypgeom\_bessel\_i}(\texttt{acb\_t res}, const \texttt{acb\_t nu}, const \texttt{acb\_t z}, \texttt{slong prec})

void \texttt{acb_hypgeom\_bessel\_i\_scaled}(\texttt{acb\_t res}, const \texttt{acb\_t nu}, const \texttt{acb\_t z}, \texttt{slong prec})

Computes the modified Bessel function of the first kind \( I_\nu(z) = z^\nu (iz)^{-\nu} J_\nu(iz) \) respectively using asymptotic series (see \texttt{acb_hypgeom\_bessel\_j\_asymp()}), the convergent series

\[
I_\nu(z) = \frac{1}{\Gamma(\nu + 1)} \left( \frac{z}{2} \right)^\nu {}_0F_1 \left( \nu + 1, 1 - \frac{z^2}{4} \right),
\]

or an automatic algorithm choice.

The \texttt{scaled} version computes the function \( e^{-z} I_\nu(z) \). The \texttt{asymp} and \texttt{0f1} functions implement both variants and allow choosing with a flag.

void \texttt{acb_hypgeom\_bessel\_k\_asymp}(\texttt{acb\_t res}, const \texttt{acb\_t nu}, const \texttt{acb\_t z}, \texttt{int scaled}, \texttt{slong prec})

Computes the modified Bessel function of the second kind via \texttt{acb_hypgeom\_u\_asymp()}. For all \( \nu \) and all \( z \neq 0 \), we have

\[
K_\nu(z) = \left( \frac{2z}{\pi} \right)^{-1/2} e^{-z} U^*(\nu + \frac{1}{2}, 2\nu + 1, 2z).
\]

If \texttt{scaled} is set, computes the function \( e^z K_\nu(z) \).

void \texttt{acb_hypgeom\_bessel\_k\_0f1\_series}(\texttt{acb\_poly\_t res}, const \texttt{acb\_poly\_t nu}, const \texttt{acb\_poly\_t z}, \texttt{int scaled}, \texttt{slong len}, \texttt{slong prec})

Computes the modified Bessel function of the second kind \( K_\nu(z) \) as a power series truncated to length \texttt{len}, given \( \nu, z \in \mathbb{C}[x] \). Uses the formula

\[
K_\nu(z) = \frac{1}{2} \frac{\pi}{\sin(\pi \nu)} \left[ \left( \frac{z}{2} \right)^{-\nu} {}_0\tilde{F}_1 \left( 1 - \nu, \frac{z^2}{4} \right) - \left( \frac{z}{2} \right)^\nu {}_0\tilde{F}_1 \left( 1 + \nu, \frac{z^2}{4} \right) \right].
\]

If \( \nu[0] \in \mathbb{Z} \), it computes one extra derivative and removes the singularity (it is then assumed that \( \nu[1] \neq 0 \)). As currently implemented, the output is indeterminate if \( \nu[0] \) is nonexact and contains an integer.

If \texttt{scaled} is set, computes the function \( e^z K_\nu(z) \).
void \texttt{acb_hypgeom_bessel\_k\_0f1}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{int scaled}, \texttt{slong prec})

Computes the modified Bessel function of the second kind from

$$K_\nu(z) = \frac{1}{2} \left[ \left( \frac{z}{2} \right)^{-\nu} \Gamma(\nu) \mathcal{F}_1 \left( 1 - \nu, \frac{-z^2}{4} \right) - \left( \frac{z}{2} \right)^{\nu} \frac{\pi}{\nu \sin(\pi \nu) \Gamma(\nu)} \mathcal{F}_1 \left( \nu + 1, \frac{-z^2}{4} \right) \right]$$

if $\nu \notin \mathbb{Z}$. If $\nu \in \mathbb{Z}$, it computes the limit value via \texttt{acb\_hypgeom\_bessel\_k\_0f1\_series()}. As currently implemented, the output is indeterminate if $\nu$ is nonexact and contains an integer.

If \texttt{scaled} is set, computes the function $e^z K_\nu(z)$.

void \texttt{acb\_hypgeom\_bessel\_k}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes the modified Bessel function of the second kind $K_\nu(z)$ using an automatic algorithm choice.

void \texttt{acb\_hypgeom\_bessel\_k\_scaled}(\texttt{acb\_t res}, \texttt{const acb\_t nu}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes the function $e^z K_\nu(z)$.

### 9.16.10 Airy functions

The Airy functions are linearly independent solutions of the differential equation $y'' - zy = 0$. All solutions are entire functions. The standard solutions are denoted $\text{Ai}(z), \text{Bi}(z)$. For negative $z$, both functions are oscillatory. For positive $z$, the first function decreases exponentially while the second increases exponentially.

The Airy functions can be expressed in terms of Bessel functions of fractional order, but this is inconvenient since such formulas only hold piecewise (due to the Stokes phenomenon). Computation of the Airy functions can also be optimized more than Bessel functions in general. We therefore provide a dedicated interface for evaluating Airy functions.

The following methods optionally compute ($\text{Ai}(z), \text{Ai}'(z), \text{Bi}(z), \text{Bi}'(z)$) simultaneously. Any of the four function values can be omitted by passing \texttt{NULL} for the unwanted output variables, speeding up the evaluation.

void \texttt{acb\_hypgeom\_airy\_direct}(\texttt{acb\_t ai}, \texttt{acb\_t ai\_prime}, \texttt{acb\_t bi}, \texttt{acb\_t bi\_prime}, \texttt{const acb\_t z}, \texttt{slong n}, \texttt{slong prec})

Computes the Airy functions using direct series expansions truncated at $n$ terms. Error bounds are included in the output.

void \texttt{acb\_hypgeom\_airy\_asym}(\texttt{acb\_t ai}, \texttt{acb\_t ai\_prime}, \texttt{acb\_t bi}, \texttt{acb\_t bi\_prime}, \texttt{const acb\_t z}, \texttt{slong n}, \texttt{slong prec})

Computes the Airy functions using asymptotic expansions truncated at $n$ terms. Error bounds are included in the output. For details about how the error bounds are computed, see \textit{Asymptotic series for Airy functions}.

void \texttt{acb\_hypgeom\_airy\_bound}(\texttt{mag\_t ai}, \texttt{mag\_t ai\_prime}, \texttt{mag\_t bi}, \texttt{mag\_t bi\_prime}, \texttt{const acb\_t z})

Computes bounds for the Airy functions using first-order asymptotic expansions together with error bounds. This function uses some shortcuts to make it slightly faster than calling \texttt{acb\_hypgeom\_airy\_asym()} with $n = 1$.

void \texttt{acb\_hypgeom\_airy}(\texttt{acb\_t ai}, \texttt{acb\_t ai\_prime}, \texttt{acb\_t bi}, \texttt{acb\_t bi\_prime}, \texttt{const acb\_t z}, \texttt{slong prec})

Computes Airy functions using an automatic algorithm choice.

We use \texttt{acb\_hypgeom\_airy\_asym()} whenever this gives full accuracy and \texttt{acb\_hypgeom\_airy\_direct()} otherwise. In the latter case, we first use hardware double precision arithmetic to determine an accurate estimate of the working precision needed to compute the Airy functions accurately for given $z$. This estimate is obtained by comparing the leading-order asymptotic estimate of the Airy functions with the magnitude of the largest term in the power series. The estimate is generic in the sense that it does not take into account
vanishing near the roots of the functions. We subsequently evaluate the power series at the midpoint of \(z\) and bound the propagated error using derivatives. Derivatives are bounded using \(\texttt{acb_hypgeom_airy_bound()}\).

```c
void acb_hypgeom_airy_jet(acb_ptr ai, acb_ptr bi, const acb_t z, slong len, slong prec)
```

Writes to \(ai\) and \(bi\) the respective Taylor expansions of the Airy functions at the point \(z\), truncated to length \(len\). Either of the outputs can be \(NULL\) to avoid computing that function. The variable \(z\) is not allowed to be aliased with the outputs. To simplify the implementation, this method does not compute the series expansions of the primed versions directly; these are easily obtained by computing one extra coefficient and differentiating the output with \(\texttt{acb_poly_derivative()}\).

```c
void _acb_hypgeom_airy_series(acb_ptr ai, acb_ptr bi, const acb_t z, slong zlen, slong len, slong prec)
```

Computes the Airy functions evaluated at the power series \(z\), truncated to length \(len\). As with the other Airy methods, any of the outputs can be \(NULL\).

### 9.16.11 Coulomb wave functions

Coulomb wave functions are solutions of the Coulomb wave equation

\[
y'' + \left( 1 - \frac{2\eta}{z} - \frac{\ell(\ell + 1)}{z^2} \right)y = 0
\]

which is the radial Schrödinger equation for a charged particle in a Coulomb potential \(1/z\), where \(\ell\) is the orbital angular momentum and \(\eta\) is the Sommerfeld parameter. The standard solutions are named \(F_\ell(\eta, z)\) (regular at the origin \(z = 0\)) and \(G_\ell(\eta, z)\) (irregular at the origin). The irregular solutions \(H^{\pm}_\ell(\eta, z) = G_\ell(\eta, z) \pm iF_\ell(\eta, z)\) are also used.

Coulomb wave functions are special cases of confluent hypergeometric functions. The normalization constants and connection formulas are discussed in [DYF1999], [Gas2018], [Mic2007] and chapter 33 in [NIST2012]. In this implementation, we define the analytic continuations of all the functions so that the branch cut with respect to \(z\) is placed on the negative real axis. Precise definitions are given in [http://fungrim.org/topic/Coulomb_wave_functions/](http://fungrim.org/topic/Coulomb_wave_functions/)

The following methods optionally compute \(F_\ell(\eta, z), G_\ell(\eta, z), H^+_\ell(\eta, z), H^-_\ell(\eta, z)\) simultaneously. Any of the four function values can be omitted by passing \(NULL\) for the unwanted output variables. The redundant functions \(H^\pm\) are provided explicitly since taking the linear combination of \(F\) and \(G\) suffers from cancellation in parts of the complex plane.

```c
void acb_hypgeom_coulomb(acb_t F, acb_t G, acb_t Hpos, acb_t Hneg, const acb_t l, const acb_t eta, const acb_t z, slong prec)
```

Writes to \(F\), \(G\), \(Hpos\), \(Hneg\) the values of the respective Coulomb wave functions. Any of the outputs can be \(NULL\).

```c
void acb_hypgeom_coulomb_jet(acb_ptr F, acb_ptr G, acb_ptr Hpos, acb_ptr Hneg, const acb_t l, const acb_t eta, const acb_t z, slong len, slong prec)
```

Writes to \(F\), \(G\), \(Hpos\), \(Hneg\) the respective Taylor expansions of the Coulomb wave functions at the point \(z\), truncated to length \(len\). Any of the outputs can be \(NULL\).

```c
void _acb_hypgeom_coulomb_series(acb_ptr F, acb_ptr G, acb_ptr Hpos, acb_ptr Hneg, const acb_t l, const acb_t eta, const acb_t z, slong zlen, slong len, slong prec)
```

Computes the Coulomb wave functions evaluated at the power series \(z\), truncated to length \(len\). Any of the outputs can be \(NULL\).
9.16.12 Incomplete gamma and beta functions

void \texttt{acb_hypgeom_gamma_upper}\_{\texttt{asymp}}(\texttt{acb}\_t \texttt{res}, \texttt{const} \texttt{acb}\_t \texttt{s}, \texttt{const} \texttt{acb}\_t \texttt{z}, \texttt{int} \texttt{regularized}, \texttt{slong} \texttt{prec})

void \texttt{acb_hypgeom_gamma_upper}\_{\texttt{ifia}}(\texttt{acb}\_t \texttt{res}, \texttt{const} \texttt{acb}\_t \texttt{s}, \texttt{const} \texttt{acb}\_t \texttt{z}, \texttt{int} \texttt{regularized}, \texttt{slong} \texttt{prec})

void \texttt{acb_hypgeom_gamma_upper}\_{\texttt{ifib}}(\texttt{acb}\_t \texttt{res}, \texttt{const} \texttt{acb}\_t \texttt{s}, \texttt{const} \texttt{acb}\_t \texttt{z}, \texttt{int} \texttt{regularized}, \texttt{slong} \texttt{prec})

void \texttt{acb_hypgeom_gamma_upper}\_{\texttt{singular}}(\texttt{acb}\_t \texttt{res}, \texttt{slong} \texttt{s}, \texttt{const} \texttt{acb}\_t \texttt{z}, \texttt{int} \texttt{regularized}, \texttt{slong} \texttt{prec})

void \texttt{acb_hypgeom_gamma_upper}(\texttt{acb}\_t \texttt{res}, \texttt{const} \texttt{acb}\_t \texttt{s}, \texttt{const} \texttt{acb}\_t \texttt{z}, \texttt{int} \texttt{regularized}, \texttt{slong} \texttt{prec})

If \texttt{regularized} is 0, computes the upper incomplete gamma function \(\Gamma(s, z)\).

If \texttt{regularized} is 1, computes the regularized upper incomplete gamma function \(\Gamma(s, z)/\Gamma(s)\).

If \texttt{regularized} is 2, computes the generalized exponential integral \(z^{-s}\Gamma(s, z) = E_{1-s}(z)\) instead (this option is mainly intended for internal use; \texttt{acb_hypgeom_expint()} is the intended interface for computing the exponential integral).

The different methods respectively implement the formulas

\[
\Gamma(s, z) = e^{-z}U(1 - s, 1 - s, z)
\]

\[
\Gamma(s, z) = \Gamma(s) - \frac{z^s}{s} F_1(s, s + 1, -z)
\]

\[
\Gamma(s, z) = \Gamma(s) - \frac{z^s e^{-z}}{s} F_1(1, s + 1, z)
\]

\[
\Gamma(s, z) = \frac{(-1)^n}{n!}(\psi(n + 1) - \log(z)) + \frac{(-1)^n}{(n + 1)!} z^2 F_2(1, 1, 2, 2 + n, -z) - z^{-n} \sum_{k=0}^{n-1} \frac{(-z)^k}{(k - n)k!}, \quad n = -s \in \mathbb{Z}_{\geq 0}
\]

and an automatic algorithm choice. The automatic version also handles other special input such as \(z = 0\) and \(s = 1, 2, 3\). The \texttt{singular} version evaluates the finite sum directly and therefore assumes that \(s\) is not too large.

void \texttt{acb_hypgeom_gamma_upper}\_{\texttt{series}}(\texttt{acb}\_\texttt{ptr} \texttt{res}, \texttt{const} \texttt{acb}\_\texttt{t} \texttt{s}, \texttt{acb}\_\texttt{srcptr} \texttt{z}, \texttt{slong} \texttt{zlen}, \texttt{int} \texttt{regularized}, \texttt{slong} \texttt{n}, \texttt{slong} \texttt{prec})

void \texttt{acb_hypgeom_gamma_upper}\_{\texttt{series}}(\texttt{acb}\_\texttt{poly}\_\texttt{t} \texttt{res}, \texttt{const} \texttt{acb}\_\texttt{t} \texttt{s}, \texttt{acb}\_\texttt{poly}\_\texttt{t} \texttt{z}, \texttt{int} \texttt{regularized}, \texttt{slong} \texttt{n}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to an upper incomplete gamma function where \(s\) is a constant and \(z\) is a power series, truncated to length \(n\). The \texttt{regularized} argument has the same interpretation as in \texttt{acb_hypgeom_gamma_upper()}.

void \texttt{acb_hypgeom_gamma_lower}(\texttt{acb}\_\texttt{t} \texttt{res}, \texttt{const} \texttt{acb}\_\texttt{t} \texttt{s}, \texttt{const} \texttt{acb}\_\texttt{t} \texttt{z}, \texttt{int} \texttt{regularized}, \texttt{slong} \texttt{prec})

If \texttt{regularized} is 0, computes the lower incomplete gamma function \(\gamma(s, z) = \frac{1}{z} F_1(s, s + 1, -z)\).

If \texttt{regularized} is 1, computes the regularized lower incomplete gamma function \(\gamma(s, z)/\Gamma(s)\).

If \texttt{regularized} is 2, computes a further regularized lower incomplete gamma function \(\gamma^*(s, z) = z^{-s} P(s, z)\).

void \texttt{acb_hypgeom_gamma_lower}\_{\texttt{series}}(\texttt{acb}\_\texttt{ptr} \texttt{res}, \texttt{const} \texttt{acb}\_\texttt{t} \texttt{s}, \texttt{acb}\_\texttt{srcptr} \texttt{z}, \texttt{slong} \texttt{zlen}, \texttt{int} \texttt{regularized}, \texttt{slong} \texttt{n}, \texttt{slong} \texttt{prec})
void *acb_hypgeom_gamma_lower_series(acb_poly_t res, const acb_t s, const acb_poly_t z, int regularized, slong n, slong prec)

Sets res to an lower incomplete gamma function where s is a constant and z is a power series, truncated to length n. The regularized argument has the same interpretation as in *acb_hypgeom_gamma_lower()*. 

void *acb_hypgeom_beta_lower(acb_t res, const acb_t a, const acb_t b, const acb_t z, int regularized, slong prec)

Computes the (lower) incomplete beta function, defined by

\[ B(a, b; z) = \int_0^z t^{a-1}(1-t)^{b-1} \, dt, \]

where a and b are constants and z is a power series, optionally truncated to length n. The underscore method requires positive lengths and does not support aliasing.

Note that both functions with this definition are undefined for nonpositive integer a, and I is undefined for nonpositive integer a + b.

void *_acb_hypgeom_beta_lower_series(acb_ptr res, const acb_t a, const acb_t b, acb_srcptr z, slong zlen, int regularized, slong n, slong prec)

void *acb_hypgeom_beta_lower_series(acb_poly_t res, const acb_t a, const acb_t b, const acb_poly_t z, int regularized, slong n, slong prec)

Sets res to the lower incomplete beta function B(a, b; z) (optionally the regularized version I(a, b; z)) where a and b are constants and z is a power series, truncating the result to length n. The underscore method requires positive lengths and does not support aliasing.

### 9.16.13 Exponential and trigonometric integrals

The branch cut conventions of the following functions match Mathematica.

void *acb_hypgeom_expint(acb_t res, const acb_t s, const acb_t z, slong prec)

Computes the generalized exponential integral E_s(z). This is a trivial wrapper of *acb_hypgeom_gamma_upper()*. 

void *acb_hypgeom_ei_asym(acb_t res, const acb_t z, slong prec)

void *acb_hypgeom_ei_2f2(acb_t res, const acb_t z, slong prec)

void *acb_hypgeom_ei(acb_t res, const acb_t z, slong prec)

Computes the exponential integral Ei(z), respectively using

\[ Ei(z) = -e^z U(1, 1, -z) - \log(-z) + \frac{1}{2} \left( \log(z) - \log \left( \frac{1}{z} \right) \right) \]

\[ Ei(z) = z_2F_2(1, 1; 2, 2; z) + \gamma + \frac{1}{2} \left( \log(z) - \log \left( \frac{1}{z} \right) \right) \]

and an automatic algorithm choice.

void _acb_hypgeom_ei_series(acb_ptr res, acb_srcptr z, slong zlen, slong len, slong prec)

void *acb_hypgeom_ei_series(acb_poly_t res, const acb_poly_t z, slong len, slong prec)

Computes the exponential integral of the power series z, truncated to length len.

void *acb_hypgeom_epi_asym(acb_t res, const acb_t z, slong prec)
void \texttt{acb_hypgeom\_si\_1f2}(\texttt{acb\_t} \texttt{res}, \texttt{const acb\_t} \texttt{z}, \texttt{slong prec})

void \texttt{acb_hypgeom\_si}(\texttt{acb\_t} \texttt{res}, \texttt{const acb\_t} \texttt{z}, \texttt{slong prec})

Computes the sine integral $\text{Si}(z)$, respectively using

$$\text{Si}(z) = \frac{i}{2} [e^{iz}U(1,1,-iz) - e^{-iz}U(1,1,iz) + \log(-iz) - \log(iz)]$$

and an automatic algorithm choice.

void \_acb\_hypgeom\_si\_series(\texttt{acb\_ptr} \texttt{res}, \texttt{acb\_sresptr} \texttt{z}, \texttt{slong zlen}, \texttt{slong} \texttt{len}, \texttt{slong} \texttt{prec})

void \texttt{acb_hypgeom\_si\_series}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{const acb\_poly\_t} \texttt{z}, \texttt{slong len}, \texttt{slong prec})

Computes the sine integral of the power series $z$, truncated to length \texttt{len}.

void \texttt{acb_hypgeom\_ci\_asym}(\texttt{acb\_t} \texttt{res}, \texttt{const acb\_t} \texttt{z}, \texttt{slong prec})

void \texttt{acb_hypgeom\_ci\_2f3}(\texttt{acb\_t} \texttt{res}, \texttt{const acb\_t} \texttt{z}, \texttt{slong prec})

void \texttt{acb_hypgeom\_ci}(\texttt{acb\_t} \texttt{res}, \texttt{const acb\_t} \texttt{z}, \texttt{slong prec})

Computes the cosine integral $\text{Ci}(z)$, respectively using

$$\text{Ci}(z) = \log(z) - \frac{1}{2} [e^{iz}U(1,1,-iz) + e^{-iz}U(1,1,iz) + \log(-iz) + \log(iz)]$$

and an automatic algorithm choice.

void \_acb\_hypgeom\_ci\_series(\texttt{acb\_ptr} \texttt{res}, \texttt{acb\_sresptr} \texttt{z}, \texttt{slong zlen}, \texttt{slong} \texttt{len}, \texttt{slong} \texttt{prec})

void \texttt{acb_hypgeom\_ci\_series}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{const acb\_poly\_t} \texttt{z}, \texttt{slong len}, \texttt{slong prec})

Computes the cosine integral of the power series $z$, truncated to length \texttt{len}.

void \texttt{acb_hypgeom\_shi}(\texttt{acb\_t} \texttt{res}, \texttt{const acb\_t} \texttt{z}, \texttt{slong prec})

Computes the hyperbolic sine integral $\text{Shi}(z) = -i\text{Si}(iz)$. This is a trivial wrapper of \texttt{acb\_hypgeom\_si}. 

void \_acb\_hypgeom\_shi\_series(\texttt{acb\_ptr} \texttt{res}, \texttt{acb\_sresptr} \texttt{z}, \texttt{slong zlen}, \texttt{slong} \texttt{len}, \texttt{slong} \texttt{prec})

void \texttt{acb_hypgeom\_shi\_series}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{const acb\_poly\_t} \texttt{z}, \texttt{slong len}, \texttt{slong prec})

Computes the hyperbolic sine integral of the power series $z$, truncated to length \texttt{len}.

void \texttt{acb_hypgeom\_chi\_asym}(\texttt{acb\_t} \texttt{res}, \texttt{const acb\_t} \texttt{z}, \texttt{slong prec})

void \texttt{acb_hypgeom\_chi\_2f3}(\texttt{acb\_t} \texttt{res}, \texttt{const acb\_t} \texttt{z}, \texttt{slong prec})

void \texttt{acb_hypgeom\_chi}(\texttt{acb\_t} \texttt{res}, \texttt{const acb\_t} \texttt{z}, \texttt{slong prec})

Computes the hyperbolic cosine integral $\text{Chi}(z)$, respectively using

$$\text{Chi}(z) = -\frac{1}{2} [e^{iz}U(1,1,-z) + e^{-iz}U(1,1,z) + \log(-z) - \log(z)]$$

and an automatic algorithm choice.

void \_acb\_hypgeom\_chi\_series(\texttt{acb\_ptr} \texttt{res}, \texttt{acb\_sresptr} \texttt{z}, \texttt{slong zlen}, \texttt{slong} \texttt{len}, \texttt{slong} \texttt{prec})

void \texttt{acb_hypgeom\_chi\_series}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{const acb\_poly\_t} \texttt{z}, \texttt{slong len}, \texttt{slong prec})

Computes the hyperbolic cosine integral of the power series $z$, truncated to length \texttt{len}. 

650 Chapter 9. Real and complex numbers
void acb_hypgeom_li(acb_t res, const acb_t z, int offset, slong prec)
  If offset is zero, computes the logarithmic integral li(z) = Ei(log(z)).
  If offset is nonzero, computes the offset logarithmic integral Li(z) = li(z) - li(2).

void acb_hypgeom_2f1_transform(acb_ptr res, acb_srcptr z, slong zlen, int offset, slong len, slong prec)
void acb_hypgeom_2f1_direct(acb_poly_t res, const acb_poly_t z, int offset, slong len, slong prec)
  Computes the logarithmic integral (optionally the offset version) of the power series z, truncated to length len.

9.16.14 Gauss hypergeometric function

The following methods compute the Gauss hypergeometric function

\[ F(z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k k!} z^k \]

or the regularized version \( F(z) = F(a, b, c, z) = 2F_1(a, b, c, z) / \Gamma(c) \) if the flag regularized is set.

void acb_hypgeom_2f1_continuation(acb_t res0, acb_t res1, const acb_t a, const acb_t b, const acb_t c, const acb_t z0, const acb_t z1, const acb_t f0, const acb_t f1, slong prec)
  Given \( F(z_0), F'(z_0) \) in \( f0, f1 \), sets \( res0 \) and \( res1 \) to \( F(z_1), F'(z_1) \) by integrating the hypergeometric differential equation along a straight-line path. The evaluation points should be well-isolated from the singular points 0 and 1.

void acb_hypgeom_2f1_series_direct(acb_poly_t res, const acb_poly_t a, const acb_poly_t b, const acb_poly_t c, const acb_poly_t z, int regularized, slong len, slong prec)
  Computes \( F(z) \) of the given power series truncated to length \( len \), using direct summation of the hypergeometric series.

void acb_hypgeom_2f1_direct(acb_t res, const acb_t a, const acb_t b, const acb_t c, const acb_t z, int regularized, slong prec)
  Computes \( F(z) \) using direct summation of the hypergeometric series.

void acb_hypgeom_2f1_transform(acb_t res, const acb_t a, const acb_t b, const acb_t c, const acb_t z, int flags, int which, slong prec)

void acb_hypgeom_2f1_transform_limit(acb_t res, const acb_t a, const acb_t b, const acb_t c, const acb_t z, int regularized, int which, slong prec)
  Computes \( F(z) \) using an argument transformation determined by the flag which. Legal values are
  1 for \( z/(z - 1) \), 2 for \( 1/z \), 3 for \( 1/(1 - z) \), 4 for \( 1 - z \), and 5 for \( 1 - 1/z \).

  The transform_limit version assumes that which is not 1. If which is 2 or 3, it assumes that \( b - a \)
  represents an exact integer. If which is 4 or 5, it assumes that \( c - a - b \) represents an exact integer.
  In these cases, it computes the correct limit value.

  See acb_hypgeom_2f1() for the meaning of flags.

void acb_hypgeom_2f1_corner(acb_t res, const acb_t a, const acb_t b, const acb_t c, const acb_t z, int regularized, slong prec)
  Computes \( F(z) \) near the corner cases \( \exp(\pm \pi i \sqrt{3}) \) by analytic continuation.

int acb_hypgeom_2f1_choose(const acb_t z)
  Chooses a method to compute the function based on the location of \( z \) in the complex plane. If the return value is 0, direct summation should be used. If the return value is 1 to 5, the transformation with this index in acb_hypgeom_2f1_transform() should be used. If the return value is 6, the corner case algorithm should be used.
void \texttt{acb_hypgeom\_2f1}(\texttt{acb\_t} res, const \texttt{acb\_t} a, const \texttt{acb\_t} b, const \texttt{acb\_t} c, const \texttt{acb\_t} z, int flags, \texttt{slong} prec)

Computes $F(z)$ or $\mathbf{F}(z)$ using an automatic algorithm choice.

The following bit fields can be set in \texttt{flags}:

- \texttt{ACB\_HYPERGEOM\_2F1\_REGULARIZED} - computes the regularized hypergeometric function $\mathbf{F}(z)$. Setting \texttt{flags} to 1 is the same as just toggling this option.
- \texttt{ACB\_HYPERGEOM\_2F1\_AB} - $a - b$ is an integer.
- \texttt{ACB\_HYPERGEOM\_2F1\_ABC} - $a + b - c$ is an integer.
- \texttt{ACB\_HYPERGEOM\_2F1\_AC} - $a - c$ is an integer.
- \texttt{ACB\_HYPERGEOM\_2F1\_BC} - $b - c$ is an integer.

The last four flags can be set to indicate that the respective parameter differences are known to represent exact integers, even if the input intervals are inexact. This allows the correct limits to be evaluated when applying transformation formulas. For example, to evaluate $2\mathbf{F}_1(\sqrt{2}, 1/2, \sqrt{2} + 3/2, 9/10)$, the \texttt{ABC} flag should be set. If not set, the result will be an indeterminate interval due to internally dividing by an interval containing zero. If the parameters are exact floating-point numbers (including exact integers or half-integers), then the limits are computed automatically, and setting these flags is unnecessary.

Currently, only the \texttt{AB} and \texttt{ABC} flags are used this way; the \texttt{AC} and \texttt{BC} flags might be used in the future.

### 9.16.15 Orthogonal polynomials and functions

void \texttt{acb_hypgeom\_chebyshev\_t}(\texttt{acb\_t} res, const \texttt{acb\_t} n, const \texttt{acb\_t} z, \texttt{slong} prec)

void \texttt{acb_hypgeom\_chebyshev\_u}(\texttt{acb\_t} res, const \texttt{acb\_t} n, const \texttt{acb\_t} z, \texttt{slong} prec)

Computes the Chebyshev polynomial (or Chebyshev function) of first or second kind

\[
T_n(z) = 2\mathbf{F}_1(-n, n, 1, \frac{1 - z}{2})
\]

\[
U_n(z) = (n + 1)\mathbf{F}_1(-n, n + 2, \frac{3}{2}, \frac{1 - z}{2}).
\]

The hypergeometric series definitions are only used for computation near the point 1. In general, trigonometric representations are used. For word-size integer $n$, \texttt{acb\_chebyshev\_t\_ui()} and \texttt{acb\_chebyshev\_u\_ui()} are called.

void \texttt{acb_hypgeom\_jacobi\_p}(\texttt{acb\_t} res, const \texttt{acb\_t} n, const \texttt{acb\_t} a, const \texttt{acb\_t} b, const \texttt{acb\_t} z, \texttt{slong} prec)

Computes the Jacobi polynomial (or Jacobi function)

\[
P_n^{(a,b)}(z) = \frac{(a + 1)_n}{\Gamma(n + 1)^2} \mathbf{F}_1(-n, n + a + b + 1, a + 1, \frac{1 - z}{2}).
\]

For nonnegative integer $n$, this is a polynomial in $a$, $b$ and $z$, even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

void \texttt{acb_hypgeom\_gegenbauer\_c}(\texttt{acb\_t} res, const \texttt{acb\_t} n, const \texttt{acb\_t} m, const \texttt{acb\_t} z, \texttt{slong} prec)

Computes the Gegenbauer polynomial (or Gegenbauer function)

\[
C_n^m(z) = \frac{(2m)_n}{\Gamma(n + 1)} 2\mathbf{F}_1(-n, 2m + n, m + \frac{1}{2}, \frac{1 - z}{2}).
\]

For nonnegative integer $n$, this is a polynomial in $m$ and $z$, even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.
void acb_hypgeom_laguerre_l(acb_t res, const acb_t n, const acb_t m, const acb_t z, slong prec)
    Computes the Laguerre polynomial (or Laguerre function)
    \[ L_n^m(z) = \frac{(m + 1)_n}{(n + 1)_1} {}_1F_1(-n, m + 1, z). \]

For nonnegative integer \( n \), this is a polynomial in \( m \) and \( z \), even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

There are at least two incompatible ways to define the Laguerre function when \( n \) is a negative integer. One possibility when \( m = 0 \) is to define \( L_n^0(z) = e^z L_n^{(-1)}(-z) \). Another possibility is to cover this case with the recurrence relation \( L_n^m(z) + L_n^{m-1}(z) = L_n^m(z) \). Currently, we leave this case undefined (returning indeterminate).

void acb_hypgeom_hermite_h(acb_t res, const acb_t n, const acb_t z, slong prec)
    Computes the Hermite polynomial (or Hermite function)
    \[ H_n(z) = 2^n \sqrt{\pi} \left( \frac{1}{\Gamma((n-1)/2)} {}_1F_1\left(-\frac{n}{2}, 1, z^2\right) - \frac{2z}{\Gamma(-n/2)} {}_1F_1\left(\frac{1-n}{2}, \frac{3}{2}, z^2\right) \right). \]

void acb_hypgeom_legendre_p(acb_t res, const acb_t n, const acb_t m, const acb_t z, int type, slong prec)
    Sets \( \text{res} \) to the associated Legendre function of the first kind evaluated for degree \( n \), order \( m \), and argument \( z \). When \( m \) is zero, this reduces to the Legendre polynomial \( P_n(z) \).

Many different branch cut conventions appear in the literature. If \( \text{type} \) is 0, the version
    \[ P_n^m(z) = \frac{(1 + z)^{m/2}}{(1 - z)^{m/2}} {}_2F_1\left(-n, n + 1, 1 - m, \frac{1 - z}{2}\right) \]
    is computed, and if \( \text{type} \) is 1, the alternative version
    \[ P_n^m(z) = \frac{(z + 1)^{m/2}}{(z - 1)^{m/2}} {}_2F_1\left(-n, n + 1, 1 - m, \frac{1 - z}{2}\right). \]
    is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in \textit{Mathematica} and \textit{mpmath}.

void acb_hypgeom_legendre_q(acb_t res, const acb_t n, const acb_t m, const acb_t z, int type, slong prec)
    Sets \( \text{res} \) to the associated Legendre function of the second kind evaluated for degree \( n \), order \( m \), and argument \( z \). When \( m \) is zero, this reduces to the Legendre function \( Q_n(z) \).

Many different branch cut conventions appear in the literature. If \( \text{type} \) is 0, the version
    \[ Q_n^m(z) = \frac{\pi}{2 \sin(\pi m)} \left( \cos(\pi m) P_n^m(z) - \frac{\Gamma(1 + m + n)}{\Gamma(1 - m + n)} P_{n-1}^m(z) \right) \]
    is computed, and if \( \text{type} \) is 1, the alternative version
    \[ Q_n^m(z) = \frac{\pi}{2 \sin(\pi m)} e^{\pi i m} \left( P_n^m(z) - \frac{\Gamma(1 + m + n)}{\Gamma(1 - m + n)} P_{n-1}^m(z) \right) \]
    is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in \textit{Mathematica} and \textit{mpmath}.

When \( m \) is an integer, either expression is interpreted as a limit. We make use of the connection formulas \[ WQ3a \], \[ WQ3b \] and \[ WQ3c \] to allow computing the function even in the limiting case. (The formula \[ WQ3d \] would be useful, but is incorrect in the lower half plane.)
The dilogarithm function is given by

\[ \text{Li}_2(z) = -\int_0^z \frac{\log(1-t)}{t} \, dt = z \log(1-z) - \frac{1}{2} z^2. \]

For nonnegative integer \( n \) and \( m \), uses recurrence relations to evaluate \((1 - z^2)^{-m/2} P_m^m(z)\) which is a polynomial in \( z \).

void \texttt{acb_hypgeom_dilog} (\texttt{acb_t} res, \texttt{slong} n, \texttt{slong} m, \texttt{const acb_t} z, \texttt{slong} prec)

Computes the dilogarithm for \( z \) close to 0 using the hypergeometric series (effective only when \(|z| \ll 1\)).

void \texttt{acb_hypgeom_dilog_zero} (\texttt{acb_t} res, \texttt{const acb_t} z, \texttt{slong} prec)

Computes the dilogarithm using a default algorithm choice.

void \texttt{acb_hypgeom_dilog_transform} (\texttt{acb_t} res, \texttt{const acb_t} z, \texttt{int} algorithm, \texttt{slong} prec)

Computes the dilogarithm by applying one of the transformations \( 1/z, 1 - z, z/(z - 1), 1/(1 - z) \), indexed by \texttt{algorithm} from 1 to 4, and calling \texttt{acb_hypgeom_dilog_zero()} with the reduced variable. Alternatively, for \texttt{algorithm} between 5 and 7, starts from the respective point \( \pm i, (1 \pm i)/2, (1 \pm i)/2 \) (with the sign chosen according to the midpoint of \( z \)) and computes the dilogarithm by the bit-burst method.

9.16.16 Dilogarithm

The dilogarithm function is given by \( \text{Li}_2(z) = -\int_0^z \frac{\log(1-t)}{t} \, dt = z \log(1-z) - \frac{1}{2} z^2 \).

void \texttt{acb_hypgeom_dilog_continuation} (\texttt{acb_t} res, \texttt{const acb_t} a, \texttt{const acb_t} z, \texttt{slong} prec)

Computes \( \text{Li}_2(z) - \text{Li}_2(a) \) using Taylor expansion at \( a \). Binary splitting is used. Both \( a \) and \( z \) should be well isolated from the points 0 and 1, except that \( a \) may be exactly 0. If the straight line path from \( a \) to \( b \) crosses the branch cut, this method provides continuous analytic continuation instead of computing the principal branch.

void \texttt{acb_hypgeom_dilog_bitburst} (\texttt{acb_t} res, \texttt{acb_t} z0, \texttt{const acb_t} z, \texttt{slong} prec)

Sets \( z0 \) to a point with short bit expansion close to \( z \) and sets \texttt{res} to \( \text{Li}_2(z) - \text{Li}_2(z0) \), computed using the bit-burst algorithm.

void \texttt{acb_hypgeom_dilog} (\texttt{acb_t} res, \texttt{const acb_t} z, \texttt{slong} prec)

Computes the dilogarithm using a default algorithm choice.
9.17 arb_hypgeom.h – hypergeometric functions of real variables

See arb_hypgeom.h – hypergeometric functions of complex variables for the general implementation of hypergeometric functions.

For convenience, this module provides versions of the same functions for real variables represented using arb_t and arb_poly_t. Most methods are simple wrappers around the complex versions, but some of the functions in this module have been further optimized specifically for real variables.

This module also provides certain functions exclusive to real variables, such as functions for computing real roots of common special functions.

9.17.1 Rising factorials

void _arb_hypgeom_rising_coeffs_1(ulong *c, ulong k, slong n)
void _arb_hypgeom_rising_coeffs_2(ulong *c, ulong k, slong n)
void _arb_hypgeom_rising_coeffs_fmpz(fmpz *c, ulong k, slong n)
Sets c to the coefficients of the rising factorial polynomial \((X + k)^n\). The 1 and 2 versions respectively compute single-word and double-word coefficients, without checking for overflow, while the fmpz version allows arbitrarily large coefficients. These functions are mostly intended for internal use; the fmpz version does not use an asymptotically fast algorithm. The degree n must be at least 2.

void arb_hypgeom_rising_ui_forward(arb_t res, const arb_t x, ulong n, slong prec)
void arb_hypgeom_rising_ui_bs(arb_t res, const arb_t x, ulong n, slong prec)
void arb_hypgeom_rising_ui_rs(arb_t res, const arb_t x, ulong n, ulong m, slong prec)
void arb_hypgeom_rising_ui_rec(arb_t res, const arb_t x, const arb_t n, slong prec)
void arb_hypgeom_rising_ui(arb_t res, const arb_t x, ulong n, slong prec)
void arb_hypgeom_rising(arb_t res, const arb_t x, const arb_t n, slong prec)
Computes the rising factorial \((x)^n\).

The forward version uses the forward recurrence. The bs version uses binary splitting. The rs version uses rectangular splitting. It takes an extra tuning parameter m which can be set to zero to choose automatically. The rec version chooses an algorithm automatically, avoiding use of the gamma function (so that it can be used in the computation of the gamma function). The default versions (rising_ui and rising_ui) choose an algorithm automatically and may additionally fall back on the gamma function.

void arb_hypgeom_rising_ui_jet_powsum(arb_ptr res, const arb_t x, ulong n, slong len, slong prec)
void arb_hypgeom_rising_ui_jet_bs(arb_ptr res, const arb_t x, ulong n, slong len, slong prec)
void arb_hypgeom_rising_ui_jet_rs(arb_ptr res, const arb_t x, ulong n, ulong m, slong len, slong prec)
void arb_hypgeom_rising_ui_jet(arb_ptr res, const arb_t x, ulong n, slong len, slong prec)
Computes the jet of the rising factorial \((x)^n\), truncated to length len. In other words, constructs the polynomial \((X + x)^n \in \mathbb{R}[X]\), truncated if len < n + 1 (and zero-extended if len > n + 1).

The powsum version computes the sequence of powers of x and forms integral linear combinations of these. The bs version uses binary splitting. The rs version uses rectangular splitting. It takes an extra tuning parameter m which can be set to zero to choose automatically. The default version chooses an algorithm automatically.
9.17.2 Gamma function

void _arb_hypgeom_gamma_stirling_term_bounds(slong *bound, const mag_t zinv, slong N)
For \(1 \leq n < N\), sets bound to an exponent bounding the \(n\)-th term in the Stirling series for the gamma function, given a precomputed upper bound for \(|z|^{-1}\). This function is intended for internal use and does not check for underflow or overflow in the exponents.

void arb_hypgeom_gamma_stirling_sum_horner(arb_t res, const arb_t z, slong N, slong prec)
void arb_hypgeom_gamma_stirling_sum_improved(arb_t res, const arb_t z, slong N, slong K, slong prec)
Sets res to the final sum in the Stirling series for the gamma function truncated before the term with index \(N\), i.e. computes \(\sum_{n=1}^{N-1} B_{2n}/(2n(2n-1))z^{2n-1}\). The horner version uses Horner scheme with gradual precision adjustments. The improved version uses rectangular splitting for the low-index terms and reexpands the high-index terms as hypergeometric polynomials, using a splitting parameter \(K\) (which can be set to 0 to use a default value).

void arb_hypgeom_gamma(arb_t res, const arb_t x, slong prec)
void arb_hypgeom_gamma_fmpq(arb_t res, const fmpq_t x, slong prec)
void arb_hypgeom_rgamma(arb_t res, const arb_t x, slong prec)
Sets res to the gamma function of \(x\) computed using a default algorithm choice.

9.17.3 Binomial coefficients

void arb_hypgeom_central_bin_ui(arb_t res, ulong n, slong prec)
Computes the central binomial coefficient \(\binom{2n}{n}\).

9.17.4 Generalized hypergeometric function

void arb_hypgeom_pfq(arb_t res, arb_srcreal_t a, slong p, arb_srcreal_t b, slong q, const arb_t z, int regularized, slong prec)
Computes the generalized hypergeometric function \(pFq(z)\), or the regularized version if regularized is set.
9.17.5 Confluent hypergeometric functions

void \texttt{arb\_hypgeom\_0f1}(\texttt{arb\_t\ res, const arb\_t\ a, const arb\_t\ z, int regularized,\ slong\ prec})

Computes the confluent hypergeometric limit function \( _0F_1(a, z) \) or \( \frac{1}{\Gamma(a)}_0F_1(a, z) \) if \texttt{regularized} is set.

void \texttt{arb\_hypgeom\_m}(\texttt{arb\_t\ res, const arb\_t\ a, const arb\_t\ b, const arb\_t\ z, int regularized,\ slong\ prec})

Computes the confluent hypergeometric function \( M(a, b, z) = {}_1F_1(a, b, z) \), or \( \frac{1}{\Gamma(b)}M(a, b, z) \) if \texttt{regularized} is set.

void \texttt{arb\_hypgeom\_1f1}(\texttt{arb\_t\ res, const arb\_t\ a, const arb\_t\ b, const arb\_t\ z, int regularized,\ slong\ prec})

Alias for \texttt{arb\_hypgeom\_m}().

void \texttt{arb\_hypgeom\_1f1\_integration}(\texttt{arb\_t\ res, const arb\_t\ a, const arb\_t\ b, const arb\_t\ z, int regularized,\ slong\ prec})

Computes the confluent hypergeometric function using numerical integration of the representation

\[
_1F_1(a, b, z) = \frac{\Gamma(b)}{\Gamma(a)\Gamma(b - a)} \int_0^1 e^{zt}t^{a-1}(1 - t)^{b-a-1}dt.
\]

This algorithm can be useful if the parameters are large. This will currently only return a finite enclosure if \( a \geq 1 \) and \( b - a \geq 1 \).

void \texttt{arb\_hypgeom\_u}(\texttt{arb\_t\ res, const arb\_t\ a, const arb\_t\ b, const arb\_t\ z,\ slong\ prec})

Computes the confluent hypergeometric function \( U(a, b, z) \).

void \texttt{arb\_hypgeom\_u\_integration}(\texttt{arb\_t\ res, const arb\_t\ a, const arb\_t\ b, const arb\_t\ z,\ slong\ prec})

Computes the confluent hypergeometric function \( U(a, b, z) \) using numerical integration of the representation

\[
U(a, b, z) = \frac{1}{\Gamma(a)} \int_0^\infty e^{-zt}t^{a-1}(1 + t)^{b-a-1}dt.
\]

This algorithm can be useful if the parameters are large. This will currently only return a finite enclosure if \( a \geq 1 \) and \( z > 0 \).

9.17.6 Gauss hypergeometric function

void \texttt{arb\_hypgeom\_2f1}(\texttt{arb\_t\ res, const arb\_t\ a, const arb\_t\ b, const arb\_t\ c, const arb\_t\ z, int regularized,\ slong\ prec})

Computes the Gauss hypergeometric function \( {}_2F_1(a, b, c, z) \), or \( F(a, b, c, z) = \frac{1}{\Gamma(c)}_2F_1(a, b, c, z) \) if \texttt{regularized} is set.

Additional evaluation flags can be passed via the \texttt{regularized} argument; see \texttt{acb\_hypgeom\_2f1()} for documentation.

void \texttt{arb\_hypgeom\_2f1\_integration}(\texttt{arb\_t\ res, const arb\_t\ a, const arb\_t\ b, const arb\_t\ c, const arb\_t\ z,\ slong\ prec})

Computes the Gauss hypergeometric function using numerical integration of the representation

\[
{}_2F_1(a, b, c, z) = \frac{\Gamma(a)}{\Gamma(b)\Gamma(c - b)} \int_0^1 t^{b-1}(1 - t)^{c-b-1}(1 - zt)^{-a}dt.
\]

This algorithm can be useful if the parameters are large. This will currently only return a finite enclosure if \( b \geq 1 \) and \( c - b \geq 1 \) and \( z < 1 \), possibly with \( a \) and \( b \) exchanged.
9.17.7 Error functions and Fresnel integrals

void arb_hypgeom_erf(const arb_t res, const arb_t z, slong prec)
Computes the error function \( \text{erf}(z) \).

void _arb_hypgeom_erf_series(const arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
void arb_hypgeom_erf_series(const arb_poly_t res, const arb_poly_t z, slong len, slong prec)
Computes the error function of the power series \( z \), truncated to length \( len \).

void arb_hypgeom_erfc(const arb_t res, const arb_t z, slong prec)
Computes the complementary error function \( \text{erfc}(z) = 1 - \text{erf}(z) \). This function avoids catastrophic cancellation for large positive \( z \).

void _arb_hypgeom_erfc_series(const arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
void arb_hypgeom_erfc_series(const arb_poly_t res, const arb_poly_t z, slong len, slong prec)
Computes the complementary error function of the power series \( z \), truncated to length \( len \).

void arb_hypgeom_erfi(const arb_t res, const arb_t z, slong prec)
Computes the imaginary error function \( \text{erfi}(z) = -i \text{erf}(iz) \).

void _arb_hypgeom_erfi_series(const arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
void arb_hypgeom_erfi_series(const arb_poly_t res, const arb_poly_t z, slong len, slong prec)
Computes the imaginary error function of the power series \( z \), truncated to length \( len \).

void arb_hypgeom_erfinv(const arb_t res, const arb_t z, slong prec)
void arb_hypgeom_erfci(const arb_t res, const arb_t z, slong prec)
void arb_hypgeom_erfcinv(const arb_t res, const arb_t z, slong prec)
Computes the inverse error function \( \text{erf}^{-1}(z) \) or inverse complementary error function \( \text{erfc}^{-1}(z) \).

void arb_hypgeom_fresnel(const arb_t res1, const arb_t res2, const arb_t z, int normalized, slong prec)
Sets \( res1 \) to the Fresnel sine integral \( S(z) \) and \( res2 \) to the Fresnel cosine integral \( C(z) \). Optionally, just a single function can be computed by passing NULL as the other output variable. The definition \( S(z) = \int_0^z \sin(t^2)dt \) is used if \( \text{normalized} \) is 0, and \( S(z) = \int_0^\infty \sin(\frac{1}{2}t^2)dt \) is used if \( \text{normalized} \) is 1 (the latter is the Abramowitz & Stegun convention). \( C(z) \) is defined analogously.

void _arb_hypgeom_fresnel_series(const arb_ptr res1, const arb_ptr res2, arb_srcptr z, slong zlen, int normalized, slong len, slong prec)
void arb_hypgeom_fresnel_series(const arb_poly_t res1, const arb_poly_t res2, const arb_poly_t z, int normalized, slong len, slong prec)
Sets \( res1 \) to the Fresnel sine integral and \( res2 \) to the Fresnel cosine integral of the power series \( z \), truncated to length \( len \). Optionally, just a single function can be computed by passing NULL as the other output variable.

9.17.8 Incomplete gamma and beta functions

void arb_hypgeom_gamma_upper(const arb_t res, const arb_t s, const arb_t z, int regularized, slong prec)
If \( \text{regularized} \) is 0, computes the upper incomplete gamma function \( \Gamma(s, z) \).
If \( \text{regularized} \) is 1, computes the regularized upper incomplete gamma function \( Q(s, z) = \Gamma(s, z) / \Gamma(s) \).
If \( \text{regularized} \) is 2, computes the generalized exponential integral \( z^{-s} \Gamma(s, z) = E_{1-s}(z) \) instead (this option is mainly intended for internal use; \text{arb_hypgeom_expint()} \) is the intended interface for computing the exponential integral).

void arb_hypgeom_gamma_upper_integration(const arb_t res, const arb_t s, const arb_t z, int regularized, slong prec)
Computes the upper incomplete gamma function using numerical integration.
void _arb_hypgeom_gamma_upper_series (arb_ptr res, const arb_t s, arb_srcptr z, slong zlen, int regularized, slong n, slong prec)

void _arb_hypgeom_gamma_upper_series (arb_poly_t res, const arb_t s, const arb_poly_t z, int regularized, slong n, slong prec)

Sets res to an upper incomplete gamma function where s is a constant and z is a power series, truncated to length n. The regularized argument has the same interpretation as in _arb_hypgeom_gamma_upper().

void arb_hypgeom_gamma_upper_series (arb_t res, const arb_t s, const arb_t z, int regularized, slong prec)

If regularized is 0, computes the lower incomplete gamma function \( \gamma(s, z) = \sum_{k=0}^{\infty} \frac{z^k}{\Gamma(k+1)} \). If regularized is 1, computes the regularized lower incomplete gamma function \( \Gamma(s, z) = \gamma(s, z)/\Gamma(s) \). If regularized is 2, computes a further regularized lower incomplete gamma function \( \gamma^*(s, z) = z^{-s} P(s, z) \).

void _arb_hypgeom_gamma_lower_series (arb_ptr res, const arb_t s, arb_srcptr z, slong zlen, int regularized, slong n, slong prec)

void _arb_hypgeom_gamma_lower_series (arb_poly_t res, const arb_t s, const arb_poly_t z, int regularized, slong n, slong prec)

Sets res to an lower incomplete gamma function where s is a constant and z is a power series, truncated to length n. The regularized argument has the same interpretation as in _arb_hypgeom_gamma_lower().

void arb_hypgeom_beta_lower_series (arb_t res, const arb_t a, const arb_t b, const arb_t z, int regularized, slong prec)

Computes the (lower) incomplete beta function, defined by \( B(a, b; z) = \int_0^z t^{a-1}(1-t)^{b-1} \), optionally the regularized incomplete beta function \( I(a, b; z) = B(a, b; z)/B(a, b; 1) \).

void _arb_hypgeom_beta_lower_series (arb_ptr res, const arb_t a, const arb_t b, arb_srcptr z, slong zlen, int regularized, slong n, slong prec)

void _arb_hypgeom_beta_lower_series (arb_poly_t res, const arb_t a, const arb_t b, const arb_poly_t z, int regularized, slong n, slong prec)

Sets res to the lower incomplete beta function \( B(a, b; z) \) (optionally the regularized version \( I(a, b; z) \)) where a and b are constants and z is a power series, truncating the result to length n. The underscore method requires positive lengths and does not support aliasing.

**Internal evaluation functions**

void _arb_hypgeom_lower_sum rs 1 (arb_t res, ulong p, ulong q, const arb_t z, slong N, slong prec)

Computes \( \sum_{k=0}^{N-1} \frac{z^k}{(a)_k} \) where \( a = p/q \) using rectangular splitting. It is assumed that \( p+qN \) fits in a limb.

void _arb_hypgeom_upper_sum rs 1 (arb_t res, ulong p, ulong q, const arb_t z, slong N, slong prec)

Computes \( \sum_{k=0}^{N-1} \frac{1}{(a)_k} \) where \( a = p/q \) using rectangular splitting. It is assumed that \( p+qN \) fits in a limb.

slong _arb_hypgeom_upper_fmpq_inf_choose_N (mag_t err, const fmpq_t a, const arb_t z, const mag_t abs_tol)

Returns number of terms \( N \) and sets \( err \) to the truncation error for evaluating \( \Gamma(a, z) \) using the asymptotic series at infinity, targeting an absolute tolerance of \( abs\_tol \). The error may be set to \( err \) if the tolerance cannot be achieved. Assumes that \( z \) is positive.
void _arb_hypgeom_gamma_upper_fmpq_inf_bsplit(arb_t res, const fmpq_t a, const arb_t z, slong N, slong prec)

Sets res to the approximation of $\Gamma(a, z)$ obtained by truncating the asymptotic series at infinity before term $N$. The truncation error bound has to be added separately.

slong _arb_hypgeom_gamma_lower_fmpq_0_choose_N(mag_t err, const fmpq_t a, const arb_t z, const mag_t abs_tol)

Returns number of terms $N$ and sets err to the truncation error for evaluating $\gamma(a, z)$ using the Taylor series at zero, targeting an absolute tolerance of abs_tol. Assumes that $z$ is positive.

void _arb_hypgeom_gamma_lower_fmpq_0_bsplit(arb_t res, const fmpq_t a, const arb_t z, slong N, slong prec)

Sets res to the approximation of $\gamma(a, z)$ obtained by truncating the Taylor series at zero before term $N$. The truncation error bound has to be added separately.

slong _arb_hypgeom_upper_singular_si_choose_N(mag_t err, slong n, const arb_t z, const mag_t abs_tol)

Returns number of terms $N$ and sets err to the truncation error for evaluating $\Gamma(-n, z)$ using the Taylor series at zero, targeting an absolute tolerance of abs_tol.

void _arb_hypgeom_upper_singular_si_bsplit(arb_t res, slong n, const arb_t z, slong N, slong prec)

Sets res to the approximation of $\Gamma(-n, z)$ obtained by truncating the Taylor series at zero before term $N$. The truncation error bound has to be added separately.

void _arb_gamma_upper_fmpq_step_bsplit(arb_t Gz1, const fmpq_t a, const arb_t z0, const arb_t z1, const arb_t Gz0, const arb_t expmz0, const mag_t abs_tol, slong prec)

Given $Gz0$ and $expmz0$ representing the values $\Gamma(a, z_0)$ and $\exp(-z_0)$, computes $\Gamma(a, z_1)$ using the Taylor series at $z_0$ evaluated using binary splitting, targeting an absolute error of abs_tol. Assumes that $z_0$ and $z_1$ are positive.

### 9.17.9 Exponential and trigonometric integrals

void arb_hypgeom_expint(arb_t res, const arb_t s, const arb_t z, slong prec)

Computes the generalized exponential integral $E_s(z)$.

void arb_hypgeom_ei(arb_t res, const arb_t z, slong prec)

Computes the exponential integral $\text{Ei}(z)$.

void _arb_hypgeom_ei_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)

void _arb_hypgeom_ei_series(arb_poly_t res, const arb_poly_t z, slong len, slong prec)

Computes the exponential integral of the power series $z$, truncated to length $len$.

void _arb_hypgeom_si_asymp(arb_t res, const arb_t z, slong N, slong prec)

void _arb_hypgeom_si_if2(arb_t res, const arb_t z, slong N, slong wp, slong prec)

void _arb_hypgeom_si(arb_t res, const arb_t z, slong prec)

Computes the sine integral $\text{Si}(z)$.

void _arb_hypgeom_si_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)

void _arb_hypgeom_si_series(arb_poly_t res, const arb_poly_t z, slong len, slong prec)

Computes the sine integral of the power series $z$, truncated to length $len$.

void _arb_hypgeom_ci_asymp(arb_t res, const arb_t z, slong N, slong prec)

void _arb_hypgeom_ci_2f3(arb_t res, const arb_t z, slong N, slong wp, slong prec)
 Computes the modified Bessel function of the first kind \( I_\nu(z) \).

void \texttt{arb\_hypgeom\_bessel\_j}(\texttt{arb\_t} res, const \texttt{arb\_t} nu, const \texttt{arb\_t} z, \texttt{slong} prec)
 Computes the Bessel function of the first kind \( J_\nu(z) \).

void \texttt{arb\_hypgeom\_bessel\_y}(\texttt{arb\_t} res, const \texttt{arb\_t} nu, const \texttt{arb\_t} z, \texttt{slong} prec)
 Computes the Bessel function of the second kind \( Y_\nu(z) \).

void \texttt{arb\_hypgeom\_bessel\_jy}(\texttt{arb\_t} res1, \texttt{arb\_t} res2, const \texttt{arb\_t} nu, const \texttt{arb\_t} z, \texttt{slong} prec)
 Sets \texttt{res1} to \( J_\nu(z) \) and \texttt{res2} to \( Y_\nu(z) \), computed simultaneously.

void \texttt{arb\_hypgeom\_bessel\_i}(\texttt{arb\_t} res, const \texttt{arb\_t} nu, const \texttt{arb\_t} z, \texttt{slong} prec)
 Computes the modified Bessel function of the first kind \( I_\nu(z) = e^z(iz)^{-\nu}J_\nu(iz) \).

void \texttt{arb\_hypgeom\_bessel\_i\_scaled}(\texttt{arb\_t} res, const \texttt{arb\_t} nu, const \texttt{arb\_t} z, \texttt{slong} prec)
 Computes the function \( e^{-z}I_\nu(z) \).

void \texttt{arb\_hypgeom\_bessel\_k}(\texttt{arb\_t} res, const \texttt{arb\_t} nu, const \texttt{arb\_t} z, \texttt{slong} prec)
 Computes the modified Bessel function of the second kind \( K_\nu(z) \).

void \texttt{arb\_hypgeom\_bessel\_k\_scaled}(\texttt{arb\_t} res, const \texttt{arb\_t} nu, const \texttt{arb\_t} z, \texttt{slong} prec)
 Computes the function \( e^zK_\nu(z) \).

void \texttt{arb\_hypgeom\_bessel\_i\_integration}(\texttt{arb\_t} res, const \texttt{arb\_t} nu, const \texttt{arb\_t} z, int scaled, \texttt{slong} prec)

9.17.10 Bessel functions

void \texttt{arb\_hypgeom\_shiname}(\texttt{arb\_t} res, const \texttt{arb\_t} z, \texttt{slong} prec)
 Computes the hyperbolic sine integral \( \text{Shi}(z) \).

void \texttt{arb\_hypgeom\_shiname\_series}(\texttt{arb\_ptr} res, \texttt{arb\_srcptr} z, \texttt{slong} len, \texttt{slong} prec)
 Computes the hyperbolic sine integral of the power series \( z \), truncated to length \( len \).

void \texttt{arb\_hypgeom\_chi}(\texttt{arb\_t} res, const \texttt{arb\_t} z, \texttt{slong} prec)
 Computes the hyperbolic cosine integral \( \text{Chi}(z) \).

void \texttt{arb\_hypgeom\_chi\_series}(\texttt{arb\_ptr} res, \texttt{arb\_srcptr} z, \texttt{slong} len, \texttt{slong} prec)
 Computes the hyperbolic cosine integral of the power series \( z \), truncated to length \( len \).

void \texttt{arb\_hypgeom\_li\_series}(\texttt{arb\_ptr} res, \texttt{arb\_srcptr} z, \texttt{slong} len, int offset, \texttt{slong} prec)
 Computes the logarithmic integral (optionally the offset version) of the power series \( z \), truncated to length \( len \).

9.17.10 Bessel functions

void \texttt{arb\_hypgeom\_bessel\_j\_series}(\texttt{arb\_poly\_t} res, const \texttt{arb\_poly\_t} z, \texttt{slong} prec)
 Computes the function \( J_\nu(z) \) for each polynomial in the power series \( z \).
void _arb_hypgeom_bessel_k_integration(arb_t res, const arb_t nu, const arb_t z, int scaled, slong prec)

Computes the modified Bessel functions using numerical integration.

### 9.17.11 Airy functions

void arb_hypgeom_airy(arb_t ai, arb_t ai_prime, arb_t bi, arb_t bi_prime, const arb_t z, slong prec)

Computes the Airy functions \( \text{Ai}(z), \text{Ai}'(z), \text{Bi}(z), \text{Bi}'(z) \) simultaneously. Any of the four function values can be omitted by passing \texttt{NULL} for the unwanted output variables, speeding up the evaluation.

void _arb_hypgeom_airy_series(arb_ptr ai, arb_ptr ai_prime, arb_ptr bi, arb_ptr bi_prime, arb_srcptr z, slong zlen, slong len, slong prec)

void arb_hypgeom_airy_series(arb_poly_t ai, arb_poly_t ai_prime, arb_poly_t bi, arb_poly_t bi_prime, const arb_poly_t z, slong len, slong prec)

Computes the Airy functions evaluated at the power series \( z \), truncated to length \( \text{len} \). As with the other Airy methods, any of the outputs can be \texttt{NULL}.

void arb_hypgeom_airy_zero(arb_t a, arb_t a_prime, arb_t b, arb_t b_prime, const fmpz_t n, slong prec)

Computes the \( n \)-th real zero \( a_n, a'_n, b_n, b'_n \) for the respective Airy function or Airy function derivative. Any combination of the four output variables can be \texttt{NULL}. The zeros are indexed by increasing magnitude, starting with \( n = 1 \) to follow the convention in the literature. An index \( n \) that is not positive is invalid input. The implementation uses asymptotic expansions for the zeros [PS1991] together with the interval Newton method for refinement.

### 9.17.12 Coulomb wave functions

void arb_hypgeom_coulomb(arb_t F, arb_t G, const arb_t l, const arb_t eta, const arb_t z, slong prec)

Writes to \( F, G \) the values of the respective Coulomb wave functions \( F_\ell(\eta, z) \) and \( G_\ell(\eta, z) \). Either of the outputs can be \texttt{NULL}.

void _arb_hypgeom_coulomb_series(arb_ptr F, arb_ptr G, const arb_t l, const arb_t eta, arb_srcptr z, slong zlen, slong len, slong prec)

void arb_hypgeom_coulomb_series(arb_poly_t F, arb_poly_t G, const arb_t l, const arb_t eta, const arb_poly_t z, slong len, slong prec)

Computes the Coulomb wave functions evaluated at the power series \( z \), truncated to length \( \text{len} \). Either of the outputs can be \texttt{NULL}.
9.17.13 Orthogonal polynomials and functions

void arb_hypgeom_chebyshev_t (arb_t res, const arb_t nu, const arb_t z, slong prec)
void arb_hypgeom_chebyshev_u (arb_t res, const arb_t nu, const arb_t z, slong prec)
void arb_hypgeom_jacobi_p (arb_t res, const arb_t n, const arb_t a, const arb_t b, const arb_t z, slong prec)
void arb_hypgeom_gegenbauer_c (arb_t res, const arb_t n, const arb_t m, const arb_t z, slong prec)
void arb_hypgeom_chebyshev_u (arb_t res, const arb_t nu, const arb_t z, slong prec)
void arb_hypgeom_hermite_h (arb_t res, const arb_t nu, const arb_t z, slong prec)

Computes Chebyshev, Jacobi, Gegenbauer, Laguerre or Hermite polynomials, or their extensions to non-integer orders.

void arb_hypgeom_legendre_p (arb_t res, const arb_t n, const arb_t m, const arb_t z, int type, slong prec)
void arb_hypgeom_legendre_q (arb_t res, const arb_t n, const arb_t m, const arb_t z, int type, slong prec)

Computes Legendre functions of the first and second kind. See \texttt{acb\_hypgeom\_legendre\_p()} and \texttt{acb\_hypgeom\_legendre\_q()} for definitions.

void arb_hypgeom_legendre_p ui deriv bound (mag_t dp, mag_t dp2, ulong n, const arb_t x, const arb_t x sub 1)

Sets \( dp \) to an upper bound for \( P'_n(x) \) and \( dp2 \) to an upper bound for \( P''_n(x) \) given \( x \) assumed to represent a real number with \( |x| \leq 1 \). The variable \( x2sub1 \) must contain the precomputed value \( 1 - x^2 \) (or \( x^2 - 1 \)). This method is used internally to bound the propagated error for Legendre polynomials.

void arb_hypgeom_legendre_p ui zero (arb_t res, arb_t res prime, ulong n, const arb_t x, slong K, slong prec)
void arb_hypgeom_legendre_p ui one (arb_t res, arb_t res prime, ulong n, const arb_t x, slong K, slong prec)
void arb_hypgeom_legendre_p ui asymp (arb_t res, arb_t res prime, ulong n, const arb_t x, slong K, slong prec)
void arb_hypgeom_legendre_p ui rec (arb_t res, arb_t res prime, ulong n, const arb_t x, slong prec)
void arb_hypgeom_legendre_p ui (arb_t res, arb_t res prime, ulong n, const arb_t x, slong prec)

Evaluates the ordinary Legendre polynomial \( P_n(x) \). If \( \text{res prime} \) is non-NULL, simultaneously evaluates the derivative \( P'_n(x) \).

The overall algorithm is described in [JM2018].

The versions \texttt{zero}, \texttt{one} respectively use the hypergeometric series expansions at \( x = 0 \) and \( x = 1 \) while the \texttt{asymp} version uses an asymptotic series on \((-1, 1)\) intended for large \( n \). The parameter \( K \) specifies the exact number of expansion terms to use (if the series expansion truncated at this point does not give the exact polynomial, an error bound is computed automatically). The asymptotic expansion with error bounds is given in [Bog2012]. The \texttt{rec} version uses the forward recurrence implemented using fixed-point arithmetic; it is only intended for the interval \((-1, 1)\), moderate \( n \) and modest precision.

The default version attempts to choose the best algorithm automatically. It also estimates the amount of cancellation in the hypergeometric series and increases the working precision to compensate, bounding the propagated error using derivative bounds.

void arb_hypgeom_legendre_p ui root (arb_t res, arb_t weight, ulong n, ulong k, slong prec)

Sets \( \text{res} \) to the \( k \)-th root of the Legendre polynomial \( P_n(x) \). We index the roots in decreasing order

\[
1 > x_0 > x_1 > \ldots > x_{n-1} > -1
\]
(which corresponds to ordering the roots of $P_n(\cos(\theta))$ in order of increasing $\theta$). If weight is non-NULL, it is set to the weight corresponding to the node $x_k$ for Gaussian quadrature on $[-1, 1]$. Note that only $[n/2]$ roots need to be computed, since the remaining roots are given by $x_k = -x_{n-1-k}$.

We compute an enclosing interval using an asymptotic approximation followed by some number of Newton iterations, using the error bounds given in [Pet1999]. If very high precision is requested, the root is subsequently refined using interval Newton steps with doubling working precision.

### 9.17.14 Dilogarithm

```c
void arb_hypgeom_dilog(arb_t res, const arb_t z, slong prec)
```

Computes the dilogarithm $\text{Li}_2(z)$.

### 9.17.15 Hypergeometric sums

```c
void arb_hypgeom_sum_fmpq_arb_forward(arb_t res, const fmpq *a, slong alen, const fmpq *b, slong blen, const arb_t z, int reciprocal, slong N, slong prec)
void arb_hypgeom_sum_fmpq_arb_rs(arb_t res, const fmpq *a, slong alen, const fmpq *b, slong blen, const arb_t z, int reciprocal, slong N, slong prec)
void arb_hypgeom_sum_fmpq_arb(arb_t res, const fmpq *a, slong alen, const fmpq *b, slong blen, const arb_t z, int reciprocal, slong N, slong prec)
```

Sets res to the finite hypergeometric sum $\sum_{n=0}^{N-1} (a)_{n} z^n/(b)_n$ where $x_n = (x_1)_{n}(x_2)_{n} \cdots$, given vectors of rational parameters $a$ (of length alen) and $b$ (of length blen). If reciprocal is set, replace $z$ by $1/z$. The forward version uses the forward recurrence, optimized by delaying divisions, the rs version uses rectangular splitting, and the default version uses an automatic algorithm choice.

```c
void arb_hypgeom_sum_fmpq_imag_arb_forward(arb_t res1, arb_t res2, const fmpq *a, slong alen, const fmpq *b, slong blen, const arb_t z, int reciprocal, slong N, slong prec)
void arb_hypgeom_sum_fmpq_imag_arb_rs(arb_t res1, arb_t res2, const fmpq *a, slong alen, const fmpq *b, slong blen, const arb_t z, int reciprocal, slong N, slong prec)
void arb_hypgeom_sum_fmpq_imag_arb(arb_t res1, arb_t res2, const fmpq *a, slong alen, const fmpq *b, slong blen, const arb_t z, int reciprocal, slong N, slong prec)
```

Sets res1 and res2 to the real and imaginary part of the finite hypergeometric sum $\sum_{n=0}^{N-1} (a)_{n} (iz)^n/(b)_n$. If reciprocal is set, replace $z$ by $1/z$.

### 9.18 acb_elliptic.h – elliptic integrals and functions of complex variables

This module supports computation of elliptic (doubly periodic) functions, and their inverses, elliptic integrals. See `acb_modular.h` for the closely related modular forms and Jacobi theta functions.

Warning: incomplete elliptic integrals have very complicated branch structure when extended to complex variables. For some functions in this module, branch cuts may be artifacts of the evaluation algorithm rather than having a natural mathematical justification. The user should, accordingly, watch out for edge cases where the functions implemented here may differ from other systems or literature. There may also exist points where a function should be well-defined but the implemented algorithm fails to produce a finite result due to artificial internal singularities.
9.18.1 Complete elliptic integrals

```c
void acb_elliptic_k(acb_t res, const acb_t m, slong prec)
```

Computes the complete elliptic integral of the first kind

\[
K(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \int_0^1 \frac{dt}{\sqrt{1 - t^2} \sqrt{1 - mt^2}}
\]

using the arithmetic-geometric mean: \( K(m) = \pi/(2M(\sqrt{1-m})) \).

```c
void acb_elliptic_k_jet(acb_ptr res, const acb_t m, slong len, slong prec)
```

Sets the coefficients in the array \( res \) to the power series expansion of the complete elliptic integral of the first kind at the point \( m \) truncated to length \( len \), i.e.

\[
K(m + x) \in \mathbb{C}[x].
\]

```c
void _acb_elliptic_k_series(acb_ptr res, acb_srcptr m, slong mlen, slong len, slong prec)
```

```c
void acb_elliptic_k_series(acb_poly_t res, const acb_poly_t m, slong len, slong prec)
```

Sets \( res \) to the complete elliptic integral of the first kind of the power series \( m \), truncated to length \( len \).

```c
void acb_elliptic_e(acb_t res, const acb_t m, slong prec)
```

Computes the complete elliptic integral of the second kind

\[
E(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \int_0^1 \frac{\sqrt{1 - mt^2}}{\sqrt{1 - t^2}} dt
\]

using \( E(m) = (1 - m)(2mK'(m) + K(m)) \) (where the prime denotes a derivative, not a complementary integral).

```c
void acb_elliptic_pi(acb_t res, const acb_t n, const acb_t m, slong prec)
```

Evaluates the complete elliptic integral of the third kind

\[
\Pi(n, m) = \int_0^{\pi/2} \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}} = \int_0^1 \frac{dt}{(1 - nt^2)\sqrt{1 - t^2}\sqrt{1 - mt^2}}
\]

This implementation currently uses the same algorithm as the corresponding incomplete integral. It is therefore less efficient than the implementations of the first two complete elliptic integrals which use the AGM.

9.18.2 Legendre incomplete elliptic integrals

```c
void acb_elliptic_f(acb_t res, const acb_t phi, const acb_t m, int pi, slong prec)
```

Evaluates the Legendre incomplete elliptic integral of the first kind, given by

\[
F(\phi, m) = \int_0^\phi \frac{dt}{\sqrt{1 - m \sin^2 t}} = \int_0^{\sin \phi} \frac{dt}{(\sqrt{1 - t^2})(\sqrt{1 - mt^2})}
\]

on the standard strip \(-\pi/2 \leq \text{Re}(\phi) \leq \pi/2\). Outside this strip, the function extends quasiperiodically as

\[
F(\phi + n\pi, m) = 2nK(m) + F(\phi, m), n \in \mathbb{Z}.
\]

Inside the standard strip, the function is computed via the symmetric integral \( R_F \).

If the flag \( pi \) is set to 1, the variable \( \phi \) is replaced by \( \pi \phi \), changing the quasiperiod to 1.

The function reduces to a complete elliptic integral of the first kind when \( \phi = \pi/2 \); that is, \( F(\pi/2, m) = K(m) \).
void `acb_elliptic_e_inc`(`acb_t res`, const `acb_t phi`, const `acb_t m`, int `pi`, slong `prec`)

Evaluates the Legendre incomplete elliptic integral of the second kind, given by

\[ E(\phi, m) = \int_{0}^{\phi} \sqrt{1 - m \sin^2 t} \, dt \]

on the standard strip \(-\pi/2 \leq \text{Re}(\phi) \leq \pi/2\). Outside this strip, the function extends quasiperiodically as

\[ E(\phi + n\pi, m) = 2n E(m) + E(\phi, m), \, n \in \mathbb{Z}. \]

Inside the standard strip, the function is computed via the symmetric integrals \(R_F\) and \(R_D\).

If the flag \(pi\) is set to 1, the variable \(\phi\) is replaced by \(\pi\varphi\), changing the quasiperiod to 1.

The function reduces to a complete elliptic integral of the second kind when \(\phi = \pi/2\); that is, \(E(\pi/2, m) = E(m)\).

void `acb_elliptic_pi_inc`(`acb_t res`, const `acb_t n`, const `acb_t phi`, const `acb_t m`, int `pi`, slong `prec`)

Evaluates the Legendre incomplete elliptic integral of the third kind, given by

\[ \Pi(n, \phi, m) = \int_{0}^{\phi} \frac{dt}{(1 - n \sin^2 t)\sqrt{1 - m \sin^2 t}} = \int_{0}^{\sin \phi} \frac{dt}{(1 - nt^2)\sqrt{1 - t^2}\sqrt{1 - mt^2}} \]

on the standard strip \(-\pi/2 \leq \text{Re}(\phi) \leq \pi/2\). Outside this strip, the function extends quasiperiodically as

\[ \Pi(n, \phi + k\pi, m) = 2k \Pi(n, m) + \Pi(n, \phi, m), \, k \in \mathbb{Z}. \]

Inside the standard strip, the function is computed via the symmetric integrals \(R_F\) and \(R_J\).

If the flag \(pi\) is set to 1, the variable \(\phi\) is replaced by \(\pi\varphi\), changing the quasiperiod to 1.

The function reduces to a complete elliptic integral of the third kind when \(\phi = \pi/2\); that is, \(\Pi(n, \pi/2, m) = \Pi(n, m)\).

### 9.18.3 Carlson symmetric elliptic integrals

Carlson symmetric forms are the preferred form of incomplete elliptic integrals, due to their neat properties and relatively simple computation based on duplication theorems. There are five named functions: \(R_F, R_G, R_J,\) and \(R_C, R_D\) which are special cases of \(R_F\) and \(R_J\) respectively. We largely follow the definitions and algorithms in [Car1995] and chapter 19 in [NIST2012].

void `acb_elliptic_rf`(`acb_t res`, const `acb_t x`, const `acb_t y`, const `acb_t z`, int `flags`, slong `prec`)

Evaluates the Carlson symmetric elliptic integral of the first kind

\[ R_F(x, y, z) = \frac{1}{2} \int_{0}^{\infty} \frac{dt}{(t + x)(t + y)(t + z)} \]

where the square root extends continuously from positive infinity. The integral is well-defined for \(x, y, z \notin (-\infty, 0)\), and with at most one of \(x, y, z\) being zero. When some parameters are negative real numbers, the function is still defined by analytic continuation.

In general, one or more duplication steps are applied until \(x, y, z\) are close enough to use a multivariate Taylor series.
The special case \( R_C(x, y) = R_F(x, y, y) = \frac{1}{2} \int_0^\infty (t+x)^{-1/2}(t+y)^{-1} dt \) may be computed by setting \( y \) and \( z \) to the same variable. (This case is not yet handled specially, but might be optimized in the future.)

The **flags** parameter is reserved for future use and currently does nothing. Passing 0 results in default behavior.

```c
void acb_elliptic_rg(acb_t res, const acb_t x, const acb_t y, const acb_t z, int flags, slong prec)
```

Evaluates the Carlson symmetric elliptic integral of the second kind

\[
R_G(x, y, z) = \frac{1}{4} \int_0^\infty \frac{t}{\sqrt{(t+x)(t+y)(t+z)}} \left( \frac{x}{t+x} + \frac{y}{t+y} + \frac{z}{t+z} \right) dt
\]

where the square root is taken continuously as in \( R_F \). The evaluation is done by expressing \( R_G \) in terms of \( R_F \) and \( R_D \). There are no restrictions on the variables.

```c
void acb_elliptic_rj(acb_t res, const acb_t x, const acb_t y, const acb_t z, const acb_t p, int flags, slong prec)
```

```c
void acb_elliptic_rj_carlson(acb_t res, const acb_t x, const acb_t y, const acb_t z, const acb_t p, int flags, slong prec)
```

```c
void acb_elliptic_rj_integration(acb_t res, const acb_t x, const acb_t y, const acb_t z, const acb_t p, int flags, slong prec)
```

Evaluates the Carlson symmetric elliptic integral of the third kind

\[
R_J(x, y, z, p) = \frac{3}{2} \int_0^\infty \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}}
\]

where the square root is taken continuously as in \( R_F \).

Three versions of this function are available: the **carlson** version applies one or more duplication steps until \( x, y, z, p \) are close enough to use a multivariate Taylor series.

The duplication algorithm is not correct for all possible combinations of complex variables, since the square roots taken during the computation can introduce spurious branch cuts. According to [Car1995], a sufficient (but not necessary) condition for correctness is that \( x, y, z \) have nonnegative real part and that \( p \) has positive real part.

In other cases, the algorithm might still be correct, but no attempt is made to check this; it is up to the user to verify that the duplication algorithm is appropriate for the given parameters before calling this function.

The **integration** algorithm uses explicit numerical integration to translate the parameters to the right half-plane. This is reliable but can be slow.

The default method uses the **carlson** algorithm when it is certain to be correct, and otherwise falls back to the slow **integration** algorithm.

The special case \( R_D(x, y, z) = R_J(x, y, z, z) \) may be computed by setting \( z \) and \( p \) to the same variable. This case is handled specially to avoid redundant arithmetic operations. In this case, the **carlson** algorithm is correct for all \( x, y \) and \( z \).

The **flags** parameter is reserved for future use and currently does nothing. Passing 0 results in default behavior.

```c
void acb_elliptic_rc1(acb_t res, const acb_t x, slong prec)
```

This helper function computes the special case \( R_C(1,1+x) = \frac{\arctan(\sqrt{x})}{\sqrt{x}} = 2F_1(1,1/2,3/2,-x) \), which is needed in the evaluation of \( R_J \).
9.18.4 Weierstrass elliptic functions

Elliptic functions may be defined on a general lattice \( \Lambda = \{m\omega_1 + n\omega_2 : m, n \in \mathbb{Z} \} \) with half-periods \( \omega_1, \omega_2 \). We simplify by setting \( 2\omega_1 = 1, 2\omega_2 = \tau \) with \( \text{im}(\tau) > 0 \). To evaluate the functions on a general lattice, it is enough to make a linear change of variables. The main reference is chapter 23 in [NIST2012].

void _acb_elliptic_p_series(acb_ptr res, acb_srcptr z, long zlen, const acb_t tau, long long slong prec)

Computes the formal power series \( \wp(\tau) \) of Weierstrass’s elliptic function \( \wp(z, \tau) \). This function is given by the elliptic integral

\[
\wp^{-1}(z, \tau) = \frac{1}{2} \int_0^\infty \frac{dt}{\sqrt{(t-e_1)(t-e_2)(t-e_3)}} = R_F(z - e_1, z - e_2, z - e_3).
\]

void acb_elliptic_zeta(acb_t res, const acb_t z, const acb_t tau, slong prec)

Computes the Weierstrass zeta function

\[
\zeta(z, \tau) = \frac{1}{z} + \sum_{n^2+m^2 \neq 0} \left[ \frac{1}{z - m + n\tau} + \frac{1}{m + n\tau} + \frac{z}{(m + n\tau)^2} \right]
\]

which is quasiperiodic with \( \zeta(z + 1, \tau) = \zeta(z, \tau) + \zeta(1/2, \tau) \) and \( \zeta(z + \tau, \tau) = \zeta(z, \tau) + \zeta(\tau/2, \tau) \).
void acb_elliptic_sigma(acb_t res, const acb_t z, const acb_t tau, slong prec)

Computes the Weierstrass sigma function

$$\sigma(z, \tau) = z \prod_{n^2+m^2 \neq 0} \left( 1 - \frac{z}{m + n\tau} \right) \exp \left( \frac{z}{m + n\tau} + \frac{z^2}{2(m + n\tau)^2} \right)$$

which is quasiperiodic with $\sigma(z + 1, \tau) = -e^{2\pi i (1/2, \tau)}(z+1/2)\sigma(z, \tau)$ and $\sigma(z + \tau, \tau) =$...

9.19 acb_modular.h – modular forms of complex variables

This module provides methods for numerical evaluation of modular forms and Jacobi theta functions. See acb_elliptic.h for the closely related elliptic functions and integrals.

In the context of this module, tau or $\tau$ always denotes an element of the complex upper half-plane $\mathbb{H} = \{ z \in \mathbb{C} : \text{Im}(z) > 0 \}$. We also often use the variable $q$, variously defined as $q = e^{2\pi i \tau}$ (usually in relation to modular forms) or $q = e^{\pi i \tau}$ (usually in relation to theta functions) and satisfying $|q| < 1$. We will clarify the local meaning of $q$ every time such a quantity appears as a function of $\tau$.

As usual, the numerical functions in this module compute strict error bounds: if tau is represented by an acb_t whose content overlaps with the real line (or lies in the lower half-plane), and tau is passed to a function defined only on $\mathbb{H}$, then the output will have an infinite radius. The analogous behavior holds for functions requiring $|q| < 1$.

9.19.1 The modular group

type psl2z_struct

type psl2z_t

Represents an element of the modular group $\text{PSL}(2, \mathbb{Z})$, namely an integer matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

with $ad - bc = 1$, and with signs canonicalized such that $c \geq 0$, and $d > 0$ if $c = 0$. The struct members $a$, $b$, $c$, $d$ are of type fmpz.

void psl2z_init(psl2z_t g)

Initializes $g$ and set it to the identity element.

void psl2z_clear(psl2z_t g)

Clears $g$.

void psl2z_swap(psl2z_t f, psl2z_t g)

Swaps $f$ and $g$ efficiently.

void psl2z_set(psl2z_t f, const psl2z_t g)

Sets $f$ to a copy of $g$.

void psl2z_one(psl2z_t g)

Sets $g$ to the identity element.

int psl2z_is_one(const psl2z_t g)

Returns nonzero iff $g$ is the identity element.

void psl2z_print(const psl2z_t g)

Prints $g$ to standard output.
void `psl2z_fprint`(FILE *file, const `psl2z_t` g)
Prints `g` to the stream `file`.

int `psl2z_equal`(const `psl2z_t` f, const `psl2z_t` g)
Returns nonzero iff `f` and `g` are equal.

void `psl2z_mul`(psl2z_t h, const `psl2z_t` f, const `psl2z_t` g)
Sets `h` to the product of `f` and `g`, namely the matrix product with the signs canonicalized.

void `psl2z_inv`(psl2z_t h, const `psl2z_t` g)
Sets `h` to the inverse of `g`.

int `psl2z_is_correct`(const `psl2z_t` g)
Returns nonzero iff `g` contains correct data, i.e. satisfying \(ad - bc = 1\), \(c \geq 0\), and \(d > 0\) if \(c = 0\).

void `psl2z_randtest`(psl2z_t g, flint_rand_t state, slong bits)
Sets `g` to a random element of \(\text{PSL}(2, \mathbb{Z})\) with entries of bit length at most \(\text{bits}\) (or 1, if \(\text{bits}\) is not positive). We first generate \(a\) and \(d\), compute their Bezout coefficients, divide by the GCD, and then correct the signs.

9.19.2 Modular transformations

void `acb_modular_transform`(acb_t w, const `psl2z_t` g, const acb_t z, slong prec)
Applies the modular transformation `g` to the complex number `z`, evaluating \(w = gz = \frac{az + b}{cz + d}\).

void `acb_modular_fundamental_domain_approx_d`(psl2z_t g, double x, double y, double one_minus_eps)
Attempts to determine a modular transformation `g` that maps the complex number \(x + yi\) to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by `one_minus_eps`.

The inputs are assumed to be finite numbers, with \(y\) positive.

Uses floating-point iteration, repeatedly applying either the transformation \(z \leftarrow z + b\) or \(z \leftarrow -1/z\). The iteration is terminated if \(|x| \leq 1/2\) and \(x^2 + y^2 \geq 1 - \varepsilon\) where \(1 - \varepsilon\) is passed as `one_minus_eps`.

It is also terminated if too many steps have been taken without convergence, or if the numbers end up too large or too small for the working precision.

The algorithm can fail to produce a satisfactory transformation. The output `g` is always set to some correct modular transformation, but it is up to the user to verify a posteriori that `g` maps \(x + yi\) close enough to the fundamental domain.

This function first tries to use `acb_modular_fundamental_domain_approx_d()` and checks if the result is acceptable. If this fails, it calls `acb_modular_fundamental_domain_approx_arf()` with higher precision. Finally, \(w = gz\) is evaluated by a single application of `g`.

The algorithm can fail to produce a satisfactory transformation. The output `g` is always set to some correct modular transformation, but it is up to the user to verify a posteriori that `w` is close enough to the fundamental domain.
int acb_modular_is_in_fundamental_domain(const acb_t z, const arf_t tol, slong prec)

Returns nonzero if it is certainly true that \(|z| \geq 1 - \varepsilon\) and \(|\Re(z)| \leq 1/2 + \varepsilon\) where \(\varepsilon\) is specified by \(tol\). Returns zero if this is false or cannot be determined.

9.19.3 Addition sequences

void acb_modular_fill_addseq(slong *tab, slong len)

Builds a near-optimal addition sequence for a sequence of integers which is assumed to be reasonably dense.

As input, the caller should set each entry in \(tab\) to \(-1\) if that index is to be part of the addition sequence, and to \(0\) otherwise. On output, entry \(i\) in \(tab\) will either be zero (if the number is not part of the sequence), or a value \(j\) such that both \(j\) and \(i - j\) are also marked. The first two entries in \(tab\) are ignored (the number 1 is always assumed to be part of the sequence).

9.19.4 Jacobi theta functions

Unfortunately, there are many inconsistent notational variations for Jacobi theta functions in the literature. Unless otherwise noted, we use the functions

\[
\begin{align*}
\theta_1(z, \tau) &= -i \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i ((n + 1/2)^2 \tau + (2n + 1)z)) = 2q_{1/4} \sum_{n=0}^{\infty} (-1)^n q^{(n+1)} \sin((2n + 1)\pi z) \\
\theta_2(z, \tau) &= \sum_{n=-\infty}^{\infty} \exp(\pi i ((n + 1/2)^2 \tau + (2n + 1)z)) = 2q_{1/4} \sum_{n=0}^{\infty} q^{(n+1)} \cos((2n + 1)\pi z) \\
\theta_3(z, \tau) &= \sum_{n=-\infty}^{\infty} \exp(\pi i n^2 \tau + 2nz) = 1 + 2 \sum_{n=1}^{\infty} q^{n^2} \cos(2n\pi z) \\
\theta_4(z, \tau) &= \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i n^2 \tau + 2nz) = 1 + 2 \sum_{n=1}^{\infty} (-1)^n q^{n^2} \cos(2n\pi z)
\end{align*}
\]

where \(q = \exp(\pi i \tau)\) and \(q_{1/4} = \exp(\pi i \tau/4)\). Note that many authors write \(q_{1/4}\) as \(q^{1/4}\), but the principal fourth root \((q^{1/4}) = \exp(1/4 \log q)\) differs from \(q_{1/4}\) in general and some formulas are only correct if one reads \(q^{1/4} = \exp(\pi i \tau/4)\). To avoid confusion, we only write \(q^k\) when \(k\) is an integer.

void acb_modular_theta_transform(int *R, int *S, int *C, const psl2z_t g)

We wish to write a theta function with quasiperiod \(\tau' = g\tau\), given some \(g = (a,b,c,d) \in \text{PSL}(2,\mathbb{Z})\). For \(i = 0, 1, 2, 3\), this function computes integers \(R_i\) and \(S_i\) (and \(R\) and \(S\) should be arrays of length \(4\)) such that

\[
\theta_{1+i}(z, \tau) = \exp(\pi i R_i/4) \cdot A \cdot B \cdot \theta_{1+i}(z', \tau')
\]

where \(z' = z, A = B = 1\) if \(C = 0\), and

\[
z' = \frac{-z}{cr + d}, \quad A = \sqrt{\frac{i}{cr + d}}, \quad B = \exp\left(-\pi ic\frac{z^2}{cr + d}\right)
\]

if \(C = 1\). Note that \(A\) is well-defined with the principal branch of the square root since \(A^2 = i/(cr + d)\) lies in the right half-plane.

Firstly, if \(c = 0\), we have \(\theta_i(z, \tau) = \exp(-\pi i b/4)\theta_i(z, \tau + b)\) for \(i = 1, 2\), whereas \(\theta_3\) and \(\theta_4\) remain unchanged when \(b\) is even and swap places with each other when \(b\) is odd. In this case we set \(C = 0\).

For an arbitrary \(g\) with \(c > 0\), we set \(C = 1\). The general transformations are given by Rademacher [Rad1973]. We need the function \(\theta_{m,n}(z, \tau)\) defined for \(m, n \in \mathbb{Z}\) by (beware of the typos in [Rad1973])

\[
\theta_{0,0}(z, \tau) = \theta_3(z, \tau), \quad \theta_{0,1}(z, \tau) = \theta_4(z, \tau)
\]
\[ \theta_{1,0}(z, \tau) = \theta_2(z, \tau), \quad \theta_{1,1}(z, \tau) = i \theta_1(z, \tau) \]
\[ \theta_{m+2,n}(z, \tau) = (-1)^n \theta_{m,n}(z, \tau) \]
\[ \theta_{m,n+2}(z, \tau) = \theta_{m,n}(z, \tau). \]

Then we may write
\[ \theta_1(z, \tau) = \varepsilon_1 AB \theta_1(z', \tau') \]
\[ \theta_2(z, \tau) = \varepsilon_2 AB \theta_1(-c+1+a(z', \tau') \]
\[ \theta_3(z, \tau) = \varepsilon_3 AB \theta_1+d-c+1-b+a(z', \tau') \]
\[ \theta_4(z, \tau) = \varepsilon_4 AB \theta_1+d-c+1-b(z', \tau') \]

where \( \varepsilon_1 \) is an 8th root of unity. Specifically, if we denote the 24th root of unity in the transformation formula of the Dedekind eta function by \( \varepsilon(a, b, c, d) = \exp(\pi i R(a, b, c, d)/12) \) (see `acb_modular_epsilon_arg()`, then:
\[ \varepsilon_1(a, b, c, d) = \exp(\pi i [R(-d, b, c, -a) + 1]/4) \]
\[ \varepsilon_2(a, b, c, d) = \exp(\pi i [R(a, b, c, d) + (5 + (2 - c)a)]/4) \]
\[ \varepsilon_3(a, b, c, d) = \exp(\pi i [-R(a, b, c, d) + (4 + (c - d - 2) (b - a))] / 4) \]
\[ \varepsilon_4(a, b, c, d) = \exp(\pi i [-R(a, b, c, d) + (3 - (2 + d) b)] / 4) \]

These formulas are easily derived from the formulas in [Rad1973] (Rademacher has the transformed/untransformed variables exchanged, and his “\( \varepsilon \)” differs from ours by a constant offset in the phase).

void `acb_modular_addseq_theta(slong *exponents, slong *aindex, slong *bindex, slong num)`

Constructs an addition sequence for the first `num` squares and triangular numbers interleaved (excluding zero), i.e. 1, 2, 4, 6, 9, 12, 16, 20, 25, 30 etc.

void `acb_modular_theta_sum(acb_ptr theta1, acb_ptr theta2, acb_ptr theta3, acb_ptr theta4, const acb_t w, int w_is_unit, const acb_t q, slong len, slong prec)`

Simultaneously computes the first `len` coefficients of each of the formal power series
\[ \theta_1(z + x, \tau)/q_{1/4} \in \mathbb{C}[x] \]
\[ \theta_2(z + x, \tau)/q_{1/4} \in \mathbb{C}[x] \]
\[ \theta_3(z + x, \tau) \in \mathbb{C}[x] \]
\[ \theta_4(z + x, \tau) \in \mathbb{C}[x] \]

given \( w = \exp(\pi i z) \) and \( q = \exp(\pi i \tau) \), by summing a finite truncation of the respective theta function series. In particular, with `len` equal to 1, computes the respective value of the theta function at the point \( z \). We require `len` to be positive. If `w_is_unit` is nonzero, \( w \) is assumed to lie on the unit circle, i.e. \( z \) is assumed to be real.

Note that the factor \( q_{1/4} \) is removed from \( \theta_1 \) and \( \theta_2 \). To get the true theta function values, the user has to multiply this factor back. This convention avoids unnecessary computations, since the user can compute \( q_{1/4} = \exp(\pi i \tau / 4) \) followed by \( q = (q_{1/4})^4 \), and in many cases when computing products or quotients of theta functions, the factor \( q_{1/4} \) can be eliminated entirely.

This function is intended for \( |q| \ll 1 \). It can be called with any \( q \), but will return useless intervals if convergence is not rapid. For general evaluation of theta functions, the user should only call this function after applying a suitable modular transformation.

We consider the sums together, alternatingly updating \((\theta_1, \theta_2)\) or \((\theta_3, \theta_4)\). For \( k = 0, 1, 2, \ldots \), the powers of \( q \) are \( [(k + 2)^2/4] = 1, 2, 4, 6, 9 \) etc. and the powers of \( w \) are \( \pm (k + 2) = \pm 2, \pm 3, \pm 4, \ldots \) etc. The scheme is illustrated by the following table:
\[
\begin{array}{|c|c|c|}
\hline
k & \theta_1, \theta_2 & q^k & \pm (w^1 \pm w^{-1}) \\
0 & \theta_3, \theta_4 & q^1 & \pm (w^2 \pm w^{-2}) \\
1 & \theta_1, \theta_2 & q^2 & \pm (w^3 \pm w^{-3}) \\
2 & \theta_3, \theta_4 & q^4 & \pm (w^6 \pm w^{-6}) \\
3 & \theta_1, \theta_2 & q^6 & \pm (w^9 \pm w^{-9}) \\
4 & \theta_3, \theta_4 & q^9 & \pm (w^{12} \pm w^{-12}) \\
5 & \theta_1, \theta_2 & q^{12} & \pm (w^{15} \pm w^{-15}) \\
\hline
\end{array}
\]
For some integer $N \geq 1$, the summation is stopped just before term $k = N$. Let $Q = |q|$, $W = \max(|w|, |w^{-1}|)$, $E = [(N + 2)^2/4]$ and $F = [(N + 1)/2] + 1$. The error of the zeroth derivative can be bounded as

$$2QFW^{N+2}[1 + QFW + Q^2W^2 + \ldots] = \frac{2QFW^{N+2}}{1 - QFW}$$

provided that the denominator is positive (otherwise we set the error bound to infinity). When $len$ is greater than 1, consider the derivative of order $r$. The term of index $k$ and order $r$ picks up a factor of magnitude $(k + 2)^r$ from differentiation of $w^{k+2}$ (it also picks up a factor $\pi^r$, but we omit this until we rescale the coefficients at the end of the computation). Thus we have the error bound

$$2QFW^{N+2}(N + 2)^r \left[1 + QFW\frac{(N + 3)^r}{(N + 2)^r} + Q^2FW^2\frac{(N + 4)^r}{(N + 2)^r} + \ldots\right]$$

which by the inequality $(1 + m/(N + 2))^r \leq \exp(mr/(N + 2))$ can be bounded as

$$2QFW^{N+2}(N + 2)^r \frac{1 - QFW \exp(r/(N + 2))}{1 - QFW}$$

again valid when the denominator is positive.

To actually evaluate the series, we write the even cosine terms as $w^{2n} + w^{-2n}$, the odd cosine terms as $w(w^{2n} + w^{-2n-2})$, and the sine terms as $w(w^{2n} - w^{-2n-2})$. This way we only need even powers of $w$ and $w^{-1}$. The implementation is not yet optimized for real $z$, in which case further work can be saved.

This function does not permit aliasing between input and output arguments.

```c
void acb_modular_theta_const_sum_basecase(acb_t theta2, acb_t theta3, acb_t theta4, const acb_t q, slong N, slong prec)

void acb_modular_theta_const_sum_rs(acb_t theta2, acb_t theta3, acb_t theta4, const acb_t q, slong N, slong prec)

Computes the truncated theta constant sums $θ_2 = \sum_{k(k+1) < N} q^{k(k+1)}$, $θ_3 = \sum_{k^2 < N} q^{k^2}$, $θ_4 = \sum_{k^2 < N} (-1)^k q^{k^2}$. The basecase version uses a short addition sequence. The rs version uses rectangular splitting. The algorithms are described in [EHJ2016].

void acb_modular_theta_const_sum(acb_t theta2, acb_t theta3, acb_t theta4, const acb_t q, slong prec)

Computes the respective theta constants by direct summation (without applying modular transformations). This function selects an appropriate $N$, calls either `acb_modular_theta_const_sum_basecase()` or `acb_modular_theta_const_sum_rs()` depending on $N$, and adds a bound for the truncation error.

void acb_modular_theta_notransform(acb_t theta1, acb_t theta2, acb_t theta3, acb_t theta4, const acb_t z, const acb_t tau, slong prec)

Evaluates the Jacobi theta functions $θ_i(z, τ)$, $i = 1, 2, 3, 4$ simultaneously. This function does not move $τ$ to the fundamental domain. This is generally worse than `acb_modular_theta()`, but can be slightly better for moderate input.

void acb_modular_theta(acb_t theta1, acb_t theta2, acb_t theta3, acb_t theta4, const acb_t z, const acb_t tau, slong prec)

Evaluates the Jacobi theta functions $θ_i(z, τ)$, $i = 1, 2, 3, 4$ simultaneously. This function moves $τ$ to the fundamental domain and then also reduces $z$ modulo $τ$ before calling `acb_modular_theta_sum()`.

void acb_modular_theta_jet_notransform(acb_ptr theta1, acb_ptr theta2, acb_ptr theta3, acb_ptr theta4, const acb_t z, const acb_t tau, slong len, slong prec)
void \texttt{acb\_modular\_theta\_jet}(acb\_ptr theta1, acb\_ptr theta2, acb\_ptr theta3, acb\_ptr theta4, const acb\_t z, const acb\_t tau, slong len, slong prec)

Evaluates the Jacobi theta functions along with their derivatives with respect to \( z \), writing the first \( \text{len} \) coefficients in the power series \( \theta_i(z + x, \tau) \in C[[x]] \) to each respective output variable. The \texttt{notransform} version does not move \( \tau \) to the fundamental domain or reduce \( z \) during the computation.

void \_acb\_modular\_theta\_series(acb\_ptr theta1, acb\_ptr theta2, acb\_ptr theta3, acb\_ptr theta4, acb\_srcptr z, slong zlen, const acb\_t tau, slong len, slong prec)

void \texttt{acb\_modular\_theta\_series}(acb\_poly\_t theta1, acb\_poly\_t theta2, acb\_poly\_t theta3, acb\_poly\_t theta4, acb\_poly\_t z, const acb\_t tau, slong len, slong prec)

Evaluates the respective Jacobi theta functions of the power series \( z \), truncated to length \( \text{len} \). Either of the output variables can be \texttt{NULL}.

9.19.5 Dedekind eta function

void \texttt{acb\_modular\_addseq\_eta}(slong \*exponents, slong \*aindex, slong \*bindex, slong num)

Constructs an addition sequence for the first \( \text{num} \) generalized pentagonal numbers (excluding zero), i.e. 1, 2, 5, 7, 12, 15, 22, 26, 35, 40 etc.

void \texttt{acb\_modular\_eta\_sum}(acb\_t eta, const acb\_t q, slong prec)

Evaluates the Dedekind eta function without the leading 24th root, i.e.

\[
\exp(-\pi i \tau/12) \eta(\tau) = \sum_{n=-\infty}^{\infty} (-1)^n q^{(3n^2-n)/2}
\]

given \( q = \exp(2\pi i \tau) \), by summing the defining series.

This function is intended for \(|q| \ll 1\). It can be called with any \( q \), but will return useless intervals if convergence is not rapid. For general evaluation of the eta function, the user should only call this function after applying a suitable modular transformation.

The series is evaluated using either a short addition sequence or rectangular splitting, depending on the number of terms. The algorithms are described in [EHJ2016].

int \texttt{acb\_modular\_epsilon\_arg}(const psl2z\_t g)

Given \( g = (a, b; c, d) \), computes an integer \( R \) such that \( \varepsilon(a, b, c, d) = \exp(\pi i R/12) \) is the 24th root of unity in the transformation formula for the Dedekind eta function,

\[
\eta\left(\frac{a\tau + b}{c\tau + d}\right) = \varepsilon(a, b, c, d) \sqrt{c\tau + d} \eta(\tau).
\]

void \texttt{acb\_modular\_eta}(acb\_t r, const acb\_t tau, slong prec)

Computes the Dedekind eta function \( \eta(\tau) \) given \( \tau \) in the upper half-plane. This function applies the functional equation to move \( \tau \) to the fundamental domain before calling \texttt{acb\_modular\_eta\_sum()}.  

9.19.6 Modular forms

void \texttt{acb\_modular\_j}(acb\_t r, const acb\_t tau, slong prec)

Computes Klein’s \( j \)-invariant \( j(\tau) \) given \( \tau \) in the upper half-plane. The function is normalized so that \( j(i) = 1728 \). We first move \( \tau \) to the fundamental domain, which does not change the value of the function. Then we use the formula \( j(\tau) = 32(\theta_2^4 + \theta_4^4 + \theta_6^4)^3/(\theta_2\theta_3\theta_4)^4 \) where \( \theta_i = \theta_i(0, \tau) \).
void \textbf{acb\_modular\_lambda}(acb\_t r, const acb\_t tau, slong prec)

Computes the lambda function \( \lambda(\tau) = \theta_3^4(0,\tau)/\theta_2^4(0,\tau) \), which is invariant under modular transformations \((a, b; c, d)\) where \(a, d\) are odd and \(b, c\) are even.

void \textbf{acb\_modular\_delta}(acb\_t r, const acb\_t tau, slong prec)

Computes the modular discriminant \( \Delta(\tau) = \eta(\tau)^{24} \), which transforms as

\[
\Delta\left( \frac{a\tau + b}{c\tau + d} \right) = (c\tau + d)^{12} \Delta(\tau).
\]

The modular discriminant is sometimes defined with an extra factor \((2\pi)^{12}\), which we omit in this implementation.

void \textbf{acb\_modular\_eisenstein}(acb\_ptr r, const acb\_t tau, slong len, slong prec)

Computes simultaneously the first \(len\) entries in the sequence of Eisenstein series \( G_4(\tau), G_6(\tau), G_8(\tau), \ldots \), defined by

\[
G_{2k}(\tau) = \sum_{m^2 + n^2 \neq 0} \frac{1}{(m + n\tau)^{2k}}
\]

and satisfying

\[
G_{2k}\left( \frac{a\tau + b}{c\tau + d} \right) = (c\tau + d)^{2k} G_{2k}(\tau).
\]

We first evaluate \( G_4(\tau) \) and \( G_6(\tau) \) on the fundamental domain using theta functions, and then compute the Eisenstein series of higher index using a recurrence relation.

9.19.7 Elliptic integrals and functions

See the \texttt{acb\_elliptic.h} module for elliptic integrals and functions. The following wrappers are available for backwards compatibility.

void \textbf{acb\_modular\_elliptic\_k}(acb\_t w, const acb\_t m, slong prec)

void \textbf{acb\_modular\_elliptic\_k\_cpx}(acb\_ptr w, const acb\_t m, slong len, slong prec)

void \textbf{acb\_modular\_elliptic\_e}(acb\_t w, const acb\_t m, slong prec)

void \textbf{acb\_modular\_elliptic\_p}(acb\_t wp, const acb\_t z, const acb\_t tau, slong prec)

void \textbf{acb\_modular\_elliptic\_p\_cpx}(acb\_ptr wp, const acb\_t z, const acb\_t tau, slong len, slong prec)

9.19.8 Class polynomials

void \textbf{acb\_modular\_hilbert\_class\_poly}(fmpz\_poly\_t res, slong D)

Sets \( res \) to the Hilbert class polynomial of discriminant \( D \), defined as

\[
H_D(x) = \prod_{(a,b,c)} \left( x - j \left( \frac{-b + \sqrt{D}}{2a} \right) \right)
\]

where \((a, b, c)\) ranges over the primitive reduced positive definite binary quadratic forms of discriminant \( b^2 - 4ac = D \).

The Hilbert class polynomial is only defined if \( D < 0 \) and \( D \) is congruent to 0 or 1 mod 4. If some other value of \( D \) is passed as input, \( res \) is set to the zero polynomial.
9.20 acb_theta.h – Riemann theta functions

This module provides methods for the numerical evaluation of theta functions in any dimension $g \geq 1$. The algorithms will be detailed in the forthcoming paper [EK2023]. In the case $g = 1$, we rely on, but also improve on functionality from acb_module.h.

In the context of this module, $\tau$ or $\tau$ always denotes an element of the Siegel upper half-space $\mathbb{H}_g$, which consists of all symmetric $g \times g$ complex matrices with positive definite imaginary part. The letter $z$ denotes an element of $\mathbb{C}^g$. For each $a, b \in \{0, 1\}^g$, the Riemann theta function of characteristic $(a, b)$ is the following analytic function in $\tau \in \mathbb{H}_g$ and $z \in \mathbb{C}^g$:

$$
\theta_{a,b}(z, \tau) = \sum_{n \in \mathbb{Z}^g + \frac{1}{2}} \exp(\pi i n^T \tau + 2\pi i n^T (z + \frac{b}{2})),
$$

considering $a, b$ and $z$ as column vectors.

We encode a theta characteristic $a \in \{0, 1\}^g$ as the ulong between 0 and $2^g - 1$ that has the corresponding expansion in base 2: thus $a = (1, 0, 0)$ for $g = 3$ will be numbered 8. We also use this encoding to order vectors of theta values throughout. Similarly, a pair of characteristics $(a, b)$ is encoded as an ulong between 0 and $2^{2g} - 1$, where $a$ corresponds to the $g$ more significant bits. With these conventions, the output of acb_module_theta() is $(-\theta_1, \theta_2, \theta_0, \theta_1)$.

The main user-facing function to evaluate theta functions is acb_theta_all(). This function first reduces the input $(z, \tau)$ using the action of the Siegel modular group $\text{Sp}_{2g}(\mathbb{Z})$ on $\mathbb{C}^g \times \mathbb{H}_g$, then uses a quasi-linear algorithm to compute theta values on the reduced domain. At low precisions and when $\tau$ is reasonably reduced, one may also consider using “naive algorithms” directly, which consist in evaluating a partial sum of the theta series. The main functions to do so are acb_theta_naive_fixed_ab() and acb_theta_naive_all(). We also provide functionality to evaluate derivatives of theta functions, and to evaluate Siegel modular forms in terms of theta functions when $g = 2$.

The numerical functions in this module compute certified error bounds: for instance, if $\tau$ is represented by an acb_mat_t which is not certainly positive definite at the chosen working precision, the output will have an infinite radius. Throughout, $g$ must be at least 1 (this is not checked.)

9.20.1 Main user functions

void acb_theta_all(acb_ptr th, acb_srcptr z, const acb_mat_t tau, int sqr, slong prec)

Sets $th$ to the vector of theta values $\theta_{a,b}(z, \tau)$ or $\theta_{a,b}(z, \tau)^2$ for all $a, b \in \{0, 1\}^g$, depending on whether $sqr$ is 0 (false) or nonzero (true).

void acb_theta_naive_fixed_ab(acb_ptr th, ulong ab, acb_srcptr zs, slong nb, const acb_mat_t tau, slong prec)

void acb_theta_naive_all(acb_ptr th, acb_srcptr zs, slong nb, const acb_mat_t tau, slong prec)

Assuming that $zs$ is the concatenation of $nb$ vectors $z$ of length $g$, evaluates $\theta_{a,b}(z, \tau)$ using the naive algorithm, for either the given value of $(a, b)$ or all $(a, b) \in \{0, 1\}^{2g}$. The result $th$ will be a concatenation of $nb$ vectors of length 1 or $2^{2g}$ respectively. The user should ensure that $\tau$ is reasonably reduced before calling these functions.

void acb_theta_jet_all(acb_ptr dth, acb_srcptr z, const acb_mat_t tau, slong ord, slong prec)

Sets $dth$ to the partial derivatives with respect to $z$ up to total order $ord$ of all functions $\theta_{a,b}$ for $a, b \in \{0, 1\}^g$ at the given point $(z, \tau)$, as a concatenation of $2^{2g}$ vectors. (See below for conventions on the numbering and normalization of derivatives.)

void acb_theta_jet_naive_fixed_ab(acb_ptr dth, ulong ab, acb_srcptr z, const acb_mat_t tau, slong ord, slong prec)
void acb_theta_jet_naive_all(acb_ptr dth, acb_srcptr z, const acb_mat_t tau, slong ord, slong prec)

Sets \( dth \) to the partial derivatives with respect to \( z \) up to total order \( ord \) of \( \theta_{a,b} \) for the given (resp. all) \((a,b) \in \{0,1\}^g \) at the given point \((z,\tau)\) using the naive algorithm. The user should ensure that \( \tau \) is reasonably reduced before calling these functions.

### 9.20.2 Example of usage

The following code snippet constructs the period matrix \( \tau = i I_2 \) for \( g = 2 \), computes the associated theta values at \( z = 0 \) at 10000 bits of precision in roughly 0.1s, and prints them.

```c
#include "acb_theta.h"

int main()
{
    acb_mat_t tau;
    acb_ptr th, z;
    slong prec = 10000;

    acb_mat_init(tau, 2, 2);
    z = _acb_vec_init(2);
    th = _acb_vec_init(16);

    acb_mat_onei(tau);
    acb_theta_all(th, z, tau, 0, prec);
    _acb_vec_printd(th, 16, 5);

    acb_mat_clear(tau);
    _acb_vec_clear(z, 2);
    _acb_vec_clear(th, 16);
    flint_cleanup();
    return 0;
}
```

\[(1.1803 + 0j) +/- (2.23e-3010, 1.23e-3010j), (0.99254 + 0j) +/- (1.73e-3010, 1.23e-3010j), (0.99254 + 0j) +/- (1.73e-3010, 1.23e-3010j), (0 + 0j) +/- (1.23e-3010, 1.23e-3010j)\]

### 9.20.3 The Siegel modular group

We use the type \texttt{fmpz_mat_t} to handle matrices in \( \text{Sp}_{2g}(Z) \). In addition to the functions in this section, methods from \texttt{fmpz_mat.h} such as \texttt{fmpz_mat_equal()} can thus be used on symplectic matrices directly.

In the following functions (with the exception of \texttt{sp2gz_is_correct()} ) we always assume that the input matrix \( \text{mat} \) is square of even size \( 2g \), and write it as

\[
m = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}
\]
where \( \alpha, \beta, \gamma, \delta \) are \( g \times g \) blocks.

```c
slong sp2gz_dim(const fmpz_mat_t mat)

Returns \( g \), which is half the number of rows (or columns) of \( \text{mat} \). This is an inline function only.
```

```c
void sp2gz_set_blocks(fmpz_mat_t mat, const fmpz_mat_t alpha, const fmpz_mat_t beta, const fmpz_mat_t gamma, const fmpz_mat_t delta)

Sets \( \text{mat} \) to \( \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \). The dimensions must match.
```

```c
void sp2gz_j(fmpz_mat_t mat)

Sets \( \text{mat} \) to the symplectic matrix \( J = \begin{pmatrix} 0 & I_g \\ -I_g & 0 \end{pmatrix} \).
```

```c
void sp2gz_block_diag(fmpz_mat_t mat, const fmpz_mat_t U)

Sets \( \text{mat} \) to the symplectic matrix \( \begin{pmatrix} U & 0 \\ 0 & U^{-T} \end{pmatrix} \). We require that \( U \in \text{GL}_g(\mathbb{Z}) \).
```

```c
void sp2gz_trig(fmpz_mat_t mat, const fmpz_mat_t S)

Sets \( \text{mat} \) to \( \begin{pmatrix} I_g & S \\ S & I_g \end{pmatrix} \), where \( S \) is a symmetric \( g \times g \) matrix.
```

```c
void sp2gz_embed(fmpz_mat_t res, const fmpz_mat_t mat)

Assuming that \( \text{mat} \) is a symplectic matrix of size \( 2r \times 2r \) and \( \text{res} \) is square of size \( 2g \times 2g \) for some \( g \geq r \), sets \( \text{res} \) to the symplectic matrix

\[
\begin{pmatrix}
\alpha & I_{g-r} & \beta & 0_{g-r} \\
\gamma & 0_{g-r} & \delta & I_{g-r}
\end{pmatrix}
\]

where \( \alpha, \beta, \gamma, \delta \) are the \( r \times r \) blocks of \( \text{mat} \).
```

```c
void sp2gz_restrict(fmpz_mat_t res, const fmpz_mat_t mat)

Assuming that \( \text{mat} \) is a symplectic matrix of size \( 2g \times 2g \) and \( \text{res} \) is square of size \( 2r \times 2r \) for some \( r \leq g \), sets \( \text{res} \) to the matrix whose \( r \times r \) blocks are the upper left corners of the corresponding \( g \times g \) block of \( \text{mat} \). The result may not be a symplectic matrix.
```

```c
slong sp2gz_nb_fundamental(slong g)

Returns the number of fundamental symplectic matrices used in the reduction algorithm on \( \mathbb{H}_g \). This number is 1 when \( g = 1 \) (the \( J \) matrix) and 19 when \( g = 2 \) [Got1959]. When \( g > 2 \), a complete set of matrices defining the boundary of a fundamental domain for the action of \( \text{Sp}_{2g}(\mathbb{Z}) \) is not currently known. As a substitute, we consider two types of matrices: the 19\( g \)\(( g - 1) / 2 \) matrices obtained by mimicking the \( g = 2 \) matrices on any pair of indices between 0 and \( g - 1 \), and the \( 2^g \) matrices obtained by embedding a copy of a lower-dimensional \( J \) matrix on any subset of indices.
```

```c
void sp2gz_fundamental(fmpz_mat_t mat, slong j)

Sets \( \text{mat} \) to the \( j \)th fundamental symplectic matrix as defined above.
```

```c
int sp2gz_is_correct(const fmpz_mat_t mat)

Returns true (nonzero) iff \( \text{mat} \) is a symplectic matrix.
```

```c
int sp2gz_is_j(const fmpz_mat_t mat)

Returns true (nonzero) iff the symplectic matrix \( \text{mat} \) is the \( J \) matrix.
```

```c
int sp2gz_is_block_diag(const fmpz_mat_t mat)

Returns true (nonzero) iff the symplectic matrix \( \text{mat} \) is of block-diagonal form as in \( \text{sp2gz_block_diag()} \).
```

```c
int sp2gz_is_trig(const fmpz_mat_t mat)

Returns true (nonzero) iff the symplectic matrix \( \text{mat} \) is of trigonal form as in \( \text{sp2gz_trig()} \).
```
void acb_siegel_cocycle(acb_mat_t c, const fmpz_mat_t mat, const slong prec)

Sets c to $\gamma \tau + \delta$.

void acb_siegel_transform_cocycle_inv(acb_mat_t w, acb_mat_t c, acb_mat_t cinv, const fmpz_mat_t mat, const acb_mat_t tau, slong prec)

Sets w, c and cinv to $(\alpha \tau + \beta) (\gamma \tau + \delta)^{-1}$, $\gamma \tau + \delta$ and $(\gamma \tau + \delta)^{-1}$ respectively.

void acb_siegel_transform(acb_mat_t w, const fmpz_mat_t mat, const acb_mat_t tau, slong prec)

Sets w to $(\alpha \tau + \beta) (\gamma \tau + \delta)^{-1}$.

void acb_siegel_transform_z(acb_ptr r, acb_mat_t w, const fmpz_mat_t mat, acb_srcptr z, const acb_mat_t tau, slong prec)

Sets w to $(\alpha \tau + \beta) (\gamma \tau + \delta)^{-1}$ and $r$ to $(\gamma \tau + \delta)^{-T} z$.

void acb_siegel_cho(arb_mat_t C, const acb_mat_t tau, slong prec)

Sets C to an upper-triangular Cholesky matrix such that $\pi \text{Im}(\tau) = C^T C$. If one cannot determine that $\text{Im}(\tau)$ is positive definite at the current working precision, C is set to an indeterminate matrix.

void acb_siegel_yinv(arb_mat_t Yinv, const acb_mat_t tau, slong prec)

Sets Yinv to the inverse of Im($\tau$). If one cannot determine that Im($\tau$) is invertible at the current working precision, Yinv is set to an indeterminate matrix.

void acb_siegel_reduce(fmpz_mat_t mat, const acb_mat_t tau, slong prec)

Sets mat to a symplectic matrix such that mat $\cdot$ $\tau$ is as reduced as possible, repeatedly reducing the imaginary and real parts of $\tau$ and applying fundamental symplectic matrices. If the coefficients of $\tau$ do not have a reasonable size or if det Im($\tau$) is vanishingly small, we simply set mat to the identity.

9.20.4 The Siegel half space

We continue to denote by $\alpha, \beta, \gamma, \delta$ the $g \times g$ blocks of mat, which is always assumed to be symplectic.

Sets w, c and cinv to $(\alpha \tau + \beta) (\gamma \tau + \delta)^{-1}$, $\gamma \tau + \delta$ and $(\gamma \tau + \delta)^{-1}$ respectively.
int acb_siegel_is_reduced(const acb_mat_t tau, slong tol_exp, slong prec)

Returns true (nonzero) iff it is certainly true that \( \tau \) belongs to the reduced domain defined by the tolerance parameter \( \varepsilon = 2^{\text{tol.exp}} \). This means the following: \( |\text{Re}(\tau_j)j| < \frac{1}{2} + \varepsilon \) for all \( 0 \leq j, k < g \); the imaginary part of \( \tau \) passes \text{arb_mat_spd_is_lll_reduced()} with the same parameters; and for every matrix obtained from \text{sp2gz_fundamental()} , the determinant of the corresponding cocycle is at least \( 1 - \varepsilon \).

void acb_siegel_randtest(acb_mat_t tau, flint_rand_t state, slong prec, slong mag_bits)

Sets \( \tau \) to a random matrix in \( \mathbb{H}_g \), possibly far from being reduced.

void acb_siegel_randtest_reduced(acb_mat_t tau, flint_rand_t state, slong prec, slong mag_bits)

Sets \( \tau \) to a random reduced matrix in \( \mathbb{H}_g \) that is likely to trigger corner cases for several functions in this module.

void acb_siegel_randtest_vec(acb_ptr z, flint_rand_t state, slong g, slong prec)

Sets \( z \) to a random vector of length \( g \) that is likely to trigger corner cases for several functions in this module.

9.20.5 Theta characteristics

void acb_theta_char_get_slong(slong *n, ulong a, slong g)

Sets each entry of \( n \) to the corresponding bit of \( a \).

ulong acb_theta_char_get_a(const slong *n, slong g)

Returns the unique characteristic \( a \) such that \( n \in 2Z^g + a \).

void acb_theta_char_get_arb(arb_ptr v, ulong a, slong g)

void acb_theta_char_get_acb(acb_ptr v, ulong a, slong g)

Sets \( v \) to \( a/2 \) seen as an element of \( \mathbb{R}^g \) or \( \mathbb{C}^g \) respectively.

slong acb_theta_char_dot(ulong a, ulong b, slong g)

Returns \( \sum_{i=0}^{g-1} a_i b_i \) modulo 4 as an integer between 0 and 3, where \( a_i, b_i \) for \( 0 \leq i < g \) denote the bits of \( a \) and \( b \) respectively.

slong acb_theta_char_dot_slong(ulong a, const slong *n, slong g)

Returns \( \sum_{i=0}^{g-1} a_i n_i \) modulo 4 as an integer between 0 and 3.

void acb_theta_char_dot_acb(acb_t x, ulong a, acb_srcptr z, slong g, slong prec)

Sets \( x \) to \( \sum_{i=0}^{g-1} a_i z_i \).

int acb_theta_char_is_even(ulong ab, slong g)

Returns true iff the characteristic \( (a, b) \) is even, i.e. \( a^T b \) is divisible by 2.

int acb_theta_char_is_goepel(ulong ch1, ulong ch2, ulong ch3, ulong ch4, slong g)

Returns true iff the given characteristics define a Göpel quadruple, i.e. they are distinct even characteristics whose sum belongs to \( 2^Z \).

int acb_theta_char_is_syzygous(ulong ch1, ulong ch2, ulong ch3, slong g)

Returns true iff the given characteristics define a syzygous triple, i.e. they can be completed into a Göpel quadruple.
9.20.6 Ellipsoids: types and macros

Following [DHBH2004], naive algorithms will compute a partial sum of theta series over points \( n \) in the lattice \( \mathbb{Z}^g \) contained in certain ellipsoids, and finally add an error bound coming from the tail. We first gather methods to compute with ellipsoids themselves.

Fix an upper-triangular matrix \( C \) with positive diagonal entries (henceforth called a “Cholesky matrix”), a radius \( R \geq 0 \), a vector \( v \in \mathbb{R}^g \), and \( 1 \leq d \leq g \). Consider the ellipsoid \( E \) consisting of points \( n = (n_0, \ldots, n_{d-1}) \) satisfying \( (v + Cn)^T(v + Cn) \leq R^2 \) and such that their last coordinates \( n_d, \ldots, n_{g-1} \) are fixed. We encode \( E \) as follows: we store the endpoints and midpoint of the interval of allowed values for \( n_{d-1} \) as \texttt{slong}'s, and if \( d \geq 1 \), we store a \((d-1)\)-dimensional “child” of \( E \) for each value of \( n_{d-1} \) as another ellipsoid in a recursive way. Children are partitioned between left and right children depending on the position of \( n_{d-1} \) relative to the midpoint (by convention, the midpoint is a right child). When \( d = g \) and for a fixed Cholesky matrix \( C \), this representation uses \( \mathcal{O}(R^{g-1}) \) space for an ellipsoid of radius \( R \) containing approximately \( \mathcal{O}(R^g) \) points.

**type acb_theta_eld_struct**

**type acb_theta_eld_t**

An \texttt{acb_theta_eld_t} is an array of length one of type \texttt{acb_theta_eld_struct} encoding an ellipsoid as described above, permitting it to be passed by reference.

The following macros are available after \( E \) of type \texttt{acb_theta_eld_t} has been initialized using \texttt{acb_theta_eld_init()} below.

\texttt{acb_theta_eld_dim}(E)

Macro returning \( d \).

\texttt{acb_theta_eld_ambient_dim}(E)

Macro returning \( g \).

The following macros are available after \( E \) has been initialized and then computed using \texttt{acb_theta_eld_set()} below.

\texttt{acb_theta_eld_coord}(E, \( k \))

Macro returning the common coordinate \( n_k \) of the points in \( E \). This requires \( d \leq k < g \).

\texttt{acb_theta_eld_min}(E)

\texttt{acb_theta_eld_mid}(E)

\texttt{acb_theta_eld_max}(E)

Macros returning the minimum, midpoint, and maximum of \( n_{d-1} \) in \( E \) respectively.

\texttt{acb_theta_eld_nr}(E)

\texttt{acb_theta_eld_nl}(E)

Macros returning the number of right and left children of \( E \) respectively.

\texttt{acb_theta_eld_rchild}(E, \( k \))

\texttt{acb_theta_eld_lchild}(E, \( k \))

Macros returning a pointer to the \( k \)th right (resp. left) child of \( E \) as an \texttt{acb_theta_eld_t}.

\texttt{acb_theta_eld_nb_pts}(E)

Macro returning the number of points contained in \( E \).

\texttt{acb_theta_eld_nb_border}(E)

Macro returning the number of points in the border of \( E \), defined as follows. If \( d = 1 \), then it consists of the two points with \( n_0 \) equal to \( m - 1 \) and \( M + 1 \), where \( m \) and \( M \) are the result of \texttt{acb_theta_eld_max} and \texttt{acb_theta_eld_min} respectively. If \( d \geq 2 \), then it is the reunion of the borders of all children of \( E \). This is only used for testing.
acb_theta_eld_box(E, k)

Macro returning the smallest nonnegative integer $M_k$ such that all the points in $E$ satisfy $|n_k| \leq M_k$. This requires $0 \leq k < d$.

### 9.20.7 Ellipsoids: memory management and computations

```c
void acb_theta_eld_init(acb_theta_eld_t E, slong d, slong g)
```

Initializes $E$ as a $d$-dimensional ellipsoid in ambient dimension $g$. We require $1 \leq d \leq g$.

```c
void acb_theta_eld_clear(acb_theta_eld_t E)
```

Cleans $E$ as well as any recursive data contained in it.

```c
int acb_theta_eld_set(acb_theta_eld_t E, const arb_mat_t C, const arf_t R2, arb_srcptr v)
```

Assuming that $C$ is upper-triangular with positive diagonal entries, attempts to set $E$ as an ellipsoid as defined above, where $R2$ indicates $R^2$, and returns 1 upon success. If the ellipsoid points do not fit in $\text{slong}$’s or if the ellipsoid is unreasonably large, returns 0 instead and leaves $E$ undefined.

The following functions are available after `acb_theta_eld_set()` has been called successfully.

```c
void acb_theta_eld_points(slong *pts, const acb_theta_eld_t E)
```

Sets $pts$ to the list of all the points in $E$, as a concatenation of vectors of length $g$.

```c
void acb_theta_eld_border(slong *pts, const acb_theta_eld_t E)
```

Sets $pts$ to the list of all the points in the border of $E$.

```c
int acb_theta_eld_contains(const acb_theta_eld_t E, slong *pt)
```

Returns true (nonzero) iff $pt$ is contained in $E$. The vector $pt$ must be of length $g$.

```c
void acb_theta_eld_print(const acb_theta_eld_t E)
```

Prints a faithful description of $E$. This may be unwieldy in high dimensions.

### 9.20.8 Naive algorithms: error bounds

By [EK2023], for any $v \in \mathbb{R}^g$ and any upper-triangular Cholesky matrix $C$, and any $R$ such that $R^2 \geq \max(4, \text{ord})$, we have

$$
\sum_{n \in CZ^g + v, \|n\|^2 \geq R^2} |n|^{\text{ord}} e^{-|n|^2} \leq 2^{2g+2} R^{g-1+\text{ord}} e^{-R^2} \prod_{j=0}^{g-1} (1 + \gamma_j^{-1})
$$

where $\gamma_0, \ldots, \gamma_{g-1}$ are the diagonal coefficients of $C$. We use this to bound the contribution from the tail of the theta series in naive algorithms, and thus to find out which ellipsoid to consider at a given precision. When several vectors $z$ are present, we first reduce them to a common compact domain and use only one ellipsoid, following [DHBHS2004].

```c
void acb_theta_naive_radius(arf_t R2, arf_t eps, const arb_mat_t C, slong ord, slong prec)
```

Sets $R2$ and $\text{eps}$ such that the above upper bound for $R2$ and the given $\text{ord}$ is at most $\text{eps}$. We choose $\text{eps}$ so that the relative error on the output of the naive algorithm should be roughly $2^{-\text{prec}}$ if no cancellations occur in the sum, i.e. $\text{eps} \approx 2^{-\text{prec}} \prod_{j=0}^{g-1} (1 + \gamma_j^{-1})$.

```c
void acb_theta_naive_reduce(arb_ptr v, arb_ptr new_zs, arb_ptr as, arb_ptr cs, arb_ptr us,
    arb_srcptr zs, slong nb, const arb_mat_t tau, slong prec)
```

Given $zs$, a concatenation of $nb$ vectors of length $g$, performs the simultaneous reduction of these vectors with respect to the matrix $\tau$. This means the following. Let $0 \leq k < nb$, let $z$ denote the $k$th vector stored in $zs$, and let $X, Y$ (resp. $x, y$) be the real and imaginary parts of $\tau$ (resp. $z$). Write $Y^{-1}y = r + a$ where $a$ is an even integral vector and $r$ is bounded. (We set $a = 0$ instead if the entries of this vector have an unreasonably large magnitude.) Then
\[
\theta_{0,b}(z, \tau) = e^{\pi y T Y^{-1} y} \sum_{n \in \mathbb{Z}^g} e^{\pi i ((n-a)^T X (n-a) + 2(n-a)^T x + ir^T Y r)} e^{-\pi(n+r)^T Y(n+r)}
\]

\[= e^{\pi y T Y^{-1} y} e^{\pi i (a^T X a - 2a^T x + i r^T Y r)} \theta_{0,b}((x - X a) + i Y r, \tau).\]

The reduction of \(z\) is defined as \((x - X a) + i Y r\), which has a bounded imaginary part, and this vector is stored as the \(k^{th}\) vector of new_zs. The vector \(a\) is stored as the \(k^{th}\) vector of \(as\). The quantity \(u = \exp(\pi y T Y^{-1} y)\) is a multiplicative factor for the error bound, and is stored as the \(k^{th}\) entry of \(us\). The quantity

\[
c = u \exp(\pi i (a^T X a - 2a^T x + i r^T Y r))
\]

is a multiplicative factor for the theta values, and is stored as the \(k^{th}\) entry of \(cs\). The offset for the corresponding ellipsoid is \(u^{(k)} = Cr\) which is also bounded independently of \(k\), and \(v\) is set to the \(acb\_union()\) of the \(v^{(k)}\) for \(0 \leq k < nb\).

void \texttt{acb\_theta\_naive\_term}(acb\_res, \texttt{acb\_srcptr} \(z\), const \texttt{acb\_mat\_t} \(tau\), \texttt{slong*} \(\texttt{tup}\), \texttt{slong*} \(\texttt{n}\), \texttt{slong prec})

Sets \(res\) to \(n_0^{k_0} \cdots n_g^{k_g-1} \exp(\pi i (n^T \tau n + 2n^T z))\), where the \(k_j\) and \(n_j\) denotes the \(j^{th}\) entry in \(\texttt{tup}\) and \(\texttt{n}\) respectively. The vector \(\texttt{tup}\) may be \texttt{NULL}, which is understood to mean the zero tuple. This is only used for testing.

**9.20.9 Naive algorithms: main functions**

The main worker inside each version of the naive algorithm will process one line inside the computed ellipsoid. Before calling this worker, for fixed \(\tau\) and \(z\) and fixed coordinates \(n_1, \ldots, n_{g-1}\) defining a line inside the ellipsoid, if \(n_{min}\) are \(n_{max}\) are the endpoints of the interval of allowed values for \(n_0\), we (efficiently) compute:

- the vector \(v_1\) with entries \(\exp(\pi i j^2 \tau_{0,0})\) for \(n_{min} \leq j \leq n_{max}\),
- the vector \(v_2\) with entries \(x_j^1\) for \(n_{min} \leq j \leq n_{max}\), where

\[
x = \exp(2\pi i z_0) \prod_{k=1}^{g-1} \exp(2\pi i n_k \tau_{0,k})
\]

- the cofactor \(c \in \mathbb{C}\) given by

\[
c = \prod_{k=1}^{g-1} \exp(2\pi i n_k z_k) \prod_{1 \leq j \leq k < g} \exp(\pi i (2 - \delta_{j,k}) n_j n_k \tau_{j,k}).
\]

This allow us to use \texttt{acb\_dot()} in the workers while maintaining reasonable memory costs, and to use an average of strictly less than two complex multiplications per lattice point as \(R \to \infty\). Moreover, these multiplications are performed at only a fraction of the full precision for lattice points far from the ellipsoid center. Different versions of the naive algorithm will rely on slightly different workers, so introducing a function pointer type is helpful to avoid code duplication.

The methods in this section are only used when \(g \geq 2\): when \(g = 1\), the naive algorithms will call functions from \texttt{acb\_modular.h} directly.

type \texttt{acb\_theta\_naive\_worker\_t}

A function pointer type. A function \texttt{worker} of this type has the following signature:
void worker(acb_ptr th,acb_srcptr v1,acb_srcptr v2,const slong *precs,slong len,const acb_t c,const slong *coords,slong ord,slong g,slong prec,slong fullprec)

where:
- th denotes the output vector of theta values to which terms will be added,
- v1, v2 and c are precomputed as above,
- precs contains working precisions for each term \( n_{\text{min}} \leq j \leq n_{\text{max}} \),
- \( \text{len} = n_{\text{max}} - n_{\text{min}} + 1 \) is the common length of v1, v2 and precs,
- coords is \((n_{\text{min}}, n_{1}, \ldots, n_{g-1})\),
- ord is the maximal derivation order,
- prec is the working precision for this line inside the ellipsoid, and finally
- fullprec is the working precision for summing into th.

void acb_theta_naive_worker(acb_ptr th,slong len,acb_srcptr zs,slong nb,const acb_mat_t tau,acb_theta_eld_t E,slong ord,slong prec,acb_theta_naive_worker_t worker)

Returns the naive algorithm by calling worker on each line in the ellipsoid \( E \). The argument zs is a concatenation of nb vectors \( z \in \mathbb{C}^g \), len is the number of theta values computed by worker for each \( z \), and ord is passed as an argument to worker. No error bound coming from the tail is added. Considering several vectors \( z \) at the same time allows for a faster computation of \( \theta_{a,b}(z,\tau) \) for many values of \( z \) and a fixed \( \tau \), since exponentials of the entries of \( \tau \) can be computed only once.

void acb_theta_naive_00(acb_ptr th,acb_srcptr zs,slong nb,const acb_mat_t tau,slong prec)

void acb_theta_naive_0b(acb_ptr th,acb_srcptr zs,slong nb,const acb_mat_t tau,slong prec)

Evaluates either \( \theta_{0,0}(z(k),\tau) \), or alternatively \( \theta_{0,b}(z(k),\tau) \) for each \( b \in \{0,1\}^g \), for each \( 0 \leq k < nb \).

The result th will be a concatenation of nb vectors of length 1 or 2^\( g \) respectively.

The associated worker performs one acb_dot() operation.

void acb_theta_naive_fixed_a(acb_ptr th, ulong a,acb_srcptr zs,slong nb,const acb_mat_t tau,slong prec)

Evaluates \( \theta_{a,b}(z(k),\tau) \) for all \((a,b)\) where \( b \in \{0,1\}^g \) and a is fixed, for each \( 0 \leq k < nb \). The result th will be a concatenation of nb vectors of length 2^\( g \).

We reduce to calling acb_theta_naive_0b() by writing

\[
\theta_{a,b}(z,\tau) = \exp(\pi i a^T \frac{\tau}{2}) \exp(\pi i a^T (z + \frac{b}{2})) \theta_{0,b}(z + \frac{\tau}{2},\tau).
\]

We proceed similarly in acb_theta_naive_fixed_0b() and acb_theta_naive_all(), using acb_theta_naive_00() for the former.

### 9.20.10 Naive algorithms for derivatives

This section contains methods to evaluate the successive partial derivatives of \( \theta_{a,b}(z,\tau) \) with respect to the \( g \) coordinates of \( z \). Derivatives with respect to \( \tau \) are accounted for by the heat equation

\[
\frac{\partial \theta_{a,b}}{\partial \tau_{j,k}} = \frac{1}{2 \pi i (1 + \delta_{j,k})} \frac{\partial^2 \theta_{a,b}}{\partial z_j \partial z_k}.
\]

We encode tuples of derivation orders, henceforth called “derivation tuples”, as vectors of type slong and length \( g \). In agreement with acb_modular.h, we also normalize derivatives in the same way as in the Taylor expansion, so that the tuple \((k_0,\ldots,k_{g-1})\) corresponds to the differential operator...
\[
\frac{1}{k_0! \cdots k_{g-1}!} \frac{\partial^{|k|}}{\partial z_0^{k_0} \cdots \partial z_{g-1}^{k_{g-1}}}.
\]

where \(|k| := \sum k_i\). We always consider all derivation tuples up to a total order \(\text{ord}\), and order them first by their total order, then reverse-lexicographically. For example, in the case \(g = 2\), the sequence of orders is \((0,0), (1,0), (0,1), (2,0), (1,1), \ldots\).

The naive algorithms for derivatives will evaluate a partial sum of the differentiated series:

\[
\sum_{\mathbf{n} \in \mathbb{Z}^g + \frac{1}{2}} n_0^{k_0} \cdots n_{g-1}^{k_{g-1}} e^{\pi in \cdot \tau + 2\pi in \cdot (z+b/2)}.
\]

**slong acb_theta_jet_nb(slong ord, slong g)**

Returns the number of derivation tuples with total order at most \(\text{ord}\). The result will be zero if \(\text{ord}\) is negative.

**slong acb_theta_jet_total_order(const slong *tup, slong g)**

Returns the total derivation order for the given tuple \(\text{tup}\) of length \(g\).

**void acb_theta_jet_tuples(slong *tups, slong ord, slong g)**

Sets \(\text{tups}\) to the concatenation of all derivation tuples up to total order \(\text{ord}\).

**slong acb_theta_jet_index(const slong *tup, slong g)**

Returns \(n\) such that \(\text{tup}\) is the \(n^{th}\) derivation tuple of length \(g\).

**void acb_theta_jet_mul(acb_ptr res, acb_srcptr v1, acb_srcptr v2, slong ord, slong g, slong prec)**

Sets \(\text{res}\) to the vector of derivatives of the product \(f_1 f_2\), assuming that \(v1\) and \(v2\) contains the derivatives of \(f_1\) and \(f_2\) respectively.

**void acb_theta_jet_compose(acb_ptr res, acb_srcptr v, const acb_mat_t N, slong ord, slong prec)**

Sets \(\text{res}\) to the vector of derivatives of the composition \(f(Nz)\), assuming that \(v\) contains the derivatives of \(f\) at the point \(Nz\).

**void acb_theta_jet_exp_pi_i(acb_ptr res, arb_srcptr a, slong ord, slong g, slong prec)**

Sets \(\text{res}\) to the vector of derivatives of the function \(\exp(\pi i (a_0 z_1 + \cdots + a_{g-1} z_{g-1}))\) at \(z = 0\), where \(a_0, \ldots, a_{g-1}\) are the entries of \(a\).

**void acb_theta_jet_naive_radius(arf_t R2, arf_t eps, arb_srcptr v, const arb_mat_t C, slong ord, slong prec)**

Assuming that \(C\) is the upper-triangular Cholesky matrix for \(\pi Y\) and \(v = CY^{-1} y\) where \(y, Y\) are the imaginary parts of \(z\) and \(\tau\) respectively, returns \(R2\) and \(\text{eps}\) so that, when summing the above series on terms \(n \in \mathbb{Z}^g\) such that \((v + Cn)^T (v + Cn) \leq R^2\), the absolute value of the tail of the series (before multiplying by the leading factor \((2\pi i)^{|k|} e^{\pi Y^{-1} Y^{-1} y}\), see below) will be bounded above by \(\text{eps}\), for any derivation tuple \(k\) with \(|k| \leq \text{ord}\).

We can rewrite the above sum as

\[
(2\pi i)^{|k|} e^{\pi Y^{-1} Y^{-1} y} \sum_{n \in \mathbb{Z}^g + \frac{1}{2}} n_0^{k_0} \cdots n_{g-1}^{k_{g-1}} e^{\pi i n \cdot \tau} e^{-\pi (n + Y^{-1} y)^T Y (n + Y^{-1} y)}.
\]

We ignore the leading multiplicative factor. Writing \(m = Cn + v\), we have

\[
n_0^{k_0} \cdots n_{g-1}^{k_{g-1}} \leq (\|C^{-1}\|_\infty \|n\|_2 + \|Y^{-1} y\|_\infty)^{|k|}.
\]
Using the upper bound from \texttt{acb_theta_naive_radius()}, we see that the absolute value of the tail of the series is bounded above by

\[
(||C^{-1}||_{\infty} R + ||Y^{-1} y||_{\infty})|k|2^{q+2}R^q e^{-R^2} \prod_{j=0}^{q-1} (1 + \gamma_j^{-1}).
\]

Thus, we proceed as follows. We first compute \(R2\) and \(\text{eps}\) using \texttt{acb_theta_naive_radius()} with \(\text{ord} = 0\). If \(R \leq ||Y^{-1} y||_{\infty}/||C^{-1}||_{\infty}\), we simply multiply \(\text{eps}\) by \(\max\{1,2||Y^{-1} y||_{\infty}\}\)\text{ord}.

Otherwise, we compute \(R2\) and \(\text{eps}\) using \texttt{acb_theta_naive_radius()} with the given value of \(\text{ord}\). We can then set \(R2\) to the maximum of \(R2\) and \(||Y^{-1} y||_{\infty}/||C^{-1}||_{\infty}\), and multiply \(\text{eps}\) by \(\max\{1,2||C^{-1}||\}\text{ord}\).

\begin{verbatim}
void acb_theta_jet_naive_00(acb_ptr dth, acb_srcptr z, const acb_mat_t tau, slong ord, slong prec)
Sets \(dth\) to the vector of derivatives of \(\theta_{a,0}\) at the given point \((z,\tau)\) up to total order \(\text{ord}\).

In \texttt{acb_theta_jet_naive_fixed_ab()}, we reduce to this function using the same formula as in \texttt{acb_theta_jet_naive_fixed_ab()}, making suitable linear combinations of the derivatives.

In \texttt{acb_theta_jet_naive_all()}, we instead use an ellipsoid to encode points in \(\mathbb{Z}^g\), and divide \(\tau\) by \(4\) and \(z\) by \(2\) to sum the correct terms. The bounds output by \texttt{acb_theta_jet_naive_radius()} are still valid, since this just has the effect of multiplying \(||C^{-1}||\) and each \(\gamma_j^{-1}\) by \(2\).

\end{verbatim}

\begin{verbatim}
void acb_theta_jet_error_bounds(arb_ptr err, acb_srcptr z, const acb_mat_t tau, acb_srcptr dth, slong ord, slong prec)
Assuming that \(dth\) contains the derivatives of a function \(\theta_{a,b}\) up to total order \(\text{ord} + 2\), sets \(\text{err}\) to a vector with the following property. Let \((z_0,\tau_0)\) be the midpoint of \((z,\tau)\), and let \((z_1,\tau_1)\) be any point inside the ball specified by the given \(z\) and \(\tau\). Then the vectors of derivatives of \(\theta_{a,b}\) at \((z_0,\tau_0)\) and \((z_1,\tau_1)\) up to total order \(\text{ord}\) differ by at most \(\text{err}\) elementwise.

\end{verbatim}

### 9.20.11 Quasi-linear algorithms: presentation

We refer to [EK2023] for a detailed description of the quasi-linear algorithm implemented here. In a nutshell, the algorithm relies on the following duplication formula: for all \(z, z' \in \mathbb{C}^g\) and \(\tau \in \mathbb{H}_g\),

\[
\theta_{a,0}(z,\tau)\theta_{a,0}(z',\tau) = \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a',0}(z + z', 2\tau)\theta_{a+a',0}(z - z', 2\tau).
\]

In particular,

\[
\theta_{a,0}(z,\tau)^2 = \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a',0}(2z, 2\tau)\theta_{a+a',0}(0, 2\tau),
\]

\[
\theta_{a,0}(0,\tau)\theta_{a,0}(z,\tau) = \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a',0}(z, 2\tau)\theta_{a+a',0}(z, 2\tau),
\]

\[
\theta_{a,0}(0,\tau)^2 = \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a',0}(0, 2\tau)\theta_{a+a',0}(0, 2\tau).
\]

Applying one of these duplication formulas amounts to taking a step in a (generalized) AGM sequence. These formulas also have analogues for all theta values, not just \(\theta_{a,0}\): for instance, we have

\[
\theta_{a,b}(0,\tau)^2 = \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} (-1)^{a'\tau}e^{\theta_{a',0}(0, 2\tau)}\theta_{a+a',0}(0, 2\tau).
\]
Suppose that we wish to compute $\theta_{a,0}(0, \tau)$ for all $a \in \{0, 1\}^g$ and a reduced matrix $\tau \in \mathbb{H}_g$. Applying the last formula $n$ times, we reduce to evaluating $\theta_{a,0}(0, 2^n\tau)$. We expect that the absolute value of this complex number is roughly $\exp(-d^2)$ for $d = 2^n \text{Dist}_\tau(0, \mathbb{Z}^g + \frac{a}{2})$, where $\text{Dist}_\tau$ denotes the distance in $\mathbb{R}^g$ attached to the quadratic form $\text{Im}(\tau)$. Provided that $n \geq \log_2(\text{prec})$, we have to sum only $O_g(1)$ terms in the naive algorithm to evaluate $\theta_{a,0}(0, 2^n\tau)$ at “shifted absolute precision” $\text{prec}$, i.e. absolute precision $\text{prec} + d^2/\log(2)$.

In order to recover $\theta_{a,0}(0, \tau)$, we then perform $n$ AGM steps. Assuming that each $|\theta_{a,0}(0, 2^k\tau)|$ is indeed of the expected order of magnitude, we can ensure that the precision loss is $O_g(1)$ bits at each step in terms of shifted absolute precision, and we can calculate the correct sign choices of square roots at each step with the naive algorithm. However, depending on the choice of $\tau$, this assumption may not always hold.

We make the following adjustments to make the algorithm work for all $\tau$, as well as for theta values at $z \neq 0$:

- If we discover (after applying the naive algorithm) that some value $\theta_{a,0}(0, 2^k\tau)$ is too small, we introduce an auxiliary real vector $t$. At each step, starting from $\theta_{a,0}(0, 2^{k+1}\tau)$, $\theta_{a,0}(2^{k+1}t, 2^{k+1}\tau)$ and $\theta_{a,0}(2^{k+2}t, 2^{k+2}\tau)$, we compute $\theta_{a,0}(2^kt, 2k\tau)$ and $\theta_{a,0}(2^{k+1}t, 2^{k+1}\tau)$ using square roots (second formula above), then $\theta_{a,0}(0, 2^k\tau)$ using divisions (third formula). For a huge majority of such $t$, none of the values $\theta_{a,0}(2^kt, 2k\tau)$ and $\theta_{a,0}(2^{k+1}t, 2^{k+1}\tau)$ will be too small [EK2023]. In practice, we choose $t$ at random and obtain a probabilistic algorithm with a negligible failure probability.

- When computing $\theta_{a,0}(z, \tau)$ for a nonzero $z$, we compute $\theta_{a,0}(0, 2^k\tau)$ and $\theta_{a,0}(2^kz, 2^k\tau)$ using the second and fourth formulas at each step. We actually replace each occurrence of $\theta_{a,0}(z, \tau)$ by $e^{-\pi y^2} r^{-1} \theta_{a,0}(z, \tau)$, as the absolute values of the latter quantities do not increase as $y$ gets farther from zero, and they still satisfy the duplication formulas.

- These two techniques can be combined by evaluating theta values at the six vectors $2^kv$ for $v = 0, t, 2t, z, z + t, z + 2t$. Note that we only have to compute $\theta_{a,0}(2^kz, 2^k\tau)$ at the last step $k = 0$.

- Finally, if the eigenvalues of $\text{Im}(\tau)$ have different orders of magnitude, then the ellipsoid we have to sum on for the naive algorithm will become very thin in one direction while still being thick in other directions. In such a case, we can split the total sum and compute $O(1)$ theta values in a lower dimension. This increases the efficiency of the algorithm while ensuring that the absolute precisions we consider are always in $O(\text{prec})$.

### 9.20.12 Quasi-linear algorithms: distances

```c
void acb_theta_dist_pt(arb_t d, arb_srcptr v, const arb_mat_t C, slong *n, slong prec)

Sets $d$ to $\|v + Cn\|^2$ for the usual Euclidean norm.
```

```c
void acb_theta_dist_lat(arb_t d, arb_srcptr v, const arb_mat_t C, slong prec)

Sets $d$ to Dist($v, C\mathbb{Z}^g$)² for the usual Euclidean norm. We first compute an upper bound on the result by considering the $2^g$ vectors obtained by rounding the entries of $C^{-1}v$ to integers up or down, then compute an ellipsoid to find the minimum distance.
```

```c
void acb_theta_dist_a0(arb_ptr d, arb_srcptr z, const arb_mat_t tau, slong prec)

Sets $d$ to the vector containing Dist($C \cdot (Y^{-1}y + \frac{a}{2}), C \cdot \mathbb{Z}^g$)² for $a \in \{0, 1\}^g$, where $y, Y$ are the imaginary parts of $z, \tau$ respectively and $C$ is the upper-triangular Cholesky matrix for $\pi Y$. The $a\text{th}$ entry of $d$ is also Dist,$t, (-Y^{-1}y, \mathbb{Z}^g + \frac{a}{2})$, where Dist,$t$ denotes the distance attached to the quadratic form $\text{Im}(\tau)$.
```

```c
slong acb_theta_dist_addprec(const arb_t d)

Returns an integer that is close to $d$ divided by $\log(2)$ if $d$ is finite and of reasonable size, and otherwise returns 0.
```
9.20.13 Quasi-linear algorithms: AGM steps

void **acb_theta_agm_hadamard**(acb_ptr res, acb_srcptr a, slong g, slong prec)

Sets res to the product of the Hadamard matrix \( \left( \frac{1}{\sqrt{2}} \right)^g \) and the vector \( a \). Both res and \( a \) must be vectors of length \( 2^g \). In other words, for each \( k \in \{0,1\}^g \), this sets the \( k \)-th entry of res to \( \sum_{j \in \{0,1\}^*} (-1)^{a_j} a_j \).

void **acb_theta_agm_sqrt**(acb_ptr res, acb_srcptr a, acb_srcptr rts, slong nb, slong prec)

Sets the \( k \)-th entry \( r_k \) of res for \( 0 \leq k < nb \) to a square root of the corresponding entry \( a_k \) of \( a \). The choice of sign is determined by rts: each \( r_k \) will overlap the corresponding entry of rts but not its opposite. Exceptional cases are handled as follows: if both square roots of \( a_k \) overlap rts, then \( r_k \) is set to their \( \texttt{acb_union()} \); if none overlaps rts, then \( r_k \) is set to an indeterminate value.

void **acb_theta_agm_mul**(acb_ptr res, acb_srcptr a1, acb_srcptr a2, slong g, slong prec)

For each \( 0 \leq k < 2^g \), sets the \( k \)-th entry of res to \( 2^{-g} \sum_{b \in \{0,1\}^*} a_{1,b} a_{2,k+b} \), where addition is meant in \( (\mathbb{Z}/2^g) \) (a bitwise xor).

Following [LT2016], we apply the Hadamard matrix twice with multiplications in-between. This causes precision losses when the absolute values of the entries of \( a_1 \) and/or \( a_2 \) are of different orders of magnitude. This function is faster when \( a_1 \) and \( a_2 \) are equal as pointers, as we can use squarings instead of multiplications.

void **acb_theta_agm_mul_tight**(acb_ptr res, acb_srcptr a0, acb_srcptr d0, arb_srcptr d, slong g, slong prec)

Assuming that \( d \theta \) and \( d \) are obtained as the result of \( \texttt{acb_theta_dist_a0()} \) on \((0, \tau)\) and \((z, \tau)\) respectively, performs the same computation as \( \texttt{acb_theta_agm_mul()} \) on the vectors \( a_0 \) and \( a \) with a different management of error bounds. The resulting error bounds on res will be tighter when the absolute value of \( a_k \) is roughly \( e^{-d_k} \) for each \( 0 \leq k < 2^g \), and similarly for \( a_0 \) and \( d \).

When \( g > 1 \), we manage the error bounds as follows. We compute \( m, \varepsilon \) such that the following holds: for each \( 0 \leq k < nb \), if \( d_k \) (resp. \( a_k \)) denotes the \( k \)-th entry of \( d \) (resp. \( a \)), then the absolute value of \( a_k \) is at most \( m \cdot e^{-d_k} \) and the radius of the complex ball \( a_k \) is at most \( \varepsilon \cdot e^{-d_k} \). We proceed similarly on \( a_0 \) and \( d_0 \) to obtain \( m_0, \varepsilon_0 \). Then we call \( \texttt{acb_theta_agm_mul()} \) on the midpoints of \( a_0 \) and \( a \) at a higher working precision, and finally add \( e^{-d_k} (m_0 \varepsilon + m_0 \varepsilon_0 + \varepsilon \varepsilon_0) \) to the error bound on the \( k \)-th entry of res. This is valid for the following reason: keeping notation from \( \texttt{acb_theta_dist_a0()} \), for each \( b \in \{0,1\}^g \), the sum

\[
\text{Dist}_\tau(-Y^{-1} y, Z^g + \frac{k}{2})^2 + \text{Dist}_\tau(-Y^{-1} y, Z^g + \frac{k+k}{2})^2
\]

is at most \( \text{Dist}_\tau(-Y^{-1} y, Z^g + \frac{k}{2})^2 \) by the parallelogram identity.

9.20.14 Quasi-linear algorithms: main functions

The functions in this section will work best when \( \tau \) lies in the reduced domain, however \( \text{Im}(\tau) \) may have large eigenvalues.

**type acb_theta_ql_worker_t**

A function pointer type. A function worker of this type has the following signature:

```c
int worker(acb_ptr th, acb_srcptr t, acb_srcptr z, arb_srcptr d0, arb_srcptr d, const acb_mat_t tau, slong guard, slong prec)
```

Such a worker will attempt to set \( th \) to the values \( \theta_{a,0}(v, \tau) \) for \( v = 0, t, 2t, z, z + t, z + 2t \) and \( a \in \{0,1\}^g \) at shifted absolute precision \( prec \), and return 1 on success and 0 on failure. The vectors \( d \theta \) and \( d \) must contain the result of \( \texttt{acb_theta_dist_a0()} \) on \((0, \tau)\) and \((z, \tau)\). If \( z = 0, t = 0, \) or both, we only compute 3, 2, or 1 vectors of \( 2^g \) values respectively.
Two functions of this type are available: \texttt{acb_theta_ql_a0_naive()} and the main function \texttt{acb_theta_ql_a0()}. Using function pointers allows us to write independent test code for the main workhorses \texttt{acb_theta_ql_a0_steps()} and \texttt{acb_theta_ql_a0_split()} below.

\begin{verbatim}
int acb_theta_ql_a0_naive(acb_ptr th, acb_srcptr t, acb_srcptr z, arb_srcptr d0, arb_srcptr d,
const acb_mat_t tau, slong guard, slong prec)

Follows the specifications of a function of type \texttt{acb_theta_ql_worker_t} using the naive algorithm only. The return value is 1 iff the output vector \textit{th} contains finite values.

int acb_theta_ql_a0_split(acb_ptr th, acb_srcptr t, acb_srcptr z, arb_srcptr d, const acb_mat_t tau, slong s, slong guard, slong prec, \texttt{acb_theta_ql_worker_t} worker)

Follows the specifications of a function of type \texttt{acb_theta_ql_worker_t}, except for the additional arguments \textit{s} and \texttt{worker}. We split the theta series according to the first \textit{s} coordinates of \textit{n} \in \mathbb{Z}^g, writing \textit{n} = (n_0,n_1) where \textit{n}_0 \in \mathbb{Z}^s and \textit{n}_1 \in \mathbb{Z}^{g-s}. We must have 1 \leq \textit{s} \leq g - 1. Then \texttt{worker} is called to evaluate the sum corresponding to each \textit{n}_1. The return value is 1 iff all the calls to \texttt{worker} succeed.

For each 0 \leq \textit{a} < 2^g, we compute \textit{R}_2 and \textit{eps} as in \texttt{acb_theta_naive_radius()} at shifted absolute precision \textit{prec}. Note that \(n^T \Im(\tau)n \geq \|C_1 n_1\|^2\), where \(C_1\) denotes the lower-right block of \(C\) of dimensions \((g - s) \times (g - s)\). Thus, in order to compute \(\theta_{\textit{a},0}(z; 2^g \tau)\) at shifted absolute precision \textit{prec}, it is enough to consider those \(n_1 \in \mathbb{Z}^{g-s}\) in an ellipsoid \(E_1\) of radius \(R_2\) for the Cholesky matrix \(C_1\). This ellipsoid is meant to contain very few points, and we list all of them. Then, for a given choice of \(n_1\), the sum of the corresponding terms in the theta series is

\[
e^{\pi i (\tau_0 + \tau_1)} \theta_{\textit{a},0}(z_0 + x(n_1 + \frac{\textit{a}}{2} \tau_1), \tau_0),
\]

where \(\tau = \left(\begin{array}{c}
\tau_0 \\
\frac{x}{2} \\
\tau_1
\end{array}\right)\) and \(z = (z_0, z_1)\). When calling \texttt{worker}, we adjust the shifted absolute precision according to the distance between \(n_1\) and the center of \(E_1\).

\begin{verbatim}
int acb_theta_ql_a0_steps(acb_ptr th, acb_srcptr t, acb_srcptr z, arb_srcptr d0, arb_srcptr d,
const acb_mat_t tau, slong nb_steps, slong s, slong guard, slong prec,
acb_theta_ql_worker_t worker)

Follows the specifications of a function of type \texttt{acb_theta_ql_worker_t}, except for the additional arguments \textit{nb_steps}, \textit{s} and \texttt{worker}. We first compute low-precision approximations (more precisely, at shifted absolute precision \textit{guard}) of the square roots we must take to perform \textit{nb_steps} AGM steps; we hope that none of these approximations contains zero. Then we call \texttt{acb_theta_ql_a0_naive()} or \texttt{acb_theta_ql_a0_split()} (with the given \texttt{worker}) depending on whether \textit{s} is zero or not, and finally perform the AGM steps. The return value is 1 iff each subprocedure succeeds.

The user should ensure that the eigenvalues of \(2^{\textit{nb_steps}} \Im(\tau)\) are not too large when calling this function.

\begin{verbatim}
slong acb_theta_ql_a0_nb_steps(const acb_mat_t C, slong s, slong prec)

Returns an integer \textit{n} such that \(2^n \gamma_2 \approx \textit{prec}\) where \(\gamma_0, \ldots, \gamma_{g-1}\) denote the diagonal coefficients of \(C\). This \textit{n} is meant to be the number of AGM steps to use in \texttt{acb_theta_ql_a0_steps()}, and its precise value is chosen to optimize performance. We require 0 \leq \textit{s} < \textit{g}.

int acb_theta_ql_a0(acb_ptr th, acb_srcptr t, acb_srcptr z, arb_srcptr d0, arb_srcptr d,
const acb_mat_t tau, slong guard, slong prec)

Follows the specifications of a function of type \texttt{acb_theta_ql_worker_t}.

We first decide how many AGM steps we should use and whether we should use the splitting strategy. Then we run \texttt{acb_theta_ql_a0_steps()} on the midpoints of \(t, z\) and \(\tau\) at a slightly higher precision to account for precision losses in the duplication formulas, using a recursive call to \texttt{acb_theta_ql_a0()} as \texttt{worker}. If the return value is 1, we finally compute provable error bounds on the result using \texttt{acb_theta_jet_naive_fixed_ab()} and \texttt{acb_theta_jet_error_bounds()}.
\end{verbatim}
\end{verbatim}
The function \texttt{acb\_theta\_ql\_a0()} may fail for an unlucky choice of auxiliary vector \( t \) or when \texttt{guard} is too small. Thus, we implement a probabilistic algorithm where we gradually increase \texttt{guard} and first choose \( t = 0 \), then make a random choice of \( t \) at each step.

\begin{verbatim}
slong acb_theta_ql_reduce(acb_ptr new_z, acb_t c, arb_t u, slong *n1, acb_srcptr z, const acb_mat_t tau, slong prec)
Sets new\_z, c, u, n1 and returns \(-1 \leq s \leq g \) such that the following holds. If \( s \geq 0 \) is returned, then \( z' = \text{new\_z} \) is a vector of length \( s \) and \( n1 \) is a vector of length \( g - s \), and for each characteristic \((a, b)\), we have (borrowing notation from \texttt{acb\_theta\_ql\_a0\_split()}) : either

\[ |\theta_{a,b}(z,\tau) - c_{n1}^{T} b \cdot \theta_{a0,b0}(z',\tau0)| \leq u \]

when the last \( g - s \) coordinates of \( a \) equal \( n1 \) modulo 2, or

\[ |\theta_{a,b}(z,\tau)| \leq u \]

otherwise. If \( s = -1 \) is returned, then \( n1, c \) and \texttt{new\_z} are left undefined and we have \( \theta_{a,b}(z,\tau) \leq u \) for all characteristics \((a, b)\). This filters out very large eigenvalues of \texttt{Im(\tau)} that have a negligible impact on theta values but would give rise to unreasonable choices of precisions in the final duplication formula for computing all theta values \( \theta_{a,b} \).

This works as follows. We first compute \( R2 \) and \texttt{eps} as in \texttt{acb\_theta\_naive\_radius()} , then set \( c, u \) and \texttt{new\_z} as in \texttt{acb\_theta\_naive\_reduce()} in dimension \( g \). We then set \( s \) such that the ellipsoid \( E \) of radius \( R2 \) that we are interested in is either empty or contains points whose \( g - s \) last coordinates are fixed. In the former case, we return \( s = -1 \). Now assume that \( E \) is not empty, let \( n1 \) be the vector of these fixed last \( g - s \) coordinates, and let \( a1 \in \{0,1\}^{g-s} \) be the corresponding characteristic. We can then write the sum defining \( \theta_{a,b} \) over \( E \) as

\[ e^{\pi i (\frac{a}{2} T \tau_1 n_1 + n_1 T (z_1 + \frac{b}{2} z_1))} \sum_{n0 \in E0 \cap (\mathbb{Z}^{T} + \frac{b0}{2})} e^{\pi i (n_0 T \tau_0 n_0 + 2n_0 T (z_0 + z \frac{b_0}{2} + \frac{b}{2} z_1))} \]

if the last \( g - s \) coordinates of \( a \) are equal to \( n1 \) modulo 2; the sum is zero otherwise. Thus we can set \( z' \) to \( z_0 + x \frac{b_0}{2} \) and multiply \( c \) by \( \exp(\pi i (\frac{a}{2} T \tau_1 n_1 + n_1 T z_1)) \).

\begin{verbatim}
void acb_theta_ql_all(acb_ptr th, acb_srcptr z, const acb_mat_t tau, int sqr, slong prec)
Sets \( \texttt{th} \) to the collection of \( \theta_{a,b}(z,\tau) \) or \( \theta_{a,b}(z,\tau)^2 \) for all \( a, b \in \{0,1\}^{g} \), depending on whether \texttt{sqr} is 0 (false) or nonzero (true).

After calling \texttt{acb\_theta\_ql\_reduce()}, we generally use the duplication formula on the result of \texttt{acb\_theta\_ql\_a0()}. When \texttt{sqr} is zero, we add a final square-root step.
\end{verbatim}

### 9.20.15 Quasi-linear algorithms: derivatives

We implement an algorithm for derivatives of theta functions on the reduced domain based on finite differences. Consider the Taylor expansion:

\[ \theta_{a,b}(z + h, \tau) = \sum_{k \in \mathbb{Z}^{g}, k \geq 0} a_k b_0^k \cdots b_{g-1} h_{g-1}^k. \]

If one chooses \( h = h_n = (\varepsilon \zeta^{n_0}, \ldots, \varepsilon \zeta^{n_{g-1}}) \) where \( \varepsilon > 0 \) and \( \zeta \) is a primitive \( m^{th} \) root of unity and lets \( n \) run through all vectors in \( \{0, \ldots, m - 1\}^{g} \), then taking a discrete Fourier transform of the resulting values will compute the individual Taylor coefficient for each derivation tuple that is bounded by \( m - 1 \) elementwise. A constant proportion, for fixed \( g \), of this set consists of all tuples of total order at most \( m - 1 \). More precisely, fix \( p \in \mathbb{Z}^{g} \). Then
\[ \sum_{n \in \{0, \ldots, m-1\}^g} \zeta^{-n^T \eta} \theta_{a,b}(z + h_n, \tau) = m^g \sum_{k \in \mathbb{Z}^g, k \geq 0, \ k = p \mod m} a_k \varepsilon^{k}. \]

We obtain an upper bound on the tail of this series from the Cauchy integration formula: if \(|\theta_{a,b}(z, \tau)| \leq c\) uniformly on a ball of radius \(\rho\) centered in \(z\) for \(\|z\|_\infty\), then the sum is \(m^g(a_p \varepsilon^{p} + T)\) with

\[ |T| \leq 2c \varepsilon^{p+m} \rho^m. \]

Since we divide by \(\varepsilon^{p}\) to get \(a_p\), we will add an error of \(2c \varepsilon^{m+p} \rho^{m+p}\) to the result of the discrete Fourier transform.

```c
void acb_theta_jet_ql_bounds(arb_t c, arb_t rho, acb_srcptr z, const acb_mat_t tau, slong ord)

Sets \(c\) and \(rho\) such that on every ball centered at (a point contained in) \(z\) of radius \(rho\), the functions \(|\theta_{a,b}|\) for all characteristics \((a,b)\) are uniformly bounded by \(c\). The choice of \(rho\) is tuned to get interesting upper bounds on derivatives of \(\theta_{a,b}\) up to order \(ord\).

We proceed as follows. First, we compute \(c_0, c_1, c_2\) such that for any choice of \(rho\), one can take \(c = c_0 \exp((c_1 + c_2 \rho)^2)\) above. We can take

\[
c_0 = 2^g \prod_{j=0}^{g-1} (1 + 2 \gamma_j^{-1}),
\]

\[
c_1 = \sqrt{\pi y T^{-1} y},
\]

\[
c_2 = \sup_{\|x\|_\infty \leq 1} \sqrt{\pi x T \text{Im}(\tau)^{-1} x}.
\]

One can easily compute an upper bound on \(c_2\) from the Cholesky decomposition of \(\pi \text{Im}(\tau)^{-1}\). We then look for a value of \(rho\) that minimizes \(\exp((c_1 + c_2 \rho)^2)/\rho^{2m+1}\) where \(m = ord + 1\), i.e. we set \(rho\) to the positive root of \(2c \varepsilon^{m}(c_1 + c_2 \rho) = 2m - 1\).

```c
void acb_theta_jet_ql_radius(arb_t eps, arb_t err, const arb_t c, const arb_t rho, slong ord, slong g, slong prec)

Sets \(eps\) and \(err\) to be a suitable radius and error bound for computing derivatives up to total order \(ord\) at precision \(prec\), given \(c\) and \(rho\) as above.

We set \(\text{varepsilon}\) such that \((2g)^{1/m} \varepsilon \leq \rho\) and \(2c \varepsilon^{m} / \rho^{m+|p|} \leq 2^{-prec}\) where \(m = ord + 1\). We also set \(err\) to \(2^{-prec}\).

```c
void acb_theta_jet_qlFiniteDiff(arb_ptr dth, const arb_t eps, const arb_t err, arb_srcptr val, slong ord, slong g, slong prec)

Assuming that \(val\) contains the values \(\theta_{a,b}(z + h_n, \tau)\) where \(h_n = (\varepsilon \zeta^n, \ldots, \varepsilon \zeta^{n-1})\) for a root of unity \(\zeta\) of order \(ord + 1\), and assuming that \(eps\) and \(err\) have been computed as in \(\text{acb_theta_jet_ql_radius()}\), sets \(dth\) to the vector of partial derivatives of \(\theta_{a,b}\) at \((z, \tau)\) up to total order \(ord\). The vector \(val\) should be indexed in lexicographic order as in \(\text{acb_dft()}\), i.e. writing \(j = a_g \cdots a_1\) in basis \(m\), the \(j\)th entry of \(val\) corresponds to \(n = (a_0, \ldots, a_{g-1})\). The output derivatives are normalized as in the Taylor expansion.

```c
void acb_theta_jet_ql_all(arb_ptr dth, arb_srcptr z, const acb_mat_t tau, slong ord, slong prec)

Sets \(dth\) to the derivatives of all functions \(\theta_{a,b}\) for \(a,b \in \{0,1\}^g\) at \((z, \tau)\), as a concatenation of \(2^g\) vectors of length \(N\), the total number of derivation tuples of total order at most \(ord\). This algorithm runs in quasi-linear time in \(prec \cdot ord^9\) for any fixed \(g\) provided that \((z, \tau)\) is reduced.

We first compute \(c, rho, err\) and \(eps\) as above, then compute theta values \(\theta_{a,b}(z + h_n, \tau)\) at a higher precision at the midpoints of \(z\) and \(\tau\) to account for division by \(\varepsilon^{ord} \cdot (ord + 1)^g\). Finally, we adjust the error bounds using \(\text{acb_theta_jet_error_bounds()}\) and the naive algorithm for derivatives of order at most \(ord + 2\).
The functions in this section implement the theta transformation formula of \[\text{Igu1972}\], p. 176 and \[\text{Mum1983}\], p. 189: for any symplectic matrix \(m\), any \((z, \tau) \in \mathbb{C} \times \mathbb{H}_g\), and any characteristic \((a, b)\), we have

\[
\theta_{a, b}(m \cdot (z, \tau)) = \kappa(m) \zeta^e(m, a, b) \det(\gamma \tau + \delta)^{1/2} e^{\pi i z^T (\gamma \tau + \delta)^{-1} \nu \gamma z} \theta_{a', b'}(z, \tau)
\]

where

- \(\gamma, \delta\) are the lower \(g \times g\) blocks of \(m\),
- \(a', b'\) is another characteristic depending on \(m, a, b\),
- \(\zeta = \exp(i\pi/4)\),
- \(e(m, a, b)\) is an integer given by an explicit formula in terms of \(m, a, b\) (this is \(\phi_m\) in Igusa’s notation), and
- \(\kappa(m)\) is an \(8^{th}\) root of unity, only well-defined up to sign unless we choose a particular branch of \(\det(\gamma \tau + \delta)^{1/2}\) on \(\mathbb{H}_g\).

### Function Definitions

```c
ulong acb_theta_transform_char(slong *e, const fmpz_mat_t mat, ulong ab)
```

Returns the theta characteristic \((a', b')\) and sets \(e\) to \(e(m, a, b)\).

```c
void acb_theta_transform_sqrtdet(acb_t res, const acb_mat_t tau, slong prec)
```

Sets \(res\) to \(\det(\tau)^{1/2}\), where the branch of the square root is chosen such that the result is \(i^{9/2} \det(Y)\) when \(\tau = iy\) is purely imaginary.

We pick a purely imaginary matrix \(A\) and consider the polynomial \(P(t) = \det(A + i\frac{t+1}{2} (\tau - A))\). Up to choosing another \(A\), we may assume that it has degree \(g\) and that its roots (as complex balls) do not intersect the segment \([-1, 1] \subset \mathbb{C}\). We then find the correct branch of \(P(t)^{1/2}\) between \(t = -1\) and \(t = 1\) following \[\text{MN2019}\].

```c
slong acb_theta_transform_kappa(acb_t sqrtdet, const fmpz_mat_t mat, const acb_mat_t tau, slong prec)
```

Returns \(0 \leq r < 8\) such that \(\kappa(m) = \zeta^r\) and sets \(sqrtdet\) to the corresponding square root of \(\det(\gamma \tau + \delta)^{1/2}\).

After applying \(sp2gz_decompose()\), we only have to consider four special cases for \(mat\). If \(mat\) is trigonal or block-diagonal, one can compute its action on \(\theta_{0, 0}\) directly. If \(mat\) is an embedded matrix from \(SL_2(\mathbb{Z})\), we rely on \(acb_modular_theta_transform()\). Finally, if \(mat\) is an embedded \(J\) matrix from dimension \(0 \leq r < 8\), then \(\kappa(m) \zeta^r \nu^{1/2} \det(\tau_0)^{1/2} = 1\), where \(\tau_0\) denotes the upper left \(r \times r\) submatrix of \(\tau\) and the square root is computed as in \(acb_theta_transform_sqrtdet()\).

```c
slong acb_theta_transform_kappa2(const fmpz_mat_t mat)
```

Returns \(0 \leq r < 3\) such that \(\kappa(m)^2 = ir\), which makes sense without reference to a branch of \(\det(\gamma \tau + \delta)^{1/2}\).

We adopt a similar strategy to \(acb_theta_transform_kappa()\) but do not call \(acb_theta_transform_sqrtdet()\).

```c
void acb_theta_transform_proj(acb_ptr res, const fmpz_mat_t mat, acb_srcptr th, int sqr, slong prec)
```

Assuming that \(sqr\) is 0 (false) and that \(th\) contains \(\theta_{a, b}(z, \tau)\) for some \((z, \tau)\), sets \(res\) to contain the values \(\theta_{a, b}(mat \cdot (z, \tau))\) up to a common scalar factor in \(\mathbb{C}^\times\). This only permutes the theta values and multiplies them by a suitable \(8^{th}\) root of unity. If \(sqr\) is nonzero (true), does the same computation for squared theta values \(\theta_{a, b}(z, \tau)^2\) instead.

In \(acb_theta_all()\) and \(acb_theta_jet_all()\), we first reduce \(\tau\) using \(acb_siegel_reduce()\), then call \(acb_theta_gl_all()\), or \(acb_theta_jet_gl_all()\) on the reduced matrix, and finally apply the transformation formula. If the reduction step is not successful, we set the result to indeterminate values.
9.20.17 Dimension 2 specifics

In the \( g = 2 \) case, one can use theta functions to evaluate many fundamental Siegel modular forms. This section contains methods to do so, in analogy with `acb_modular_delta()` or `acb_modular_eisenstein()` when \( g = 1 \).

We use the following notation. Fix \( k, j \geq 0 \). A Siegel modular form of weight \( \det^k \otimes \text{Sym}^j \) is defined by an analytic function \( f : \mathbb{H}_g \to \mathbb{C}[X] \) (the vector space of polynomials of degree at most \( j \)) such that for any \( \tau \in \mathbb{H}_g \) and \( m \in \text{Sp}_4(\mathbb{Z}) \), we have

\[
\det((\alpha \tau + \beta)(\gamma \tau + \delta)^{-1}) = \det(\gamma \tau + \delta)^k \cdot \text{Sym}^j(\gamma \tau + \delta)(f(\tau)).
\]

Here \( \alpha, \beta, \gamma, \delta \) are the \( g \times g \) blocks of \( m \), and the notation \( \text{Sym}^j(r) \) where \( r = \left( \begin{array}{cc} a & b \\ c & d \end{array} \right) \in \text{GL}_2(\mathbb{C}) \) stands for the map

\[
P(X) \mapsto (hX + d)^j P\left(\frac{aX + c}{hX + d}\right).
\]

For a nonzero \( f \) to exist, \( j \) must be even.

Siegel modular forms generate a bi-graded ring which is not finitely generated. However, if we relax the definition of a Siegel modular form and allow them to have a pole along the diagonal \( \mathbb{H}_2^2 = \left\{ \left( \begin{array}{cc} a & b \\ c & d \end{array} \right) \right\} \subset \mathbb{H}_2 \) of a certain order (depending on the weight), we indeed find a finitely generated ring corresponding to classical “covariants” of a binary sextic. Historically, covariants are classified in terms of their degree \( k \) and index \( j \), corresponding to Siegel modular functions of weight \( \det^{k-j/2} \otimes \text{Sym}^j \). See [CFG2017] for more details on the correspondence between modular forms and covariants.

**ACB_THETA_G2_COV_NB**

Macro giving the number of generators of the ring of covariants, equal to 26.

```c
void acb_theta_g2_jet_naive_1(acb_ptr dth, const acb_mat_t tau, slong prec)
{  Sets dth in the same way as acb_theta_jet_naive_all() at order 1 for z = 0.
   We take advantage of the fact that the value (resp. gradients) of \( \theta_{a,b}(z, \tau) \) at \( z = 0 \) vanish if \( (a, b) \) is an odd (resp. even) characteristic. The attached worker of type acb_theta_naive_worker_t uses one of two available strategies (doing multiplications and then summing, or calling acb_dot() twice) depending on prec.
}
```

```c
void acb_theta_g2_detk_symj(acb_poly_t res, const acb_mat_t m, const acb_poly_t f, slong k, slong prec)
{  Sets res to \( \det^k(\text{Sym}^j(f)) \). The polynomial \( f \) should be of degree at most \( j \) (any coefficients of larger degree are ignored).
}
```

```c
void acb_theta_g2_transvectant(acb_poly_t res, const acb_poly_t g, const acb_poly_t h, slong m, slong n, slong k, slong prec)
{  Sets res to the \( k \)-th transvectant of the polynomials \( g \) and \( h \) of degrees \( m \) and \( n \): considering \( g \) and \( h \) as homogeneous polynomials of degree \( m \) (resp. \( n \)) in \( x_1, x_2 \), this sets res to

\[
(g, h)_k := \frac{(m-k)!(n-k)!}{m!n!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} \frac{\partial^k g}{\partial x_1^j \partial x_2^{k-j}} \frac{\partial^k h}{\partial x_1^j \partial x_2^{k-j}}.
\]

Any coefficients of \( g \) or \( h \) of larger degree than \( m \) (resp. \( n \)) are ignored.
}
```

```c
void acb_theta_g2_transvectant_lead(acb_t res, const acb_poly_t g, const acb_poly_t h, slong m, slong n, slong k, slong prec)
{  Sets res to the leading coefficient of \( (g, h)_k \) in \( x_1 \), with the same conventions as in acb_theta_g2_transvectant().
}  ```

9.20. acb_theta.h – Riemann theta functions
slong acb_theta_g2_character(const fmpz_mat_t mat)
    Returns the value in \( \mathbb{Z}/2\mathbb{Z} \) (0 or 1) of the unique nontrivial character of \( \text{Sp}_4(\mathbb{Z}) \) at \( \mat \), following [CFG2019], §12.

void acb_theta_g2_psi4(acb_t res, acb_srcptr th2, slong prec)
void acb_theta_g2_psi6(acb_t res, acb_srcptr th2, slong prec)
void acb_theta_g2_ch10(acb_t res, acb_srcptr th2, slong prec)
void acb_theta_g2_ch12(acb_t res, acb_srcptr th2, slong prec)

Sets \( \text{res} \) to the value of the Eisenstein series \( \psi_4, \psi_6 \) or the cusp forms \( \chi_{10}, \chi_{12} \) corresponding to the given vector \( \text{th2} \) of squared theta values (of length 16).

We use the formulas from §7.1 in [Str2014], with the following normalizations:

\[
\psi_4 = h_4/4, \quad \psi_6 = h_6/4, \quad \chi_{10} = -2^{-12}h_{10}, \quad \chi_{12} = 2^{-15}h_{12}.
\]

We warn that \( \chi_{10} \) and \( \chi_{12} \) differ from the classical notation of Igusa [Igu1979] by scalar factors. Writing \( \tau = \begin{pmatrix} \tau_1 & \tau_2 \\ \tau_2 & \tau_3 \end{pmatrix} \) and \( q_j = \exp(2\pi i \tau_j) \), the Fourier expansions of these modular forms begin as follows:

\[
\psi_4(\tau) = 1 + 240(q_1 + q_3) + \cdots \\
\psi_6(\tau) = 1 - 504(q_1 + q_3) + \cdots \\
\chi_{10}(\tau) = (q_2 - 2 + q_2^{-1})q_1q_3 + \cdots \\
\chi_{12}(\tau) = (q_2 + 10 + q_2^{-1})q_1q_3 + \cdots.
\]

void acb_theta_g2_ch15(acb_t res, acb_srcptr th, slong prec)

Sets \( \text{res} \) to the value of \( \chi_5 = -2^{-6} \prod_{(a,b) \text{ even}} \theta_{a,b} \) corresponding to the given theta values \( \text{th} \). The form \( \chi_5 \) is a Siegel cusp form with character: see [CFG2019] for more details.

void acb_theta_g2_ch35(acb_t res, acb_srcptr th, slong prec)

Sets \( \text{res} \) to the value of the cusp form \( \chi_{35} \) corresponding to the vector of theta values \( \text{th} \). The form \( \chi_{35} \) is the unique scalar-valued Siegel modular form of weight \( \text{det}^{35} \otimes \text{Sym}^0 \) up to scalars, and is normalized as follows:

\[
\chi_{35}(\tau) = q_1^2q_3^2(q_1 - q_3)(q_2 - q_2^{-1}) + \cdots.
\]

An explicit formula for \( \chi_{35} \) in terms of theta values is given in [Bol1887]. See also [Mum1984], Prop. 6.2 p. 98 for how to translate Bolza’s notation in terms of theta characteristics.

void acb_theta_g2_ch3_6(acb_poly_t res, acb_srcptr dth, slong prec)

Sets \( \text{res} \) to the value of the vector-valued cusp form with character \( \chi_{6,3} \) of weight \( \text{det}^3 \otimes \text{Sym}^6 \) corresponding to the given values of \( \text{dth} \), computed as in e.g. \( \text{acb_theta_g2_jet_naive_1()} \). We have by [CFG2017]:

\[
\chi_{6,3}(\tau) = \frac{1}{64\pi^{6}} \prod_{(a,b) \text{ odd}} \left( \frac{\partial \theta_{a,b}}{\partial x_1}(0, \tau)x_1 + \frac{\partial \theta_{a,b}}{\partial x_2}(0, \tau)x_2 \right).
\]
void \texttt{acb\_theta\_g2\_sextic}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{const} \texttt{acb\_mat\_t} \texttt{tau}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to the value of $\chi_{-2,6} := \chi_{3.6}/\chi_5$ at $\tau$. We reduce $\tau$ to the Siegel fundamental domain and call either \texttt{acb\_theta\_g2\_jet\_naive\_1()} or \texttt{acb\_theta\_jet\_gl\_all()} to compute theta gradients, depending on \texttt{prec}. Under the correspondence between Siegel modular functions and covariants of binary sextics, $\chi_{-2,6}$ corresponds to the binary sextic itself, hence the name.

void \texttt{acb\_theta\_g2\_sextic\_chi5}(\texttt{acb\_poly\_t} \texttt{res}, \texttt{acb\_t} \texttt{chi5}, \texttt{const} \texttt{acb\_mat\_t} \texttt{tau}, \texttt{slong} \texttt{prec})

Sets \texttt{res} and \texttt{chi5} to the values of $\chi_{-2,6}$ and $\chi_5$ at $\tau$. Theta values are computed only once.

void \texttt{acb\_theta\_g2\_covariants}(\texttt{acb\_poly\_struct *res}, \texttt{const} \texttt{acb\_poly\_t} \texttt{f}, \texttt{slong} \texttt{prec})

Sets \texttt{res} to the vector of 26 generators of the ring of covariants evaluated at the sextic $f$ (any terms of degree $> 6$ are ignored), in the following order:

0. $C_{1,6} = f$
1. $C_{2,0} = 60(f,f)_6$
2. $C_{2,4} = 75(f,f)_4$
3. $C_{2,8} = 90(f,f)_2$
4. $C_{3,2} = 30(f,C_{2,4})_4$
5. $C_{3,6} = 30(f,C_{2,4})_2$
6. $C_{3,8} = 6(f,C_{2,4})_1$
7. $C_{3,12} = 6(f,C_{2,8})_1$
8. $C_{4,0} = 2(C_{2,4},C_{2,4})_4$
9. $C_{4,4} = 30(f,C_{3,2})_2$
10. $C_{4,6} = 6(f,C_{3,2})_1$
11. $C_{4,10} = 2(C_{2,8},C_{2,4})_1$
12. $C_{5,2} = (C_{2,4},C_{3,2})_2$
13. $C_{5,4} = \frac{2}{3}(C_{2,4},C_{3,2})_1$
14. $C_{5,8} = 2(C_{2,8},C_{3,2})_1$
15. $C_{6,0} = 2(C_{3,2},C_{3,2})_2$
16. $C_{6,6} = \frac{2}{3}(C_{3,6},C_{3,2})_1$
17. $C_{6,6} = \frac{8}{3}(C_{3,8},C_{3,2})_2$
18. $C_{7,2} = 30(f,C_{3,2})_4$
19. $C_{7,4} = 12(f,C_{3,2})_3$
20. $C_{8,2} = \frac{2}{3}(C_{2,4},C_{3,2})_3$
21. $C_{9,4} = 4(C_{3,8},C_{3,2})_4$
22. $C_{10,0} = 20(f,C_{3,2})_6$
23. $C_{10,2} = \frac{6}{5}(f,C_{3,2})_5$
24. $C_{12,2} = \frac{6}{5}(C_{3,8},C_{3,2})_6$
25. $C_{15,0} = \frac{1}{12}(C_{3,8},C_{3,2})_8$.

The scalar factors are chosen so that when evaluated at a formal sextic $f = \sum a_i x_1^{i-1} x_2$, the covariants are integral and primitive as multivariate polynomials in $a_0, \ldots, a_6, x_1, x_2$. 

9.20. \texttt{acb\_theta.h} – Riemann theta functions
void \texttt{acb_theta_g2_covariants_lead}(acb\_ptr res, const acb\_poly\_t f, slong prec)

Sets \texttt{res} to the vector of leading coefficients in $x_1$ of the 26 covariants evaluated at $f$. This is more efficient than taking leading coefficients of \texttt{acb_theta_g2_covariants()}, since we can use \texttt{acb_theta_g2_transvectant\_lead()} instead of \texttt{acb_theta_g2_transvectant()}.

9.20.18 Tests

\begin{itemize}
\item \texttt{./build/acb_theta/test/main sp2gz_set_blocks}
\end{itemize}

Generates a random $2g \times 2g$ matrix, calls \texttt{sp2gz_set_blocks()} on its four $g \times g$ windows, and checks that the result equals the original matrix.

\begin{itemize}
\item \texttt{./build/acb_theta/test/main sp2gz_is_correct}
\end{itemize}

Checks that the return value of \texttt{sp2gz_is_correct()} is 1 on matrices generated by \texttt{sp2gz_j()}, \texttt{sp2gz_block_diag()}, \texttt{sp2gz_trig()} and \texttt{sp2gz_fundamental()}, and 0 on the identity matrix if it is not square of even size.

\begin{itemize}
\item \texttt{./build/acb_theta/test/main sp2gz_inv}
\end{itemize}

Checks that the result of \texttt{sp2gz_inv()} agrees with \texttt{fmpz_mat_inv()} on random input.

\begin{itemize}
\item \texttt{./build/acb_theta/test/main sp2gz_decompose}
\end{itemize}

Checks that the result of \texttt{sp2gz_decompose()} on random input only consists of symplectic matrices of the allowed types, and that their product equals the original matrix.

\begin{itemize}
\item \texttt{./build/acb_theta/test/main acb_siegel_cocycle}
\end{itemize}

Checks that the chain rule holds: if $m'' = m'm$ is a product of two symplectic matrices and $\tau \in \mathbb{H}_g$, then $\gamma''\tau + \delta'' = (\gamma'\tau' + \delta')(\gamma\tau + \delta)$ where $\tau' = m\tau$. These quantities are computed using \texttt{acb_siegel_cocycle()} and \texttt{acb_siegel_transform()}.

\begin{itemize}
\item \texttt{./build/acb_theta/test/main acb_siegel_transform}
\end{itemize}

Checks that the chain rule holds, i.e. \texttt{acb_siegel_transform()} defines an action of the group $\text{Sp}_{2g}(\mathbb{Z})$ on $\mathbb{H}_g$.

\begin{itemize}
\item \texttt{./build/acb_theta/test/main acb_siegel_transform_z}
\end{itemize}

Checks that \texttt{acb_siegel_transform()} and \texttt{acb_siegel_transform_z()} agree on random input, and that \texttt{acb_siegel_transform_z()} on the inverse of any matrix yields the inverse transformation.

\begin{itemize}
\item \texttt{./build/acb_theta/test/main acb_siegel_reduce}
\end{itemize}

Generates an input matrix $\tau$ at a working precision that is not too low compared to the size of its coefficients, and calls \texttt{acb_siegel_reduce()}. Checks that the resulting matrix $m$ is symplectic and that $m\tau$ is reduced with a tolerance of $2^{-10}$ using \texttt{acb_siegel_is_reduced()}. 

\begin{itemize}
\item \texttt{./build/acb_theta/test/main acb_siegel_is_reduced}
\end{itemize}

Checks that \texttt{acb_siegel_is_reduced()} returns 1 on the matrix $iI_g$, but 0 on other matrices specially constructed to not be reduced.

\begin{itemize}
\item \texttt{./build/acb_theta/test/main acb_theta_char_get_a}
\end{itemize}

Generates a random characteristic $a$, sets $n$ to the result of \texttt{acb_theta_char_get_slong()} on $a$, and checks that the result of \texttt{acb_theta_char_get_a()} on $n$ gives back $a$. 

696 Chapter 9. Real and complex numbers
Checks that dot products computed by `acb_theta_char_dot()`, `acb_theta_char_dot_slong()` and `acb_theta_char_dot_acb()` agree on random input.

Checks that the 10 even theta characteristics for $g = 2$ are 0, 1, 2, 3, 4, 6, 8, 9, 12, 15.

Checks that there are exactly 15 Göpel quadruples for $g = 2$.

Checks that there are exactly 60 syzygous triples for $g = 2$.

Generates a random ellipsoid $E$ using `acb_theta_eld_set()`, computes its points using `acb_theta_eld_points()`, and checks that each of these points lies within the box specified by `acb_theta_eld_box`. Then, generates random points pt: if pt is in $E$ according to `acb_theta_eld_contains()`, then pt must appear in the list of points, otherwise the norm of pt according to the chosen Cholesky matrix must be at least the radius of $E$.

Generates a random ellipsoid $E$, computes its border using `acb_theta_eld_border()`, and checks that none of these border points lie in $E$ nor any of its children.

Generates a reduced matrix $\tau$ in $\mathbb{H}_g$ and vector $z \in \mathbb{C}^g$, calls `acb_theta_naive_radius()`, constructs the associated ellipsoid $E$, and checks that the sums of absolute values of terms of the theta series on the border of $E$ is at most the specified bound.

Checks that the results of `acb_theta_naive_reduce()` are sound on some special values of the input, namely when $z$s has only real entries and when $\text{Im}(z) = -\text{Im}(\tau)n + \varepsilon$ where $n$ is an even integral vector and $\varepsilon$ is small.

Checks that the result of `acb_theta_naive_term()` is $n^k \exp(i\pi(n^2\tau + 2nz))$ in the $g = 1$ case.

Checks that the output of `acb_theta_naive_00()` overlaps the first entry of the output of `acb_theta_naive_0b()`.

Checks that the results of `acb_theta_naive_all()` agree with `acb_modular_theta()` as follows: if the input matrix $\tau$ is diagonal with coefficients $\tau_0, \ldots, \tau_{g-1}$, then for all characteristics $(a, b)$ and vectors $z$, we have

$$\theta_{a, b}(z, \tau) = \prod_{j=0}^{g-1} \theta_{a_j, b_j}(z_j, \tau_j).$$
Checks that the output of `acb_theta_naive_fixed_a()` overlaps the relevant entries of `acb_theta_naive_all()` on random input.

Checks that the output of `acb_theta_naive_fixed_ab()` overlaps the relevant entries of `acb_theta_naive_all()` on random input.

For random $g$ and $ord$, generates the list of derivation tuples using `acb_theta_jet_tuples()`, picks an index $i$ at random, and checks that the result of `acb_theta_jet_index()` on the $i$th tuple is indeed $i$.

Checks that the results of `acb_theta_jet_mul()` agrees with the result of `fmpz_mpoly_mul()` on any input with integral entries.

Checks that the chain rule holds: if $N_3 = N_2 N_1$, then applying `acb_theta_jet_compose()` with $N_2$, then $N_1$ corresponds to applying `acb_theta_jet_compose()` with $N_3$ directly.

Generates a reduced matrix $\tau$ in $\mathbb{H}_g$ and vector $z \in \mathbb{C}^g$, chooses a random order of derivation, calls `acb_theta_jet_naive_radius()`, constructs the associated ellipsoid $E$, and checks that the sums of absolute values of terms of the differentiated theta series on the border of $E$ is at most the specified bound.

Checks that the results of `acb_theta_jet_naive_all()` agree with `acb_modular_theta_jet()` as follows: if the input matrix $\tau$ is diagonal with coefficients $\tau_0, \ldots, \tau_{g-1}$, then for all characteristics $(a, b)$, any vector $z$, and any derivation tuple $(k_0, \ldots, k_{g-1})$, we have

$$\frac{\partial^{k_0} \theta_{a,b}}{\partial z_0^{k_0} \cdots \partial z_{g-1}^{k_{g-1}}} (z, \tau) = \prod_{j=0}^{g-1} \frac{\partial^{k_j} \theta_{a_j,b_j}}{\partial z_j^{k_j}} (z_j, \tau_j).$$

Checks that the output of `acb_theta_jet_naive_00()` agrees with the relevant entries of `acb_theta_jet_naive_all()` on random input.

Checks that the output of `acb_theta_jet_naive_fixed_ab()` agrees with the relevant entries of `acb_theta_jet_naive_all()` on random input.

Generates two pairs $(z_1, \tau_1)$ and $(z_2, \tau_2)$ close to each other but not overlapping, sets $(z, \tau)$ to be their reunion (as complex balls on each coefficient), and calls `acb_theta_jet_error_bounds()` on $(z, \tau)$ for some choice of derivation order. The difference between the results of `acb_theta_jet_naive_all()` on $(z_1, \tau_1)$ and $(z_2, \tau_2)$ must then be at most two times the computed error.
Checks that for a random Cholesky matrix $C$ and integral vectors $n_1, n_2$, the results of \texttt{acb\_theta\_dist\_pt()} on $(v, n) = (Cn_1, n_2)$ and $(Cn_2, n_1)$ agree.

Picks a random Cholesky matrix $C$ and vector $v$, calls \texttt{acb\_theta\_dist\_lat()}, and computes the ellipsoid $E$ whose radius is the computed distance. Checks that $E$ contains at least one point and that the minimum distance is correct by looping over all the points in $E$.

Checks that when $z = \text{Im}(\tau)^{\frac{3}{2}}$ for some theta characteristic $a$, the result of \texttt{acb\_theta\_dist\_a0()} on $(z, \tau)$ contains zero in its $a^{th}$ entry.

Checks that calling \texttt{acb\_theta\_hadamard()} twice on random input is equivalent to multiplying by $2^2$.

Generates a random complex number $t$, sets $rts$ to a low-precision rounding of $t$ (possibly containing zero), and sets $a$ to the square of $t$. Checks that the result of \texttt{acb\_theta\_sqrt()} on this input is finite, contains $t$, and that the precision loss is small when $rts$ does not contain zero.

Checks that the duplication formula holds: the result of \texttt{acb\_theta\_mul()} on vectors containing $\theta_0, b(0, \tau)$ and $\theta_0, b(2z, 2\tau)$ for all $b \in \{0, 1\}$ and any choice of $(z, \tau)$ contains the squared theta values $\theta_0, b(2z, 2\tau)$.

Generates random $\tau$ and $z$ at working precision $\text{prec}$, computes the associated vectors of distances $d0$ and $d$ using \texttt{acb\_theta\_dist\_a0()}, and constructs vectors $a0$ and $a$ with entries of the form $xe^{-t}$ where $x$ is uniformly random with $|x| \leq 1$ (generated by \texttt{acb\_urandom()} ) and $t$ is the corresponding entry of $d0$ (resp. $d$). Calls \texttt{acb\_theta\_mul\_tight()} at a lower precision $mprec$. For each $0 \leq k < 2^g$, checks that the absolute value of $k^{th}$ entry of the result $res$ is at most $e^{-d_k}$, and that the error bound on that entry is at most $2^{-mprec+\delta e^{-d_k}}$ for a reasonable value of $\delta$ (e.g. 25).

Checks that the result of \texttt{acb\_theta\_split()} (using \texttt{acb\_theta\_split naive()} as worker) agrees with that of \texttt{acb\_theta\_split naive()} in case of success.

Checks that the result of \texttt{acb\_theta\_steps()} (using \texttt{acb\_theta\_split naive()} as worker) agrees with that of \texttt{acb\_theta\_split naive()} in case of success.

Checks that \texttt{acb\_theta\_a0()} , if successful, agrees with \texttt{acb\_theta\_a0 naive()} on random input.
Generates random values $\tau$ and $z$ in such a way that \texttt{acb\_theta\_ql\_reduce()} is likely to output $s < g$ and a nonzero $n1$, and checks that the claimed inequalities in that function’s documentation hold when computing theta values using \texttt{acb\_theta\_naive\_all()}.

```bash
./build/acb_theta/test/main acb_theta_ql_all
```

Checks that \texttt{acb\_theta\_ql\_all()} agrees with \texttt{acb\_theta\_naive\_all()} on random input.

```bash
./build/acb_theta/test/main acb_theta_jet_ql_bounds
```

Generates random $(z,\tau)$ at a working precision that is not too low and calls \texttt{acb\_theta\_jet\_ql\_bounds()} to compute the bounds $c$ and $\rho$. Checks that they are finite and that their definition is satisfied by sampling theta values on the corresponding neighborhood of $z$ at low precisions with \texttt{acb\_theta\_naive\_all()}.

```bash
./build/acb_theta/test/main acb_theta_jet_ql_radius
```

Checks that the result of \texttt{acb\_theta\_jet\_ql\_radius()} on random input satisfies the required inequalities.

```bash
./build/acb_theta/test/main acb_theta_jet_ql_finite_diff
```

Checks that \texttt{acb\_theta\_jet\_ql\_finite\_diff()} computes the correct Taylor coefficients for the function $\exp(z_0 + \cdots + z_{g-1})$ at zero. Correct input can be generated by \texttt{acb\_theta\_jet\_ql\_radius()}, as the bounds $c$ and $\rho$ can be computed directly for this function.

```bash
./build/acb_theta/test/main acb_theta_jet_ql_all
```

Checks that \texttt{acb\_theta\_jet\_ql\_all()} agrees with \texttt{acb\_theta\_jet\_naive\_all()} on random input.

```bash
./build/acb_theta/test/main acb_theta_transform_char
```

Checks that the $a$ component of any theta characteristic remains the same after applying \texttt{acb\_theta\_transform\_char()} when the symplectic matrix is trigonal as in \texttt{sp2gs\_trig()}.

```bash
./build/acb_theta/test/main acb_theta_transform_sqrtdet
```

Checks that the result of \texttt{acb\_theta\_transform\_sqrtdet()} on any input $\tau \in \mathbb{H}_g$ squares to $\det(\tau)$.

```bash
./build/acb_theta/test/main acb_theta_transform_kappa
```

Checks that \texttt{acb\_theta\_transform\_kappa()} and \texttt{acb\_theta\_transform\_kappa2()} agree on random input (i.e. they are congruent modulo 4).

```bash
./build/acb_theta/test/main acb_theta_transform_proj
```

Checks that applying \texttt{acb\_theta\_transform\_proj()} with a random symplectic matrix, then its inverse gives back the initial vector up to scaling.

```bash
./build/acb_theta/test/main acb_theta_all
```

Checks that \texttt{acb\_theta\_all()} agrees with \texttt{acb\_theta\_naive\_all()} on random input. The matrix $\tau$ is chosen to be a priori non-reduced but still reasonably close to the reduced domain.

```bash
./build/acb_theta/test/main acb_theta_jet_all
```

Checks that \texttt{acb\_theta\_jet\_all()} agrees with \texttt{acb\_theta\_jet\_naive\_all()} on random input. The matrix $\tau$ is chosen to be a priori non-reduced but still reasonably close to the reduced domain.
Checks that \( \texttt{acb\_theta\_g2\_jet naïve\_1()} \) agrees with \( \texttt{acb\_theta\_jet naïve\_all()} \) with \( g = 2, \ z = 0 \) and \( \text{ord} = 1 \) on a random matrix \( \tau \).

Checks that the chain rule holds for the representation \( \text{det}^k \text{Sym}^j \) of \( \text{GL}_2(\mathbb{C}) \) as computed by \( \texttt{acb\_theta\_g2\_detk\_symj()} \).

Checks that on any sextic polynomial \( f = \sum_{j=0}^{6} a_j x^{6-j} \), the transvectant \( (f, f)_6 \) as computed by \( \texttt{acb\_theta\_g2\_transvectant()} \) is \(-3a_3^2 + 8a_2a_4 - 20a_1a_5 + 120a_0a_6\).

Checks that the result of \( \texttt{acb\_theta\_g2\_transvectant\_lead()} \) is indeed the leading term of the result of \( \texttt{acb\_theta\_g2\_transvectant()} \) on random input.

Checks that the results of \( \texttt{acb\_theta\_g2\_character()} \) and \( \texttt{acb\_theta\_transform\_kappa2()} \) for \( g = 2 \) are compatible, using the fact that the product \( \chi_5 \) of the ten even theta constants is a Siegel modular form with character.

Checks that the result of \( \texttt{acb\_theta\_g2\_psi4()} \) is invariant when applying \( \texttt{acb\_theta\_transform\_proj()} \) on any input vector.

Checks that the result of \( \texttt{acb\_theta\_g2\_psi6()} \) is multiplied by \( \pm 1 \) when applying \( \texttt{acb\_theta\_transform\_proj()} \) on any input vector. The correct sign is given by \( \texttt{acb\_theta\_transform\_kappa2()} \).

Checks that the result of \( \texttt{acb\_theta\_g2\_chi10()} \) is multiplied by \( \pm 1 \) when applying \( \texttt{acb\_theta\_transform\_proj()} \) on any input vector. The correct sign is given by \( \texttt{acb\_theta\_transform\_kappa2()} \).

Checks that the result of \( \texttt{acb\_theta\_g2\_chi12()} \) is invariant when applying \( \texttt{acb\_theta\_transform\_proj()} \) on any input vector.

Checks that the result of \( \texttt{acb\_theta\_g2\_chi5()} \) squares to the result of \( \texttt{acb\_theta\_g2\_chi10()} \) on any input vector.

Checks that the result of \( \texttt{acb\_theta\_g2\_chi35()} \) is multiplied by \( i^k \) when applying \( \texttt{acb\_theta\_transform\_proj()} \) on an input vector of theta values. The exponent \( k \) is given by \( \texttt{acb\_theta\_transform\_kappa2()} \).
Checks that the product \( \chi_{8,6} = \chi_{5,8} \chi_{3,6} \) computed using `acb_theta_g2_chi5()` and `acb_theta_g2_chi3_6()` indeed defines a modular form of weight \( \det^8 \Sym^6 \) by evaluating both sides of the transformation law on random input.

Checks that the discriminant of the result of `acb_theta_g2_sextic()` on a random matrix \( \tau \) is \( 2^{12} \chi_{10}(\tau) \), as computed by `acb_theta_g2_chi10()`.

Checks that the results of `acb_theta_g2_sextic_chi5()` agree with those of `acb_theta_g2_sextic()` and `acb_theta_g2_chi5()` on random input.

Checks that the output of `acb_theta_g2_covariants()` agrees with that of `acb_theta_g2_psi4()` using the relation \( 20 \psi_4 = -C_{2,0} + 3C_{1,0} \). Also checks that each covariant, when evaluated on the result of `acb_theta_g2_sextic()`, defines a Siegel modular function of the correct weight by evaluating the transformation law, and that covariants take integral values when the input polynomial is integral.

Checks that the results of `acb_theta_g2_covariants_lead()` are indeed the leading terms of the results of `acb_theta_g2_covariants()` on random input.

### 9.20.19 Profiling

Prints quick performance measurements for `acb_siegel_reduce()`: for the given \( g \), for \( d \leq \text{dmax} \) by steps of \( \text{dstep} \), and \( \text{prec} \leq \text{pmax} \) by steps of \( \text{pstep} \), constructs an input matrix \( \tau/d \) where \( \tau \) is generated by `acb_siegel_randtest_reduced()` and runs `acb_siegel_reduce()` on \( w \) at working precision \( \text{prec} \).

This is meant to show that reduction is generally not a critical step when evaluating theta functions.

Prints quick performance measurements for `acb_theta_ql_a0_split()`: for the given \( g \) and at the given working precision \( \text{prec} \), generates an input matrix \( \tau \) as in `acb_siegel_randtest_reduced()`, but whose lower right \( (g-s) \times (g-s) \) submatrix is subsequently multiplied by \( c \), where \( s \) runs between 1 and \( g-1 \) and \( c \leq \text{cmax} \) is increased by steps of \( \text{cstep} \). The running times of `acb_theta_ql_a0_steps()` with or without splitting at \( s \) are then compared on each of these matrices, as well as the running time of `acb_theta_ql_a0()`.

This is meant to provide information on how the choice of splitting in `acb_theta_ql_a0()` should be made.

Prints quick performance measurements for `acb_theta_ql_a0_steps()`: for the given \( g \) and for a working precision \( \text{prec} \leq \text{pmax} \) increasing by steps of \( \text{pstep} \), generates a random matrix \( \tau \) in the reduced domain and compares the running time of `acb_theta_ql_a0_steps()` with different parameters \( \text{nb_steps} \).

This is meant to provide information on the correct value to return in `acb_theta_ql_a0_nb_steps()`.
Prints quick performance measurements for the functions \( \text{acb\_theta\_all()} \), \( \text{acb\_theta\_ql\_a0()} \), \( \text{acb\_theta\_ql\_all()} \) and \( \text{acb\_theta\_naive\_all()} \) at different precisions on a specific input matrix of the specified dimension \( g \). We start at precision 32, then double it \( nb\_steps \) times. The parameter \( hasz \) should be either 0 (theta constants) or 1 (theta values at a nonzero point).

This is meant to show whether the main user function is slower than naive algorithms at low precisions. (This is currently the case.)

Prints quick performance measurements for the functions \( \text{acb\_theta\_jet\_all()} \) and \( \text{acb\_theta\_jet\_naive\_all()} \) at different precisions and order 1 on a specific input matrix for \( g = 2 \).

This is meant to show whether the main user function is slower than naive algorithms at low precisions. (This is currently the case.)

### 9.21 \texttt{acb\_dirichlet.h} – Dirichlet L-functions, Riemann zeta and related functions

This module allows working with values of Dirichlet characters, Dirichlet L-functions, and related functions. A Dirichlet L-function is the analytic continuation of an L-series

\[
L(s, \chi) = \sum_{k=1}^{\infty} \frac{\chi(k)}{k^s}
\]

where \( \chi(k) \) is a Dirichlet character. The trivial character \( \chi(k) = 1 \) gives the Riemann zeta function. Working with Dirichlet characters is documented in \texttt{dirichlet.h – Dirichlet characters}.

The code in other modules for computing the Riemann zeta function, Hurwitz zeta function and polylogarithm will possibly be migrated to this module in the future.

#### 9.21.1 Roots of unity

**Type** \texttt{acb\_dirichlet\_roots\_struct}

**Type** \texttt{acb\_dirichlet\_roots\_t}

**Function** \texttt{acb\_dirichlet\_roots\_init} (\texttt{acb\_dirichlet\_roots\_t roots, ulong n, slong num, slong prec})

Initializes \( \text{roots} \) with precomputed data for fast evaluation of roots of unity \( e^{2\pi i k/n} \) of a fixed order \( n \). The precomputation is optimized for \( num \) evaluations.

For very small \( num \), only the single root \( e^{2\pi i /n} \) will be precomputed, which can then be raised to a power. For small \( prec \) and large \( n \), this method might even skip precomputing this single root if it estimates that evaluating roots of unity from scratch will be faster than powering.

If \( num \) is large enough, the whole set of roots in the first quadrant will be precomputed at once. However, this is automatically avoided for large \( n \) if too much memory would be used. For intermediate \( num \), baby-step giant-step tables are computed.

**Function** \texttt{acb\_dirichlet\_roots\_clear} (\texttt{acb\_dirichlet\_roots\_t roots})

Clears the structure.

**Function** \texttt{acb\_dirichlet\_root} (\texttt{acb\_t res, const acb\_dirichlet\_roots\_t roots, ulong k, slong prec})

Computes \( e^{2\pi i k/n} \).
9.21.2 Truncated L-series and power sums

void \texttt{acb\_dirichlet\_powsum\_term}(acb\_ptr \texttt{res}, \texttt{arb\_t log\_prev}, ulong *prev, const \texttt{acb\_t s}, ulong \texttt{k}, int integer, int \texttt{critical\_line}, slong \texttt{len}, slong \texttt{prec})

Sets \texttt{res} to $k^{-(s+x)}$ as a power series in $x$ truncated to length \texttt{len}. The flags \texttt{integer} and \texttt{critical\_line} respectively specify optimizing for $s$ being an integer or having real part 1/2.

On input \texttt{log\_prev} should contain the natural logarithm of the integer at \texttt{prev}. If \texttt{prev} is close to $k$, this can be used to speed up computations. If log($k$) is computed internally by this function, then \texttt{log\_prev} is overwritten by this value, and the integer at \texttt{prev} is overwritten by $k$, allowing \texttt{log\_prev} to be recycled for the next term when evaluating a power sum.

void \texttt{acb\_dirichlet\_powsum\_sieved}(acb\_ptr \texttt{res}, const \texttt{acb\_t s}, ulong \texttt{n}, slong \texttt{len}, slong \texttt{prec})

Sets \texttt{res} to $\sum_{k=1}^{\texttt{n}} k^{-(s+x)}$ as a power series in $x$ truncated to length \texttt{len}. This function stores a table of powers that have already been calculated, computing $(ij)^r$ as $i^r j^r$ whenever $k = ij$ is composite. As a further optimization, it groups all even $k$ and evaluates the sum as a polynomial in $2^{-(s+x)}$. This scheme requires about $n/\log n$ powers, $n/2$ multiplications, and temporary storage of $n/6$ power series. Due to the extra power series multiplications, it is only faster than the naive algorithm when \texttt{len} is small.

void \texttt{acb\_dirichlet\_powsum\_smooth}(acb\_ptr \texttt{res}, const \texttt{acb\_t s}, ulong \texttt{n}, slong \texttt{len}, slong \texttt{prec})

Sets \texttt{res} to $\sum_{k=1}^{\texttt{n}} k^{-(s+x)}$ as a power series in $x$ truncated to length \texttt{len}. This function performs partial sieving by adding multiples of 5-smooth $k$ into separate buckets. Asymptotically, this requires computing $4/15$ of the powers, which is slower than \texttt{sieved}, but only requires logarithmic extra space. It is also faster for large \texttt{len}, since most power series multiplications are traded for additions. A slightly bigger gain for larger $n$ could be achieved by using more small prime factors, at the expense of space.

9.21.3 Riemann zeta function

void \texttt{acb\_dirichlet\_zeta}(acb\_t \texttt{res}, const \texttt{acb\_t s}, slong \texttt{prec})

Computes $\zeta(s)$ using an automatic choice of algorithm.

void \texttt{acb\_dirichlet\_zeta\_jet}(acb\_t \texttt{res}, const \texttt{acb\_t s}, int \texttt{deflate}, slong \texttt{len}, slong \texttt{prec})

Computes the first \texttt{len} terms of the Taylor series of the Riemann zeta function at $s$. If \texttt{deflate} is nonzero, computes the deflated function $\zeta(s) - 1/(s - 1)$ instead.

void \texttt{acb\_dirichlet\_zeta\_bound}(mag\_t \texttt{res}, const \texttt{acb\_t s})

Computes an upper bound for $|\zeta(s)|$ quickly. On the critical strip (and slightly outside of it), formula (43.3) in [Rad1973] is used. To the right, evaluating at the real part of $s$ gives a trivial bound. To the left, the functional equation is used.

void \texttt{acb\_dirichlet\_zeta\_deriv\_bound}(mag\_t \texttt{der1}, mag\_t \texttt{der2}, const \texttt{acb\_t s})

Sets \texttt{der1} to a bound for $|\zeta'(s)|$ and \texttt{der2} to a bound for $|\zeta''(s)|$. These bounds are mainly intended for use in the critical strip and will not be tight.

void \texttt{acb\_dirichlet\_eta}(acb\_t \texttt{res}, const \texttt{acb\_t s}, slong \texttt{prec})

Sets \texttt{res} to the Dirichlet eta function $\eta(s) = \sum_{k=1}^{\infty} (-1)^{k+1}/k^s = (1 - 2^{1-s})\zeta(s)$, also known as the alternating zeta function. Note that the alternating character $\{1, -1\}$ is not itself a Dirichlet character.

void \texttt{acb\_dirichlet\_xi}(acb\_t \texttt{res}, const \texttt{acb\_t s}, slong \texttt{prec})

Sets \texttt{res} to the Riemann xi function $\xi(s) = \frac{1}{2} s(s - 1)\pi^{-s/2}\Gamma(\frac{1}{2}s)\zeta(s)$. The functional equation for $\xi$ is $\xi(1 - s) = \xi(s)$. 

704 Chapter 9. Real and complex numbers
9.21.4 Riemann-Siegel formula

The Riemann-Siegel (RS) formula is implemented closely following J. Arias de Reyna [Ari2011]. For \( s = \sigma + it \) with \( t > 0 \), the expansion takes the form

\[
\zeta(s) = \mathcal{R}(s) + X(s)\mathcal{R}(1 - s), \quad X(s) = \pi^{s-1/2}\frac{\Gamma((1-s)/2)}{\Gamma(s/2)}
\]

where

\[
\mathcal{R}(s) = \sum_{k=1}^{N} \frac{1}{k^s} + (-1)^{N-1}Ua^{-\sigma}\left[\sum_{k=0}^{K} \frac{C_k(p)}{a^k} + RS_K\right]
\]

\[
U = \exp\left(-i\left[\frac{t}{2}\log\left(\frac{t}{2\pi}\right) - \frac{t}{2} - \frac{\pi}{8}\right]\right), \quad a = \sqrt{\frac{t}{2\pi}}, \quad N = \lfloor a \rfloor, \quad p = 1 - 2(a - N).
\]

The coefficients \( C_k(p) \) in the asymptotic part of the expansion are expressed in terms of certain auxiliary coefficients \( d_j^{(k)} \) and \( F_j^{(k)}(p) \). Because of artificial discontinuities, \( s \) should be exact inside the evaluation.

void \texttt{acb\_dirichlet\_zeta\_rs\_f\_coeffs}(\texttt{acb\_ptr f}, \texttt{arb\_t p}, \texttt{slong n}, \texttt{slong prec})

Computes the coefficients \( F_j^{(k)}(p) \) for \( 0 \leq j < n \). Uses power series division. This method breaks down when \( p = \pm 1/2 \) (which is not problem if \( s \) is an exact floating-point number).

void \texttt{acb\_dirichlet\_zeta\_rs\_d\_coeffs}(\texttt{arb\_ptr d}, \texttt{arb\_t sigma}, \texttt{slong k}, \texttt{slong prec})

Computes the coefficients \( d_j^{(k)} \) for \( 0 \leq j \leq \lfloor 3k/2 \rfloor + 1 \). On input, the array \( d \) must contain the coefficients for \( d_j^{(k-1)} \) unless \( k = 0 \), and these coefficients will be updated in-place.

void \texttt{acb\_dirichlet\_zeta\_rs\_bound}(\texttt{mag\_t err}, \texttt{acb\_t s}, \texttt{slong K})

Bounds the error term \( RS_K \) following Theorem 4.2 in Arias de Reyna.

void \texttt{acb\_dirichlet\_zeta\_rs\_r}(\texttt{acb\_t res}, \texttt{acb\_t s}, \texttt{slong K}, \texttt{slong prec})

Computes \( \mathcal{R}(s) \) in the upper half plane. Uses precisely \( K \) asymptotic terms in the RS formula if this input parameter is positive; otherwise chooses the number of terms automatically based on \( s \) and the precision.

void \texttt{acb\_dirichlet\_zeta\_rs}(\texttt{acb\_t res}, \texttt{acb\_t s}, \texttt{slong K}, \texttt{slong prec})

Computes \( \zeta(s) \) using the Riemann-Siegel formula. Uses precisely \( K \) asymptotic terms in the RS formula if this input parameter is positive; otherwise chooses the number of terms automatically based on \( s \) and the precision.

void \texttt{acb\_dirichlet\_zeta\_jet\_rs}(\texttt{acb\_ptr res}, \texttt{acb\_t s}, \texttt{slong len}, \texttt{slong prec})

Computes the first \( len \) terms of the Taylor series of the Riemann zeta function at \( s \) using the Riemann Siegel formula. This function currently only supports \( len = 1 \) or \( len = 2 \). A finite difference is used to compute the first derivative.

9.21.5 Hurwitz zeta function

void \texttt{acb\_dirichlet\_hurwitz}(\texttt{acb\_t res}, \texttt{acb\_t s}, \texttt{acb\_t a}, \texttt{slong prec})

Computes the Hurwitz zeta function \( \zeta(s, a) \). This function automatically delegates to the code for the Riemann zeta function when \( a = 1 \). Some other special cases may also be handled by direct formulas. In general, Euler-Maclaurin summation is used.
9.21.6 Hurwitz zeta function precomputation

type acb_dirichlet_hurwitz_precomp_struct

type acb_dirichlet_hurwitz_precomp_t

void acb_dirichlet_hurwitz_precomp_init(acb_dirichlet_hurwitz_precomp_t pre, const acb_t s, int deflate, slong A, slong K, slong N, slong prec)

Precomputes a grid of Taylor polynomials for fast evaluation of $\zeta(s, a)$ on $a \in (0, 1]$ with fixed $s$. $A$ is the initial shift to apply to $a$, $K$ is the number of Taylor terms, $N$ is the number of grid points. The precomputation requires $NK$ evaluations of the Hurwitz zeta function, and each subsequent evaluation requires $2K$ simple arithmetic operations (polynomial evaluation) plus $A$ powers. As $K$ grows, the error is at most $O(1/(2AN)^K)$.

This function can be called with $A$ set to zero, in which case no Taylor series precomputation is performed. This means that evaluation will be identical to calling acb_dirichlet_hurwitz() directly.

Otherwise, we require that $A$, $K$ and $N$ are all positive. For a finite error bound, we require $K + \text{re}(s) > 1$. To avoid an initial “bump” that steals precision and slows convergence, $AN$ should be at least roughly as large as $|s|$, e.g. it is a good idea to have at least $AN > 0.5|s|$.

If deflate is set, the deflated Hurwitz zeta function is used, removing the pole at $s = 1$.

void acb_dirichlet_hurwitz_precomp_init_num(acb_dirichlet_hurwitz_precomp_t pre, const acb_t s, int deflate, double num_eval, slong prec)

Initializes $pre$, choosing the parameters $A$, $K$, and $N$ automatically to minimize the cost of num_eval evaluations of the Hurwitz zeta function at argument $s$ to precision $prec$.

void acb_dirichlet_hurwitz_precomp_clear(acb_dirichlet_hurwitz_precomp_t pre)

Clears the precomputed data.

void acb_dirichlet_hurwitz_precomp_choose_param(ulong *A, ulong *K, ulong *N, const acb_t s, double num_eval, slong prec)

Chooses precomputation parameters $A$, $K$, and $N$ to minimize the cost of num_eval evaluations of the Hurwitz zeta function at argument $s$ to precision $prec$. If it is estimated that evaluating each Hurwitz zeta function from scratch would be better than performing a precomputation, $A$, $K$ and $N$ are all set to 0.

void acb_dirichlet_hurwitz_precomp_bound(mag_t res, const acb_t s, slong A, slong K, slong N)

Computes an upper bound for the truncation error (not accounting for roundoff error) when evaluating $\zeta(s, a)$ with precomputation parameters $A$, $K$, $N$, assuming that $0 < a \leq 1$. For details, see Algorithms for the Hurwitz zeta function.

void acb_dirichlet_hurwitz_precomp_eval(acb_t res, const acb_dirichlet_hurwitz_precomp_t pre, along p, along q, slong prec)

Evaluates $\zeta(s, p/q)$ using precomputed data, assuming that $0 < p/q \leq 1$.

9.21.7 Lerch transcendent

void acb_dirichlet_lerch_phi_integral(acb_t res, const acb_t z, const acb_t s, const acb_t a, slong prec)

void acb_dirichlet_lerch_phi_direct(acb_t res, const acb_t z, const acb_t s, const acb_t a, slong prec)

void acb_dirichlet_lerch_phi(acb_t res, const acb_t z, const acb_t s, const acb_t a, slong prec)

Computes the Lerch transcendent

$$\Phi(z, s, a) = \sum_{k=0}^{\infty} \frac{z^k}{(k + a)^s}$$
which is analytically continued for $|z| \geq 1$.

The direct version evaluates a truncation of the defining series. The integral version uses the Hankel contour integral

$$
\Phi(z, s, a) = -\frac{\Gamma(1-s)}{2\pi i} \int_C \frac{(-t)^{s-1}e^{-at}}{1 - ze^{-t}} dt
$$

where the path is deformed as needed to avoid poles and branch cuts of the integrand. The default method chooses an algorithm automatically and also checks for some special cases where the function can be expressed in terms of simpler functions (Hurwitz zeta, polylogarithms).

### 9.21.8 Stieltjes constants

#### void acb_dirichlet_stieltjes(acb_t res, const fmpz_t n, const acb_t a, slong prec)

Given a nonnegative integer $n$, sets res to the generalized Stieltjes constant $\gamma_n(a)$ which is the coefficient in the Laurent series of the Hurwitz zeta function at the pole

$$
\zeta(s) = \frac{1}{s-1} + \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \gamma_n(a)(s-1)^n.
$$

With $a = 1$, this gives the ordinary Stieltjes constants for the Riemann zeta function.

This function uses an integral representation to permit fast computation for extremely large $n$ [JB2018]. If $n$ is moderate and the precision is high enough, it falls back to evaluating the Hurwitz zeta function of a power series and reading off the last coefficient.

Note that for computing a range of values $\gamma_0(a), \ldots, \gamma_n(a)$, it is generally more efficient to evaluate the Hurwitz zeta function series expansion once at $s = 1$ than to call this function repeatedly, unless $n$ is extremely large (at least several hundred).

### 9.21.9 Dirichlet character evaluation

#### void acb_dirichlet_chi(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, ulong n, slong prec)

Sets res to $\chi(n)$, the value of the Dirichlet character $\chi$ at the integer $n$.

#### void acb_dirichlet_chi_vec(acb_ptr v, const dirichlet_group_t G, const dirichlet_char_t chi, slong nv, slong prec)

Compute the $nv$ first Dirichlet values.

#### void acb_dirichlet_pairing(acb_t res, const dirichlet_group_t G, ulong m, ulong n, slong prec)

#### void acb_dirichlet_pairing_char(acb_t res, const dirichlet_group_t G, const dirichlet_char_t a, const dirichlet_char_t b, slong prec)

Sets res to the value of the Dirichlet pairing $\chi(m, n)$ at numbers $m$ and $n$. The second form takes two characters as input.

### 9.21.10 Dirichlet character Gauss, Jacobi and theta sums

#### void acb_dirichlet_gauss_sum_naive(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

#### void acb_dirichlet_gauss_sum_factor(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

9.21.10 acb_dirichlet.h – Dirichlet L-functions, Riemann zeta and related functions 707
Sets $res$ to the Gauss sum

$$G_q(a) = \sum_{x \mod q} \chi_q(a, x)e^{2\pi i x/q}$$

- the naive version computes the sum as defined.
- the factor version writes it as a product of local Gauss sums by Chinese remainder theorem.
- the order2 version assumes $\chi$ is real and primitive and returns $i\sqrt{q}$ where $p$ is the parity of $\chi$.
- the theta version assumes that $\chi$ is primitive to obtain the Gauss sum by functional equation of the theta series at $t = 1$. An abort will be raised if the theta series vanishes at $t = 1$. Only 4 exceptional characters of conductor 300 and 600 are known to have this particularity, and none with primepower modulus.
- the default version automatically combines the above methods.
- the ui version only takes the Conrey number $a$ as parameter.

Computes the Jacobi sum

$$J_q(a, b) = \sum_{x \mod q} \chi_q(a, x)\chi_q(b, 1 - x)$$

- the naive version computes the sum as defined.
- the factor version writes it as a product of local Jacobi sums
- the gauss version assumes $ab$ is primitive and uses the formula $J_q(a, b)G_q(ab) = G_q(a)G_q(b)$
- the default version automatically combines the above methods.
- the ui version only takes the Conrey numbers $a$ and $b$ as parameters.
void *acb_dirichlet_ui_theta_arb(acb_t res, const dirichlet_group_t G, ulong a, const arb_t t, slong prec)

Compute the theta series $\Theta_q(a,t)$ for real argument $t > 0$. Beware that if $t < 1$ the functional equation

$$t \Theta(a,t) = \epsilon(\chi) \frac{1}{t}$$

should be used, which is not done automatically (to avoid recomputing the Gauss sum).

We call theta series of a Dirichlet character the quadratic series

$$\Theta_q(a) = \sum_{n \geq 0} \chi_q(a,n)n^p x^{n^2}$$

where $p$ is the parity of the character $\chi_q(a,\cdot)$.

For $\Re(t) > 0$ we write $x(t) = \exp(-\frac{\pi}{N} t^2)$ and define

$$\Theta_q(a,t) = \sum_{n \geq 0} \chi_q(a,n)x(t)^{n^2}.$$ 

ulong *acb_dirichlet_theta_length(ulong q, const arb_t t, slong prec)

Compute the number of terms to be summed in the theta series of argument $t$ so that the tail is less than $2^{-\text{prec}}$.

void *acb_dirichlet_qseries_arb_powers_naive(acb_t res, const arb_t x, int p, const ulong *a, const acb_dirichlet_roots_t z, slong len, slong prec)

void *acb_dirichlet_qseries_arb_powers_smallorder(acb_t res, const arb_t x, int p, const ulong *a, const acb_dirichlet_roots_t z, slong len, slong prec)

Compute the series $\sum n^p z^a x^{n^2}$ for exponent list $a$, precomputed powers $z$ and parity $p$ (being 0 or 1).

The naive version sums the series as defined, while the smallorder variant evaluates the series on the quotient ring by a cyclotomic polynomial before evaluating at the root of unity, ignoring its argument $z$.

9.21.11 Discrete Fourier transforms

If $f$ is a function $\mathbb{Z}/q\mathbb{Z} \to \mathbb{C}$, its discrete Fourier transform is the function defined on Dirichlet characters mod $q$ by

$$\hat{f}(\chi) = \sum_{x \mod q} \overline{\chi(x)} f(x)$$

See the *acb_dft.h – Discrete Fourier transform* module.

Here we take advantage of the Conrey isomorphism $G \to \hat{G}$ to consider the Fourier transform on Conrey labels as

$$g(a) = \sum_{b \mod q} \overline{\chi_q(a,b)} f(b)$$
void \texttt{acb_dirichlet_dft_conrey} (\texttt{acb_ptr w, acb_srcptr v, const dirichlet_group_t G, slong prec})

Compute the DFT of \( v \) using Conrey indices. This function assumes \( v \) and \( w \) are vectors of size \( G->\text{phi}_q \), whose values correspond to a lexicographic ordering of Conrey logs (as obtained using \texttt{dirichlet_char_next()} or by \texttt{dirichlet_char_index()}).

For example, if \( q = 15 \), the Conrey elements are stored in following order

<table>
<thead>
<tr>
<th>index</th>
<th>( \log = [e,f] )</th>
<th>number = ( 7^e 11^f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0, 0]</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[0, 1]</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>[0, 2]</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>[0, 3]</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>[0, 4]</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>[1, 0]</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>[1, 1]</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>[1, 2]</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>[1, 3]</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>[1, 4]</td>
<td>11</td>
</tr>
</tbody>
</table>

void \texttt{acb_dirichlet_dft} (\texttt{acb_ptr w, acb_srcptr v, const dirichlet_group_t G, slong prec})

Compute the DFT of \( v \) using Conrey numbers. This function assumes \( v \) and \( w \) are vectors of size \( G->q \). All values at index not coprime to \( G->q \) are ignored.

### 9.21.12 Dirichlet L-functions

void \texttt{acb_dirichlet_root_number_theta} (\texttt{acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec})

Sets \( \text{res} \) to the root number \( \epsilon(\chi) \) for a primitive character \( \chi \), which appears in the functional equation (where \( p \) is the parity of \( \chi \)):

\[
\left( \frac{q}{\pi} \right)^{s+p/2} \Gamma \left( \frac{s+p}{2} \right) L(s, \chi) = \epsilon(\chi) \left( \frac{q}{\pi} \right)^{1-s+p/2} \Gamma \left( \frac{1-s+p}{2} \right) L(1-s, \chi) \]

- The \texttt{theta} variant uses the evaluation at \( t = 1 \) of the Theta series.
- The default version computes it via the Gauss sum.

void \texttt{acb_dirichlet_l_hurwitz} (\texttt{acb_t res, const acb_t s, const acb_dirichlet_hurwitz_precomp_t precomp, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec})

Computes \( L(s, \chi) \) using decomposition in terms of the Hurwitz zeta function

\[
L(s, \chi) = q^{-s} \sum_{k=1}^{q} \chi(k) \zeta \left( s, \frac{k}{q} \right).
\]

If \( s = 1 \) and \( \chi \) is non-principal, the deflated Hurwitz zeta function is used to avoid poles.

If \texttt{precomp} is \texttt{NULL}, each Hurwitz zeta function value is computed directly. If a pre-initialized \texttt{precomp} object is provided, this will be used instead to evaluate the Hurwitz zeta function.

void \texttt{acb_dirichlet_l_euler_product} (\texttt{acb_t res, const acb_t s, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec})

710 Chapter 9. Real and complex numbers
void _acb_dirichlet_euler_product_real_ui(arb_t res, along s, const signed char *chi, int mod, int reciprocal, slong prec)

Computes $L(s, \chi)$ directly using the Euler product. This is efficient if $s$ has large positive real part. As implemented, this function only gives a finite result if $\Re(s) \geq 2$.

An error bound is computed via mag_hurwitz_zeta_uui(). If $s$ is complex, replace it with its real part. Since

$$
\frac{1}{L(s, \chi)} = \prod_p \left(1 - \frac{\chi(p)}{p^s}\right) = \sum_{k=1}^{\infty} \frac{\mu(k)\chi(k)}{k^s}
$$

and the truncated product gives all smooth-index terms in the series, we have

$$
\left| \prod_{p \leq N} \left(1 - \frac{\chi(p)}{p^s}\right) - \frac{1}{L(s, \chi)} \right| \leq \sum_{k=N}^{\infty} \frac{1}{k^s} = \zeta(s, N).
$$

The underscore version specialized for integer $s$ assumes that $\chi$ is a real Dirichlet character given by the explicit list chi of character values at 0, 1, ..., mod - 1. If reciprocal is set, it computes $1/L(s, \chi)$ (this is faster if the reciprocal can be used directly).

void acb_dirichlet_l(acb_t res, const acb_t s, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

Computes $L(s, \chi)$ using a default choice of algorithm.

void acb_dirichlet_l_fmpq(acb_t res, const fmpq_t s, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

void acb_dirichlet_l_fmpq_afe(acb_t res, const fmpq_t s, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)

Computes $L(s, \chi)$ where $s$ is a rational number. The afe version uses the approximate functional equation; the default version chooses an algorithm automatically.

void acb_dirichlet_l_vec_hurwitz(acb_ptr res, const acb_t s, const acb_dirichlet_hurwitz_precomp_t precomp, const dirichlet_group_t G, slong prec)

Compute all values $L(s, \chi)$ for $\chi mod q$, using the Hurwitz zeta function and a discrete Fourier transform. The output res is assumed to have length $G->\phi_q$ and values are stored by lexicographically ordered Conrey logs. See acb_dirichlet_dft_conrey().

If precomp is NULL, each Hurwitz zeta function value is computed directly. If a pre-initialized precomp object is provided, this will be used instead to evaluate the Hurwitz zeta function.

void acb_dirichlet_l_jet(acb_ptr res, const acb_t s, const dirichlet_group_t G, const dirichlet_char_t chi, int deflate, slong len, slong prec)

Computes the Taylor expansion of $L(s, \chi)$ to length len, i.e. $L(s), L'(s), \ldots, L^{(len-1)}(s)/(len - 1)!$. If deflate is set, computes the expansion of

$$
L(s, \chi) = \sum_{k=1}^{q} \frac{\chi(k)}{(s - 1)q}
$$

instead. If chi is a principal character, then this has the effect of subtracting the pole with residue $\sum_{k=1}^{\phi(q)} \chi(k) = \phi(q)/q$ that is located at $s = 1$.

In particular, when evaluated at $s = 1$, this gives the regular part of the Laurent expansion. When chi is non-principal, deflate has no effect.

void _acb_dirichlet_l_series(acb_ptr res, acb_srcptr s, slong slen, const dirichlet_group_t G, const dirichlet_char_t chi, int deflate, slong len, slong prec)

void acb_dirichlet_l_series(acb_poly_t res, const acb_poly_t s, const dirichlet_group_t G, const dirichlet_char_t chi, int deflate, slong len, slong prec)

Sets res to the power series $L(s, \chi)$ where $s$ is a given power series, truncating the result to length len. See acb_dirichlet_l_jet() for the meaning of the deflate flag.
9.21.13 Hardy Z-functions

For convenience, setting both $G$ and $\chi$ to $NULL$ in the following methods selects the Riemann zeta function.

Currently, these methods require $\chi$ to be a primitive character.

```c
void acb_dirichlet_hardy_theta(acb_ptr res, const acb_t t, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)
```

Computes the phase function used to construct the Z-function. We have

$$\theta(t) = -\frac{t}{2} \log(\pi/q) - \frac{i \log(\epsilon)}{2} + \frac{\log \Gamma((s + \delta)/2) - \log \Gamma((1 - s + \delta)/2)}{2i}$$

where $s = 1/2 + it$, $\delta$ is the parity of $\chi$, and $\epsilon$ is the root number as computed by `acb_dirichlet_root_number()`. The first $\text{len}$ terms in the Taylor expansion are written to the output.

```c
void acb_dirichlet_hardy_z(acb_ptr res, const acb_t t, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)
```

Computes the Hardy Z-function, also known as the Riemann-Siegel Z-function $Z(t) = e^{\theta(t)} L(1/2 + it)$, which is real-valued for real $t$. The first $\text{len}$ terms in the Taylor expansion are written to the output.

```c
void _acb_dirichlet_hardy_theta_series(acb_ptr res, acb_srcptr t, slong tlen, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)
```

Sets $\text{res}$ to the power series $\theta(t)$ where $t$ is a given power series, truncating the result to length $\text{len}$.

```c
void _acb_dirichlet_hardy_z_series(acb_ptr res, acb_srcptr t, slong tlen, const dirichlet_group_t G, const dirichlet_char_t chi, slong len, slong prec)
```

Sets $\text{res}$ to the power series $Z(t)$ where $t$ is a given power series, truncating the result to length $\text{len}$.

9.21.14 Gram points

```c
void acb_dirichlet_gram_point(arb_t res, const fmpz_t n, const dirichlet_group_t G, const dirichlet_char_t chi, slong prec)
```

Sets $\text{res}$ to the $n$-th Gram point $g_n$, defined as the unique solution in $[7, \infty)$ of $\theta(g_n) = \pi n$. Currently only the Gram points corresponding to the Riemann zeta function are supported and $G$ and $\chi$ must both be set to $NULL$. Requires $n \geq -1$. 

Chapter 9. Real and complex numbers
9.21.15 Riemann zeta function zeros

The following functions for counting and isolating zeros of the Riemann zeta function use the ideas from Turing’s method in mpmath [Joh2018b] by Juan Arias de Reyna, described in [Ari2012].

`ulong acb_dirichlet_turing_method_bound(const fmpz_t p)`

Computes an upper bound $B$ for the minimum number of consecutive good Gram blocks sufficient to count nontrivial zeros of the Riemann zeta function using Turing’s method [Tur1953] as updated by [Leh1970], [Bre1979], and [Tru2011].

Let $N(T)$ denote the number of zeros (counted according to their multiplicities) of $\zeta(s)$ in the region $0 < \text{Im}(s) \leq T$. If at least $B$ consecutive Gram blocks with union $[g_n,g_p)$ satisfy Rosser’s rule, then $N(g_n) \leq n + 1$ and $N(g_p) \geq p + 1$.

`int acb_dirichlet_definite_hardy_z_zero(arb_t res, const arf_t t, slong *pprec)`

Sets $\rho$ to the Hardy Z-function $Z(t)$. The initial precision (* pprec) is increased as necessary to determine the sign of $Z(t)$. The sign is returned.

`void acb_dirichlet_isolate_gram_hardy_z_zero(arf_t a, arf_t b, const fmpz_t n)`

Uses Gram’s law to compute an interval $(a,b)$ that contains the $n$-th zero of the Hardy Z-function and no other zero. Requires $1 \leq n \leq 126$.

`void acb_dirichlet_isolate_rosser_hardy_z_zero(arf_t a, arf_t b, const fmpz_t n)`

Uses Rosser’s rule to compute an interval $(a,b)$ that contains the $n$-th zero of the Hardy Z-function and no other zero. Requires $1 \leq n \leq 13999526$.

`void acb_dirichlet_isolate_turing_hardy_z_zero(arf_t a, arf_t b, const fmpz_t n)`

Computes an interval $(a,b)$ that contains the $n$-th zero of the Hardy Z-function and no other zero, following Turing’s method. Requires $n \geq 2$.

`void acb_dirichlet_isolate_hardy_z_zero(arf_t a, arf_t b, const fmpz_t n)`

Computes an interval $(a,b)$ that contains the $n$-th zero of the Hardy Z-function and contains no other zero, using the most appropriate underscore version of this function. Requires $n \geq 1$.

`void acb_dirichlet_refine_hardy_z_zero(arb_t res, const arf_t a, const arf_t b, slong prec)`

Sets $\rho$ to the unique zero of the Hardy Z-function in the interval $(a,b)$.

`void acb_dirichlet_hardy_z_zero(arb_t res, const fmpz_t n, slong prec)`

Sets $\rho$ to the $n$-th zero of the Hardy Z-function, requiring $n \geq 1$.

`void acb_dirichlet_hardy_z_zeros(arb_ptr res, const fmpz_t n, slong len, slong prec)`

Sets the entries of $\rho$ to $\rho_1$ consecutive nontrivial zeros of $\zeta(s)$ beginning with the $n$-th zero. Requires positive $n$.

`void acb_dirichlet_zeta_zero(arb_t res, const fmpz_t n, slong prec)`

Sets $\rho$ to the $n$-th nontrivial zero of $\zeta(s)$, requiring $n \geq 1$.

`void acb_dirichlet_zeta_zeros(arb_ptr res, const fmpz_t n, slong len, slong prec)`

Sets the entries of $\rho$ to $\rho_1$ consecutive nontrivial zeros of $\zeta(s)$ beginning with the $n$-th zero. Requires positive $n$.

`void acb_dirichlet_exact_zeta_nzeros(fmpz_t res, const arf_t t)`

Computes the number of zeros (counted according to their multiplicities) of $\zeta(s)$ in the region $0 < \text{Im}(s) \leq t$.

`void acb_dirichlet_backlund_s(arb_t res, const arf_t t, slong prec)`

Computes $S(t) = \frac{1}{2} \text{arg} \zeta(\frac{1}{2} + it)$ where the argument is defined by continuous variation of $s$ in $\zeta(s)$ starting at $s = 2$, then vertically to $s = 2 + it$, then horizontally to $s = \frac{1}{2} + it$. In particular, arg in this context is not the principal value of the argument, and it cannot be computed.
directly by \textit{acb_arg}(). In practice \(S(t)\) is computed as \(S(t) = N(t) - \frac{1}{\pi} \theta(t) - 1\) where \(N(t)\) is \textit{acb_dirichlet_zeta_nzeros()} and \(\theta(t)\) is \textit{acb_dirichlet_hardy_theta()}.

\begin{verbatim}
void acb_dirichlet_backlund_s_bound(mag_t res, const arb_t t)
  Compute an upper bound for |\(S(t)\)| quickly. Theorem 1 and the bounds in (1.2) in [Tru2014] are used.

void acb_dirichlet_zeta_nzeros_gram(fmpz_t res, const fmpz_t n)
  Compute \(N(g_n)\). That is, compute the number of zeros (counted according to their multiplicities) of \(\zeta(s)\) in the region \(0 < \text{Im}(s) \leq g_n\) where \(g_n\) is the \(n\)-th Gram point. Requires \(n \geq -1\).

slong acb_dirichlet_backlund_s_gram(const fmpz_t n)
  Compute \(S(g_n)\) where \(g_n\) is the \(n\)-th Gram point. Requires \(n \geq -1\).
\end{verbatim}

### 9.21.16 Riemann zeta function zeros (Platt’s method)

The following functions related to the Riemann zeta function use the ideas and formulas described by David J. Platt in [Pla2017].

\begin{verbatim}
void acb_dirichlet_platt_scaled_lambda(arb_t res, const arb_t t, slong prec)
  Compute \(\Lambda(t)\) using \(\pi^{\alpha} \Gamma\left(\frac{\pi + it}{2}\right) \zeta\left(\frac{1}{2} + it\right)\) and is designed to counteract the decay of the gamma factor as \(t\) increases.

void acb_dirichlet_platt_scaled_lambda_vec(arb_ptr res, const fmpz_t T, slong A, slong B, slong prec)

void acb_dirichlet_platt_multieval(arb_ptr res, const fmpz_t T, slong A, slong B, const arb_t h, const fmpz_t J, slong K, slong sigma, slong prec)

void acb_dirichlet_platt_multieval_threaded(arb_ptr res, const fmpz_t T, slong A, slong B, const arb_t h, const fmpz_t J, slong K, slong sigma, slong prec)
  Compute \(\textit{acb_dirichlet_platt_scaled_lambda()}\) at \(N = AB\) points on a grid, following the notation of [Pla2017]. The first point on the grid is \(T-B/2\) and the distance between grid points is \(1/A\). The product \(N = AB\) must be an even integer. The multieval versions evaluate the function at all points on the grid simultaneously using discrete Fourier transforms, and they require the four additional tuning parameters \(h\), \(J\), \(K\), and \(\sigma\). The \textit{threaded} multieval version splits the computation over the number of threads returned by \textit{flint_get_num_threads()}, while the default multieval version chooses whether to use multithreading automatically.

void acb_dirichlet_platt_ws_interpolation(arb_t res, arf_t deriv, const arb_t t0, arb_srcptr p, const fmpz_t T, slong A, slong B, slong Nmax, const arb_t H, slong sigma, slong prec)

Compute \(\textit{acb_dirichlet_platt_scaled_lambda()}\) at \(t0\) by Gaussian-windowed Whittaker-Shannon interpolation of points evaluated by \(\textit{acb_dirichlet_platt_scaled_lambda_vec()}\). The derivative is also approximated if the output parameter \(\textit{deriv}\) is not \textit{NULL}. \(\textit{Ns_max}\) defines the maximum number of supporting points to be used in the interpolation on either side of \(t0\). \(H\) is the standard deviation of the Gaussian window centered on \(t0\) to be applied before the interpolation. \(\sigma\) is an odd positive integer tuning parameter \(\sigma \in 2\mathbb{Z}_{>0} + 1\) used in computing error bounds.

slong _acb_dirichlet_platt_local_hardy_z_zeros(arb_ptr res, const fmpz_t n, slong len, const fmpz_t T, slong A, slong B, const arb_t h, const fmpz_t J, slong K, slong sigma_grid, slong Nmax, const arb_t H, slong sigma_interp, slong prec)
\end{verbatim}
slong acb_dirichlet_platt_local_hardy_z_zeros(arb_ptr res, const fmpz_t n, slong len, slong prec)

slong acb_dirichlet_platt_hardy_z_zeros(arb_ptr res, const fmpz_t n, slong len, slong prec)
Sets at most the first len entries of res to consecutive zeros of the Hardy Z-function starting with the n-th zero. The number of obtained consecutive zeros is returned. The first two function variants each make a single call to Platt’s grid evaluation of the scaled Lambda function, whereas the third variant performs as many evaluations as necessary to obtain len consecutive zeros. The final several parameters of the underscored local variant have the same meanings as in the functions acb_dirichlet_platt_multieval() and acb_dirichlet_platt_ws_interpolation(). The non-underscored variants currently expect \(10^4 \leq n \leq 10^{23}\). The user has the option of multi-threading through flint_set_num_threads(numthreads).

slong acb_dirichlet_platt_zeta_zeros(acb_ptr res, const fmpz_t n, slong len, slong prec)
Sets at most the first len entries of res to consecutive zeros of the Riemann zeta function starting with the n-th zero. The number of obtained consecutive zeros is returned. It currently expects \(10^4 \leq n \leq 10^{23}\). The user has the option of multi-threading through flint_set_num_threads(numthreads).

9.22 bernoulli.h – support for Bernoulli numbers

This module provides helper functions for exact or approximate calculation of the Bernoulli numbers, which are defined by the exponential generating function
\[
\frac{x}{e^x-1} = \sum_{n=0}^{\infty} B_n \frac{x^n}{n!}.
\]

Efficient algorithms are implemented for both multi-evaluation and calculation of isolated Bernoulli numbers. A global (or thread-local) cache is also provided, to support fast repeated evaluation of various special functions that depend on the Bernoulli numbers (including the gamma function and the Riemann zeta function).

9.22.1 Generation of Bernoulli numbers

type bernoulli_rev_t
An iterator object for generating a range of even-indexed Bernoulli numbers exactly in reverse order, i.e. computing the exact fractions \(B_n, B_{n-2}, B_{n-4}, \ldots, B_0\). The Bernoulli numbers are generated from scratch, i.e. no caching is performed.

The Bernoulli numbers are computed by direct summation of the zeta series. This is made fast by storing a table of powers (as done by [Blo2009]). As an optimization, we only include the odd powers, and use fixed-point arithmetic.

The reverse iteration order is preferred for performance reasons, as the powers can be updated using multiplications instead of divisions, and we avoid having to periodically recompute terms to higher precision. To generate Bernoulli numbers in the forward direction without having to store all of them, one can split the desired range into smaller blocks and compute each block with a single reverse pass.

void bernoulli_rev_init(bernoulli_rev_t iter, ulong n)
Initializes the iterator iter. The first Bernoulli number to be generated by calling bernoulli_rev_next() is \(B_n\). It is assumed that \(n\) is even.

void bernoulli_rev_next(fmpz_t numer, fmpz_t denom, bernoulli_rev_t iter)
Sets numer and denom to the exact, reduced numerator and denominator of the Bernoulli number \(B_k\) and advances the state of iter so that the next invocation generates \(B_{k-2}\).
void `bernoulli_rev_clear`(`bernoulli_rev_t` `iter`)

Frees all memory allocated internally by `iter`.

void `bernoulli_fmpq_vec_no_cache`(`fmpq` *`res`, `ulong` `a`, `slong` `num`)

Writes `num` consecutive Bernoulli numbers to `res` starting with $B_a$. This function is not currently optimized for a small count `num`. The entries are not read from or written to the Bernoulli number cache; if retrieving a vector of Bernoulli numbers is needed more than once, use `bernoulli_cache_compute()` followed by `bernoulli_fmpq_ui()` instead.

This function is a wrapper for the `rev` iterators. It can use multiple threads internally.

### 9.22.2 Caching

`slong` `bernoulli_cache_num`

`fmpq` *`bernoulli_cache`

Cache of Bernoulli numbers. Uses thread-local storage if enabled in FLINT.

void `bernoulli_cache_compute`(`slong` `n`)

Makes sure that the Bernoulli numbers up to at least $B_{n-1}$ are cached. Calling `flint_cleanup()` frees the cache.

The cache is extended by calling `bernoulli_fmpq_vec_no_cache()` internally.

### 9.22.3 Bounding

`slong` `bernoulli_bound_2exp_si`(`ulong` `n`)

Returns an integer $b$ such that $|B_n| \leq 2^b$. Uses a lookup table for small $n$, and for larger $n$ uses the inequality $|B_n| < 4n!/(2\pi)^n < 4(n + 1)^n e^{-n}/(2\pi)^n$. Uses integer arithmetic throughout, with the bound for the logarithm being looked up from a table. If $|B_n| = 0$, returns `LONG_MIN`. Otherwise, the returned exponent $b$ is never more than one percent larger than the true magnitude.

This function is intended for use when $n$ small enough that one might comfortably compute $B_n$ exactly. It aborts if $n$ is so large that internal overflow occurs.

### 9.22.4 Isolated Bernoulli numbers

`ulong` `bernoulli_mod_p_harvey`(`ulong` `n`, `ulong` `p`)

Returns the $B_n$ modulo the prime number $p$, computed using Harvey’s algorithm [Har2010]. The running time is linear in $p$. If $p$ divides the numerator of $B_n$, `UWORD_MAX` is returned as an error code.

void `_bernoulli_fmpq_ui_zeta`(`fmpz_t` `num`, `fmpz_t` `den`, `ulong` `n`)

void `_bernoulli_fmpq_ui_multi_mod`(`fmpz_t` `num`, `fmpz_t` `den`, `ulong` `n`, `double` `alpha`)

Sets `num` and `den` to the reduced numerator and denominator of the Bernoulli number $B_n$.

The `zeta` version computes the denominator $d$ using the von Staudt-Clausen theorem, numerically approximates $B_n$ using `arb_bernoulli_ui_zeta()`, and then rounds $dB_n$ to the correct numerator.

The `multi_mod` version reconstructs $B_n$ by computing the high bits via the Riemann zeta function and the low bits via Harvey’s multimodular algorithm. The tuning parameter `alpha` should be a fraction between 0 and 1 controlling the number of bits to compute by the multimodular algorithm. If set to a negative number, a default value will be used.

void `_bernoulli_fmpq_ui`(`fmpz_t` `num`, `fmpz_t` `den`, `ulong` `n`)

716 Chapter 9. Real and complex numbers
void bernoulli_fmpq_ui(fmpq_t b, ulong n)

Computes the Bernoulli number $B_n$ as an exact fraction, for an isolated integer $n$. This function reads $B_n$ from the global cache if the number is already cached, but does not automatically extend the cache by itself.

## 9.23 hypgeom.h – support for hypergeometric series

This module provides functions for high-precision evaluation of series of the form

$$
\sum_{k=0}^{n-1} \frac{A(k)}{B(k)} \prod_{j=1}^{k} \frac{P(j)}{Q(j)} z^k
$$

where $A, B, P, Q$ are polynomials. The present version only supports $A, B, P, Q \in \mathbb{Z}[k]$ (represented using the FLINT fmpz_poly_t type). This module also provides functions for high-precision evaluation of infinite series ($n \to \infty$), with automatic, rigorous error bounding.

Note that we can standardize to $A = B = 1$ by setting $\tilde{P}(k) = P(k)A(k)B(k-1), \tilde{Q}(k) = Q(k)A(k-1)B(k)$. However, separating out $A$ and $B$ is convenient and improves efficiency during evaluation.

### 9.23.1 Strategy for error bounding

We wish to evaluate $S(z) = \sum_{k=n}^{\infty} T(k)z^k$ where $T(k)$ satisfies $T(0) = 1$ and

$$
T(k) = R(k)T(k-1) = \left(\frac{P(k)}{Q(k)}\right) T(k-1)
$$

for given polynomials

$$
P(k) = a_p k^p + a_{p-1} k^{p-1} + \ldots a_0 \\
Q(k) = b_q k^q + b_{q-1} k^{q-1} + \ldots b_0.
$$

For convergence, we require $p < q$, or $p = q$ with $|z||a_p| < |b_q|$. We also assume that $P(k)$ and $Q(k)$ have no roots among the positive integers (if there are positive integer roots, the sum is either finite or undefined). With these conditions satisfied, our goal is to find a parameter $n \geq 0$ such that

$$
\left| \sum_{k=n}^{\infty} T(k)z^k \right| \leq 2^{-d}.
$$

We can rewrite the hypergeometric term ratio as

$$
z R(k) = z \frac{P(k)}{Q(k)} = z \left(\frac{a_p}{b_q}\right) \frac{1}{k^{q-p}} F(k)
$$

where

$$
F(k) = \frac{1 + \tilde{a}_1/k + \tilde{a}_2/k^2 + \ldots + \tilde{a}_q/k^p}{1 + b_1/k + b_2/k^2 + \ldots + b_q/k^q} = 1 + O(1/k)
$$

and where $\tilde{a}_i = a_{p-i}/a_p$, $\tilde{b}_i = b_{q-i}/b_q$. Next, we define

$$
C = \max_{1 \leq i \leq p} |\tilde{a}_i|^{(1/i)}, \quad D = \max_{1 \leq i \leq q} |\tilde{b}_i|^{(1/i)}.
$$

Now, if $k > C$, the magnitude of the numerator of $F(k)$ is bounded from above by

$$
1 + \sum_{i=1}^{p} \left(\frac{C}{k}\right)^i \leq 1 + \frac{C}{k-C}
$$

9.23. hypgeom.h – support for hypergeometric series 717
and if $k > 2D$, the magnitude of the denominator of $F(k)$ is bounded from below by

$$1 - \frac{q}{k} \left(\frac{D}{k}\right)^i \geq 1 + \frac{D}{D-k}.$$ 

Putting the inequalities together gives the following bound, valid for $k > K = \max(C, 2D)$:

$$|F(k)| \leq \frac{k(k-D)}{(k-C)(k-2D)} = \left(1 + \frac{C}{k-C}\right) \left(1 + \frac{D}{k-2D}\right).$$

Let $r = q - p$ and $\tilde{z} = |pz_p/b_q|$. Assuming $k > \max(C, 2D, \tilde{z}^{1/r})$, we have

$$|zR(k)| \leq G(k) = \frac{\tilde{z}F(k)}{k^r}$$

where $G(k)$ is monotonically decreasing. Now we just need to find an $n$ such that $G(n) < 1$ and for which $|T(n)|/(1 - G(n)) \leq 2^{-d}$. This can be done by computing a floating-point guess for $n$ then trying successively larger values.

This strategy leaves room for some improvement. For example, if $\tilde{b}_1$ is positive and large, the bound $B$ becomes very pessimistic (a larger positive $\tilde{b}_1$ causes faster convergence, not slower convergence).

### 9.23.2 Types, macros and constants

type hypgeom_struct

type hypgeom_t
Stores polynomials $A$, $B$, $P$, $Q$ and precomputed bounds, representing a fixed hypergeometric series.

### 9.23.3 Memory management

void hypgeom_init(hypgeom_t hyp)

void hypgeom_clear(hypgeom_t hyp)

### 9.23.4 Error bounding

**slong hypgeom_estimate_terms**(const mag_t z, int r, slong d)
Computes an approximation of the largest $n$ such that $|z|^n/(n!)^r = 2^{-d}$, giving a first-order estimate of the number of terms needed to approximate the sum of a hypergeometric series of weight $r \geq 0$ and argument $z$ to an absolute precision of $d \geq 0$ bits. If $r = 0$, the direct solution of the equation is given by $n = (\log(1 - z) - d \log 2)/\log z$. If $r > 0$, using $\log n! \approx n \log n - n$ gives an equation that can be solved in terms of the Lambert $W$-function as $n = (d \log 2)/(r W(t))$ where $t = (d \log 2)/(\varepsilon r^{1/r})$.

The evaluation is done using double precision arithmetic. The function aborts if the computed value of $n$ is greater than or equal to LONG_MAX / 2.

**slong hypgeom_bound**(mag_t error, int r, slong C, slong D, slong K, const mag_t TK, const mag_t z, slong prec)
Computes a truncation parameter sufficient to achieve $\text{prec}$ bits of absolute accuracy, according to the strategy described above. The input consists of $r$, $C$, $D$, $K$, precomputed bound for $T(K)$, and $\tilde{z} = z(z_p/b_q)$, such that for $k > K$, the hypergeometric term ratio is bounded by

$$\frac{\tilde{z} k(k-D)}{k^r (k-C)(k-2D)}.$$ 

Given this information, we compute a $\varepsilon$ and an integer $n$ such that $\sum_{k=n}^{\infty} T(k) \leq \varepsilon \leq 2^{-\text{prec}}$. The output variable error is set to the value of $\varepsilon$, and $n$ is returned.
void hypgeom_precompute(hypgeom_t hyp)

Precomputes the bounds data $C$, $D$, $K$ and an upper bound for $T(K)$.

### 9.23.5 Summation

void arb_hypgeom_sum(arb_t P, arb_t Q, const hypgeom_t hyp, slong n, slong prec)

Computes $P, Q$ such that $P/Q = \sum_{k=0}^{n-1} T(k)$ where $T(k)$ is defined by $hyp$, using binary splitting and a working precision of $prec$ bits.

void arb_hypgeom_infsun(arb_t P, arb_t Q, hypgeom_t hyp, slong tol, slong prec)

Computes $P, Q$ such that $P/Q = \sum_{k=0}^{\infty} T(k)$ where $T(k)$ is defined by $hyp$, using binary splitting and working precision of $prec$ bits. The number of terms is chosen automatically to bound the truncation error by at most $2^{-tol}$. The bound for the truncation error is included in the output as part of $P$.

### 9.24 partitions.h – computation of the partition function

This module implements the asymptotically fast algorithm for evaluating the integer partition function $p(n)$ described in [Joh2012]. The idea is to evaluate a truncation of the Hardy-Ramanujan-Rademacher series using tight precision estimates, and symbolically factoring the occurring exponential sums.

An implementation based on floating-point arithmetic can also be found in FLINT. That version relies on some numerical subroutines that have not been proved correct.

The implementation provided here uses ball arithmetic throughout to guarantee a correct error bound for the numerical approximation of $p(n)$. Optionally, hardware double arithmetic can be used for low-precision terms. This gives a significant speedup for small (e.g. $n < 10^6$).

void partitions_rademacher_bound(arf_t b, const fmpz_t n, ulong N)

Sets $b$ to an upper bound for

$$M(n, N) = \frac{44n^2}{225\sqrt{3}} N^{-1/2} + \frac{\pi\sqrt{2}}{75} \left( \frac{N}{n-1} \right)^{1/2} \sinh \left( \frac{\pi}{N} \sqrt{\frac{2n}{3}} \right).$$

This formula gives an upper bound for the truncation error in the Hardy-Ramanujan-Rademacher formula when the series is taken up to the term $t(n, N)$ inclusive.

void partitions_hrr_sum_arb(arb_t x, const fmpz_t n, slong N0, slong N, int use_doubles)

Evaluates the partial sum $\sum_{k=N0}^{N} t(n, k)$ of the Hardy-Ramanujan-Rademacher series.

If `use_doubles` is nonzero, doubles and the system’s standard library math functions are used to evaluate the smallest terms. This significantly speeds up evaluation for small $n$ (e.g. $n < 10^6$), and gives a small speed improvement for larger $n$, but the result is not guaranteed to be correct.

In practice, the error is estimated very conservatively, and unless the system’s standard library is broken, use of doubles can be considered safe. Setting `use_doubles` to zero gives a fully guaranteed bound.

void partitions_fmpz_fmpz(fmpz_t p, const fmpz_t n, int use_doubles)

Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.

If $n$ is sufficiently large and a number of threads greater than 1 has been selected with `flint_set_num_threads()`, the computation time will be reduced by using two threads.

See `partitions_hrr_sum_arb()` for an explanation of the `use_doubles` option.
void partitions_fmpz_ui(fmpz_t p, ulong n)
Computes the partition function \( p(n) \) using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing \( p(n) \) and verifies that the ball contains a unique integer.

void partitions_fmpz_ui_using_doubles(fmpz_t p, ulong n)
Computes the partition function \( p(n) \), enabling the use of doubles internally. This significantly speeds up evaluation for small \( n \) (e.g. \( n < 10^6 \)), but the error bounds are not certified (see remarks for partitions_hrr_sum_arb()).

void partitions_leading_fmpz(arb_t res, const fmpz_t n, slong prec)
Sets \( \text{res} \) to the leading term in the Hardy-Ramanujan series for \( p(n) \) (without Rademacher’s correction of this term, which is vanishingly small when \( n \) is large), that is, \( \sqrt{12(1 - \frac{1}{t})e^{t} / (24n - 1)} \) where \( t = \pi \sqrt{24n - 1}/6 \).

9.25 arb_calc.h – calculus with real-valued functions

This module provides functions for operations of calculus over the real numbers (intended to include root-finding, optimization, integration, and so on). It is planned that the module will include two types of algorithms:

- Interval algorithms that give provably correct results. An example would be numerical integration on an interval by dividing the interval into small balls and evaluating the function on each ball, giving rigorous upper and lower bounds.

- Conventional numerical algorithms that use heuristics to estimate the accuracy of a result, without guaranteeing that it is correct. An example would be numerical integration based on pointwise evaluation, where the error is estimated by comparing the results with two different sets of evaluation points. Ball arithmetic then still tracks the accuracy of the function evaluations.

Any algorithms of the second kind will be clearly marked as such.

9.25.1 Types, macros and constants

type arb_calc_func_t
Typedef for a pointer to a function with signature:

```c
int func(arb_ptr out, const arb_t inp, void * param, slong order, slong prec)
```

implementing a univariate real function \( f(x) \). When called, \( \text{func} \) should write to \( \text{out} \) the first \( \text{order} \) coefficients in the Taylor series expansion of \( f(x) \) at the point \( \text{inp} \), evaluated at a precision of \( \text{prec} \) bits. The \( \text{param} \) argument may be used to pass through additional parameters to the function. The return value is reserved for future use as an error code. It can be assumed that \( \text{out} \) and \( \text{inp} \) are not aliased and that \( \text{order} \) is positive.

ARB_CALC_SUCCESS
Return value indicating that an operation is successful.

ARB_CALC_IMPRECISE_INPUT
Return value indicating that the input to a function probably needs to be computed more accurately.

ARB_CALC_NO_CONVERGENCE
Return value indicating that an algorithm has failed to convergence, possibly due to the problem not having a solution, the algorithm not being applicable, or the precision being insufficient.
9.25.2 Debugging

int arb_calc_verbose
    If set, enables printing information about the calculation to standard output.

9.25.3 Subdivision-based root finding

type arf_interval_struct

type arf_interval_t
    An arf_interval_struct consists of a pair of arf_struct, representing an interval used for subdivision-based root-finding. An arf_interval_t is defined as an array of length one of type arf_interval_struct, permitting an arf_interval_t to be passed by reference.

type arf_interval_ptr
    Alias for arf_interval_struct *, used for vectors of intervals.

type arf_interval_srcptr
    Alias for const arf_interval_struct *, used for vectors of intervals.

void arf_interval_init(arf_interval_t v)
void arf_interval_clear(arf_interval_t v)

arf_interval_ptr _arf_interval_vec_init(slong n)

void _arf_interval_vec_clear(arf_interval_ptr v, slong n)

void arf_interval_set(arf_interval_t v, const arf_interval_t u)

void arf_interval_swap(arf_interval_t v, arf_interval_t u)

void arf_interval_get_arb(arb_t x, const arf_interval_t v, slong prec)

void arf_interval_printd(const arf_interval_t v, slong n)
    Helper functions for endpoint-based intervals.

void arf_interval_fprintd(FILE *file, const arf_interval_t v, slong n)
    Helper functions for endpoint-based intervals.

slong arb_calc_isolate_roots(arf_interval_ptr *found, int **flags, arb_calc_func_t func, void *param, const arf_interval_t interval, slong maxdepth, slong maxeval, slong maxfound, slong prec)
    Rigorously isolates single roots of a real analytic function on the interior of an interval.

    This routine writes an array of n interesting subintervals of interval to found and corresponding flags to flags, returning the integer n. The output has the following properties:
    • The function has no roots on interval outside of the output subintervals.
    • Subintervals are sorted in increasing order (with no overlap except possibly starting and ending with the same point).
    • Subintervals with a flag of 1 contain exactly one (single) root.
    • Subintervals with any other flag may or may not contain roots.

    If no flags other than 1 occur, all roots of the function on interval have been isolated. If there are output subintervals on which the existence or nonexistence of roots could not be determined, the user may attempt further searches on those subintervals (possibly with increased precision and/or increased bounds for the breaking criteria). Note that roots of multiplicity higher than one and roots located exactly at endpoints cannot be isolated by the algorithm.

    The following breaking criteria are implemented:
- At most $maxdepth$ recursive subdivisions are attempted. The smallest details that can be distinguished are therefore about $2^{-maxdepth}$ times the width of interval. A typical, reasonable value might be between 20 and 50.

- If the total number of tested subintervals exceeds $maxeval$, the algorithm is terminated and any untested subintervals are added to the output. The total number of calls to $func$ is thereby restricted to a small multiple of $maxeval$ (the actual count can be slightly higher depending on implementation details). A typical, reasonable value might be between 100 and 100000.

- The algorithm terminates if $maxfound$ roots have been isolated. In particular, setting $maxfound$ to 1 can be used to locate just one root of the function even if there are numerous roots. To try to find all roots, $LONG_MAX$ may be passed.

The argument $prec$ denotes the precision used to evaluate the function. It is possibly also used for some other arithmetic operations performed internally by the algorithm. Note that it probably does not make sense for $maxdepth$ to exceed $prec$.

Warning: it is assumed that subdivision points of $interval$ can be represented exactly as floating-point numbers in memory. Do not pass $1 \pm 2^{-100}$ as input.

```c
int arb_calc_refine_root_bisect(arf_interval_t r, arb_calc_func_t func, void *param, const arb_interval_t start, slong iter, slong prec)
```

Given an interval $start$ known to contain a single root of $func$, refines it using $iter$ bisection steps. The algorithm can return a failure code if the sign of the function at an evaluation point is ambiguous. The output $r$ is set to a valid isolating interval (possibly just $start$) even if the algorithm fails.

### 9.25.4 Newton-based root finding

```c
void arb_calc_newton_conv_factor(arf_t conv_factor, arb_calc_func_t func, void *param, const arb_t conv_region, slong prec)
```

Given an interval $I$ specified by $conv_region$, evaluates a bound for $C = \sup_{t,u \in I} \frac{1}{2} |f''(t)||f'(u)|$, where $f$ is the function specified by $func$ and $param$. The bound is obtained by evaluating $f'(I)$ and $f''(I)$ directly. If $f$ is ill-conditioned, $I$ may need to be extremely precise in order to get an effective, finite bound for $C$.

```c
int arb_calc_newton_step(arb_t xnew, arb_calc_func_t func, void *param, const arb_t x, const arb_t conv_region, const arb_t conv_factor, slong prec)
```

Performs a single step with an interval version of Newton’s method. The input consists of the function $f$ specified by $func$ and $param$, a ball $x = [m - r, m + r]$ known to contain a single root of $f$, a ball $I$ ($conv_region$) containing $x$ with an associated bound ($conv_factor$) for $C = \sup_{t,u \in I} \frac{1}{2} |f''(t)||f'(u)|$, and a working precision $prec$.

The Newton update consists of setting $x' = [m' - r', m' + r']$ where $m' = m - f(m)/f'(m)$ and $r' = Cr^2$. The expression $m - f(m)/f'(m)$ is evaluated using ball arithmetic at a working precision of $prec$ bits, and the rounding error during this evaluation is accounted for in the output. We now check that $x' \in I$ and $r' < r$. If both conditions are satisfied, we set $xnew$ to $x'$ and return $ARB_CALC_SUCCESS$. If either condition fails, we set $xnew$ to $x$ and return $ARB_CALC_NO_CONVERGENCE$, indicating that no progress is made.

```c
int arb_calc_refine_root_newton(arb_t r, arb_calc_func_t func, void *param, const arb_t start, const arb_t conv_region, const arb_t conv_factor, slong eval_extra_prec, slong prec)
```

Refines a precise estimate of a single root of a function to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for $arb_calc_newton_step$, except for the precision parameters: $prec$ is the target accuracy and $eval_extra_prec$ is the estimated number of guard bits that need to be added to evaluate the function accurately close to the root (for example, if the function is a
polynomial with large coefficients of alternating signs and Horner’s rule is used to evaluate it, the extra precision should typically be approximately the bit size of the coefficients).

This function returns ARB_CALC_SUCCESS if all attempted Newton steps are successful (note that this does not guarantee that the computed root is accurate to prec bits, which has to be verified by the user), only that it is more accurate than the starting ball.

On failure, ARB_CALC_IMPRECISE_INPUT or ARB_CALC_NO_CONVERGENCE may be returned. In this case, r is set to a ball for the root which is valid but likely does have full accuracy (it can possibly just be equal to the starting ball).

9.26 acb_calc.h – calculus with complex-valued functions

This module provides functions for operations of calculus over the complex numbers (intended to include root-finding, integration, and so on). The numerical integration code is described in [Joh2018a].

9.26.1 Types, macros and constants

type acb_calc_func_t

Typedef for a pointer to a function with signature:

```c
int func(acb_ptr out, const acb_t inp, void * param, slong order, slong prec)
```

implementing a univariate complex function \( f(z) \). The `param` argument may be used to pass through additional parameters to the function. The return value is reserved for future use as an error code. It can be assumed that `out` and `inp` are not aliased.

When called with `order = 0`, `func` should write to `out` the value of \( f(z) \) at the point `inp`, evaluated at a precision of `prec` bits. In this case, \( f \) can be an arbitrary complex function, which may have branch cuts or even be non-holomorphic.

When called with `order = n` for \( n \geq 1 \), `func` should write to `out` the first \( n \) coefficients in the Taylor series expansion of \( f(z) \) at the point `inp`, evaluated at a precision of `prec` bits. In this case, the implementation of `func` must verify that \( f \) is holomorphic on the complex interval defined by \( z \), and set the coefficients in `out` to non-finite values otherwise.

For algorithms that do not rely on derivatives, `func` will always get called with `order = 0` or `order = 1`, in which case the user only needs to implement evaluation of the direct function value \( f(z) \) (without derivatives). With `order = 1`, `func` must verify holomorphicity (unlike the `order = 0` case).

If \( f \) is built from field operations and meromorphic functions, then no special action is necessary when `order` is positive since division by zero or evaluation of builtin functions at poles automatically produces infinite enclosures. However, manual action is needed for bounded functions with branch cuts. For example, when evaluating \( \sqrt{z} \), the output must be set to an non-finite value if \( z \) overlaps with the branch cut \( [-\infty,0] \). The easiest way to accomplish this is to use versions of basic functions (sqrt, log, pow, etc.) that test holomorphicity of their arguments individually.

Some functions with branch cut detection are available as builtins: see `acb_sqrt_analytic()`, `acb_rsqrt_analytic()`, `acb_log_analytic()`, `acb_pow_analytic()`. It is not difficult to write custom functions of this type, using the following pattern:

```c
/* Square root function on C with detection of the branch cut. */
void sqrt_analytic(acb_t res, const acb_t z, int analytic, slong prec)
{
    if (analytic &&
        arb_contains_zero(acb_imagref(z)) &&
        arb_contains_nonpositive(acb_realref(z))))
```
The built-in methods `acb_real_abs()`, `acb_real_sgn()`, `acb_real_heaviside()`, `acb_real_floor()`, `acb_real ceil()`, `acb_real_max()`, `acb_real_min()` provide piecewise holomorphic functions that are useful for integrating piecewise-defined real functions.

For example, here we define a piecewise holomorphic extension of the function $f(z) = \sqrt{\lfloor z \rfloor}$ (for simplicity, without implementing derivatives):

```c
int func(acb_ptr out, const acb_t inp, void * param, slong order, slong prec)
{
    if (order > 1) flint_abort(); /* derivatives not implemented */
    acb_real_floor(out, inp, order != 0, prec);
    acb_sqrt_analytic(out, out, order != 0, prec);
    return 0;
}
```

(Here, `acb_real_sqrtpos()` may be slightly better if it is known that $z$ will be nonnegative on the path.)

See the demo program `examples/integrals.c` for more examples.

### 9.26.2 Integration

```c
int acb_calc_integrate(acb_t res, acb_calc_func_t func, void *param, const acb_t a, const acb_t b, slong rel_goal, const mag_t abs_tol, const acb_calc_integrate_opt_t options, slong prec)
```

Computes a rigorous enclosure of the integral

$$I = \int_{a}^{b} f(t)dt$$

where $f$ is specified by (`func`, `param`), following a straight-line path between the complex numbers $a$ and $b$. For finite results, $a$, $b$ must be finite and $f$ must be bounded on the path of integration. To compute improper integrals, the user should therefore truncate the path of integration manually (or make a regularizing change of variables, if possible). Returns `ARB_CALC_SUCCESS` if the integration converged to the target accuracy on all subintervals, and returns `ARB_CALC_NO_CONVERGENCE` otherwise.

By default, the integrand `func` will only be called with `order = 0` or `order = 1`; that is, derivatives are not required.

- The integrand will be called with `order = 0` to evaluate $f$ normally on the integration path (either at a single point or on a subinterval). In this case, $f$ is treated as a pointwise defined function and can have arbitrary discontinuities.
- The integrand will be called with `order = 1` to evaluate $f$ on a domain surrounding a segment of the integration path for the purpose of bounding the error of a quadrature formula. In this case, `func` must verify that $f$ is holomorphic on this domain (and output a non-finite value if it is not).
The integration algorithm combines direct interval enclosures, Gauss-Legendre quadrature where \( f \) is holomorphic, and adaptive subdivision. This strategy supports integrands with discontinuities while providing exponential convergence for typical piecewise holomorphic integrands.

The following parameters control accuracy:

- `rel_goal` - relative accuracy goal as a number of bits, i.e. target a relative error less than \( \varepsilon_{rel} = 2^{-r} \) where \( r = rel\_goal \) (note the sign: `rel_goal` should be nonnegative).
- `abs_tol` - absolute accuracy goal as a `mag_t` describing the error tolerance, i.e. target an absolute error less than \( \varepsilon_{abs} = abs\_tol \).
- `prec` - working precision. This is the working precision used to evaluate the integrand and manipulate interval endpoints. As currently implemented, the algorithm does not attempt to adjust the working precision by itself, and adaptive control of the working precision must be handled by the user.

For typical usage, set `rel_goal = prec` and `abs_tol = 2^{-prec}`. It usually only makes sense to have `rel_goal` between 0 and `prec`. The algorithm attempts to achieve an error of \( \max(\varepsilon_{abs}, M\varepsilon_{rel}) \) on each subinterval, where \( M \) is the magnitude of the integral. These parameters are only guidelines; the cumulative error may be larger than both the prescribed absolute and relative error goals, depending on the number of subdivisions, cancellation between segments of the integral, and numerical errors in the evaluation of the integrand.

To compute tiny integrals with high relative accuracy, one should set \( \varepsilon_{abs} \approx M\varepsilon_{rel} \) where \( M \) is a known estimate of the magnitude. Setting \( \varepsilon_{abs} \) to 0 is also allowed, forcing use of a relative instead of an absolute tolerance goal. This can be handy for exponentially small or large functions of unknown magnitude. It is recommended to avoid setting \( \varepsilon_{abs} \) very small if possible since the algorithm might need many extra subdivisions to estimate \( M \) automatically; if the approximate magnitude can be estimated by some external means (for example if a midpoint-width or endpoint-width estimate is known to be accurate), providing an appropriate \( \varepsilon_{abs} \approx M\varepsilon_{rel} \) will be more efficient.

If the integral has very large magnitude, setting the absolute tolerance to a corresponding large value is recommended for best performance, but it is not necessary for convergence since the absolute tolerance is increased automatically during the execution of the algorithm if the partial integrals are found to have larger error.

Additional options for the integration can be provided via the `options` parameter (documented below). To use all defaults, `NULL` can be passed for `options`.

### Options for integration

```c
#define acb_calc_integrate_opt_struct

#define acb_calc_integrate_opt_t

This structure contains several fields, explained below. An `acb_calc_integrate_opt_t` is defined as an array of `acb_calc_integrate_opt_struct` of length 1, permitting it to be passed by reference. An `acb_calc_integrate_opt_t` must be initialized before use, which sets all fields to 0 or `NULL`. For fields that have not been set to other values, the integration algorithm will choose defaults automatically (based on the precision and accuracy goals). This structure will most likely be extended in the future to accommodate more options.

```c
typedef slong deg_limit

Maximum quadrature degree for each subinterval. If a zero or negative value is provided, the limit is set to a default value which currently equals \( 0.5 \cdot \min(prec, rel\_goal) + 60 \) for Gauss-Legendre quadrature. A higher quadrature degree can be beneficial for functions that are holomorphic on a large domain around the integration path and yet behave irregularly, such as oscillatory entire functions. The drawback of increasing the degree is that the precomputation time for quadrature nodes increases.

9.26. acb_calc.h — calculus with complex-valued functions
**slong eval_limit**

Maximum number of function evaluations. If a zero or negative value is provided, the limit is set to a default value which currently equals $1000 \cdot prec + prec^2$. This is the main parameter used to limit the amount of work before aborting due to possible slow convergence or non-convergence. A lower limit allows aborting faster. A higher limit may be needed for integrands with many discontinuities or many singularities close to the integration path. This limit is only taken as a rough guideline, and the actual number of function evaluations may be slightly higher depending on the actual subdivisions.

**slong depth_limit**

Maximum search depth for adaptive subdivision. Technically, this is not the limit on the local bisection depth but the limit on the number of simultaneously queued subintervals. If a zero or negative value is provided, the limit is set to the default value $2 \cdot prec$. Warning: memory usage may increase in proportion to this limit.

**int use_heap**

By default (if set to 0), new subintervals generated by adaptive bisection will be appended to the top of a stack. If set to 1, a binary heap will be used to maintain a priority queue where the subintervals with larger error have higher priority. This sometimes gives better results in case of convergence failure, but can lead to a much larger array of subintervals (requiring a higher depth_limit) when many global bisections are needed.

**int verbose**

If set to 1, some information about the overall integration process is printed to standard output. If set to 2, information about each subinterval is printed.

```c
void acb_calc_integrate_opt_init(acb_calc_integrate_opt_t options)
```

Initializes options for use, setting all fields to 0 indicating default values.

### 9.26.3 Local integration algorithms

```c
int acb_calc_integrate_gl_auto_deg(acb_t res, slong *num_eval, acb_calc_func_t func, void *param, const acb_t a, const acb_t b, const mag_t tol, slong deg_limit, int flags, slong prec)
```

Attempts to compute $I = \int_a^b f(t)dt$ using a single application of Gauss-Legendre quadrature with automatic determination of the quadrature degree so that the error is smaller than $tol$. Returns `ARB_CALC_SUCCESS` if the integral has been evaluated successfully or `ARB_CALC_NO_CONVERGENCE` if the tolerance could not be met. The total number of function evaluations is written to `num_eval`.

For the interval $[-1, 1]$, the error of the $n$-point Gauss-Legendre rule is bounded by

$$|I - \sum_{k=0}^{n-1} w_k f(x_k)| \leq \frac{64M}{15(\rho - 1)\rho^{2n-1}}$$

if $f$ is holomorphic with $|f(z)| \leq M$ inside the ellipse $E$ with foci $\pm 1$ and semiaxes $X$ and $Y = \sqrt{X^2 - 1}$ such that $\rho = X + Y$ with $\rho > 1$ [Tre2008].

For an arbitrary interval, we use $I = \int_a^b f(t)dt = \int_1^1 g(t)dt$ where $g(t) = \Delta f(\Delta t + m)$, $\Delta = \frac{1}{2}(b - a)$, $m = \frac{1}{2}(a + b)$. With $I = [\pm X] + [\pm Y]i$, this means that we evaluate $\Delta f(\Delta I + m)$ to get the bound $M$. (An improvement would be to reduce the wrapping effect of rotating the ellipse when the path is not rectilinear).

We search for an $X$ that makes the error small by trying steps $2^{2k}$. Larger $X$ will give smaller $1/\rho^{2n-1}$ but larger $M$. If we try successive larger values of $k$, we can abort when $M = \infty$ since this either means that we have hit a singularity or a branch cut or that overestimation in the evaluation of $f$ is becoming too severe.
9.26.4 Integration (old)

void \texttt{acb\_calc\_cauchy\_bound}(\texttt{arb\_t bound, acb\_calc\_func\_t func, void \*param, const acb\_t x, const arb\_t radius, slong maxdepth, slong prec})

Sets \texttt{bound} to a ball containing the value of the integral

\[
C(x, r) = \frac{1}{2\pi r} \int_{|z-x|=r} |f(z)|dz = \int_0^1 |f(x + re^{2\pi it})|dt
\]

where \( f \) is specified by (\texttt{func, param}) and \( r \) is given by \texttt{radius}. The integral is computed using a simple step sum. The integration range is subdivided until the order of magnitude of \( b \) can be determined (i.e. its error bound is smaller than its midpoint), or until the step length has been cut in half \texttt{maxdepth} times. This function is currently implemented completely naively, and repeatedly subdivides the whole integration range instead of performing adaptive subdivisions.

int \texttt{acb\_calc\_integrate\_taylor}(\texttt{acb\_t res, acb\_calc\_func\_t func, void \*param, const acb\_t a, const acb\_t b, const arf\_t inner\_radius, const arf\_t outer\_radius, slong accuracy\_goal, slong prec})

Computes the integral

\[
I = \int_a^b f(t)dt
\]

where \( f \) is specified by (\texttt{func, param}), following a straight-line path between the complex numbers \( a \) and \( b \) which both must be finite.

The integral is approximated by piecewise centered Taylor polynomials. Rigorous truncation error bounds are calculated using the Cauchy integral formula. More precisely, if the Taylor series of \( f \) centered at the point \( m \) is \( f(m + x) = \sum_{n=0}^{\infty} a_n x^n \), then

\[
\int f(m + x) = \left( \sum_{n=0}^{N-1} a_n \frac{x^{n+1}}{n+1} \right) + \left( \sum_{n=N}^{\infty} a_n \frac{x^{n+1}}{n+1} \right).
\]

For sufficiently small \( x \), the second series converges and its absolute value is bounded by

\[
\sum_{n=N}^{\infty} C(m, R) \frac{|x|^{n+1}}{R^n} \frac{|x|^{n+1}}{N+1} = C(m, R) \frac{Rx}{(R-x)(N+1)} \left( \frac{R}{R} \right)^N.
\]

It is required that any singularities of \( f \) are isolated from the path of integration by a distance strictly greater than the positive value \texttt{outer\_radius} (which is the integration radius used for the Cauchy bound). Taylor series step lengths are chosen so as not to exceed \texttt{inner\_radius}, which must be strictly smaller than \texttt{outer\_radius} for convergence. A smaller \texttt{inner\_radius} gives more rapid convergence of each Taylor series but means that more series might have to be used. A reasonable choice might be to set \texttt{inner\_radius} to half the value of \texttt{outer\_radius}, giving roughly one accurate bit per term.

The truncation point of each Taylor series is chosen so that the absolute truncation error is roughly \( 2^{-p} \) where \( p \) is given by \texttt{accuracy\_goal} (in the future, this might change to a relative accuracy). Arithmetic operations and function evaluations are performed at a precision of \texttt{prec} bits. Note that due to accumulation of numerical errors, both values may have to be set higher (and the endpoints may have to be computed more accurately) to achieve a desired accuracy.

This function chooses the evaluation points uniformly rather than implementing adaptive subdivision.
9.27 arb_fpwrap.h – floating-point wrappers of Arb mathematical functions

This module provides wrappers of Arb functions intended users who want accurate floating-point mathematical functions without necessarily caring about ball arithmetic. The wrappers take floating-point input, give floating-point output, and automatically increase the internal working precision to ensure that the output is accurate (in the rare case of failure, they output NaN along with an error code).

**Warning:** This module is experimental (as of Arb 2.21). It has not been extensively tested, and interfaces may change in the future.

**Supported types:**
- double and complex_double (53-bit precision)

**Limitations:**
- The wrappers currently only handle finite input and points where function value is finite. For example, they do not know that \( \log(0) = -\infty \) or that \( \exp(-\infty) = 0 \). Singular input or output result in FPWRAP_UNABLE and a NaN output value. Evaluation of limit values may be implemented in the future for some functions.
- The wrappers currently treat \(-0.0\) as \(+0.0\). Users who need to distinguish signs of zero, e.g. on branch cuts, currently need to do so manually.
- When requesting *correct rounding*, the wrappers can fail to converge in asymptotic or exact cases (where special algorithms are required).
- If the value is computed accurately internally but is too small to represent as a floating-point number, the result will be \(-0.0\) or \(+0.0\) (on underflow) or \(-\text{Inf}\) or \(+\text{Inf}\) (on overflow). Since the underflowed or overflowed result is the best possible floating-point approximation of the true value, this outcome is considered correct and the flag FPWRAP_SUCCESS is returned. In the future, return status flags may be added to indicate that underflow or overflow has occurred.
- Different rounding modes are not yet implemented.

### 9.27.1 Option and return flags

Functions return an int flag indicating the status.

**FPWRAP_SUCCESS**
- Indicates an accurate result. (Up to inevitable underflow or overflow in the final conversion to a floating-point result; see above.)
  - This flag has the numerical value 0.

**FPWRAP_UNABLE**
- Indicates failure (unable to achieve to target accuracy, possibly because of a singularity). The output is set to NaN.
  - This flag has the numerical value 1.

Functions take a flags parameter specifying optional rounding and termination behavior. This can be set to 0 to use defaults.

**FPWRAP_ACCURATE_PARTS**
- For complex output, compute both real and imaginary parts to full relative accuracy. By default (if this flag is not set), complex results are computed to at least 53-bit accuracy as a whole, but if either the real or imaginary part is much smaller than the other, that part can have a large relative error. Setting this flag can result in slower evaluation or failure to converge in some cases.
  - This flag has the numerical value 1.
FPWRAP_CORRECT_ROUNDING

Guarantees *correct rounding*. By default (if this flag is not set), real results are accurate up to the rounding of the last bit, but the last bit is not guaranteed to be rounded optimally. Setting this flag can result in slower evaluation or failure to converge in some cases. Correct rounding automatically applies to both real and imaginary parts of complex numbers, so it is unnecessary to set both this flag and *FPWRAP_ACCURATE_PARTS*.

This flag has the numerical value 2.

FPWRAP_WORK_LIMIT

Multiplied by an integer, specifies the maximum working precision to use before giving up. With \( n \times \text{FPWRAP_WORK_LIMIT} \) added to flags, \( n \) levels of precision will be used. The default \( n = 0 \) is equivalent to \( n = 8 \), which for double means trying with a working precision of 64, 128, 256, 512, 1024, 2048, 4096, 8192 bits. With flags = 2 * FPWRAP_WORK_LIMIT, we only try 64 and 128 bits, and with flags = 16 * FPWRAP_WORK_LIMIT we go up to 2097152 bits.

This flag has the numerical value 65536.

### 9.27.2 Types

Outputs are passed by reference so that we can return status flags and so that the interface is uniform for functions with multiple outputs.

**type complex_double**

A struct of two double components (*real* and *imag*), used to represent a machine-precision complex number. We use this custom type instead of the complex types defined in <complex.h> since Arb does not depend on C99. Users should easily be able to convert to the C99 complex type since the layout in memory is identical.

### 9.27.3 Functions

**Elementary functions**

```c
int arb_fpwrap_double_exp(double *res, double x, int flags)
int arb_fpwrap_cdouble_exp(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_expm1(double *res, double x, int flags)
int arb_fpwrap_cdouble_expm1(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_log(double *res, double x, int flags)
int arb_fpwrap_cdouble_log(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_log1p(double *res, double x, int flags)
int arb_fpwrap_cdouble_log1p(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_pow(double *res, double x, double y, int flags)
int arb_fpwrap_cdouble_pow(complex_double *res, complex_double x, complex_double y, int flags)
int arb_fpwrap_double_sqrt(double *res, double x, int flags)
int arb_fpwrap_cdouble_sqrt(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_rsqrt(double *res, double x, int flags)
int arb_fpwrap_cdouble_rsqrt(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_cbrt(double *res, double x, int flags)
int arb_fpwrap_cdouble_cbrt(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_sin(double *res, double x, int flags)
```

9.27. *arb_fpwrap.h* – floating-point wrappers of Arb mathematical functions 729
int arb_fpwrap_cdouble_sin(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_cos(double *res, double x, int flags)
int arb_fpwrap_cdouble_cos(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_tan(double *res, double x, int flags)
int arb_fpwrap_cdouble_tan(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_cot(double *res, double x, int flags)
int arb_fpwrap_cdouble_cot(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_sec(double *res, double x, int flags)
int arb_fpwrap_cdouble_sec(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_csc(double *res, double x, int flags)
int arb_fpwrap_cdouble_csc(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_sinc(double *res, double x, int flags)
int arb_fpwrap_cdouble_sinc(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_sin_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_sin_pi(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_cos_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_cos_pi(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_tan_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_tan_pi(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_cot_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_cot_pi(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_sinc_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_sinc_pi(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_asin(double *res, double x, int flags)
int arb_fpwrap_cdouble_asin(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_acos(double *res, double x, int flags)
int arb_fpwrap_cdouble_acos(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_atan(double *res, double x, int flags)
int arb_fpwrap_cdouble_atan(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_atan2(double *res, double x1, double x2, int flags)
int arb_fpwrap_double_asinh(double *res, double x, int flags)
int arb_fpwrap_cdouble_asinh(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_acosh(double *res, double x, int flags)
int arb_fpwrap_cdouble_acosh(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_atanh(double *res, double x, int flags)
int arb_fpwrap_cdouble_atanh(\texttt{complex\_double} *res, \texttt{complex\_double} x, int flags)
int arb_fpwrap_double_lambertw(double *res, double x, slong branch, int flags)
int arb_fpwrap_cdouble_lambertw(\texttt{complex\_double} *res, \texttt{complex\_double} x, slong branch, int flags)
Gamma, zeta and related functions

```c
int arb_fpwrap_double_rising(double *res, double x, double n, int flags)
int arb_fpwrap_cdouble_rising(complex_double *res, complex_double x, complex_double n, int flags)
```

Rising factorial.

```c
int arb_fpwrap_double_gamma(double *res, double x, int flags)
int arb_fpwrap_cdouble_gamma(complex_double *res, complex_double x, int flags)
```
Gamma function.

```c
int arb_fpwrap_double_rgamma(double *res, double x, int flags)
int arb_fpwrap_cdouble_rgamma(complex_double *res, complex_double x, int flags)
```
Reciprocal gamma function.

```c
int arb_fpwrap_double_lgamma(double *res, double x, int flags)
int arb_fpwrap_cdouble_lgamma(complex_double *res, complex_double x, int flags)
```
Log-gamma function.

```c
int arb_fpwrap_double_digamma(double *res, double x, int flags)
int arb_fpwrap_cdouble_digamma(complex_double *res, complex_double x, int flags)
```
Digamma function.

```c
int arb_fpwrap_double_zeta(double *res, double x, int flags)
int arb_fpwrap_cdouble_zeta(complex_double *res, complex_double x, int flags)
```
Riemann zeta function.

```c
int arb_fpwrap_double_hurwitz_zeta(double *res, double s, double z, int flags)
int arb_fpwrap_cdouble_hurwitz_zeta(complex_double *res, complex_double s, complex_double z, int flags)
```
Hurwitz zeta function.

```c
int arb_fpwrap_double_lerch_phi(double *res, double z, double s, double a, int flags)
int arb_fpwrap_cdouble_lerch_phi(complex_double *res, complex_double z, complex_double s, complex_double a, int flags)
```
Lerch transcendent.

```c
int arb_fpwrap_double_barnes_g(double *res, double x, int flags)
int arb_fpwrap_cdouble_barnes_g(complex_double *res, complex_double x, int flags)
```
Barnes G-function.

```c
int arb_fpwrap_double_log_barnes_g(double *res, double x, int flags)
int arb_fpwrap_cdouble_log_barnes_g(complex_double *res, complex_double x, int flags)
```
Logarithmic Barnes G-function.

```c
int arb_fpwrap_double_polygamma(double *res, double s, double z, int flags)
int arb_fpwrap_cdouble_polygamma(complex_double *res, complex_double s, complex_double z, int flags)
```
Polygamma function.

```c
int arb_fpwrap_double_polylog(double *res, double s, double z, int flags)
int arb_fpwrap_cdouble_polylog(complex_double *res, complex_double s, complex_double z, int flags)
```
Polylogarithm.

```c
int arb_fpwrap_cdouble_dirichlet_eta(complex_double *res, complex_double s, int flags)
```
int arb_fpwrap_cdouble_riemann_xi(complex_double *res, complex_double s, int flags)
int arb_fpwrap_cdouble_hardy_theta(complex_double *res, complex_double z, int flags)
int arb_fpwrap_cdouble_hardy_z(complex_double *res, complex_double z, int flags)
int arb.fpwrap_cdouble_zeta_zero(complex_double *res, ulong n, int flags)

Error functions and exponential integrals

int arb_fpwrap_double_erf(double *res, double x, int flags)
int arb_fpwrap_cdouble_erf(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_erfc(double *res, double x, int flags)
int arb_fpwrap_cdouble_erfc(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_erfi(double *res, double x, int flags)
int arb_fpwrap_cdouble_erfi(complex_double *res, complex_double x, int flags)
int arb.fpwrap_double_erfinv(double *res, double x, int flags)
int arb_fpwrap_double_erfcinv(double *res, double x, int flags)
int arb.fpwrap_double_fresnel_s(double *res, double x, int normalized, int flags)
int arb_fpwrap_cdouble_fresnel_s(complex_double *res, complex_double x, int normalized, int flags)
int arb.fpwrap_double_fresnel_c(double *res, double x, int normalized, int flags)
int arb_fpwrap_cdouble_fresnel_c(complex_double *res, complex_double x, int normalized, int flags)
int arb.fpwrap_double_gamma_upper(double *res, double s, double z, int regularized, int flags)
int arb.fpwrap_cdouble_gamma_upper(complex_double *res, complex_double s, complex_double z, int regularized, int flags)
int arb.fpwrap_double_gamma_lower(double *res, double s, double z, int regularized, int flags)
int arb.fpwrap_cdouble_gamma_lower(complex_double *res, complex_double s, complex_double z, int regularized, int flags)
int arb.fpwrap_double_beta_lower(double *res, double a, double b, double z, int regularized, int flags)
int arb.fpwrap_cdouble_beta_lower(complex_double *res, complex_double a, complex_double b, complex_double z, int regularized, int flags)
int arb.fpwrap_double_exp_integral_e(double *res, double s, double z, int flags)
int arb.fpwrap_cdouble_exp_integral_e(complex_double *res, complex_double s, complex_double z, int flags)
int arb.fpwrap_double_exp_integral_ei(double *res, double x, int flags)
int arb.fpwrap_cdouble_exp_integral_ei(complex_double *res, complex_double x, int flags)
int arb.fpwrap_double_sin_integral(double *res, complex_double x, int flags)
int arb.fpwrap_cdouble_sin_integral(complex_double *res, complex_double x, int flags)
int arb.fpwrap_double_cos_integral(double *res, double x, int flags)
int arb.fpwrap_cdouble_cos_integral(complex_double *res, complex_double x, int flags)
int arb.fpwrap_double_sinh_integral(double *res, double x, int flags)
int arb_fpwrap_cdouble_sinh_integral(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_cosh_integral(double *res, double x, int flags)
int arb_fpwrap_cdouble_cosh_integral(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_log_integral(double *res, double x, int offset, int flags)
int arb_fpwrap_cdouble_log_integral(complex_double *res, complex_double x, int offset, int flags)

int arb_fpwrap_double_dilog(double *res, double x, int flags)
int arb_fpwrap_cdouble_dilog(complex_double *res, complex_double x, int flags)

Bessel, Airy and Coulomb functions

int arb_fpwrap_double_bessel_j(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_j(complex_double *res, complex_double nu, complex_double x, int flags)

int arb_fpwrap_double_bessel_y(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_y(complex_double *res, complex_double nu, complex_double x, int flags)

int arb_fpwrap_double_bessel_i(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_i(complex_double *res, complex_double nu, complex_double x, int flags)

int arb_fpwrap_double_bessel_k(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_k(complex_double *res, complex_double nu, complex_double x, int flags)

int arb_fpwrap_double_bessel_k_scaled(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_k_scaled(complex_double *res, complex_double nu, complex_double x, int flags)

int arb_fpwrap_double_airy_ai(double *res, double x, int flags)
int arb_fpwrap_cdouble_airy_ai(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_airy_ai_prime(double *res, double x, int flags)
int arb_fpwrap_cdouble_airy_ai_prime(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_airy_bi(double *res, double x, int flags)
int arb_fpwrap_cdouble_airy_bi(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_airy_bi_prime(double *res, double x, int flags)
int arb_fpwrap_cdouble_airy_bi_prime(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_airy_ai_zero(double *res, ulong n, int flags)
int arb_fpwrap_double_airy_ai_prime_zero(double *res, ulong n, int flags)
int arb_fpwrap_double_airy_bi_zero(double *res, ulong n, int flags)
int arb_fpwrap_double_airy_bi_prime_zero(double *res, ulong n, int flags)

int arb_fpwrap_double_coulomb_f(double *res, double l, double eta, double x, int flags)
int arb_fpwrap_cdouble_coulomb_f(complex_double *res, complex_double l, complex_double eta, complex_double x, int flags)

int arb_fpwrap_double_coulomb_g(double *res, double l, double eta, double x, int flags)
int arb_fpwrap_cdouble_coulomb_g(complex_double *res, complex_double l, complex_double eta, complex_double x, int flags)

int arb_fpwrap_cdouble_coulomb_hpos(complex_double *res, complex_double l, complex_double eta, complex_double x, int flags)

int arb_fpwrap_cdouble_coulomb_hneg(complex_double *res, complex_double l, complex_double eta, complex_double x, int flags)

Orthogonal polynomials

int arb_fpwrap_double_chebyshev_t(double *res, double n, double x, int flags)
int arb_fpwrap_cdouble_chebyshev_t(complex_double *res, complex_double n, complex_double x, int flags)

int arb_fpwrap_double_chebyshev_u(double *res, double n, double x, int flags)
int arb_fpwrap_cdouble_chebyshev_u(complex_double *res, complex_double n, complex_double x, int flags)

int arb_fpwrap_double_jacobi_p(double *res, double n, double a, double b, double x, int flags)
int arb_fpwrap_cdouble_jacobi_p(complex_double *res, complex_double n, complex_double a, complex_double b, complex_double x, int flags)

int arb_fpwrap_double_gegenbauer_c(double *res, double n, double m, double x, int flags)
int arb_fpwrap_cdouble_gegenbauer_c(complex_double *res, complex_double n, complex_double m, complex_double x, int flags)

int arb_fpwrap_double_laguerre_l(double *res, double n, double m, double x, int flags)
int arb_fpwrap_cdouble_laguerre_l(complex_double *res, complex_double n, complex_double m, complex_double x, int flags)

int arb_fpwrap_double_hermite_h(double *res, double n, double x, int flags)
int arb_fpwrap_cdouble_hermite_h(complex_double *res, complex_double n, complex_double x, int flags)

int arb_fpwrap_double_legendre_p(double *res, double n, double m, double x, int type, int flags)
int arb_fpwrap_cdouble_legendre_p(complex_double *res, complex_double n, complex_double m, complex_double x, int type, int flags)

int arb_fpwrap_double_legendre_q(double *res, double n, double m, double x, int type, int flags)
int arb_fpwrap_cdouble_legendre_q(complex_double *res, complex_double n, complex_double m, complex_double x, int type, int flags)

int arb_fpwrap_double_legendre_root(double *res1, double *res2, ulong n, ulong k, int flags)
    Sets res1 to the index k root of the Legendre polynomial \( P_n(x) \), and simultaneously sets res2 to the corresponding weight for Gauss-Legendre quadrature.

int arb_fpwrap_cdouble_spherical_y(complex_double *res, slong n, slong m, complex_double x1, complex_double x2, int flags)
Hypergeometric functions

```c
int arb_fpwrap_double_hypgeom_0f1(double *res, double a, double x, int regularized, int flags);
int arb_fpwrap_cdouble_hypgeom_0f1(complex_double *res, complex_double a, complex_double x, int regularized, int flags);
int arb_fpwrap_double_hypgeom_1f1(double *res, double a, double b, double x, int regularized, int flags);
int arb_fpwrap_cdouble_hypgeom_1f1(complex_double *res, complex_double a, complex_double b, complex_double x, int regularized, int flags);
int arb_fpwrap_double_hypgeom_u(double *res, double a, double b, double x, int flags);
int arb_fpwrap_cdouble_hypgeom_u(complex_double *res, complex_double a, complex_double b, complex_double x, int flags);
int arb_fpwrap_double_hypgeom_2f1(double *res, double a, double b, double c, double x, int regularized, int flags);
int arb_fpwrap_cdouble_hypgeom_2f1(complex_double *res, complex_double a, complex_double b, complex_double c, complex_double x, int regularized, int flags);
int arb_fpwrap_double_hypgeom_pfq(double *res, const double *a, slong p, const double *b, slong q, double z, int regularized, int flags);
int arb_fpwrap_cdouble_hypgeom_pfq(complex_double *res, const complex_double *a, slong p, const complex_double *b, slong q, complex_double z, int regularized, int flags);
```

Elliptic integrals, elliptic functions and modular forms

```c
int arb_fpwrap_double_agm(double *res, double x, double y, int flags);
int arb_fpwrap_cdouble_agm(complex_double *res, complex_double x, complex_double y, int flags);
int arb_fpwrap_cdouble_elliptic_k(complex_double *res, complex_double m, int flags);
int arb_fpwrap_cdouble_elliptic_e(complex_double *res, complex_double m, int flags);
int arb_fpwrap_cdouble_elliptic_pi(complex_double *res, complex_double n, complex_double m, int flags);
int arb_fpwrap_cdouble_elliptic_f(complex_double *res, complex_double phi, complex_double m, int pi, int flags);
int arb_fpwrap_cdouble_elliptic_e_inc(complex_double *res, complex_double phi, complex_double m, int pi, int flags);
int arb_fpwrap_cdouble_elliptic_pi_inc(complex_double *res, complex_double n, complex_double m, int pi, int flags);
int arb_fpwrap_cdouble_elliptic_rf(complex_double *res, complex_double x, complex_double y, complex_double z, int option, int flags);
int arb_fpwrap_cdouble_elliptic_rg(complex_double *res, complex_double x, complex_double y, complex_double z, int option, int flags);
int arb_fpwrap_cdouble_elliptic_rj(complex_double *res, complex_double x, complex_double y, complex_double z, complex_double w, int option, int flags);
```

Arithmetic-geometric mean.

Complete and incomplete elliptic integrals.

Carlson symmetric elliptic integrals.
Weierstrass elliptic functions.

Jacobi theta functions.

9.27.4 Calling from C

The program examples/fpwrap.c provides a usage example:

```c
#include "arb_fpwrap.h"

int main()
{
    double x, y;
    complex_double cx, cy;
    int flags = 0; /* default options */

    x = 2.0;
    cx.real = 0.5;
    cx.imag = 123.0;

    arb_fpwrap_double_zeta(&y, x, flags);
    arb_fpwrap_cdouble_zeta(&cy, cx, flags);

    printf("zeta(%.16g) = %.16g\n", x, y);
    printf("zeta(%.16g + %.16gi) = %.16g + %.16gi\n", cx.real, cx.imag, cy.real, cy.imag);

    (continues on next page)
```
This should print:

```bash
> build/examples/fpwrap
zeta(2) = 1.644934066848226
zeta(0.5 + 123i) = 0.006252861175594465 + 0.08206030514520983i
```

Note that this program does not check the return flag to perform error handling.

### 9.27.5 Interfacing from Python

This illustrates how to call functions from Python using **ctypes**:

```python
import ctypes
import ctypes.util

libarb_path = ctypes.util.find_library('arb')
libarb = ctypes.CDLL(libarb_path)

class _complex_double(ctypes.Structure):
    _fields_ = [('real', ctypes.c_double), ('imag', ctypes.c_double)]

def wrap_double_fun(fun):
    def f(x):
        y = ctypes.c_double()
        if fun(ctypes.byref(y), ctypes.c_double(x), 0):
            raise ValueError(f"unable to evaluate function accurately at \{x\}")
        return y.value
    return f

def wrap_cdouble_fun(fun):
    def f(x):
        x = complex(x)
        cx = _complex_double()
        cy = _complex_double()
        cx.real = x.real
        cx.imag = x.imag
        if fun(ctypes.byref(cy), cx, 0):
            raise ValueError(f"unable to evaluate function accurately at \{x\}")
        return complex(cy.real, cy.imag)
    return f

zeta = wrap_double_fun(libarb.arb_fpwrap_double_zeta)
czeta = wrap_cdouble_fun(libarb.arb_fpwrap_cdouble_zeta)

print(zeta(2.0))
print(czeta(0.5+1e9j))
print(zeta(1.0))  # pole, where wrapper throws exception
```

This should print:
1.6449340668482264
(-2.761748029838061 - 1.6775122409894598j)
Traceback (most recent call last):
  ...
ValueError: unable to evaluate function accurately at 1.0

### 9.27.6 Interfacing from Julia

This illustrates how to call functions from Julia using `ccall`:

```julia
using Libdl
dlopen("/home/fredrik/src/arb/libarb.so")

function zeta(x::Float64)
    cy = Ref{Float64}()
    if Bool(ccall((:arb_fpwrap_double_zeta, :libarb), Cint, (Ptr{Float64}, Float64, → Cint), cy, x, 0))
        error("unable to evaluate accurately at ", x)
    end
    return cy[]
end

function zeta(x::Complex{Float64})
    cy = Ref{Complex{Float64}()};
    if Bool(ccall((:arb_fpwrap_cdouble_zeta, :libarb), Cint, (Ptr{Complex{Float64}}, Complex{Float64}, Cint), cy, x, 0))
        error("unable to evaluate accurately at ", x)
    end
    return cy[]
end

println(zeta(2.0))
println(zeta(0.5 + 1e9im))
println(zeta(1.0))  # pole, where wrapper throws exception
```

This should print:

1.6449340668482264
-2.761748029838061 - 1.6775122409894598im
ERROR: unable to evaluate accurately at 1.0
Stacktrace:
  ...

### 9.28 fmpz_extras.h – extra methods for FLINT integers

This module implements a few utility methods for the FLINT multiprecision integer type (`fmpz_t`). It is mainly intended for internal use.
9.28.1 Memory-related methods

\texttt{slong fmpz_allocated_bytes}(const \texttt{fmpz\_t x})

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add \texttt{sizeof(fmpz)} to get the size of the object as a whole.

9.28.2 Convenience methods

void \texttt{fmpz\_adiv\_q\_2exp} (\texttt{fmpz\_t z}, const \texttt{fmpz\_t x}, \texttt{flint\_bitcnt\_t exp})

Sets \( z \) to \( x / 2^e \), rounded away from zero.

void \texttt{fmpz\_ui\_mul\_ui} (\texttt{fmpz\_t x}, \texttt{ulong a}, \texttt{ulong b})

Sets \( x \) to \( a \times b \).

void \texttt{fmpz\_max} (\texttt{fmpz\_t z}, const \texttt{fmpz\_t x}, const \texttt{fmpz\_t y})

void \texttt{fmpz\_min} (\texttt{fmpz\_t z}, const \texttt{fmpz\_t x}, const \texttt{fmpz\_t y})

Sets \( z \) to the maximum (respectively minimum) of \( x \) and \( y \).

9.28.3 Inlined arithmetic

The \texttt{fmpz\_t} bignum type uses an immediate representation for small integers, specifically when the absolute value is at most \( 2^{62} - 1 \) (on 64-bit machines) or \( 2^{30} - 1 \) (on 32-bit machines). The following methods completely inline the case where all operands (and possibly some intermediate values in the calculation) are known to be small. This is faster in code where all values almost certainly will be much smaller than a full word. In particular, these methods are used within Arb for manipulating exponents of floating-point numbers. Inlining slows down the general case, and increases code size, so these methods should not be used gratuitously.

void \texttt{fmpz\_add\_inline} (\texttt{fmpz\_t z}, const \texttt{fmpz\_t x}, const \texttt{fmpz\_t y})

void \texttt{fmpz\_add\_si\_inline} (\texttt{fmpz\_t z}, const \texttt{fmpz\_t x}, \texttt{slong y})

void \texttt{fmpz\_add\_ui\_inline} (\texttt{fmpz\_t z}, const \texttt{fmpz\_t x}, \texttt{ulong y})

Sets \( z \) to the sum of \( x \) and \( y \).

void \texttt{fmpz\_sub\_si\_inline} (\texttt{fmpz\_t z}, const \texttt{fmpz\_t x}, \texttt{slong y})

Sets \( z \) to the difference of \( x \) and \( y \).

void \texttt{fmpz\_add2\_fmpz\_si\_inline} (\texttt{fmpz\_t z}, const \texttt{fmpz\_t x}, const \texttt{fmpz\_t y}, \texttt{slong c})

Sets \( z \) to the sum of \( x \), \( y \), and \( c \).

\texttt{mp\_size\_t fmpz\_size} (const \texttt{fmpz\_t x})

Returns the number of limbs required to represent \( x \).

\texttt{slong fmpz\_sub\_small} (const \texttt{fmpz\_t x}, const \texttt{fmpz\_t y})

Computes the difference of \( x \) and \( y \) and returns the result as an \texttt{slong}. The result is clamped between \(-\text{WORD\_MAX} \) and \text{WORD\_MAX}, i.e. between \( \pm (2^{63} - 1) \) inclusive on a 64-bit machine.

void \texttt{fmpz\_set\_si\_small} (\texttt{fmpz\_t x}, \texttt{slong v})

Sets \( x \) to the integer \( v \) which is required to be a value between \text{COEFF\_MIN} and \text{COEFF\_MAX} so that promotion to a bignum cannot occur.
9.28.4 Low-level conversions

void fmpz_set_mpn_large (fmpz_t z, mp_srcptr src, mp_size_t n, int negative)

Sets z to the integer represented by the n limbs in the array src, or minus this value if negative is 1. Requires $n \geq 2$ and that the top limb of src is nonzero. Note that fmpz_set_ui, fmpz_neg_ui can be used for single-limb integers.

FMPZ_GET_MPN_READONLY (zsign, zn, zptr, zv)

Given an fmpz_t zv, this macro sets zptr to a pointer to the limbs of zv, zn to the number of limbs, and zsign to a sign bit (0 if nonnegative, 1 if negative). The variable ztmp must be a single mp_limb_t, which is used as a buffer. If zv is a small value, zv itself contains no limb array that ztmp could point to, so the single limb is copied to ztmp and zptr is set to point to ztmp. The case where zv is zero is not handled specially, and zn is set to 1.

void fmpz_lshift_mpn (fmpz_t z, mp_srcptr src, mp_size_t n, int negative, flint_bitcnt_t shift)

Sets z to the integer represented by the n limbs in the array src, or minus this value if negative is 1, shifted left by shift bits. Requires $n \geq 1$ and that the top limb of src is nonzero.

9.29 General formulas and bounds

This section collects some results from real and complex analysis that are useful when deriving error bounds. Beware of typos.

9.29.1 Error propagation

We want to bound the error when $f(x + a)$ is approximated by $f(x)$. Specifically, the goal is to bound $f(x + a) - f(x)$ in terms of $r$ for the set of values $a$ with $|a| \leq r$. Most bounds will be monotone increasing with $|a|$ (assuming that $x$ is fixed), so for brevity we simply express the bounds in terms of $|a|$.

Theorem (generic first-order bound):

$$|f(x + a) - f(x)| \leq \min(2C_0, C_1|a|)$$

where

$$C_0 = \sup_{|t| \leq |a|} |f(x + t)|, \quad C_1 = \sup_{|t| \leq |a|} |f'(x + t)|.$$

The statement is valid with either $a, t \in \mathbb{R}$ or $a, t \in \mathbb{C}$.

Theorem (product): For $x, y \in \mathbb{C}$ and $a, b \in \mathbb{C}$,

$$|(x + a)(y + b) - xy| \leq |xb| + |ya| + |ab|.$$

Theorem (quotient): For $x, y \in \mathbb{C}$ and $a, b \in \mathbb{C}$ with $|b| < |y|$,

$$\left|\frac{x}{y} - \frac{x + a}{y + b}\right| \leq \frac{|xb| + |ya|}{|y|(|y| - |b|)}.$$

Theorem (square root): For $x, a \in \mathbb{R}$ with $0 \leq |a| \leq x$,

$$|\sqrt{x + a} - \sqrt{x}| \leq \sqrt{x} \left(1 - \sqrt{1 - \frac{|a|}{x}}\right) \leq \frac{\sqrt{2}}{2} \left(\frac{|a|}{x} + \frac{|a|^2}{x^2}\right)$$

where the first inequality is an equality if $a \leq 0$. (When $x = a = 0$, the limiting value is 0.)

Theorem (reciprocal square root): For $x, a \in \mathbb{R}$ with $0 \leq |a| < x$,

$$\left|\frac{1}{\sqrt{x + a}} - \frac{1}{\sqrt{x}}\right| \leq \frac{|a|}{2(x - |a|)^{3/2}}.$$
Theorem (k-th root): For $k > 1$ and $x, a \in \mathbb{R}$ with $0 \leq |a| \leq x$,\[
| (x + a)^{1/k} - x^{1/k} | \leq x^{1/k} \min \left( 1, \frac{1}{k} \log \left( 1 + \frac{|a|}{x - |a|} \right) \right).
\]

Proof: The error is largest when $a = -r$ is negative, and\[
x^{1/k} - (x - r)^{1/k} = x^{1/k} [1 - (1 - r/x)^{1/k}]
= x^{1/k} [1 - \exp(\log(1 - r/x)/k)] \leq x^{1/k} \min(1, -\log(1 - r/x)/k)
= x^{1/k} \min(1, \log(1 + r/(x - r))/k).
\]

Theorem (sine, cosine): For $x, a \in \mathbb{R}$,\[
| \sin(x + a) - \sin(x) | \leq \min(2, |a|).
\]

Theorem (logarithm): For $x, a \in \mathbb{R}$ with $0 \leq |a| < x$,\[
| \log(x + a) - \log(x) | \leq \log \left( 1 + \frac{|a|}{x - |a|} \right),
\]
with equality if $a \leq 0$.

Theorem (exponential): For $x, a \in \mathbb{R}$,\[
| e^{x+a} - e^x | = e^x (e^a - 1) \leq e^x (e^{|a|} - 1), \text{ with equality if } a \geq 0.
\]

Theorem (inverse tangent): For $x, a \in \mathbb{R}$,\[
| \arctan(x + a) - \arctan(x) | \leq \min(\pi, C_1 |a|).
\]

where\[
C_1 = \sup_{|t| \leq |a|} \frac{1}{1 + (x + t)^2}.
\]
If $|a| < |x|$, then $C_1 = (1 + (|x| - |a|)^2)^{-1}$ gives a monotone bound.

An exact bound: if $|a| < |x|$ or $|x(x + a)| < 1$, then\[
| \arctan(x + a) - \arctan(x) | = \arctan \left( \frac{|a|}{1 + x(x + a)} \right).
\]
In the last formula, a case distinction has to be made depending on the signs of $x$ and $a$.

9.29.2 Sums and series

Theorem (geometric bound): If $|c_k| \leq C$ and $|z| \leq D < 1$, then\[
\left| \sum_{k=N}^{\infty} c_k z^k \right| \leq \frac{CD^N}{1-D}.
\]

Theorem (integral bound): If $f(x)$ is nonnegative and monotone decreasing, then\[
\int_N^\infty f(x) \leq \sum_{k=N}^{\infty} f(k) \leq f(N) + \int_N^\infty f(x) dx.
\]

9.29.3 Complex analytic functions

Theorem (Cauchy’s integral formula): If $f(z) = \sum_{k=0}^{\infty} c_k z^k$ is analytic (on an open subset of $\mathbb{C}$ containing the disk $D = \{ z : |z| \leq R \}$) in its interior, where $R > 0$, then\[
c_k = \frac{1}{2\pi i} \int_{|z|=R} \frac{f(z)}{z^{k+1}} dz.
\]
Corollary (derivative bound):

\[ |c_k| \leq \frac{C}{R^k}, \quad C = \max_{|z|=R} |f(z)|. \]

Corollary (Taylor series tail): If \( 0 \leq r < R \) and \( |z| \leq r \), then

\[ \left| \sum_{k=N}^{\infty} c_k z^k \right| \leq \frac{CD^N}{1-D}, \quad D = \left| \frac{r}{R} \right|. \]

9.29.4 Euler-Maclaurin formula

Theorem (Euler-Maclaurin): If \( f(t) \) is \( 2M \)-times differentiable, then

\[ \sum_{k=L}^{U} f(k) = S + I + T + R \]

\[ S = \sum_{k=L}^{N-1} f(k), \quad I = \int_{N}^{U} f(t) dt, \]

\[ T = \frac{1}{2} (f(N) + f(U)) + \sum_{k=1}^{M} \frac{B_{2k}}{(2k)!} \left( f^{(2k-1)}(U) - f^{(2k-1)}(N) \right), \]

\[ R = - \int_{N}^{U} \frac{B_{2M}(t - \lfloor t \rfloor)}{(2M)!} f^{(2M)}(t) dt. \]

Lemma (Bernoulli polynomials): \( |B_n(t - \lfloor t \rfloor)| \leq 4n!/\pi^n \).

Theorem (remainder bound):

\[ |R| \leq \frac{4}{(2\pi)^{2M}} \int_{N}^{U} \left| f^{(2M)}(t) \right| dt. \]

Theorem (parameter derivatives): If \( f(t) = f(t, x) = \sum_{k=0}^{\infty} a_k(t)x^k \) and \( R(x) = \sum_{k=0}^{\infty} c_k x^k \) are analytic functions of \( x \), then

\[ |c_k| \leq \frac{4}{(2\pi)^{2M}} \int_{N}^{U} \left| a_k^{(2M)}(t) \right| dt. \]

9.30 Algorithms for mathematical constants

Most mathematical constants are evaluated using the generic hypergeometric summation code.

9.30.1 Pi

\( \pi \) is computed using the Chudnovsky series

\[ \frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)! (13591409 + 545140134k)}{(3k)! (k)!^3 640320^{3k+3/2}} \]

which is hypergeometric and adds roughly 14 digits per term. Methods based on the arithmetic-geometric mean seem to be slower by a factor three in practice.

A small trick is to compute \( 1/\sqrt{640320} \) instead of \( \sqrt{640320} \) at the end.
9.30.2 Logarithms of integers

We use the formulas

\[
\log(2) = 3 \sum_{k=0}^{\infty} \frac{(-1)^k (k!)^2}{2^k (2k+1)!}
\]

\[
\log(10) = 46 \text{atanh}(1/31) + 34 \text{atanh}(1/49) + 20 \text{atanh}(1/161)
\]

9.30.3 Euler’s constant

Euler’s constant \( \gamma \) is computed using the Brent-McMillan formula ([BM1980], [MPFR2012])

\[
\gamma = \frac{S_0(2n) - K_0(2n)}{I_0(2n)} - \log(n)
\]

in which \( n \) is a free parameter and

\[
S_0(x) = \sum_{k=0}^{\infty} \frac{H_k}{(k!)^2} \left(\frac{x}{2}\right)^{2k}, \quad I_0(x) = \sum_{k=0}^{\infty} \frac{1}{(k!)^2} \left(\frac{x}{2}\right)^{2k}
\]

\[
2xI_0(x)K_0(x) \sim \sum_{k=0}^{\infty} \frac{[(2k)!]^3}{(k!)^4 8^{2k} x^{2k}}.
\]

All series are evaluated using binary splitting. The first two series are evaluated simultaneously, with the summation taken up to \( k = N - 1 \) inclusive where \( N \geq \alpha n + 1 \) and \( \alpha \approx 4.9706257595442318644 \) satisfies \( \alpha (\log \alpha - 1) = 3 \). The third series is taken up to \( k = 2n - 1 \) inclusive. With these parameters, it is shown in [BJ2013] that the error is bounded by \( 24 e^{-8n} \).

9.30.4 Catalan’s constant

Catalan’s constant is computed using the hypergeometric series

\[
C = \frac{1}{768} \sum_{k=0}^{\infty} \frac{(-4096)^k P(k)}{k^3 (2k-1)(3k-1)(3k-2)(6k-1)(6k-5)(10k)(12k)}
\]

where

\[
P(k) = -43203456k^6 + 92809152k^5 - 76613904k^4 + 536620k^3 - 17325,
\]

discovered by Zuniga [Zun2023]. It was previously computed using a series given in [PP2010].

9.30.5 Apery’s constant

Apery’s constant \( \zeta(3) \) is computed using the hypergeometric series

\[
\zeta(3) = \frac{1}{48} \sum_{k=1}^{\infty} \frac{(-1)^{k-1} P(k)}{k^5 (2k-1)^3 (3k-1)(3k-2)(4k-1)(4k-3)(6k-1)(6k-5)(15k)(30k)(35k)(50k)}
\]

where

\[
P(k) = 1565994397644288k^{11} - 6719460725627136k^{10} + 12632254526031264k^9
\]

\[
-1368432515879536k^8 + 945122351851808k^7 - 4348596587040104k^6
\]

\[
+ 1352700034136826k^5 - 28205786014979k^4 + 38721705264979k^3
\]

\[
-3292502315430k^2 + 156286859400k - 3143448000,
\]

discovered by Zuniga [Zun2023].
9.30.6 Khinchin’s constant

Khinchin’s constant $K_0$ is computed using the formula

$$\log K_0 = \frac{1}{\log 2} \left( \sum_{k=2}^{N-1} \log \left( \frac{k-1}{k} \right) \log \left( \frac{k+1}{k} \right) + \sum_{n=1}^{\infty} \frac{\zeta(2n,N)}{n} \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} \right)$$

where $N \geq 2$ is a free parameter that can be used for tuning [BBC1997]. If the infinite series is truncated after $n = M$, the remainder is smaller in absolute value than

$$\sum_{n=M+1}^{\infty} \zeta(2n,N) = \sum_{n=M+1}^{\infty} \sum_{k=0}^{\infty} (k+N)^{-2n} \leq \sum_{n=M+1}^{\infty} \left( N^{-2n} + \int_{0}^{\infty} (t+N)^{-2n} dt \right)$$

$$= \sum_{n=M+1}^{\infty} \frac{1}{N^{2n}} \left( 1 + \frac{N}{2n-1} \right) \leq \sum_{n=M+1}^{\infty} \frac{N+1}{N^{2n}} = \frac{1}{N^{2M}(N-1)} \leq \frac{1}{N^{2M}}.$$ 

Thus, for an error of at most $2^{-p}$ in the series, it is sufficient to choose $M \geq p/(2 \log_2 N)$.

9.30.7 Glaisher’s constant

Glaisher’s constant $A = \exp(1/12 - \zeta'(1))$ is computed directly from this formula. We don’t use the reflection formula for the zeta function, as the arithmetic in Euler-Maclaurin summation is faster at $s = -1$ than at $s = 2$.

9.30.8 Reciprocal Fibonacci constant

We use Gosper’s series ([Gos1974], corrected in [Arn2012])

$$\sum_{n=1}^{\infty} \frac{1}{F_n} = \sum_{n=0}^{\infty} \frac{(-1)^{n+1}/2}{F_{2n+1}F_{2n+2}L_1L_2 \cdots L_{2n+1}}$$

where $L_n = 2F_{n-1} + F_n$ denotes a Lucas number. The truncation error after $N \geq 1$ terms is bounded by $(1/\phi)^{N^2}$. The series is not of hypergeometric type, but we can evaluate it in quasilinear time using binary splitting; factoring out a multiplicative recurrence for $L_1L_2 \cdots$ allows computing the series as a product of $O(\sqrt{p})$ matrices with $O(\sqrt{p})$-bit entries.

9.31 Algorithms for the gamma function

9.31.1 The Stirling series

In general, the gamma function is computed via the Stirling series

$$\log \Gamma(z) = \left( z - \frac{1}{2} \right) \log z - z + \frac{\ln 2\pi}{2} + \sum_{k=1}^{n-1} \frac{B_{2k}}{2k(2k-1)} z^{2k-1} + R_n(z)$$

where ([Olv1997] pp. 293-295) the remainder term is exactly

$$R_n(z) = \int_{0}^{\infty} \frac{B_{2n} - \tilde{B}_{2n}(x)}{2n(x+z)^{2n}} dx.$$ 

To evaluate the gamma function of a power series argument, we substitute $z \to z + t \in \mathbb{C}[t]$.

Using the bound for $|x+z|$ given by [Olv1997] and the fact that the numerator of the integrand is bounded in absolute value by $2|B_{2n}|$, the remainder can be shown to satisfy the bound

$$|[t^k]R_n(z+t)| \leq 2|B_{2n}| \frac{\Gamma(2n+k-1)}{\Gamma(k+1)\Gamma(2n+1)} |z| \left( \frac{b}{|z|} \right)^{2n+k}.$$
where \( b = 1 / \cos(\arg(z)/2) \). Note that by trigonometric identities, assuming that \( z = x + yi \), we have \( b = \sqrt{1 + u^2} \) where
\[
u = \frac{y}{\sqrt{x^2 + y^2 + x}} = \frac{\sqrt{x^2 + y^2} - x}{y}.
\]

To use the Stirling series at \( p \)-bit precision, we select parameters \( r, n \) such that the remainder \( R(n, z) \) approximately is bounded by \( 2^{-p} \). If \( |z| \) is too small for the Stirling series to give sufficient accuracy directly, we first translate to \( z + r \) using the formula \( \Gamma(z) = \Gamma(z + r)/(z(z + 1)(z + 2) \cdots (z + r - 1)) \).

To obtain a remainder smaller than \( 2^{-p} \), we must choose an \( r \) such that, in the real case, \( z + r > \beta p \), where \( \beta > \log(2)/(2\pi) \approx 0.11 \). In practice, a slightly larger factor \( \beta \approx 0.2 \) more closely balances \( n \) and \( r \). A much larger \( \beta \) (e.g. \( \beta = 1 \)) could be used to reduce the number of Bernoulli numbers that have to be precomputed, at the expense of slower repeated evaluation.

### 9.31.2 Rational arguments

We use efficient methods to compute \( y = \Gamma(p/q) \) where \( q \) is one of 1, 2, 3, 4, 6 and \( p \) is a small integer.

The cases \( \Gamma(1) = 1 \) and \( \Gamma(1/2) = \sqrt{\pi} \) are trivial. We reduce all remaining cases to \( \Gamma(1/3) \) or \( \Gamma(1/4) \) using the following relations:
\[
\Gamma(2/3) = \frac{2\pi}{3^{1/2}\Gamma(1/3)}, \quad \Gamma(3/4) = \frac{2^{1/2}\pi}{\Gamma(1/4)},
\]
\[
\Gamma(1/6) = \frac{\Gamma(1/3)^2}{(\pi/3)^{1/2}2^{1/3}}, \quad \Gamma(5/6) = \frac{2\pi(\pi/3)^{1/2}2^{1/3}}{\Gamma(1/3)^2}.
\]

We compute \( \Gamma(1/3) \) and \( \Gamma(1/4) \) rapidly to high precision using
\[
\Gamma(1/3) = \left(\frac{12\pi^4}{\sqrt{10}} \sum_{k=0}^{\infty} \frac{(6k)!(-1)^k}{(k!)^3(3k)!3^k160^{2k}}\right)^{1/6}, \quad \Gamma(1/4) = \sqrt{\frac{(2\pi)^{3/2}}{\operatorname{agm}(1, \sqrt{2})}}.
\]

An alternative formula which could be used for \( \Gamma(1/3) \) is
\[
\Gamma(1/3) = \frac{2^{4/9}\pi^{2/3}}{3^{1/12}\left(\operatorname{agm}\left(1, \frac{1}{2}\sqrt{2 + \sqrt{3}}\right)\right)^{1/3}},
\]
but this appears to be slightly slower in practice.

### 9.32 Algorithms for the Hurwitz zeta function

#### 9.32.1 Euler-Maclaurin summation

The Euler-Maclaurin formula allows evaluating the Hurwitz zeta function and its derivatives for general complex input. The algorithm is described in [Joh2013].

#### 9.32.2 Parameter Taylor series

To evaluate \( \zeta(s, a) \) for several nearby parameter values, the following Taylor expansion is useful:
\[
\zeta(s, a + x) = \sum_{k=0}^{\infty} \frac{(-x)^k (s)_k}{k!} \zeta(s + k, a)
\]
We assume that \(a \geq 1\) is real and that \(\sigma = \text{re}(s)\) with \(K + \sigma > 1\). The tail is bounded by
\[
\sum_{k=K}^{\infty} |x|^k \frac{|(s)_k|}{k!} \zeta(\sigma + k, a) \leq \sum_{k=K}^{\infty} |x|^k \frac{|(s)_k|}{k!} \left[ \frac{1}{a^{\sigma+k}} + \frac{1}{(\sigma + k - 1)a^{\sigma+k-1}} \right].
\]

Denote the term on the right by \(T(k)\). Then
\[
\left| \frac{T(k+1)}{T(k)} \right| = \left| \frac{x}{a} \frac{(k + \sigma - 1)(k + \sigma + a)}{(k + \sigma)(k + \sigma + a - 1)(k + 1)} \right| \leq \left| \frac{x}{a} \left( 1 + \frac{1}{K + \sigma + a - 1} \right) \left( 1 + \frac{|s - 1|}{K + 1} \right) = C \right|
\]
and if \(C < 1\),
\[
\sum_{k=K}^{\infty} T(k) \leq \frac{T(K)}{1 - C}.
\]

### 9.33 Algorithms for polylogarithms

The polylogarithm is defined for \(s, z \in \mathbb{C}\) with \(|z| < 1\) by
\[
\text{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}
\]
and for \(|z| \geq 1\) by analytic continuation, except for the singular point \(z = 1\).

#### 9.33.1 Computation for small \(z\)

The power sum converges rapidly when \(|z| \ll 1\). To compute the series expansion with respect to \(s\), we substitute \(s \rightarrow s + x \in \mathbb{C}[\llbracket x \rrbracket]\) and obtain
\[
\text{Li}_{s+x}(z) = \sum_{d=0}^{\infty} x^d \frac{(-1)^d}{d!} \sum_{k=1}^{\infty} T(k)
\]
where
\[
T(k) = \frac{z^k \log^d(k)}{k^s}.
\]
The remainder term \(|\sum_{k=N}^{\infty} T(k)|\) is bounded via the following strategy, implemented in `mag_polylog_tail()`.

Denote the terms by \(T(k)\). We pick a nonincreasing function \(U(k)\) such that
\[
\frac{T(k+1)}{T(k)} = z \left( \frac{k}{k+1} \right)^s \left( \frac{\log(k+1)}{\log(k)} \right)^d \leq U(k).
\]
Then, as soon as \(U(N) < 1\),
\[
\sum_{k=N}^{\infty} T(k) \leq T(N) \sum_{k=0}^{\infty} U(N)^k = \frac{T(N)}{1 - U(N)}.
\]
In particular, we take
\[
U(k) = z B(k, \max(0, -s)) B(k \log(k), d)
\]
where \(B(m, n) = (1 + 1/m)^n\). This follows from the bounds
\[
\left( \frac{k}{k+1} \right)^s \leq \begin{cases} 1 & \text{if } s \geq 0 \\ (1 + 1/k)^{-s} & \text{if } s < 0 \end{cases}
\]
and
\[
\left( \frac{\log(k+1)}{\log(k)} \right)^d \leq \left( 1 + \frac{1}{k \log(k)} \right)^d.
\]
9.33.2 Expansion for general $z$

For general complex $s$, $z$, we write the polylogarithm as a sum of two Hurwitz zeta functions

$$
\text{Li}_s(z) = \frac{\Gamma(v)}{(2\pi i)^v} \left[ i^v \zeta \left( v, \frac{1}{2} + \frac{\log(-z)}{2\pi i} \right) + i^{-v} \zeta \left( v, \frac{1}{2} - \frac{\log(-z)}{2\pi i} \right) \right]
$$

in which $s = 1 - v$. With the principal branch of $\log(-z)$, we obtain the conventional analytic continuation of the polylogarithm with a branch cut on $z \in (1, +\infty)$.

To compute the series expansion with respect to $v$, we substitute $v \to v + x \in \mathbb{C}[[x]]$ in this formula (at the end of the computation, we map $x \to -x$ to obtain the power series for $\text{Li}_{1+x}(z)$). The right hand side becomes

$$
\Gamma(v+x)[E_1Z_1 + E_2Z_2]
$$

where $E_1 = (i/(2\pi))^{v+x}$, $Z_1 = \zeta(v+x,\ldots)$, $E_2 = (1/(2\pi i))^{v+x}$, $Z_2 = \zeta(v+x,\ldots)$.

When $v = 1$, the $Z_1$ and $Z_2$ terms become Laurent series with a leading $1/x$ term. In this case, we compute the deflated series $\tilde{Z}_1, \tilde{Z}_2 = \zeta(x,\ldots) - 1/x$. Then

$$
E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2 = (E_1 + E_2)/x + E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2.
$$

Note that $(E_1 + E_2)/x$ is a power series, since the constant term in $E_1 + E_2$ is zero when $v = 1$. So we simply compute one extra derivative of both $E_1$ and $E_2$, and shift them one step. When $v = 0, -1, -2, \ldots$, the $\Gamma(v+x)$ prefactor has a pole. In this case, we proceed analogously and formally multiply $x \Gamma(v+x)$ with $[E_1Z_1 + E_2Z_2]/x$.

Note that the formal cancellation only works when the order $s$ (or $v$) is an exact integer: it is not currently possible to use this method when $s$ is a small ball containing any of $0, 1, 2, \ldots$ (then the result becomes indeterminate).

The Hurwitz zeta method becomes inefficient when $|z| \to 0$ (it gives an indeterminate result when $z = 0$). This is not a problem since we just use the defining series for the polylogarithm in that region. It also becomes inefficient when $|z| \to \infty$, for which an asymptotic expansion would better.

9.34 Algorithms for hypergeometric functions

The algorithms used to compute hypergeometric functions are described in [Joh2016]. Here, we state the most important error bounds.

9.34.1 Convergent series

Let

$$
T(k) = \frac{\prod_{i=0}^{b-1} (a_i)_k}{\prod_{i=0}^{n-1} (b_i)_k} z^k.
$$

We compute a factor $C$ such that

$$
\left| \sum_{k=n}^{\infty} T(k) \right| \leq C |T(n)|.
$$

We check that $\text{Re}(b + n) > 0$ for all lower parameters $b$. If this does not hold, $C$ is set to infinity. Otherwise, we cancel out pairs of parameters $a$ and $b$ against each other. We have

$$
\left| \frac{a + k}{b + k} \right| = \left| 1 + \frac{a - b}{b + k} \right| \leq 1 + \left| \frac{a - b}{b + n} \right|
$$
and

\[
\left| \frac{1}{b + k} \right| \leq \frac{1}{|b + n|}
\]

for all \( k \geq n \). This gives us a constant \( D \) such that \( T(k + 1) \leq DT(k) \) for all \( k \geq n \). If \( D \geq 1 \), we set \( C \) to infinity. Otherwise, we take \( C = \sum_{k=0}^{\infty} D^k = (1 - D)^{-1} \).

### 9.34.2 Convergent series of power series

The same principle is used to get tail bounds for with \( a_i, b_i, z \in \mathbb{C}[[x]] \), or more precisely, bounds for each coefficient in \( \sum_{k=0}^{\infty} T(k) \in \mathbb{C}[[x]]/(x^n) \) given \( a_i, b_i, z \in \mathbb{C}[[x]]/(x^n) \). First, we fix some notation, assuming that \( A \) and \( B \) are power series:

- \( A[k] \) denotes the coefficient of \( x^k \) in \( A \), and \( A[m,n] \) denotes the power series \( \sum_{k=m}^{n-1} A[k]x^k \).
- \( |A| \) denotes \( \sum_{k=0}^{\infty} |A[k]|x^k \) (this can be viewed as an element of \( \mathbb{R}_0[[x]] \)).
- \( A \leq B \) signifies that \( |A[k]| \leq |B[k]| \) holds for all \( k \).
- We define \( \mathcal{R}(B) = |B[0]| - |B[1:\infty]| \).

Using the formulas

\[
(AB)[k] = \sum_{j=0}^{k} A[j]B[k-j], \quad (1/B)[k] = \frac{1}{B[0]} \sum_{j=1}^{k} -B[j](1/B)[k-j],
\]

it is easy prove the following bounds for the coefficients of sums, products and quotients of formal power series:

\[
|A + B| \leq |A| + |B|, \quad |AB| \leq |A||B|, \quad |A/B| \leq |A|/\mathcal{R}(B).
\]

If \( p \leq q \) and \( \text{Re}(b_{i[0]} + N) > 0 \) for all \( b_{i} \), then we may take

\[
D = |z| \prod_{i=1}^{p} \left( 1 + \frac{|a_i - b_i|}{\mathcal{R}(b_i + N)} \right) \prod_{i=p+1}^{q} \frac{1}{\mathcal{R}(b_i + N)}.
\]

If \( D[0] < 1 \), then \( (1 - D)^{-1}[T(n)] \) gives the error bound.

Note when adding and multiplying power series with (complex) interval coefficients, we can use point-valued upper bounds for the absolute values instead of performing interval arithmetic throughout. For \( \mathcal{R}(B) \), we must then pick a lower bound for \( |B[0]| \) and upper bounds for the coefficients of \( |B[1:\infty]| \).

### 9.34.3 Asymptotic series for the confluent hypergeometric function

Let \( U(a, b, z) \) denote the confluent hypergeometric function of the second kind with the principal branch cut, and let \( U^* = z^aU(a, b, z) \). For all \( z \neq 0 \) and \( b \notin \mathbb{Z} \) (but valid for all \( b \) as a limit), we have (DLMF 13.2.42)

\[
U(a, b, z) = \frac{\Gamma(1-b)}{\Gamma(a-b+1)} M(a, b, z) + \frac{\Gamma(b-1)}{\Gamma(a)} z^{1-b} M(a-b+1, 2-b, z).
\]

Moreover, for all \( z \neq 0 \) we have

\[
\frac{\text{}_1F_1(a, b, z)}{\Gamma(b)} = \frac{(-z)^{-a}}{\Gamma(b-a)} U'(a, b, z) + \frac{z^{a-b} \text{e}^z}{\Gamma(a)} U^*(b-a, b, -z)
\]

which is equivalent to DLMF 13.2.41 (but simpler in form).

We have the asymptotic expansion

\[
U^*(a, b, z) \sim \text{}_2F_0(a, a-b+1, -1/z)
\]
where \( _2F_0(a, b, z) \) denotes a formal hypergeometric series, i.e.

\[
U^*(a, b, z) = \sum_{k=0}^{n-1} \frac{(a)_k(a - b + 1)_k}{k!(a)_k} + \varepsilon_n(z).
\]

The error term \( \varepsilon_n(z) \) is bounded according to DLMF 13.7. A case distinction is made depending on whether \( z \) lies in one of three regions which we index by \( R \). Our formula for the error bound increases with the value of \( R \), so we can always choose the larger out of two indices if \( z \) lies in the union of two regions.

Let \( r = |b - 2a| \). If \( \text{Re}(z) \geq r \), set \( R = 1 \). Otherwise, if \( \text{Im}(z) \geq \text{Re}(z) \geq 0 \land |z| \geq r \), set \( R = 2 \). Otherwise, if \( |z| \geq 2r \), set \( R = 3 \). Otherwise, the bound is infinite. If the bound is finite, we have

\[
|\varepsilon_n(z)| \leq 2\alpha C_n \left| \frac{(a)_n(a - b + 1)_n}{n!z^n} \right| \exp(2\alpha \rho C_1/|z|)
\]

in terms of the following auxiliary quantities

\[
\sigma = |(b - 2a)/z| \\
C_n = \begin{cases} 1 & \text{if } R = 1 \\ \chi(n) & \text{if } R = 2 \\ (\chi(n) + \sigma \nu^2 n) \nu^n & \text{if } R = 3 \end{cases} \\
\nu = \left( \frac{1}{2} + \frac{1}{2} \sqrt{1 - 4 \sigma^2} \right)^{-1/2} \leq 1 + 2 \sigma^2 \\
\chi(n) = \sqrt{\pi} \Gamma(\frac{1}{2} n + 1)/\Gamma(\frac{1}{2} n + \frac{1}{2}) \\
\sigma' = \begin{cases} \sigma & \text{if } R \neq 3 \\ \nu \sigma & \text{if } R = 3 \end{cases} \\
\alpha = (1 - \sigma')^{-1} \\
\rho = \frac{1}{2} |2a^2 - 2ab + b| + \sigma'(1 + 4 \sigma')(1 - \sigma')^{-2}
\]

### 9.34.4 Asymptotic series for Airy functions

Error bounds are based on Olver (DLMF section 9.7). For \( \arg(z) < \pi \) and \( \zeta = (2/3)z^{3/2} \), we have

\[
\text{Ai}(z) = \frac{e^{-\zeta}}{2\sqrt{\pi}z^{1/4}} \left[ S_n(\zeta) + R_n(z) \right], \quad \text{Ai}’(z) = -\frac{2^{1/4}e^{-\zeta}}{2\sqrt{\pi}} \left[ S_n’(\zeta) + R_n’(z) \right]
\]

\[
S_n(\zeta) = \sum_{k=0}^{n-1} (-1)^k \frac{u(k)}{\zeta^k}, \quad S_n’(\zeta) = \sum_{k=0}^{n-1} (-1)^k \frac{v(k)}{\zeta^k}
\]

\[
u(k) = \frac{(1/6)k(5/6)k}{2^k k!}, \quad v(k) = \frac{6k + 1}{1 - 6k} u(k).
\]

Assuming that \( n \) is positive, the error terms are bounded by

\[
|R_n(z)| \leq C|u(n)||\zeta|^{-n}, \quad |R_n’(z)| \leq C|v(n)||\zeta|^{-n}
\]

where

\[
C = \begin{cases} 2 \exp(7/(36|\zeta|)) & |\arg(z)| \leq \pi/3 \\ 2 \chi(n) \exp(7\pi/(72|\zeta|)) & \pi/3 \leq |\arg(z)| \leq 2\pi/3 \\ 4 \chi(n) \exp(7\pi/(36 \text{re}(\zeta))) \cos(\arg(\zeta)) |\zeta|^{-n} & 2\pi/3 \leq |\arg(z)| < \pi. \end{cases}
\]
For computing $B_i$ when $z$ is roughly in the positive half-plane, we use the connection formulas

\[
B_i(z) = -i(2w^+ Ai(zw^{-2}) - A_i(z)) \\
B_i(z) = +i(2w^- Ai(zw^+2) - A_i(z))
\]

where $w = \exp(\pi i/3)$. Combining roots of unity gives

\[
B_i(z) = \frac{1}{2\sqrt{\pi z}^{1/4}}[2X + iY] \\
B_i(z) = \frac{1}{2\sqrt{\pi z}^{1/4}}[2X - iY]
\]

$X = \exp(+\zeta)[S_n(-\zeta) + R_n(zw^{-2})]$, $Y = \exp(-\zeta)[S_n(\zeta) + R_n(z)]$

where the upper formula is valid for $-\pi < \arg(\zeta) < \pi$ and the lower formula is valid for $-\pi/3 < \arg(z) < \pi/3$. We proceed analogously for the derivative of $B_i$.

In the negative half-plane, we use the connection formulas

\[
A_i(z) = e^{+\pi i/3} A_i(z_1) + e^{-\pi i/3} A_i(z_2) \\
B_i(z) = e^{-\pi i/6} A_i(z_1) + e^{+\pi i/6} A_i(z_2)
\]

where $z_1 = -ze^{+\pi i/3}, z_2 = -ze^{-\pi i/3}$. Provided that $|\arg(-z)| < 2\pi/3$, we have $|\arg(z_1)|, |\arg(z_2)| < \pi$, and thus the asymptotic expansion for $A_i$ can be used. As before, we collect roots of unity to obtain

\[
A_i(z) = A_1[S_n(i\zeta) + R_n(z_1)] + A_2[S_n(-i\zeta) + R_n(z_2)] \\
B_i(z) = A_3[S_n(i\zeta) + R_n(z_1)] + A_4[S_n(-i\zeta) + R_n(z_2)]
\]

where $\zeta = (2/3)(-z)^{3/2}$ and

\[
A_1 = \frac{\exp(-i(\zeta - \pi/4))}{2\sqrt{\pi(-z)^{1/4}}}, \quad A_2 = \frac{\exp(i(\zeta - \pi/4))}{2\sqrt{\pi(-z)^{1/4}}}, \quad A_3 = -iA_1, \quad A_4 = +iA_2.
\]

The differentiated formulas are analogous.

### 9.34.5 Corner case of the Gauss hypergeometric function

In the corner case where $z$ is near $\exp(\pm \pi i/3)$, none of the linear fractional transformations is effective. In this case, we use Taylor series to analytically continue the solution of the hypergeometric differential equation from the origin. The function $f(z) = {}_2F_1(a, b, c, z_0 + z)$ satisfies

\[
f''(z) = -\frac{(a + b + 1 - c)(a + b + 1 - c)}{(z_0 + z)(z_0 - 1 + z)} f'(z) - \frac{ab}{(z_0 + z)(z_0 - 1 + z)} f(z).
\]

Knowing $f(0), f'(0)$, we can compute the consecutive derivatives recursively, and evaluating the truncated Taylor series allows us to compute $f(z), f'(z)$ to high accuracy for sufficiently small $z$. Some experimentation showed that two continuation steps

\[
0 \rightarrow 0.375 \pm 0.625i \rightarrow 0.5 \pm 0.8125i \rightarrow z
\]

gives good performance. Error bounds for the truncated Taylor series are obtained using the Cauchy-Kovalevskaya majorant method, following the outline in [Hoe2001]. The differential equation is majorized by

\[
g''(z) = \frac{N + 1}{2} \left( \frac{\nu}{1 - \nu z} \right) g'(z) + \frac{(N + 1)N}{2} \left( \frac{\nu}{1 - \nu z} \right)^2 g(z)
\]

provided that $N$ and $\nu \geq \max(1/|z_0|, 1/|z_0 - 1|)$ are chosen sufficiently large. It follows that we can compute explicit numbers $A, N, \nu$ such that the simple solution $g(z) = A(1 - \nu z)^{-N}$ of the differential equation provides the bound

\[
|f_{\{k\}}| \leq g_{\{k\}} = A\left( \frac{N + k}{k} \right)^{\nu^k}.
\]
9.35 Algorithms for the arithmetic-geometric mean

With complex variables, it is convenient to work with the univariate function \( M(z) = \text{agm}(1, z) \). The general case is given by \( \text{agm}(a, b) = aM(1, b/a) \).

9.35.1 Functional equation

If the real part of \( z \) initially is not completely nonnegative, we apply the functional equation \( M(z) = (z + 1)M(u)/2 \) where \( u = \sqrt{z}/(z + 1) \).

Note that \( u \) has nonnegative real part, absent rounding error. It is not a problem for correctness if rounding makes the interval contain negative points, as this just inflates the final result.

For the derivative, the functional equation becomes \( M'(z) = [M(u) - (z - 1)M'(u)/((1 + z)\sqrt{z})]/2 \).

9.35.2 AGM iteration

Once \( z \) is in the right half plane, we can apply the AGM iteration \( (2a_{n+1} = a_n + b_n, b_{n+1}^2 = a_n b_n) \) directly. The correct square root is given by \( \sqrt{\frac{a}{b}} \), which is computed as \( \sqrt{a b}, -i\sqrt{-a b}, -\sqrt{-a b}, \sqrt{a b} \) respectively if both \( a \) and \( b \) have positive real part, nonnegative imaginary part, nonpositive imaginary part, or otherwise.

The iteration should be terminated when \( a_n \) and \( b_n \) are close enough. For positive real variables, we can simply take lower and upper bounds to get a correct enclosure at this point. For complex variables, it is shown in [Dup2006], p. 87 that, for \( z \) with nonnegative real part, \( |M(z) - a_n| \leq |a_n - b_n| \), giving a convenient error bound.

Rather than running the AGM iteration until \( a_n \) and \( b_n \) agree to \( p \) bits, it is slightly more efficient to iterate until they agree to about \( p/10 \) bits and finish with a series expansion. With \( z = (a - b)/(a + b) \), we have

\[
\text{agm}(a, b) = \frac{(a + b)\pi}{4K(z^2)},
\]

valid at least when \( |z| < 1 \) and \( a, b \) have nonnegative real part, and

\[
\frac{\pi}{4K(z^2)} = \frac{1}{2} - \frac{1}{8} z^2 - \frac{5}{128} z^4 - \frac{11}{512} z^6 - \frac{469}{32768} z^8 + \ldots
\]

where the tail is bounded by \( \sum_{k=10}^\infty |z|^k/64 \).

9.35.3 First derivative

Assuming that \( z \) is exact and that \( |\arg(z)| \leq 3\pi/4 \), we compute \((M(z), M'(z))\) simultaneously using a finite difference.

The basic inequality we need is \( |M(z)| \leq \max(1, |z|) \), which is an immediate consequence of the AGM iteration.

By Cauchy’s integral formula, \( |M^{(k)}(z)/k!| \leq CD^k \) where \( C = \max(1, |z| + r) \) and \( D = 1/r \), for any \( 0 < r < |z| \) (we choose \( r \) to be of the order \( |z|/4 \)). Taylor expansion now gives

\[
\left| \frac{M(z + h) - M(z)}{h} - M'(z) \right| \leq \frac{CD^2 h}{1 - Dh}
\]

\[
\left| \frac{M(z + h) - M(z - h)}{2h} - M'(z) \right| \leq \frac{CD^2 h^2}{1 - Dh}
\]

\[
\left| \frac{M(z + h) + M(z - h)}{2} - M(z) \right| \leq \frac{CD^2 h^2}{1 - Dh}
\]
assuming that \( h \) is chosen so that it satisfies \( hD < 1 \).

The forward finite difference would require two function evaluations at doubled precision. We use the central difference as it only requires 1.5 times the precision.

When \( z \) is not exact, we evaluate at the midpoint as above and bound the propagated error using derivatives. Again by Cauchy’s integral formula, we have

\[
|M'(z + \varepsilon)| \leq \frac{\max(1, |z| + |\varepsilon| + r)}{r}
\]

\[
|M''(z + \varepsilon)| \leq \frac{2 \max(1, |z| + |\varepsilon| + r)}{r^2}
\]

assuming that the circle centered on \( z \) with radius \( |\varepsilon| + r \) does not cross the negative half axis. We choose \( r \) of order \( |z|/2 \) and verify that all assumptions hold.

### 9.35.4 Higher derivatives

The function \( W(z) = 1/M(z) \) is D-finite. The coefficients of \( W(z + x) = \sum_{k=0}^{\infty} c_k x^k \) satisfy

\[
-2z(z^2 - 1)c_2 = (3z^2 - 1)c_1 + zc_0,
\]

\[
-(k + 2)(k + 3)z(z^2 - 1)c_{k+3} = (k + 2)^2(3z^2 - 1)c_{k+2} + (3(k + 3) + 7)zc_{k+1} + (k + 1)^2c_k
\]

in general, and

\[
-(k + 2)^2c_{k+2} = (3(k + 3) + 7)c_{k+1} + (k + 1)^2c_k
\]

when \( z = 1 \).
CHAPTER
TEN

EXACT REAL AND COMPLEX NUMBERS

10.1 Introduction

10.1.1 Exact numbers in Calcium

The core idea behind Calcium is to represent real and complex numbers as elements of extension fields

\[ \mathbb{Q}(a_1, \ldots, a_n) \]

of the rational numbers, where the extension numbers \( a_k \) are described by symbolic expressions (which may depend on other fields recursively). The system constructs such fields automatically as needed to represent the results of computations. Any extension field is isomorphic to a formal field

\[ \mathbb{Q}(a_1, \ldots, a_n) \cong K_{\text{formal}} := \text{Frac}(\mathbb{Q}[X_1, \ldots, X_n]/I) \]

where \( I \) is the ideal of algebraic relations among the extension numbers. The relations may involve algebraic numbers (for example: \( i^2 + 1 = 0 \)), transcendental numbers (for example: \( e^{-\pi} \cdot e^\pi = 1 \)), or combinations thereof.

Computation in the formal field depends (in general) on multivariate polynomial arithmetic together with use of a Gröbner basis for the ideal. The map from the formal field to the true complex field is maintained using arbitrary-precision ball arithmetic where necessary.

As an important special case, Calcium can be used for arithmetic in algebraic number fields (embedded explicitly in \( \mathbb{C} \))

\[ \mathbb{Q}(a) \cong \mathbb{Q}[X]/(f(X)) \]

with excellent performance thanks to internal use of the Antic library.

It will not always work perfectly: although Calcium by design should never give a mathematically erroneous answer, it may be unable to simplify a result as much as expected and it may be unable to decide a predicate (in which case it can return “Unknown”). Equality is at least decidable over the algebraic numbers \( \mathbb{Q} \) (for practical degrees and bit sizes of the numbers!), and in certain cases involving transcendentals. We hope to improve Calcium’s capabilities gradually through enhancements to its built-in algorithms and through customization options.

Usage details

To understand how Calcium works more concretely, see Calcium example programs and the documentation for the main Calcium number type (ca_t):

- ca.h – exact real and complex numbers

Implementation details for extension numbers and formal fields can be found in the documentation of the corresponding modules:

- ca_ext.h – real and complex extension numbers
• **ca_field.h** – extension fields

The following modules are used internally for arithmetic in transcendental number fields (rational function fields) \( \mathbb{Q}(x_1, \ldots, x_n) \) and over the field of algebraic numbers \( \overline{\mathbb{Q}} \), respectively. They may be of independent interest:

• **fmpz_mpoly_q.h** – multivariate rational functions over \( \mathbb{Q} \)

• **qqbar.h** – algebraic numbers represented by minimal polynomials

### 10.1.2 FAQ

Isn’t \( x = 0 \) undecidable?

In general, yes: equality over the reals is undecidable. In practice, much of calculus and elementary number theory can be done with numbers that are simple algebraic combinations of well-known elementary and special functions, and there are heuristics that work quite well for deciding predicates about such numbers. Calcium will be able to give a definitive answer at least in simple cases (for example, proving \( 16 \arctan\left(\frac{1}{5}\right) - 4 \arctan\left(\frac{1}{239}\right) = \pi \) or \( \sqrt{5 + 2\sqrt{6}} = \sqrt{2} + \sqrt{3} \)), and will simply answer “Unknown” when its heuristics are not powerful enough.

How does Calcium compare to ordinary numerical computing?

Calcium is far too slow to replace floating-point numbers for 99.93% of scientific computing. The target is symbolic and algebraic computation. Nevertheless, Calcium may well be useful as a tool to test and enhance the capabilities of numerical programs.

How does Calcium compare to Arb arithmetic?

The main advantage of Calcium over ball arithmetic alone is the ability to do exact comparisons. The automatic precision management in Calcium can also be convenient.

Calcium will usually be slower than Arb arithmetic. If a computation is mostly numerical, it is probably better to try using Arb first, and fall back on an exact calculation with Calcium only if that fails because an exact comparison is needed.

How does Calcium compare to symbolic computation systems (Mathematica, SymPy, etc.)?

Calculating with constant values is only a small part of what such systems have to do, but it is one of the most complex parts. Existing computer algebra systems sometimes manage this very well, and sometimes fail horribly. The most common problems are 1) getting numerical error bounds or branch cuts wrong, and 2) slowing down too much when the expressions get large. Calcium is intended to address both problems (through rigorous numerical evaluation and use of fast polynomial arithmetic).

Ultimately, Calcium will no doubt handle some problems better and others worse, and it should be considered a complement to existing computer algebra systems rather than a replacement. A symbolic expression simplifier may use Calcium evaluation as one of its tools, but this probably needs to be done selectively and in combination with many other heuristics.

Why is Calcium written in C?

The main advantage of developing Calcium as a C library is that it will not be tied to a particular programming language ecosystem: C is uniquely easy to interface from almost any other language. The second most important reason is familiarity: Calcium follows the design of Flint and Arb (coding style, naming, module layout, memory management, test code, etc.) which has proved to work quite well for libraries of this type.

There is also the performance argument. Some core functions will benefit from optimizations that are natural in C such as in-place operations and fine-grained manual memory management. However, the performance aspect should not be overemphasized: Calcium will spend most of its time in Flint and Arb kernel functions and this would probably still be true even if it were written in a slower language.

There are certainly types of mathematical functionality that will be too inconvenient to implement in C. Our intention is indeed to leave such functionality to projects written in Python, Julia, etc. which may then opt to depend on Calcium for basic operations.
What is the development status of Calcium?

Calcium is presently in early development and should be considered experimental software. The interfaces are subject to change and many important functions and optimizations have not been implemented. A more stable and functional release can be expected in late 2021.
10.2 Calcium example programs

See *Examples* for general information about example programs. Running:

```
make examples
```

will compile the programs and place the binaries in `build/examples`. The examples related to the Calcium module are documented below.

### 10.2.1 elementary.c

This program evaluates several elementary expressions. For some inputs, Calcium’s arithmetic should produce a simplified result automatically. Some inputs do not yet automatically simplify as much as one might hope. Calcium may still able to prove that such a number is zero or nonzero; the output of `ca_check_is_zero()` is then `T_TRUE` or `T_FALSE`.

Sample output:

```
> build/examples/elementary
>>> Exp(Pi*I) + 1
0

>>> Log(-1) / (Pi*I)
1

>>> Log(-I) / (Pi*I)
-0.500000 {-1/2}

>>> Log(1 / 10^{-123}) / Log(100)
-61.5000 {-123/2}

>>> Log(1 + Sqrt(2)) / Log(3 + 2*Sqrt(2))
0.500000 {1/2}

>>> Sqrt(2)*Sqrt(3) - Sqrt(6)
0

>>> Exp(1+Sqrt(2)) * Exp(1-Sqrt(2)) / (Exp(1)^2)
1

>>> I^-I - Exp(-Pi/2)
0

>>> Exp(Sqrt(3))^2 - Exp(Sqrt(12))
0

>>> 2*Log(Pi*I) - 4*Log(Sqrt(Pi)) - Pi*I
0

>>> -I*Pi/8*Log((2/3-2*I/3)^2 + I*Pi/8*Log(2/3+2*I/3)^2 + Pi^-2/12*Log(-1-I) + Pi^-2/...
0

>>> Sqrt(5 + 2*Sqrt(6)) - Sqrt(2) - Sqrt(3)
0e-1126 {a-c-d where a = 3.14626 [Sqrt(9.89898 {2*b+5})], b = 2.44949 [b^2-6=0], c ...

>>> Is zero?
```

(continues on next page)
>>> Sqrt(I) - (1+I)/Sqrt(2)
0e-1126 + 0e-1126*I {2*a-b*c-b)/2 where a = 0.707107 + 0.707107*I [Sqrt(1.00000*I {c} →]), b = 1.41421 [b^2-2=0], c = I [c^2+1=0]}

>>> Is zero?
T_TRUE

>>> Exp(Pi*Sqrt(163)) - (640320^3 + 744)
-7.49927e-13 {a-262537412640768744 where a = 2.62537e+17 [Exp(40.1092 {b*c})], b = 3.14159 [Pi], c = 12.7671 [c^2-163=0]}

>>> Erf(2*Log(Sqrt(1/2-Sqrt(2)/4))+Log(4)) - Erf(Log(2-Sqrt(2)))
0

cpu/wall(s): 0.022 0.022

10.2. Calcium example programs

This program computes the n-th Fibonacci number using Binet’s formula \(F_n = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}\) where \(\phi = \frac{1}{2}(1 + \sqrt{5})\). The program takes \(n\) as input.

Sample output:

```
> build/examples/binet 250
7.89633e+51
```

This illustrates exact arithmetic in algebraic number fields. The program also illustrates another aspect of Calcium arithmetic: evaluation limits. For example, trying to compute the index \(n = 10^6\) Fibonacci number hits an evaluation limit, so the value is not expanded to an explicit integer:

```
> build/examples/binet 1000000
1.95328e+208987
```

Calling the program with `-limit B n` raises the bit evaluation limit to \(B\). Setting this large enough allows \(F_{10^6}\) to expand to an integer (the following output has been truncated to avoid reproducing all 208988 digits):

```
> build/examples/binet -limit 10000000 1000000
1.95328e+208987
```

The exact mechanisms and interfaces for evaluation limits are still a work in progress.
### 10.2.3 machin.c

This program checks several variations of Machin’s formula

\[
\frac{\pi}{4} = 4 \arctan \left( \frac{1}{5} \right) - \arctan \left( \frac{1}{239} \right)
\]

expressing \(\pi\) or logarithms of small integers in terms of arctangents or hyperbolic arctangents of rational numbers. The program actually evaluates \(4 \arctan \left( \frac{1}{5} \right) - \arctan \left( \frac{1}{239} \right) - \frac{\pi}{4}\) (etc.) and prints the result, which should be precisely 0, proving the identity. Inverse trigonometric functions are not yet implemented in Calcium, so the example program evaluates them using logarithms.

Sample output:

```plaintext
> build/examples/machin
[(1)*Atan(1/1) - Pi/4] = 0
[(1)*Atan(1/2) + (1)*Atan(1/3) - Pi/4] = 0
[(2)*Atan(1/2) + (-1)*Atan(1/7) - Pi/4] = 0
[(2)*Atan(1/3) + (1)*Atan(1/7) - Pi/4] = 0
[(4)*Atan(1/5) + (-1)*Atan(1/239) - Pi/4] = 0
[(1)*Atan(1/2) + (1)*Atan(1/5) + (1)*Atan(1/8) - Pi/4] = 0
[(1)*Atan(1/3) + (1)*Atan(1/4) + (1)*Atan(1/7) + (1)*Atan(1/13) - Pi/4] = 0
[(12)*Atan(1/49) + (32)*Atan(1/57) + (-5)*Atan(1/239) + (12)*Atan(1/110443) - Pi/4] = 0
[(14)*Atanh(1/31) + (10)*Atanh(1/49) + (6)*Atanh(1/161) - Log(2)] = 0
[(22)*Atanh(1/31) + (16)*Atanh(1/49) + (10)*Atanh(1/161) - Log(3)] = 0
[(32)*Atanh(1/31) + (24)*Atanh(1/49) + (14)*Atanh(1/161) - Log(5)] = 0
[(144)*Atanh(1/251) + (54)*Atanh(1/449) + (-38)*Atanh(1/4801) + (62)*Atanh(1/8749) - Log(2)] = 0
[(228)*Atanh(1/251) + (86)*Atanh(1/449) + (-60)*Atanh(1/4801) + (98)*Atanh(1/8749) - Log(3)] = 0
[(334)*Atanh(1/251) + (126)*Atanh(1/449) + (-88)*Atanh(1/4801) + (144)*Atanh(1/8749) - Log(5)] = 0
[(404)*Atanh(1/251) + (152)*Atanh(1/449) + (-106)*Atanh(1/4801) + (174)*Atanh(1/8749) - Log(7)] = 0
```

CPU/wall(s): 0.016 0.016

Virt/peak/res/peak(MB): 35.57 35.57 8.80 8.80

### 10.2.4 swinnerton_dyer_poly.c

This program computes the coefficients of the Swinnerton-Dyer polynomial

\[
S_n = \prod \left( x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \ldots \pm \sqrt{p_n} \right)
\]

where \(p_n\) denotes the \(n\)-th prime number and all combinations of signs are taken. This polynomial has degree \(2^n\). The polynomial is expanded from its roots using naive polynomial multiplication over \(ca_t\) coefficients. There are far more efficient ways to construct this polynomial; this program simply illustrates that arithmetic in multivariate number fields works smoothly.

The program prints the coefficients of \(S_n\), from the constant term to the coefficient of \(x^{2^n}\).

Sample output:

```plaintext
> build/examples/swinnerton_dyer_poly 3
576
0
-960
```

(continues on next page)
A big benchmark problem (output truncated):

```
> build/examples/swinnerton_dyer_poly 10
4.35675e+809 {43567450015...212890625}
0 ...
0 1
```

```
cpu/wall(s): 0.002 0.002
virt/peak/res/peak(MB): 35.07 35.11 5.40 5.40
```

```
10.2.5 huge_expr.c
```

This program proves equality of two complicated algebraic numbers. More precisely, the program verifies that $N = -(1 - |M|^2)^2$ where $N$ and $M$ are given by huge symbolic expressions involving nested square roots (about 7000 operations in total).

By default, the program runs the computation using `qqbar_t` arithmetic:

```
> build/examples/huge_expr
Evaluating N...
cpu/wall(s): 0.193 0.193
Evaluating M...
cpu/wall(s): 0.024 0.024
Evaluating E = -(1-|M|^2)^2...
cpu/wall(s): 0.008 0.009
N ~ -0.16190853053311203695842869991458578203473645660641
E ~ -0.16190853053311203695842869991458578203473645660641
Testing E = N...
cpu/wall(s): 0.001 0
Equal = T_TRUE
Total: cpu/wall(s): 8.53 8.531
virt/peak/res/peak(MB): 54.50 64.56 24.64 34.61
```

To run the computation using `ca_t` arithmetic instead, pass the `-ca` flag:

```
> build/examples/huge_expr -ca
Evaluating N...
cpu/wall(s): 0.193 0.193
Evaluating M...
cpu/wall(s): 0.024 0.024
Evaluating E = -(1-|M|^2)^2...
cpu/wall(s): 0.008 0.009
```

(continues on next page)
This simplification problem was posted in a help request for Sage (https://ask.sagemath.org/question/52653). The C code has been generated from the symbolic expressions using a Python script.

10.2.6 hilbert_matrix.c

This program constructs the Hilbert matrix $H_n = (1/(i + j - 1))_{i,j=1}^n$, computes its eigenvalues $\lambda_1, \ldots, \lambda_n$, as exact algebraic numbers, and verifies the exact trace and determinant formulas

$$
\lambda_1 + \lambda_2 + \ldots + \lambda_n = \text{tr}(H_n), \quad \lambda_1 \lambda_2 \cdots \lambda_n = \det(H_n).
$$

Sample output:

```
> build/examples/hilbert_matrix 6
Trace:  
1.87821 {6508/3465}  
1.87821 {6508/3465}  
Equal: T_TRUE

Det:  
5.36730e-18 {1/186313420339200000}  
5.36730e-18 {1/186313420339200000}  
Equal: T_TRUE

cpu/wall(s): 0.07 0.069  
virt/peak/res/peak(MB): 36.56 36.66 9.69 9.69
```

The program accepts the following optional arguments:

- With `-vieta`, force use of Vieta’s formula internally (by default, Calcium uses Vieta’s formulas when working with algebraic conjugates, but only up to some bound on the degree).
- With `-novieta`, force Calcium not to use Vieta’s formulas internally.
- With `-qqbar`, do a similar computation using qqbar_t arithmetic.

10.2.7 dft.c

This program demonstrates the discrete Fourier transform (DFT) in exact arithmetic. For the input vector $x = (x_n)_{n=0}^{N-1}$, it verifies the identity

$$
x - \text{DFT}^{-1}(\text{DFT}(x)) = 0
$$

where

$$
\text{DFT}(x)_n = \sum_{k=0}^{N-1} \omega^{-kn}x_k, \quad \text{DFT}^{-1}(x)_n = \frac{1}{N} \sum_{k=0}^{N-1} \omega^{kn}x_k, \quad \omega = e^{2\pi i/N}.
$$
The program computes the DFT by naive $O(N^2)$ summation (not using FFT). It uses repeated multiplication of $\omega$ to precompute an array of roots of unity $1, \omega, \omega^2, \ldots, \omega^{2N-1}$ for use in both the DFT and the inverse DFT.

Usage:

```
build/examples/dft [-verbose] [-input i] [-limit B] [-timing T] N
```

The required parameter $N$ selects the length of the vector.

The optional flag `-verbose` chooses whether to print the arrays.

The optional parameter `-timing T` selects a timing method (default = 0).
- 0: run the computation once and time it
- 1: run the computation repeatedly if needed to get an accurate timing, creating a new context object for each iteration so that fields are not cached
- 2: run the computation once, then run the computation at least one more time (repeatedly if needed to get an accurate timing), recycling the same context object to measure the performance with cached fields

The optional parameter `-input i` selects an input sequence (default = 0).
- 0: $x_n = n + 2$
- 1: $x_n = \sqrt{n} + 2$
- 2: $x_n = \log(n + 2)$
- 3: $x_n = e^{2\pi i/(n+2)}$

The optional parameter `-limit B` sets the internal degree limit for algebraic numbers.

Sample output:

```
> build/examples/dft 4 -input 1 -verbose
DFT benchmark, length N = 4

[x] =
1.41421 {a where a = 1.41421 \[a^2-2=0\]}
1.73205 {a where a = 1.73205 \[a^2-3=0\]}
2
2.23607 {a where a = 2.23607 \[a^2-5=0\]}

DFT([x]) =
7.38233 {a+b+c+2 where a = 2.23607 \[a^2-5=0\], b = 1.73205 \[b^2-3=0\], c = 1.41421 \[c^2-2=0\]}
-0.585786 + 0.504017*I {a-d*b+d*c-2 where a = 2.23607 \[a^2-5=0\], b = 1.73205 \[b^2-3=0\], c = 1.41421 \[c^2-2=0\], d = I \[d^2+1=0\]}
-0.585786 - 0.504017*I {-a-b+c+2 where a = 2.23607 \[a^2-5=0\], b = 1.73205 \[b^2-3=0\], c = 1.41421 \[c^2-2=0\], d = I \[d^2+1=0\]}

IDFT(DFT([x])) =
1.41421 {c where a = 2.23607 \[a^2-5=0\], b = 1.73205 \[b^2-3=0\], c = 1.41421 \[c^2-2=0\], d = I \[d^2+1=0\]}
1.73205 {b where a = 2.23607 \[a^2-5=0\], b = 1.73205 \[b^2-3=0\], c = 1.41421 \[c^2-2=0\], d = I \[d^2+1=0\]}
2
2.23607 {a where a = 2.23607 \[a^2-5=0\], b = 1.73205 \[b^2-3=0\], c = 1.41421 \[c^2-2=0\], d = I \[d^2+1=0\]}
```
\[ x \cdot \text{IDFT}(\text{DFT}([x])) = \]

0 \quad (= 0 \quad \text{T_TRUE})
0 \quad (= 0 \quad \text{T_TRUE})
0 \quad (= 0 \quad \text{T_TRUE})
0 \quad (= 0 \quad \text{T_TRUE})

\text{cpu/wall(s): } 0.009 \quad 0.009
\text{virt/peak/res/peak(MB): } 36.28 \quad 36.28 \quad 9.14 \quad 9.14
10.3 calcium.h – global definitions

10.3.1 Version

const char *calcium_version(void)
    Returns a pointer to the version of the library as a string X.Y.Z.

10.3.2 Triple-valued logic

The Calcium modules use two kinds of predicate functions:

- Predicates with signature int foo_is_X(const foo_t x) return the usual C boolean values 1 for true and 0 for false, unless otherwise documented. Some functions may return 0 also when truth cannot be certified (this will be documented explicitly).

- Predicates with signature truth_t foo_check_is_X(const foo_t x) check a mathematical property that may not be decidable (or may be too costly to decide). The return value is a truth_t (T_TRUE, T_FALSE or T_UNKNOWN).

10.3.3 Flint, Arb and Antic extras

Here we collect various utility methods for Flint, Arb and Antic types that are missing in those libraries. Some of these functions may be migrated upstream in the future.

ulong calcium_fmpz_hash(const fmpz_t x)
    Hash function for integers. The algorithm may change; presently, this simply extracts the low word (with sign).

10.3.4 Input and output

type calcium_stream_struct
type calcium_stream_t
    A stream object which can hold either a file pointer or a string (with automatic resizing).

void calcium_stream_init_file(calcium_stream_t out, FILE *fp)
    Initializes the stream out for writing to the file fp. The file can be stdout, stderr, or any file opened for writing by the user.

void calcium_stream_init_str(calcium_stream_t out)
    Initializes the stream out for writing to a string in memory. When finished, the user should free the string (the s member of out with flint_free()).

void calcium_write(calcium_stream_t out, const char *s)
    Writes the string s to out.

void calcium_write_free(calcium_stream_t out, char *s)
    Writes s to out and then frees s by calling flint_free().

void calcium_write_si(calcium_stream_t out, slong x)

void calcium_write_fmpz(calcium_stream_t out, const fmpz_t x)
    Writes the integer x to out.

void calcium_write_arb(calcium_stream_t out, const arb_t z, slong digits, ulong flags)

void calcium_write_acb(calcium_stream_t out, const acb_t z, slong digits, ulong flags)
    Writes the Arb number z to out, showing digits digits and with the display style specified by flags (ARB_STR_NO_RADIUS, etc.).
10.4 ca.h – exact real and complex numbers

A ca_t represents a real or complex number in a form suitable for exact field arithmetic or comparison. Exceptionally, a ca_t may represent a special nonnumerical value, such as an infinity.

10.4.1 Introduction: numbers

A Calcium number is a real or complex number represented as an element of a formal field $K = \mathbb{Q}(a_1, \ldots, a_n)$ where the symbols $a_k$ denote fixed algebraic or transcendental numbers called extension numbers. For example, $e^{-\pi} - 3i$ may be represented as $(1 - 3a_1^2)/a_2^3$ in the field $\mathbb{Q}(a_1, a_2)$ with $a_1 = i, a_2 = e^\pi$. Extension numbers and fields are documented in the following separate modules:

- ca_ext.h – real and complex extension numbers
- ca_field.h – extension fields

The user does not need to construct extension numbers or formal extension fields explicitly: each ca_t contains an internal pointer to its formal field, and operations on Calcium numbers generate and cache fields automatically as needed to express the results.

This representation is not canonical (in general). A given complex number can be represented in different ways depending on the choice of formal field $K$. Even within a fixed field $K$, a number can have different representations if there are algebraic relations between the extension numbers. Two numbers $x$ and $y$ can be tested for inequality using numerical evaluation; to test for equality, it may be necessary to eliminate dependencies between extension numbers. One of the central goals of Calcium will be to implement heuristics for such elimination.

Together with each formal field $K$, Calcium stores a reduction ideal $I = \{g_1, \ldots, g_m\}$ with $g_i \in \mathbb{Z}[a_1, \ldots, a_n]$, defining a set of algebraic relations $g_i(a_1, \ldots, a_n) = 0$. Relations can be absolute, say $g_i = a_i^2 + 1$, or relative, say $g_i = 2a_j - 4a_k - a_la_m$. The reduction ideal effectively partitions $K$ into equivalence classes of complex numbers (e.g. $i^2 = -1$ or $2\log(\pi i) = 4\log(\sqrt{\pi}) + \pi i$), enabling simplifications and equality proving.

Extension numbers are always sorted $a_1 \succ a_2 \succ \ldots \succ a_n$ where $\succ$ denotes a structural ordering (see ca_cmp_repr()). If the reduction ideal is triangular and the multivariate polynomial arithmetic uses lexicographic ordering, reduction by $I$ eliminates numbers $a_i$ with higher complexity in the sense of $\succ$.

The reduction ideal is an imperfect computational crutch: it is not guaranteed to capture all algebraic relations, and reduction is not guaranteed to produce uniquely defined representatives. However, in the specific case of an absolute number field $K = \mathbb{Q}(a)$ where $a$ is a qqbar_t extension, the reduction ideal (consisting of a single minimal polynomial) is canonical and field elements of $K$ can be chosen canonically.

10.4.2 Introduction: special values

In order to provide a closed arithmetic system and express limiting cases of operators and special functions, a ca_t can hold any of the following special values besides ordinary numbers:

- **Unsigned infinity**, a formal object $\infty$ representing the value of 1/0. More generally, this is the value of meromorphic functions at poles.
- **Signed infinity**, a formal object $a \cdot \infty$ where the sign $a$ is a Calcium number with $|a| = 1$. The most common values are $+\infty, -\infty, +i\infty, -i\infty$. Signed infinities are used to denote directional limits and logarithmic singularities (for example, $\log(0) = -\infty$).
- **Undefined**, a formal object representing the value of indeterminate forms such as $0/0$ and essential singularities such as $\exp(\infty)$, where a number or infinity would not make sense as an answer.
- **Unknown**, a meta-value used to signal that the actual desired value could not be computed, either because Calcium does not (yet) have a data structure or algorithm for that case, or because doing so would be unreasonably expensive. This occurs, for example, if Calcium performs a division and is unable to decide whether the result is a number, unsigned infinity or undefined (because testing
for zero fails). Wrappers may want to check output variables for Unknown and throw an exception (e.g. NotImplementedError in Python).

The distinction between Calcium numbers (which must represent elements of \( C \)) and the different kinds of nonnumerical values (infinities, Undefined or Unknown) is essential. Nonnumerical values may not be used as field extension numbers \( a_k \), and the denominator of a formal field element must always represent a nonzero complex number. Accordingly, for any given Calcium value \( x \) that is not Unknown, it is exactly known whether \( x \) represents A) a number, B) unsigned infinity, C) a signed infinity, or D) Undefined.

### 10.4.3 Number objects

For all types, a type\_t is defined as an array of length one of type type\_struct, permitting a type\_t to be passed by reference.

```plaintext
type ca_struct

type ca_t
  A ca_t contains an index to a field \( K \), and data representing an element \( x \) of \( K \). The data is either an inline rational number (fmpq\_t), an inline Antic number field element (nf\_elem\_t) when \( K \) is an absolute algebraic number field \( \mathbb{Q}(a) \), or a pointer to a heap-allocated fmpz\_mpoly\_q\_t representing an element of a generic field \( \mathbb{Q}(a_1, \ldots, a_n) \). Special values are encoded using magic bits in the field index.

type ca_ptr
  Alias for ca_struct *, used for vectors of numbers.

type ca_srcptr
  Alias for const ca_struct *, used for vectors of numbers when passed as constant input to functions.
```

### 10.4.4 Context objects

```plaintext
type ca_ctx_struct

type ca_ctx_t
  A ca_ctx_t context object holds a cache of fields \( K \) and constituent extension numbers \( a_k \). The field index in an individual ca_t instance represents a shallow reference to the object defining the field \( K \) within the context object, so creating many elements of the same field is cheap.

  Since context objects are mutable (and may be mutated even when performing read-only operations on ca_t instances), they must not be accessed simultaneously by different threads: in multithreaded environments, the user must use a separate context object for each thread.

void ca_ctx_init(ca_ctx_t ctx)
  Initializes the context object ctx for use. Any evaluation options stored in the context object are set to default values.

void ca_ctx_clear(ca_ctx_t ctx)
  Clears the context object ctx, freeing any memory allocated internally. This function should only be called after all ca_t instances referring to this context have been cleared.

void ca_ctx_print(ca_ctx_t ctx)
  Prints a description of the context ctx to standard output. This will give a complete listing of the cached fields in ctx.
```
10.4.5 Memory management for numbers

```c
void ca_init(ca_t x, ca_ctx_t ctx)
    Initializes the variable x for use, associating it with the context object ctx. The value of x is set to the rational number 0.
void ca_clear(ca_t x, ca_ctx_t ctx)
    Clears the variable x.
void ca_swap(ca_t x, ca_t y, ca_ctx_t ctx)
    Efficiently swaps the variables x and y.
```

10.4.6 Symbolic expressions

```c
void ca_get_fexpr(fexpr_t res, const ca_t x, ulong flags, ca_ctx_t ctx)
    Sets res to a symbolic expression representing x.
int ca_set_fexpr(ca_t res, const fexpr_t expr, ca_ctx_t ctx)
    Sets res to the value represented by the symbolic expression expr. Returns 1 on success and 0 on failure. This function essentially just traverses the expression tree using ca arithmetic; it does not provide advanced symbolic evaluation. It is guaranteed to at least be able to parse the output of ca_get_fexpr() .
```

10.4.7 Printing

The style of printed output is controlled by ctx->options[CA_OPT_PRINT_FLAGS] (see Context options) which can be set to any combination of the following flags:

**CA_PRINT_N**
- Print a decimal approximation of the number. The approximation is guaranteed to be correctly rounded to within one unit in the last place.
- If combined with CA_PRINT_REPR, numbers appearing within the symbolic representation will also be printed with decimal approximations.
- Warning: printing a decimal approximation requires a computation, which can be expensive. It can also mutate cached data (numerical enclosures of extension numbers), affecting subsequent computations.

**CA_PRINT_DIGITS**
- Multiplied by a positive integer, specifies the number of decimal digits to show with CA_PRINT_N. If not given, the default precision is six digits.

**CA_PRINT_REPR**
- Print the symbolic representation of the number (including its recursive elements). If used together with CA_PRINT_N, field elements will print as decimal {symbolic} while extension numbers will print as decimal [symbolic].
- All extension numbers appearing in the field defining x and in the inner constructions of those extension numbers will be given local labels a, b, etc. for this printing.

**CA_PRINT_FIELD**
- For each field element, explicitly print its formal field along with its reduction ideal if present, e.g. QQ or QQ(a,b,c) / <a-b, c^2+1>.

**CA_PRINT_DEFAULT**
- The default print style. Equivalent to CA_PRINT_N | CA_PRINT_REPR.
CA_PRINT_DEBUG

Verbose print style for debugging. Equivalent to CA_PRINT_N | CA_PRINT_REPR | CA_PRINT_FIELD.

As a special case, small integers are always printed as simple literals.

As illustration, here are the numbers \(-7, \frac{2}{3}, (\sqrt{3} + 5)/2\) and \(\sqrt{2}(\log(\pi) + \pi i)\) printed in various styles:

```
# CA_PRINT_DEFAULT
-7
0.666667 \{2/3\}
3.36603 \{(a+5)/2 \text{ where } a = 1.73205 \text{ [}a^2-3=0\text{]}\}
1.61889 + 4.44288*\text{i} \{a+c+b+c+d \text{ where } a = 1.14473 \text{ [}\log(3.14159 \{b\})\text{]}, b = 3.14159 \text{ [}\pi\text{]},}
             c = 1.41421 \text{ [}c^2-2=0\text{]}, d = \text{i} \text{ [}d^2+1=0\text{]}\}

# CA_PRINT_N
-7
0.666667
3.36603
1.61889 + 4.44288*\text{i}

# CA_PRINT_N | (CA_PRINT_DIGITS * 20)
-7
0.66666666666666666667
3.3660254037844386468
1.618892529822026685 + 4.4428829381583662470*\text{i}

# CA_PRINT_REPR
-7
2/3
\{(a+5)/2 \text{ where } a = [a^2-3=0]\}
\{a+c+b+c+d \text{ where } a = \log(3.14159 \{b\}), b = 3.14159 \text{ [}\pi\text{]},}
             \cdots\}

# CA_PRINT_DEBUG
-7
0.666667 \{2/3 \text{ in \QQ}\}
3.36603 \{(a+5)/2 \text{ in \QQ(a)/<a^2-3> \text{ where } a = 1.73205 \text{ [}a^2-3=0\text{]}\}
1.61889 + 4.44288*\text{i} \{a+c+b+c+d \text{ in \QQ(a,b,c,d)/<c^2-2, d^2+1> \text{ where } a = 1.14473,}
             \cdots\}\}
```

void ca_print(const ca_t x, ca_ctx_t ctx)
Prints x to standard output.

void ca_fprint(FILE *fp, const ca_t x, ca_ctx_t ctx)
Prints x to the file fp.

char *ca_get_str(const ca_t x, ca_ctx_t ctx)
Prints x to a string which is returned. The user should free this string by calling flint_free.

void ca_printn(const ca_t x, slong n, ca_ctx_t ctx)
Prints an n-digit numerical representation of x to standard output.
10.4.8 Special values

```c
void ca_zero(ca_t res, ca_ctx_t ctx) {
    Sets res to the integer 0, 1 or -1. This creates a canonical representation of this number as an element of the trivial field \( \mathbb{Q} \).
}

void ca_one(ca_t res, ca_ctx_t ctx) {
}

void ca_neg_one(ca_t res, ca_ctx_t ctx) {
    Sets res to the imaginary unit \( i = \sqrt{-1}, \) or its negation \(-i\). This creates a canonical representation of \( i \) as the generator of the algebraic number field \( \mathbb{Q}(i) \).
}

void ca_i(ca_t res, ca_ctx_t ctx) {
}

void ca_neg_i(ca_t res, ca_ctx_t ctx) {
    Sets res to the constant \( \pi \). This creates an element of the transcendental number field \( \mathbb{Q}(\pi) \).
}

void ca_pi(ca_t res, ca_ctx_t ctx) {
}

void ca_pi_i(ca_t res, ca_ctx_t ctx) {
    Sets res to the meta-value \texttt{Unknown}.
}

void ca_undefined(ca_t res, ca_ctx_t ctx) {
    Sets res to \texttt{Undefined}.
}

void ca_uinf(ca_t res, ca_ctx_t ctx) {
    Sets res to unsigned infinity \( \tilde{\infty} \).
}

void ca_pos_inf(ca_t res, ca_ctx_t ctx) {
}

void ca_neg_inf(ca_t res, ca_ctx_t ctx) {
}

void ca_pos_i_inf(ca_t res, ca_ctx_t ctx) {
}

void ca_neg_i_inf(ca_t res, ca_ctx_t ctx) {
    Sets res to the signed infinity \( +\infty, -\infty, +i\infty \) or \(-i\infty\).
}
```

10.4.9 Assignment and conversion

```c
void ca_set(ca_t res, const ca_t x, ca_ctx_t ctx) {
    Sets res to a copy of x.
}

void ca_set_si(ca_t res, slong v, ca_ctx_t ctx) {
}

void ca_set_ui(ca_t res, ulong v, ca_ctx_t ctx) {
}

void ca_set_fmpz(ca_t res, const fmpz_t v, ca_ctx_t ctx) {
    Sets res to the integer or rational number \( v \). This creates a canonical representation of this number as an element of the trivial field \( \mathbb{Q} \).
}

void ca_set_d(ca_t res, double x, ca_ctx_t ctx) {
}

void ca_set_d_d(ca_t res, double x, double y, ca_ctx_t ctx) {
    Sets res to the value of \( x \), or the complex value \( x + yi \). NaN is interpreted as \texttt{Unknown} (not \texttt{Undefined}).
```
void ca_transfer(ca_t res, ca_ctx_t res_ctx, const ca_t src, ca_ctx_t src_ctx)
    Sets res to src where the corresponding context objects res_ctx and src_ctx may be different.
    This operation preserves the mathematical value represented by src, but may result in a different internal representation depending on the settings of the context objects.

10.4.10 Conversion of algebraic numbers

void ca_set_qqbar(ca_t res, const qqbar_t x, ca_ctx_t ctx)
    Sets res to the algebraic number x.
    If x is rational, res is set to the canonical representation as an element in the trivial field \( \mathbb{Q} \).
    If x is irrational, this function always sets res to an element of a univariate number field \( \mathbb{Q}(a) \). It will not, for example, identify \( \sqrt{2} + \sqrt{3} \) as an element of \( \mathbb{Q}(\sqrt{2}, \sqrt{3}) \). However, it may attempt to find a simpler number field than that generated by x itself. For example:
    - If x is quadratic, it will be expressed as an element of \( \mathbb{Q}(\sqrt{N}) \) where N has no small repeated factors (obtained by performing a smooth factorization of the discriminant).
    - TODO: if possible, coerce x to a low-degree cyclotomic field.

int ca_get_fmpz(fmpz_t res, const ca_t x, ca_ctx_t ctx)
int ca_get_fmpq(fmpq_t res, const ca_t x, ca_ctx_t ctx)
int ca_get_qqbar(qqbar_t res, const ca_t x, ca_ctx_t ctx)
    Attempts to evaluate x to an explicit integer, rational or algebraic number. If successful, sets res to this number and returns 1. If unsuccessful, returns 0.
    The conversion certainly fails if x does not represent an integer, rational or algebraic number (respectively), but can also fail if x is too expensive to compute under the current evaluation limits. In particular, the evaluation will be aborted if an intermediate algebraic number (or more precisely, the resultant polynomial prior to factorization) exceeds \texttt{CA\_OPT\_QQBAR\_DEG\_LIMIT} or the coefficients exceed some multiple of \texttt{CA\_OPT\_PREC\_LIMIT}. Note that evaluation may hit those limits even if the minimal polynomial for x itself is small. The conversion can also fail if no algorithm has been implemented for the functions appearing in the construction of x.

int ca_can_evaluate_qqbar(const ca_t x, ca_ctx_t ctx)
    Checks if \texttt{ca\_get\_qqbar()} has a chance to succeed. In effect, this checks if all extension numbers are manifestly algebraic numbers (without doing any evaluation).

10.4.11 Random generation

void ca_randtest_rational(ca_t res, flint_rand_t state, slong bits, ca_ctx_t ctx)
    Sets res to a random rational number with numerator and denominator up to bits bits in size.

void ca_randtest(ca_t res, flint_rand_t state, slong depth, slong bits, ca_ctx_t ctx)
    Sets res to a random number generated by evaluating a random expression. The algorithm randomly selects between generating a “simple” number (a random rational number or quadratic field element with coefficients up to bits in size, or a random builtin constant), or if depth is nonzero, applying a random arithmetic operation or function to operands produced through recursive calls with depth - 1. The output is guaranteed to be a number, not a special value.

void ca_randtest_special(ca_t res, flint_rand_t state, slong depth, slong bits, ca_ctx_t ctx)
    Randomly generates either a special value or a number.

void ca_randtest_same_nf(ca_t res, flint_rand_t state, const ca_t x, slong depth, slong bits, ca_ctx_t ctx)
    Sets res to a random element in the same number field as x, with numerator coefficients up to bits in size and denominator up to den_bits in size. This function requires that x is an element of an absolute number field.
10.4.12 Representation properties

The following functions deal with the representation of a `ca_t` and hence can always be decided quickly and unambiguously. The return value for predicates is 0 for false and 1 for true.

```c
int ca_equal_repr(const ca_t x, const ca_t y, ca_ctx_t ctx)
Returns whether x and y have identical representation. For field elements, this checks if x and y belong to the same formal field (with generators having identical representation) and are represented by the same rational function within that field.

For special values, this tests equality of the special values, with Unknown handled as if it were a value rather than a meta-value: that is, Unknown = Unknown gives 1, and Unknown = y gives 0 for any other kind of value y. If neither x nor y is Unknown, then representation equality implies that x and y describe to the same mathematical value, but if either operand is Unknown, the result is meaningless for mathematical comparison.

int ca_cmp_repr(const ca_t x, const ca_t y, ca_ctx_t ctx)
Compares the representations of x and y in a canonical sort order, returning -1, 0 or 1. This only performs a lexicographic comparison of the representations of x and y; the return value does not say anything meaningful about the numbers represented by x and y.

ulong ca_hash_repr(const ca_t x, ca_ctx_t ctx)
Hashes the representation of x.

int ca_is_unknown(const ca_t x, ca_ctx_t ctx)
Returns whether x is Unknown.

int ca_is_special(const ca_t x, ca_ctx_t ctx)
Returns whether x is a special value or metavalue (not a field element).

int ca_is_qq_elem(const ca_t x, ca_ctx_t ctx)
Returns whether x is represented as an element of the rational field Q.

int ca_is_qq_elem_zero(const ca_t x, ca_ctx_t ctx)
int ca_is_qq_elem_one(const ca_t x, ca_ctx_t ctx)
int ca_is_qq_elem_integer(const ca_t x, ca_ctx_t ctx)
Returns whether x is represented as the element 0, 1 or any integer in the rational field Q.

int ca_is_nf_elem(const ca_t x, ca_ctx_t ctx)
Returns whether x is represented as an element of a univariate algebraic number field Q(a).

int ca_is_cyclotomic_nf_elem(slong *p, ulong *q, const ca_t x, ca_ctx_t ctx)
Returns whether x is represented as an element of a univariate cyclotomic field, i.e. Q(a) where a is a root of unity. If p and q are not NULL and x is represented as an element of a cyclotomic field, this also sets p and q to the minimal integers with 0 ≤ p < q such that the generating root of unity is a = e^{2\pi i/p/q}. Note that the answer 0 does not prove that x is not a cyclotomic number, and the order q is also not necessarily the generator of the smallest cyclotomic field containing x. For the purposes of this function, only nontrivial cyclotomic fields count; the return value is 0 if x is represented as a rational number.

int ca_is_generic_elem(const ca_t x, ca_ctx_t ctx)
Returns whether x is represented as a generic field element; i.e. it is not a special value, not represented as an element of the rational field, and not represented as an element of a univariate algebraic number field.
10.4.13 Value predicates

The following predicates check a mathematical property which might not be effectively decidable. The result is a truth_t to allow representing an unknown outcome.

truth_t ca_check_is_number(const ca_t x, ca_ctx_t ctx)
Tests if x is a number. The result is T_TRUE if x is a field element (and hence a complex number), T_FALSE if x is an infinity or Undefined, and T_UNKNOWN if x is Unknown.

truth_t ca_check_is_zero(const ca_t x, ca_ctx_t ctx)

truth_t ca_check_is_one(const ca_t x, ca_ctx_t ctx)

truth_t ca_check_is_neg_one(const ca_t x, ca_ctx_t ctx)

truth_t ca_check_is_i(const ca_t x, ca_ctx_t ctx)

truth_t ca_check_is_neg_i(const ca_t x, ca_ctx_t ctx)
Tests if x is equal to the number 0, 1, −1, i, or −i.

truth_t ca_check_is_algebraic(const ca_t x, ca_ctx_t ctx)

truth_t ca_check_is_rational(const ca_t x, ca_ctx_t ctx)

truth_t ca_check_is_integer(const ca_t x, ca_ctx_t ctx)
Tests if x is respectively an algebraic number, a rational number, or an integer.

truth_t ca_check_is_real(const ca_t x, ca_ctx_t ctx)
Tests if x is a real number. Warning: this returns T_FALSE if x is an infinity with real sign.

truth_t ca_check_is_negative_real(const ca_t x, ca_ctx_t ctx)
Tests if x is a negative real number. Warning: this returns T_FALSE if x is negative infinity.

truth_t ca_check_is_imaginary(const ca_t x, ca_ctx_t ctx)
Tests if x is an imaginary number. Warning: this returns T_FALSE if x is an infinity with imaginary sign.

truth_t ca_check_is_undefined(const ca_t x, ca_ctx_t ctx)
Tests if x is the special value Undefined.

truth_t ca_check_is_infinity(const ca_t x, ca_ctx_t ctx)
Tests if x is any infinity (unsigned or signed).

truth_t ca_check_is_uinf(const ca_t x, ca_ctx_t ctx)
Tests if x is unsigned infinity ∞.

truth_t ca_check_is_signed_inf(const ca_t x, ca_ctx_t ctx)
Tests if x is any signed infinity.

truth_t ca_check_is_pos_inf(const ca_t x, ca_ctx_t ctx)

truth_t ca_check_is_neg_inf(const ca_t x, ca_ctx_t ctx)

truth_t ca_check_is_pos_i_inf(const ca_t x, ca_ctx_t ctx)

truth_t ca_check_is_neg_i_inf(const ca_t x, ca_ctx_t ctx)
Tests if x is equal to the signed infinity +∞, −∞, +i∞, −i∞, respectively.
10.4.14 Comparisons

truth_t ca_check_equal(const ca_t x, const ca_t y, ca_ctx_t ctx)

Tests \( x = y \) as a mathematical equality. The result is \( \text{T\_UNKNOWN} \) if either operand is \( \text{Unknown} \). The result may also be \( \text{T\_UNKNOWN} \) if \( x \) and \( y \) are numerically indistinguishable and cannot be proved equal or unequal by an exact computation.

truth_t ca_check_lt(const ca_t x, const ca_t y, ca_ctx_t ctx)

truth_t ca_check_le(const ca_t x, const ca_t y, ca_ctx_t ctx)

truth_t ca_check_gt(const ca_t x, const ca_t y, ca_ctx_t ctx)

truth_t ca_check_ge(const ca_t x, const ca_t y, ca_ctx_t ctx)

 Compares \( x \) and \( y \), implementing the respective operations \( x < y \), \( x \leq y \), \( x > y \), \( x \geq y \). Only real numbers and \(-\infty\) and \(+\infty\) are considered comparable. The result is \( \text{T\_FALSE} \) (not \( \text{T\_UNKNOWN} \)) if either operand is not comparable (being a nonreal complex number, unsigned infinity, or undefined).

10.4.15 Field structure operations

void ca_merge_fields(ca_t resx, ca_t resy, const ca_t x, const ca_t y, ca_ctx_t ctx)

Sets \( \text{res}x \) and \( \text{res}y \) to copies of \( x \) and \( y \) coerced to a common field. Both \( x \) and \( y \) must be field elements (not special values).

In the present implementation, this simply merges the lists of generators, avoiding duplication. In the future, it will be able to eliminate generators satisfying algebraic relations.

void ca_condense_field(ca_t res, ca_ctx_t ctx)

Attempts to denote the value of \( \text{res} \) to a trivial subfield of its current field by removing unused generators. In particular, this denotes any obviously rational value to the trivial field \( \mathbb{Q} \).

This function is applied automatically in most operations (arithmetic operations, etc.).

cast_ptr ca_is_gen_as_ext(const ca_t x, ca_ctx_t ctx)

If \( x \) is a generator of its formal field, \( x = a_k \in \mathbb{Q}(a_1, \ldots, a_n) \), returns a pointer to the extension number defining \( a_k \). If \( x \) is not a generator, returns \( \text{NULL} \).

10.4.16 Arithmetic

void ca_neg(ca_t res, const ca_t x, ca_ctx_t ctx)

Sets \( \text{res} \) to the negation of \( x \). For numbers, this operation amounts to a direct negation within the formal field. For a signed infinity \( c\infty \), negation gives \((-c)\infty\); all other special values are unchanged.

void ca_add_fmpz(ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)

void ca_add_fmpq(ca_t res, const ca_t x, const fmpq_t y, ca_ctx_t ctx)

void ca_add_ui(ca_t res, const ca_t x, ulong y, ca_ctx_t ctx)

void ca_add_si(ca_t res, const ca_t x, slong y, ca_ctx_t ctx)

void ca_add(ca_t res, const ca_t x, const ca_t y, ca_ctx_t ctx)

Sets \( \text{res} \) to the sum of \( x \) and \( y \). For special values, the following rules apply (\( c\infty \) denotes a signed infinity, \( |c| = 1 \)):

- \( c\infty + d\infty = c\infty \) if \( c = d \)
- \( c\infty + d\infty = \text{Undefined} \) if \( c \neq d \)
- \( \infty + c\infty = \infty + \infty = \text{Undefined} \)
- \( c\infty + z = c\infty \) if \( z \in \mathbb{C} \)
- \( \infty + z = \infty \) if \( z \in \mathbb{C} \)
- \( z + \text{Undefined} = \text{Undefined} \) for any value \( z \) (including \( \text{Unknown} \))
In any other case involving special values, or if the specific case cannot be distinguished, the result is Unknown.

```c
void ca_sub_fmpz(const ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)
void ca_sub_fmpq(const ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)
void ca_sub_ui(const ca_t res, const ca_t x, ulong y, ca_ctx_t ctx)
void ca_sub_si(const ca_t res, const ca_t x, slong y, ca_ctx_t ctx)
void ca_sub_fmpz(const ca_t res, const fmpz_t x, const ca_t y, ca_ctx_t ctx)
void ca_sub_fmpq(const ca_t res, const fmpz_t x, const ca_t y, ca_ctx_t ctx)
void caui_sub(const ca_t res, ulong x, const ca_t y, ca_ctx_t ctx)
void casi_sub(const ca_t res, slong x, const ca_t y, ca_ctx_t ctx)
void ca_sub(const ca_t res, const ca_t x, const ca_t y, ca_ctx_t ctx)
```

Sets res to the difference of x and y. This is equivalent to computing \(x + (-y)\).

```c
void ca_mul_fmpz(const ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)
void ca_mul_fmpq(const ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)
void ca_mul_ui(const ca_t res, const ca_t x, ulong y, ca_ctx_t ctx)
void ca_mul_si(const ca_t res, const ca_t x, slong y, ca_ctx_t ctx)
void ca_mul(const ca_t res, const ca_t x, const ca_t y, ca_ctx_t ctx)
```

Sets res to the product of x and y. For special values, the following rules apply (\(c\infty\) denotes a signed infinity, \(|c| = 1\):

- \(c\infty \cdot d\infty = ad\infty\)
- \(c\infty \cdot \infty = \infty\)
- \(\infty \cdot \infty = \infty\)
- \(c\infty \cdot z = \text{sgn}(z)c\infty\) if \(z \in \mathbb{C} \setminus \{0\}\)
- \(c\infty \cdot 0 = \text{Undefined}\)
- \(\infty \cdot 0 = \text{Undefined}\)
- \(z \cdot \text{Undefined} = \text{Undefined}\) for any value \(z\) (including Unknown)

In any other case involving special values, or if the specific case cannot be distinguished, the result is Unknown.

```c
void ca_inv(const ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets res to the multiplicative inverse of x. In a univariate algebraic number field, this always produces a rational denominator, but the denominator might not be rationalized in a multivariate field. For special values and zero, the following rules apply:

- \(1/(c\infty) = 1/\infty = 0\)
- \(1/\infty = \infty\)
- \(1/\text{Undefined} = \text{Undefined}\)
- \(1/\text{Unknown} = \text{Unknown}\)

If it cannot be determined whether x is zero or nonzero, the result is Unknown.
```c
void ca_div(ca_t res, const ca_t x, const ca_t y, ca_ctx_t ctx)
```

Sets \( \text{res} \) to the quotient of \( x \) and \( y \). This is equivalent to computing \( x \cdot (1/y) \). For special values and division by zero, this implies the following rules (\( \infty \) denotes a signed infinity, \( |c| = 1 \)):

- \((c\infty)/(d\infty) = (c\infty)/\infty = \infty/(c\infty) = \infty/\infty = \text{Undefined}\)
- \(c\infty/z = (c/\text{sgn}(z))\cdot \infty \) if \( z \in \mathbb{C} \setminus \{0\}\)
- \(c\infty/0 = \infty/0 = \infty\)
- \(z/(c\infty) = z/\infty = 0 \) if \( z \in \mathbb{C} \)
- \(z/0 = \infty \) if \( z \in \mathbb{C} \setminus \{0\} \)
- \(0/0 = \text{Undefined}\)
- \(z/\text{Undefined} = \text{Undefined} \) for any value \( z \) (including \( \text{Unknown} \))
- \(\text{Undefined}/z = \text{Undefined} \) for any value \( z \) (including \( \text{Unknown} \))

In any other case involving special values, or if the specific case cannot be distinguished, the result is \( \text{Unknown} \).

```c
void ca_dot(ca_t res, const ca_t initial, int subtract, ca_srcptr x, slong xstep, ca_srcptr y, slong ystep, slong len, ca_ctx_t ctx)
```

Computes the dot product of the vectors \( x \) and \( y \), setting \( \text{res} \) to \( s + (-1)^{\text{subtract}} \sum_{i=0}^{\text{len}-1} x_i y_i \).

The initial term \( s \) is optional and can be omitted by passing \( NULL \) (equivalently, \( s = 0 \)). The parameter \( \text{subtract} \) must be 0 or 1. The length \( \text{len} \) is allowed to be negative, which is equivalent to a length of zero. The parameters \( \text{xstep} \) or \( \text{ystep} \) specify a step length for traversing subsequences of the vectors \( x \) and \( y \); either can be negative to step in the reverse direction starting from the initial pointer. Aliasing is allowed between \( \text{res} \) and \( s \) but not between \( \text{res} \) and the entries of \( x \) and \( y \).

```c
void ca_fmpz_poly_evaluate(ca_t res, const fmpz_poly_t poly, const ca_t x, ca_ctx_t ctx)
```

Sets \( \text{res} \) to the polynomial \( poly \) evaluated at \( x \).

```c
void ca_fmpz_mpoly_evaluate(ca_t res, const fmpz_mpoly_t f, ca_srcptr x, const fmpz_mpoly_ctx_t mctx, ca_ctx_t ctx)
```

Sets \( \text{res} \) to the multivariate polynomial \( f \) evaluated at the vector of arguments \( x \).

```c
void ca_fmpz_mpoly_q_evaluate(ca_t res, const fmpz_mpoly_q_t f, ca_srcptr x, const fmpz_mpoly_ctx_t mctx, ca_ctx_t ctx)
```

Sets \( \text{res} \) to the multivariate rational function \( f \) evaluated at the vector of arguments \( x \).

These functions behave like the normal arithmetic functions, but assume (and do not check) that division by zero cannot occur. Division by zero will result in undefined behavior.
10.4.17 Powers and roots

void ca_sqr(ca_t res, const ca_t x, ca_ctx_t ctx)
Sets res to the square of x.

void ca_pow_fmpq(ca_t res, const ca_t x, const fmpq_t y, ca_ctx_t ctx)
void ca_pow_fmpz(ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)
void ca_pow_ui(ca_t res, const ca_t x, ulong y, ca_ctx_t ctx)
void ca_pow_si(ca_t res, const ca_t x, slong y, ca_ctx_t ctx)

void ca_pow(ca_t res, const ca_t x, const ca_t y, ca_ctx_t ctx)
Sets res to x raised to the power y. Handling of special values is not yet implemented.

void ca_pow_si_arithmetic(ca_t res, const ca_t x, slong n, ca_ctx_t ctx)
Sets res to x raised to the power n. Whereas ca_pow(), ca_pow_si() etc. may create \(x^n\) as an extension number if \(n\) is large, this function always perform the exponentiation using field arithmetic.

void ca_sqrt_inert(ca_t res, const ca_t x, ca_ctx_t ctx)
void ca_sqrt_nofactor(ca_t res, const ca_t x, ulong flags, ca_ctx_t ctx)
void ca_sqrt(ca_t res, const ca_t x, ca_ctx_t ctx)
Sets res to the principal square root of x.

For special values, the following definitions apply:
- \(\sqrt{\infty} = \sqrt{\tilde{\infty}} = +\infty\)
- Both Undefined and Unknown map to themselves.

The inert version outputs the generator in the formal field \(\mathbb{Q}(\sqrt{x})\) without simplifying.

The factor version writes x = \(A^2B\) in \(K\) where \(K\) is the field of \(x\), and outputs \(A\sqrt{B}\) or \(-A\sqrt{B}\) (whichever gives the correct sign) as an element of \(K(\sqrt{B})\) or some subfield thereof. This factorization is only a heuristic and is not guaranteed to make \(B\) minimal. Factorization options can be passed through to \(flags\): see ca_factor() for details.

The nofactor version will not perform a general factorization, but may still perform other simplifications. It may in particular attempt to simplify \(\sqrt{x}\) to a single element in \(\mathbb{Q}\).

void ca_sqrt_ui(ca_t res, ulong n, ca_ctx_t ctx)
Sets res to the principal square root of \(n\).

10.4.18 Complex parts

void ca_abs(ca_t res, const ca_t x, ca_ctx_t ctx)
Sets res to the absolute value of x.

For special values, the following definitions apply:
- \(|\infty| = |\tilde{\infty}| = +\infty\).
- Both Undefined and Unknown map to themselves.

This function will attempt to simplify its argument through an exact computation. It may in particular attempt to simplify \(|x|\) to a single element in \(\mathbb{Q}\).

In the generic case, this function outputs an element of the formal field \(\mathbb{Q}(|x|)\).
void \texttt{ca\_sgn}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

Sets \texttt{res} to the sign of \texttt{x}, defined by

\[
\text{sgn}(x) = \begin{cases} 
0 & x = 0 \\
\frac{x}{|x|} & x \neq 0
\end{cases}
\]

for numbers. For special values, the following definitions apply:

- \text{sgn}(\pm\infty) = \text{sign}(c).
- \text{sgn}(\tilde{\infty}) = \text{Undefined}.
- Both \text{Undefined} and \text{Unknown} map to themselves.

This function will attempt to simplify its argument through an exact computation. It may in particular attempt to simplify \text{sgn}(x) to a single element in \( \mathbb{Q} \).

In the generic case, this function outputs an element of the formal field \( \mathbb{Q}(\text{sgn}(x)) \).

void \texttt{ca\_csgn}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

Sets \texttt{res} to the extension of the real sign function taking the value 1 for \( z \) strictly in the right half plane, -1 for \( z \) strictly in the left half plane, and the sign of the imaginary part when \( z \) is on the imaginary axis. Equivalently, \text{csgn}(z) = z/\sqrt{z^2} \) except that the value is 0 when \( z \) is exactly zero. This function gives \text{Undefined} for unsigned infinity and \text{csgn}(\text{sgn}(\infty)) = \text{csgn}(c) for signed infinities.

void \texttt{ca\_arg}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

Sets \texttt{res} to the complex argument (phase) of \texttt{x}, normalized to the range \((-\pi, +\pi]\). The argument of 0 is defined as 0. For special values, the following definitions apply:

- \text{arg}(\pm\infty) = \text{arg}(c).
- \text{arg}(\tilde{\infty}) = \text{Undefined}.
- Both \text{Undefined} and \text{Unknown} map to themselves.

void \texttt{ca\_re}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

Sets \texttt{res} to the real part of \texttt{x}. The result is \text{Undefined} if \texttt{x} is any infinity (including a real infinity).

void \texttt{ca\_im}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

Sets \texttt{res} to the imaginary part of \texttt{x}. The result is \text{Undefined} if \texttt{x} is any infinity (including an imaginary infinity).

void \texttt{ca\_conj\_deep}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

void \texttt{ca\_conj\_shallow}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

void \texttt{ca\_conj}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

Sets \texttt{res} to the complex conjugate of \texttt{x}. The \text{shallow} version creates a new extension element \( \overline{x} \) unless \texttt{x} can be trivially conjugated in-place in the existing field. The \text{deep} version recursively conjugates the extension numbers in the field of \texttt{x}.

void \texttt{ca\_floor}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

Sets \texttt{res} to the floor function of \texttt{x}. The result is \text{Undefined} if \texttt{x} is any infinity (including a real infinity). For complex numbers, this is presently defined to take the floor of the real part.

void \texttt{ca\_ceil}(\texttt{ca\_t} \texttt{res}, \texttt{const ca\_t} \texttt{x}, \texttt{ca\_ctx\_t} \texttt{ctx})

Sets \texttt{res} to the ceiling function of \texttt{x}. The result is \text{Undefined} if \texttt{x} is any infinity (including a real infinity). For complex numbers, this is presently defined to take the ceiling of the real part.
### 10.4.19 Exponentials and logarithms

```c
void ca_exp (ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets `res` to the exponential function of `x`.

For special values, the following definitions apply:

- $e^{+\infty} = +\infty$
- $e^{-\infty} = \infty$ if $0 < \text{Re}(c) < 1$.
- $e^{0} = 0$ if $\text{Re}(c) < 0$.
- $e^{1} = 1$ if $\text{Re}(c) = 0$.
- $e^{\infty} = \text{Undefined}$ if $\text{Re}(c) = 0$.
- Both `Undefined` and `Unknown` map to themselves.

The following symbolic simplifications are performed automatically:

- $e^{0} = 1$
- $e^{\log(z)} = z$
- $e^{(p/q)\log(z)} = z^{p/q}$ (for rational $p/q$)
- $e^{(p/q)\pi i} = \text{algebraic root of unity}$ (for small rational $p/q$)

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(e^x)$.

```c
void ca_log (ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets `res` to the natural logarithm of `x`.

For special values and at the origin, the following definitions apply:

- For any infinity, $\log(c^{\infty}) = \log(\tilde{\infty}) = +\infty$.
- $\log(0) = -\infty$. The result is `Unknown` if deciding $x = 0$ fails.
- Both `Undefined` and `Unknown` map to themselves.

The following symbolic simplifications are performed automatically:

- $\log(1) = 0$
- $\log(e^{z}) = z + 2\pi ik$
- $\log(\sqrt{z}) = \frac{1}{2} \log(z) + 2\pi ik$
- $\log(z^{a}) = a \log(z) + 2\pi ik$
- $\log(x) = \log(-x) + \pi i$ for negative real $x$

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(\log(x))$.

### 10.4.20 Trigonometric functions

```c
void ca_sin_cos_exponential (ca_t res1, ca_t res2, const ca_t x, ca_ctx_t ctx)
void ca_sin_cos_direct (ca_t res1, ca_t res2, const ca_t x, ca_ctx_t ctx)
void ca_sin_cos_tangent (ca_t res1, ca_t res2, const ca_t x, ca_ctx_t ctx)
void ca_sin_cos (ca_t res1, ca_t res2, const ca_t x, ca_ctx_t ctx)
```

Sets `res1` to the sine of `x` and `res2` to the cosine of `x`. Either `res1` or `res2` can be `NULL` to compute only the other function. Various representations are implemented:

- The `exponential` version expresses the sine and cosine in terms of complex exponentials. Simple algebraic values will simplify to rational numbers or elements of cyclotomic fields.
The *direct* method expresses the sine and cosine in terms of the original functions (perhaps after applying some symmetry transformations, which may interchange sin and cos). Extremely simple algebraic values will automatically simplify to elements of real algebraic number fields.

The *tangent* version expresses the sine and cosine in terms of \( \tan(x/2) \), perhaps after applying some symmetry transformations. Extremely simple algebraic values will automatically simplify to elements of real algebraic number fields.

By default, the standard function uses the *exponential* representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the `CA_OPT_TRIG_FORM` context setting.

For special values, the following definitions apply:

- \( \sin(\pm i\infty) = \pm i\infty \)
- \( \cos(\pm i\infty) = +\infty \)
- All other infinities give Undefined

The standard function uses the exponential representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the `CA_OPT_TRIG_FORM` context setting.

For special values, the following definitions apply:

- At poles, \( \tan((n + \frac{1}{2})\pi) = \tilde{\infty} \)
- \( \tan(e^{i\theta}\infty) = +i, \quad 0 < \theta < \pi \)
- \( \tan(e^{i\theta}\infty) = -i, \quad -\pi < \theta < 0 \)
- \( \tan(\pm\infty) = \tan(\tilde{\infty}) = \text{Undefined} \)

Sets \( \text{res} \) to the cotangent of \( x \). This is equivalent to computing the reciprocal of the tangent.

For special values, the following definitions apply:

- \( \text{atan}(e^{i\theta}\infty) = +i, \quad 0 < \theta < \pi \)
- \( \tan(\pm\infty) = \tan(\tilde{\infty}) = \text{Undefined} \)

Sets \( \text{res} \) to the inverse tangent of \( x \). The direct version expresses the result as an inverse tangent (possibly after transforming the variable). The logarithm version expresses it in terms of complex logarithms. Simple algebraic inputs will automatically simplify to rational multiples of \( \pi \).

By default, the standard function uses the logarithm representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the `CA_OPT_TRIG_FORM` context setting.
changed using the \texttt{CA\_OPT\_TRIG\_FORM} context setting (exponential mode results in logarithmic forms).

For special values, the following definitions apply:

- \( \arctan(\pm i) = \pm i \infty \)
- \( \arctan(c\infty) = \text{csign}(c)\pi/2 \)
- \( \arctan(\bar{\infty}) = \text{Undefined} \)

void \texttt{ca\_asin\_logarithm}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)
void \texttt{ca\_acos\_logarithm}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)
void \texttt{ca\_asin\_direct}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)
void \texttt{ca\_acos\_direct}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)
void \texttt{ca\_asin}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)
void \texttt{ca\_acos}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)

Sets \( \text{res} \) to the inverse sine (respectively, cosine) of \( x \).

The \textit{direct} version expresses the result as an inverse sine or cosine (possibly after transforming the variable). The \textit{logarithm} version expresses it in terms of complex logarithms. Simple algebraic inputs will automatically simplify to rational multiples of \( \pi \).

By default, the standard function uses the \textit{logarithm} representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the \texttt{CA\_OPT\_TRIG\_FORM} context setting (exponential mode results in logarithmic forms).

The inverse cosine is presently implemented as \( \arccos(x) = \pi/2 - \arcsin(x) \).

### 10.4.21 Special functions

void \texttt{ca\_gamma}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)

Sets \( \text{res} \) to the gamma function of \( x \).

void \texttt{ca\_erf}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)

Sets \( \text{res} \) to the error function of \( x \).

void \texttt{ca\_erfc}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)

Sets \( \text{res} \) to the complementary error function of \( x \).

void \texttt{ca\_erfi}(ca\_t res, const ca\_t x, ca\_ctx\_t ctx)

Sets \( \text{res} \) to the imaginary error function of \( x \).

### 10.4.22 Numerical evaluation

void \texttt{ca\_get\_acb\_raw}(acb\_t res, const ca\_t x, slong prec, ca\_ctx\_t ctx)

Sets \( \text{res} \) to an enclosure of the numerical value of \( x \). A working precision of \( \text{prec} \) bits is used internally for the evaluation, without adaptive refinement. If \( x \) is any special value, \( \text{res} \) is set to \texttt{acb\_indeterminate}.

void \texttt{ca\_get\_acb}(acb\_t res, const ca\_t x, slong prec, ca\_ctx\_t ctx)

void \texttt{ca\_get\_acb\_accurate\_parts}(acb\_t res, const ca\_t x, slong prec, ca\_ctx\_t ctx)

Sets \( \text{res} \) to an enclosure of the numerical value of \( x \). The working precision is increased adaptively to try to ensure \( \text{prec} \) accurate bits in the output. The \texttt{accurate\_parts} version tries to ensure \( \text{prec} \) accurate bits for both the real and imaginary part separately.
The refinement is stopped if the working precision exceeds CA_OPT_PREC_LIMIT (or twice the initial precision, if this is larger). The user may call acb_rel_accuracy_bits to check if the calculation was successful.

The output is not rounded down to prec bits (to avoid unnecessary double rounding); the user may call acb_set_round when rounding is desired.

The refinement is stopped if the working precision exceeds CA_OPT_PREC_LIMIT (or twice the initial precision, if this is larger). The user may call acb_rel_accuracy_bits to check if the calculation was successful.

The output is not rounded down to prec bits (to avoid unnecessary double rounding); the user may call acb_set_round when rounding is desired.

char *ca_get_decimal_str(const ca_t x, slong digits, ulong flags, ca_ctx_t ctx)
Returns a decimal approximation of x with precision up to digits. The output is guaranteed to be correct within 1 ulp in the returned digits, but the number of returned digits may be smaller than digits if the numerical evaluation does not succeed.

If flags is set to 1, attempts to achieve full accuracy for both the real and imaginary parts separately.

if x is not finite or a finite enclosure cannot be produced, returns the string “?”.

The user should free the returned string with flint_free.

10.4.23 Rewriting and simplification

void ca_rewrite_complex_normal_form(ca_t res, const ca_t x, int deep, ca_ctx_t ctx)
Sets res to x rewritten using standardizing transformations over the complex numbers:

- Elementary functions are rewritten in terms of (complex) exponentials, roots and logarithms
- Complex parts are rewritten using logarithms, square roots, and (deep) complex conjugates
- Algebraic numbers are rewritten in terms of cyclotomic fields where applicable

If deep is set, the rewriting is applied recursively to the tower of extension numbers; otherwise, the rewriting is only applied to the top-level extension numbers.

The result is not a normal form in the strong sense (the same number can have many possible representations even after applying this transformation), but in practice this is a powerful heuristic for simplification.

10.4.24 Factorization

type ca_factor_struct

type ca_factor_t
Representes a real or complex number in factored form \( b_1^{e_1} b_2^{e_2} \cdots b_n^{e_n} \) where \( b_i \) and \( e_i \) are ca_t numbers (the exponents need not be integers).

void ca_factor_init(ca_factor_t fac, ca_ctx_t ctx)
Initializes fac and sets it to the empty factorization (equivalent to the number 1).

void ca_factor_clear(ca_factor_t fac, ca_ctx_t ctx)
Clears the factorization structure fac.

void ca_factor_one(ca_factor_t fac, ca_ctx_t ctx)
Sets fac to the empty factorization (equivalent to the number 1).

void ca_factor_print(const ca_factor_t fac, ca_ctx_t ctx)
Prints a description of fac to standard output.

void ca_factor_insert(ca_factor_t fac, const ca_t base, const ca_t exp, ca_ctx_t ctx)
Inserts \( b^e \) into fac where \( b \) is given by base and \( e \) is given by exp. If a base element structurally identical to base already exists in fac, the corresponding exponent is incremented by exp; otherwise, this factor is appended.
void ca_factor_get_ca(ca_t res, const ca_factor_t fac, ca_ctx_t ctx)
    Expands fac back to a single ca_t by evaluating the powers and multiplying out the result.

void ca_factor(ca_factor_t res, const ca_t x, ulong flags, ca_ctx_t ctx)
    Sets res to a factorization of x of the form \( x = b_1^{e_1} b_2^{e_2} \cdots b_n^{e_n} \). Requires that x is not a special value. The type of factorization is controlled by flags, which can be set to a combination of constants in the following section.

### Factorization options

The following flags select the structural polynomial factorization to perform over formal fields \( \mathbb{Q}(a_1, \ldots, a_n) \). Each flag in the list strictly encompasses the factorization power of the preceding flag, so it is unnecessary to pass more than one flag.

- **CA_FACTOR_POLY_NONE**
  - No polynomial factorization at all.
- **CA_FACTOR_POLY_CONTENT**
  - Only extract the rational content.
- **CA_FACTOR_POLY_SQF**
  - Perform a squarefree factorization in addition to extracting the rational content.
- **CA_FACTOR_POLY_FULL**
  - Perform a full multivariate polynomial factorization.

The following flags select the factorization to perform over \( \mathbb{Z} \). Integer factorization is applied if x is an element of \( \mathbb{Q} \), and to the extracted rational content of polynomials. Each flag in the list strictly encompasses the factorization power of the preceding flag, so it is unnecessary to pass more than one flag.

- **CA_FACTOR_ZZ_NONE**
  - No integer factorization at all.
- **CA_FACTOR_ZZ_SMOOTH**
  - Perform a smooth factorization to extract small prime factors (heuristically up to CA_OPT_SMOOTH_LIMIT bits) in addition to identifying perfect powers.
- **CA_FACTOR_ZZ_FULL**
  - Perform a complete integer factorization into prime numbers. This is prohibitively slow for general integers exceeding 70-80 digits.

### 10.4.25 Context options

The options member of a ca_ctx_t object is an array of slong values controlling simplification behavior and various other settings. The values of the array at the following indices can be changed by the user (example: ctx->options[CA_OPT_PREC_LIMIT] = 65536).

It is recommended to set options controlling evaluation only at the time when a context object is created. Changing such options later should normally be harmless, but since the update will not apply retroactively to objects that have already been computed and cached, one might not see the expected behavior. Superficial options (printing) can be changed at any time.

- **CA_OPT_VERBOSE**
  - Whether to print debug information. Default value: 0.
- **CA_OPT_PRINT_FLAGS**
  - Printing style. See Printing for details. Default value: CA_PRINT_DEFAULT.
CA_OPT_MPOLY_ORD
Monomial ordering to use for multivariate polynomials. Possible values are ORD_LEX, ORD_DEGLEX and ORD_DEGREVLEX. Default value: ORD_LEX. This option must be set before doing any computations.

CA_OPT_PREC_LIMIT
Maximum precision to use internally for numerical evaluation with Arb, and in some cases for the magnitude of exact coefficients. This parameter affects the possibility to prove inequalities and find simplifications between related extension numbers. This is not a strict limit; some calculations may use higher precision when there is a good reason to do so. Default value: 4096.

CA_OPT_QQBAR_DEG_LIMIT
Maximum degree of qqbar_t elements allowed internally during simplification of algebraic numbers. This limit may be exceeded when the user provides explicit qqbar_t input of higher degree. Default value: 120.

CA_OPT_LOW_PREC
Numerical precision to use for fast checks (typically, before attempting more expensive operations). Default value: 64.

CA_OPT_SMOOTH_LIMIT
Size in bits for factors in smooth integer factorization. Default value: 32.

CA_OPT_LLL_PREC
Precision to use to find integer relations using LLL. Default value: 128.

CA_OPT_POW_LIMIT
Largest exponent to expand powers automatically. This only applies in multivariate and transcendental fields: in number fields, CA_OPT_PREC_LIMIT applies instead. Default value: 20.

CA_OPT_USE_GROEBNER
Boolean flag for whether to use Gröbner basis computation. This flag and the following limits affect the ability to prove multivariate identities. Default value: 1.

CA_OPT_GROEBNER_LENGTH_LIMIT
Maximum length of ideal basis allowed in Buchberger’s algorithm. Default value: 100.

CA_OPT_GROEBNER_POLY_LENGTH_LIMIT
Maximum length of polynomials allowed in Buchberger’s algorithm. Default value: 1000.

CA_OPT_GROEBNER_POLY_BITS_LIMIT
Maximum coefficient size in bits of polynomials allowed in Buchberger’s algorithm. Default value: 10000.

CA_OPT_VIETA_LIMIT
Maximum degree n of algebraic numbers for which to add Vieta’s formulas to the reduction ideal. This must be set relatively low since the number of terms in Vieta’s formulas is $O(2^n)$ and the resulting Gröbner basis computations can be expensive. Default value: 6.

CA_OPT_TRIG_FORM
Default representation of trigonometric functions. The following values are possible:

CA_TRIG_DIRECT
Use the direct functions (with some exceptions).

CA_TRIG_EXPONENTIAL
Use complex exponentials.

CA_TRIG_SINE_COSINE
Use sines and cosines.
CA_TRIG_TANGENT

Use tangents.

Default value: CA_TRIG_EXPONENTIAL.

The exponential representation is currently used by default as typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. This may change in the future.

10.4.26 Internal representation

CA_FMPQ(x)

CA_FMPQ_NUMREF(x)

CA_FMPQ_DENREF(x)

Assuming that x holds an element of the trivial field \( \mathbb{Q} \), this macro returns a pointer which can be used as an \( fmpq_t \), or respectively to the numerator or denominator as an \( fmpz_t \).

CA_MPOLY_Q(x)

Assuming that x holds a generic field element as data, this macro returns a pointer which can be used as an \( fmpz_mpoly_q_t \).

CA_NF_ELEM(x)

Assuming that x holds an Antic number field element as data, this macro returns a pointer which can be used as an \( nf_elem_t \).

void _ca_make_field_element(ca_t x, ca_field_sptr new_index, ca_ctx_t ctx)

Changes the internal representation of x to that of an element of the field with index new_index in the context object ctx. This may destroy the value of x.

void _ca_make_fmpq(ca_t x, ca_ctx_t ctx)

Changes the internal representation of x to that of an element of the trivial field \( \mathbb{Q} \). This may destroy the value of x.
10.5 ca_vec.h – vectors of real and complex numbers

A `ca_vec_t` represents a vector of real or complex numbers, implemented as an array of coefficients of type `ca_struct`.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients (taking `ca_ptr` and `ca_srcptr` arguments), and a non-underscore method which takes `ca_vec_t` input and performs automatic memory management.

Unlike `ca_poly_t`, a `ca_vec_t` is not normalised by removing zero coefficients; it retains the exact length assigned by the user.

10.5.1 Types, macros and constants

```c
typedef ca_struct ca_vec_struct;
type ca_vec_t

Contains a pointer to an array of entries (`coeffs`), the used length (`length`), and the allocated size of the array (`alloc`).

A `ca_vec_t` is defined as an array of length one of type `ca_vec_struct`, permitting an `ca_vec_t` to be passed by reference.
```

```c
ca_vec_entry(vec, i)

Macro returning a pointer to entry `i` in the vector `vec`. The index must be in bounds.
```

10.5.2 Memory management

```c
ca_ptr ca_vec_init(slong len, ca_ctx_t ctx)

Returns a pointer to an array of `len` coefficients initialized to zero.
```

```c
void ca_vec_init(ca_vec_t vec, slong len, ca_ctx_t ctx)

Initializes `vec` to a length `len` vector. All entries are set to zero.
```

```c
void _ca_vec_clear(ca_ptr vec, slong len, ca_ctx_t ctx)

Clears all `len` entries in `vec` and frees the pointer `vec` itself.
```

```c
void ca_vec_clear(ca_vec_t vec, ca_ctx_t ctx)

Clears the vector `vec`.
```

```c
void _ca_vec_swap(ca_ptr vec1, ca_ptr vec2, slong len, ca_ctx_t ctx)

Swaps the entries in `vec1` and `vec2` efficiently.
```

```c
void ca_vec_swap(ca_vec_t vec1, ca_vec_t vec2, ca_ctx_t ctx)

Swaps the vectors `vec1` and `vec2` efficiently.
```

10.5.3 Length

```c
slong ca_vec_length(const ca_vec_t vec, ca_ctx_t ctx)

Returns the length of `vec`.
```

```c
void _ca_vec_fit_length(ca_vec_t vec, slong len, ca_ctx_t ctx)

Allocates space in `vec` for `len` elements.
```

```c
void ca_vec_set_length(ca_vec_t vec, slong len, ca_ctx_t ctx)

Sets the length of `vec` to `len`. If `vec` is shorter on input, it will be zero-extended. If `vec` is longer on input, it will be truncated.
```
10.5.4 Assignment

```c
void _ca_vec_set(ca_ptr res, ca_srcptr src, slong len, ca_ctx_t ctx)
    Sets res to a copy of src of length len.

void ca_vec_set(ca_vec_t res, const ca_vec_t src, ca_ctx_t ctx)
    Sets res to a copy of src.
```

10.5.5 Special vectors

```c
void _ca_vec_zero(ca_ptr res, slong len, ca_ctx_t ctx)
    Sets the len entries in res to zeros.

void ca_vec_zero(ca_vec_t res, slong len, ca_ctx_t ctx)
    Sets res to the length len zero vector.
```

10.5.6 Input and output

```c
void ca_vec_print(const ca_vec_t vec, ca_ctx_t ctx)
    Prints vec to standard output. The coefficients are printed on separate lines.

void ca_vec_printn(const ca_vec_t poly, slong digits, ca_ctx_t ctx)
    Prints a decimal representation of vec with precision specified by digits. The coefficients are comma-separated and the whole list is enclosed in square brackets.
```

10.5.7 List operations

```c
void ca_vec_append(ca_vec_t vec, const ca_t f, ca_ctx_t ctx)
    Appends f to the end of vec.
```

10.5.8 Arithmetic

```c
void _ca_vec_neg(ca_ptr res, ca_srcptr src, slong len, ca_ctx_t ctx)

void ca_vec_neg(ca_vec_t res, const ca_vec_t src, ca_ctx_t ctx)
    Sets res to the negation of src.

void _ca_vec_add(ca_ptr res, ca_srcptr vec1, ca_srcptr vec2, slong len, ca_ctx_t ctx)

void _ca_vec_sub(ca_ptr res, ca_srcptr vec1, ca_srcptr vec2, slong len, ca_ctx_t ctx)
    Sets res to the sum or difference of vec1 and vec2, all vectors having length len.

void _ca_vec_scalar_mul_ca(ca_ptr res, ca_srcptr src, slong len, const ca_t c, ca_ctx_t ctx)
    Sets res to src multiplied by c, all vectors having length len.

void _ca_vec_scalar_div_ca(ca_ptr res, ca_srcptr src, slong len, const ca_t c, ca_ctx_t ctx)
    Sets res to src divided by c, all vectors having length len.

void _ca_vec_scalar_addmul_ca(ca_ptr res, ca_srcptr src, slong len, const ca_t c, ca_ctx_t ctx)
    Adds src multiplied by c to the vector res, all vectors having length len.

void _ca_vec_scalar_submul_ca(ca_ptr res, ca_srcptr src, slong len, const ca_t c, ca_ctx_t ctx)
    Subtracts src multiplied by c from the vector res, all vectors having length len.
```
10.5.9 Comparisons and properties

\textit{truth\_t \_ca\_vec\_check\_is\_zero}(\textit{ca\_srcptr vec, slong len, ca\_ctx\_t ctx})

Returns whether \textit{vec} is the zero vector.

10.5.10 Internal representation

\textit{int \_ca\_vec\_is\_fmpq\_vec}(\textit{ca\_srcptr vec, slong len, ca\_ctx\_t ctx})

Checks if all elements of \textit{vec} are structurally rational numbers.

\textit{int \_ca\_vec\_fmpq\_vec\_is\_fmpz\_vec}(\textit{ca\_srcptr vec, slong len, ca\_ctx\_t ctx})

Assuming that all elements of \textit{vec} are structurally rational numbers, checks if all elements are integers.

\textit{void \_ca\_vec\_fmpq\_vec\_get\_fmpz\_vec\_den}(\textit{fmpz *c, fmpz_t den, ca\_srcptr vec, slong len, ca\_ctx\_t ctx})

Assuming that all elements of \textit{vec} are structurally rational numbers, converts them to a vector of integers \textit{c} on a common denominator \textit{den}.

\textit{void \_ca\_vec\_set\_fmpz\_vec\_div\_fmpz}(\textit{ca\_ptr res, const fmpz *v, const fmpz_t den, slong len, ca\_ctx\_t ctx})

Sets \textit{res} to the rational vector given by numerators \textit{v} and the common denominator \textit{den}.
10.6 ca_poly.h – dense univariate polynomials over the real and complex numbers

A \texttt{ca\_poly\_t} represents a univariate polynomial over the real or complex numbers (an element of \( \mathbb{R}[X] \) or \( \mathbb{C}[X] \)), implemented as an array of coefficients of type \texttt{ca\_struct}.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

Warnings:

- A polynomial is always normalised by removing zero coefficients at the top. Coefficients will not be removed when Calcium is unable to prove that they are zero. The represented degree can therefore be larger than the degree of the mathematical polynomial. When the correct degree is needed, it is important to verify the leading coefficient. (Of course, this will never be an issue with polynomials that are explicitly monic, for example.)

- The special values \texttt{Undefined}, unsigned infinity and signed infinity supported by the scalar \texttt{ca\_t} type are not really meaningful as coefficients of polynomials. We normally assume that the user does not assign those values to coefficients of polynomials, and the functions in this module will likewise normally not generate such coefficients. \texttt{Unknown} can still appear as a coefficient representing a number that is inaccessible for computation.

A polynomial with numerical coefficients and with a nonzero leading coefficient is called proper. The function \texttt{ca\_poly\_is\_proper()} can be used to check for violations.

10.6.1 Types, macros and constants

type \texttt{ca\_poly\_struct}

type \texttt{ca\_poly\_t}

Contains a pointer to an array of coefficients (\texttt{coeffs}), the used length (\texttt{length}), and the allocated size of the array (\texttt{alloc}).

A \texttt{ca\_poly\_t} is defined as an array of length one of type \texttt{ca\_poly\_struct}, permitting an \texttt{ca\_poly\_t} to be passed by reference.

10.6.2 Memory management

\begin{verbatim}
void ca_poly_init(ca_poly_t poly, ca_ctx_t ctx)
    Initializes the polynomial for use, setting it to the zero polynomial.

void ca_poly_clear(ca_poly_t poly, ca_ctx_t ctx)
    Clears the polynomial, deallocating all coefficients and the coefficient array.

void ca_poly_fit_length(ca_poly_t poly, slong len, ca_ctx_t ctx)
    Makes sure that the coefficient array of the polynomial contains at least \texttt{len} initialized coefficients.

void _ca_poly_set_length(ca_poly_t poly, slong len, ca_ctx_t ctx)
    Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void _ca_poly_normalise(ca_poly_t poly, ca_ctx_t ctx)
    Strips any top coefficients which can be proved identical to zero.
\end{verbatim}
10.6.3 Assignment and simple values

void \texttt{ca\_poly\_zero} (\texttt{ca\_poly\_t} poly, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Sets poly to the zero polynomial.
\end{verbatim}

void \texttt{ca\_poly\_one} (\texttt{ca\_poly\_t} poly, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Sets poly to the constant polynomial 1.
\end{verbatim}

void \texttt{ca\_poly\_x} (\texttt{ca\_poly\_t} poly, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Sets poly to the monomial $x$.
\end{verbatim}

void \texttt{ca\_poly\_set\_ca} (\texttt{ca\_poly\_t} poly, \texttt{const ca\_t} c, \texttt{ca\_ctx\_t} ctx)

void \texttt{ca\_poly\_set\_si} (\texttt{ca\_poly\_t} poly, \texttt{slong} c, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Sets poly to the constant polynomial $c$.
\end{verbatim}

void \texttt{ca\_poly\_set} (\texttt{ca\_poly\_t} res, \texttt{const ca\_poly\_t} src, \texttt{ca\_ctx\_t} ctx)

void \texttt{ca\_poly\_set\_fmpz\_poly} (\texttt{ca\_poly\_t} res, \texttt{const fmpz\_poly\_t} src, \texttt{ca\_ctx\_t} ctx)

void \texttt{ca\_poly\_set\_fmpq\_poly} (\texttt{ca\_poly\_t} res, \texttt{const fmpq\_poly\_t} src, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Sets poly the polynomial src.
\end{verbatim}

void \texttt{ca\_poly\_set\_coeff\_ca} (\texttt{ca\_poly\_t} poly, \texttt{slong} n, \texttt{const ca\_t} x, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Sets the coefficient at position $n$ in poly to $x$.
\end{verbatim}

void \texttt{ca\_poly\_transfer} (\texttt{ca\_poly\_t} res, \texttt{ca\_ctx\_t} res\_ctx, \texttt{const ca\_poly\_t} src, \texttt{ca\_ctx\_t} src\_ctx)
\begin{verbatim}
Sets res to src where the corresponding context objects res\_ctx and src\_ctx may be different.
\end{verbatim}
\begin{verbatim}
This operation preserves the mathematical value represented by src, but may result in a different internal representation depending on the settings of the context objects.
\end{verbatim}

10.6.4 Random generation

void \texttt{ca\_poly\_randtest} (\texttt{ca\_poly\_t} poly, \texttt{flint\_rand\_t} state, \texttt{slong} len, \texttt{slong} depth, \texttt{slong} bits, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Sets poly to a random polynomial of length up to len and with entries having complexity up to depth and bits (see \texttt{ca\_randtest()}).
\end{verbatim}

void \texttt{ca\_poly\_randtest\_rational} (\texttt{ca\_poly\_t} poly, \texttt{flint\_rand\_t} state, \texttt{slong} len, \texttt{slong} bits, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Sets poly to a random rational polynomial of length up to len and with entries up to bits bits in size.
\end{verbatim}

10.6.5 Input and output

void \texttt{ca\_poly\_print} (\texttt{const ca\_poly\_t} poly, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Prints poly to standard output. The coefficients are printed on separate lines.
\end{verbatim}

void \texttt{ca\_poly\_printn} (\texttt{const ca\_poly\_t} poly, \texttt{slong} digits, \texttt{ca\_ctx\_t} ctx)
\begin{verbatim}
Prints a decimal representation of poly with precision specified by digits. The coefficients are comma-separated and the whole list is enclosed in square brackets.
\end{verbatim}
10.6.6 Degree and leading coefficient

```c
int ca_poly_is_proper(const ca_poly_t poly, ca_ctx_t ctx)
```

Checks that `poly` represents an element of $\mathbb{C}[X]$ with well-defined degree. This returns 1 if the leading coefficient of `poly` is nonzero and all coefficients of `poly` are numbers (not special values). It returns 0 otherwise. It returns 1 when `poly` is precisely the zero polynomial (which does not have a leading coefficient).

```c
int ca_poly_make_monic(ca_poly_t res, const ca_poly_t poly, ca_ctx_t ctx)
```

Makes `poly` monic by dividing by the leading coefficient if possible and returns 1. Returns 0 if the leading coefficient cannot be certified to be nonzero, or if `poly` is the zero polynomial.

```c
void _ca_poly_reverse(ca_ptr res, ca_srcptr poly, slong len, slong n, ca_ctx_t ctx)
```

Sets `res` to the reversal of `poly` considered as a polynomial of length `n`, zero-padding if needed. The underscore method assumes that `len` is positive and less than or equal to `n`.

10.6.7 Comparisons

```c
truth_t _ca_poly_check_equal(ca_srcptr poly1, slong len1, ca_srcptr poly2, slong len2, ca_ctx_t ctx)
```

```c
truth_t ca_poly_check_equal(const ca_poly_t poly1, const ca_poly_t poly2, ca_ctx_t ctx)
```

Checks if `poly1` and `poly2` represent the same polynomial. The underscore method assumes that `len1` is at least as large as `len2`.

```c
truth_t ca_poly_check_is_zero(const ca_poly_t poly, ca_ctx_t ctx)
```

Checks if `poly` is the zero polynomial.

```c
truth_t ca_poly_check_is_one(const ca_poly_t poly, ca_ctx_t ctx)
```

Checks if `poly` is the constant polynomial 1.

10.6.8 Arithmetic

```c
void _ca_poly_shift_left(ca_ptr res, ca_srcptr poly, slong len, slong n, ca_ctx_t ctx)
```

```c
void ca_poly_shift_left(ca_poly_t res, const ca_poly_t poly, slong n, ca_ctx_t ctx)
```

Sets `res` to `poly` shifted `n` coefficients to the left; that is, multiplied by $x^n$.

```c
void _ca_poly_shift_right(ca_ptr res, ca_srcptr poly, slong len, slong n, ca_ctx_t ctx)
```

```c
void ca_poly_shift_right(ca_poly_t res, const ca_poly_t poly, slong n, ca_ctx_t ctx)
```

Sets `res` to `poly` shifted `n` coefficients to the right; that is, divided by $x^n$.

```c
void ca_poly_neg(ca_poly_t res, const ca_poly_t src, ca_ctx_t ctx)
```

Sets `res` to the negation of `src`.

```c
void _ca_poly_add(ca_ptr res, ca_srcptr poly1, slong len1, ca_srcptr poly2, slong len2, ca_ctx_t ctx)
```

```c
void ca_poly_add(ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, ca_ctx_t ctx)
```

Sets `res` to the sum of `poly1` and `poly2`.

```c
void _ca_poly_sub(ca_ptr res, ca_srcptr poly1, slong len1, ca_srcptr poly2, slong len2, ca_ctx_t ctx)
```

```c
void ca_poly_sub(ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, ca_ctx_t ctx)
```

Sets `res` to the difference of `poly1` and `poly2`.

```c
void _ca_poly_mul(ca_ptr res, ca_srcptr poly1, slong len1, ca_srcptr poly2, slong len2, ca_ctx_t ctx)
```

```c
void ca_poly_mul(ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, ca_ctx_t ctx)
```

10.6. ca_poly.h – dense univariate polynomials over the real and complex numbers
void ca_poly_mul(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)
Sets res to the product of poly1 and poly2.

void _ca_poly_mul_low(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)

void ca_poly_div(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)
Sets res to the quotient of poly1 divided by poly2.

int ca_poly_divrem(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)
Sets res to the quotient and remainder of poly1 divided by poly2.

void _ca_poly_divrem_basecase(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)

void ca_poly_divrem(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)
Sets res to the quotient and remainder of poly1 divided by poly2.

int ca_poly_divrem_basecase(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)

void _ca_poly_divrem(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)
Sets res to the quotient and remainder of poly1 divided by poly2.

int ca_poly_div(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)

int ca_poly_rem(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)
If the leading coefficient of poly1 can be proved invertible, sets res to the quotient of poly1 divided by poly2.

void _ca_poly_pow_ui_trunc(const ca_poly_t res, const ca_poly_t poly, const slong expo, const slong len, const ca_ctx_t ctx)

void _ca_poly_pow_ui(const ca_poly_t res, const ca_poly_t poly, const ulong expo, const slong len, const ca_ctx_t ctx)
Sets res to poly raised to the power expo, truncated to length len.

void _ca_poly_pow_ui(const ca_poly_t res, const ca_poly_t poly, const slong expo, const ulong len, const ca_ctx_t ctx)
Sets res to poly raised to the power expo.

10.6.9 Evaluation and composition

void _ca_poly_evaluate_horner(const ca_t res, const ca_srcptr f, slong len, const ca_t x, const ca_ctx_t ctx)

void _ca_poly_evaluate_horner(const ca_t res, const ca_poly_t f, const ca_t a, const ca_ctx_t ctx)

void _ca_poly_evaluate(const ca_t res, const ca_srcptr f, slong len, const ca_t x, const ca_ctx_t ctx)

void _ca_poly_evaluate(const ca_t res, const ca_poly_t f, const ca_t a, const ca_ctx_t ctx)
Sets res to f evaluated at the point a.

void _ca_poly_compose(const ca_t res, const ca_srcptr poly1, slong len1, const ca_srcptr poly2, slong len2, const ca_ctx_t ctx)

void _ca_poly_compose(const ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, const ca_ctx_t ctx)
Sets res to the composition of poly1 with poly2.
10.6.10 Derivative and integral

void _ca_poly_derivative(ca_ptr res, ca_srcptr poly, slong len, ca_ctx_t ctx)
void ca_poly_derivative(ca_poly_t res, const ca_poly_t poly, ca_ctx_t ctx)

Sets res to the derivative of poly. The underscore method needs one less coefficient than len for the output array.

void _ca_poly_integral(ca_ptr res, ca_srcptr poly, slong len, ca_ctx_t ctx)
void ca_poly_integral(ca_poly_t res, const ca_poly_t poly, ca_ctx_t ctx)

Sets res to the integral of poly. The underscore method needs one more coefficient than len for the output array.

10.6.11 Power series division

void _ca_poly_inv_series(ca_ptr res, ca_srcptr f, slong flen, slong len, ca_ctx_t ctx)
void ca_poly_inv_series(ca_poly_t res, const ca_poly_t f, slong len, ca_ctx_t ctx)

Sets res to the power series inverse of f truncated to length len.

void _ca_poly_div_series(ca_ptr res, ca_srcptr f, slong flen, ca_srcptr g, slong glen, slong len, ca_ctx_t ctx)
void ca_poly_div_series(ca_poly_t res, const ca_poly_t f, const ca_poly_t g, slong len, ca_ctx_t ctx)

Sets res to the power series quotient of f and g truncated to length len. This function divides by zero if g has constant term zero; the user should manually remove initial zeros when an exact cancellation is required.

10.6.12 Elementary functions

void _ca_poly_exp_series(ca_ptr res, ca_srcptr f, slong flen, slong len, ca_ctx_t ctx)
void ca_poly_exp_series(ca_poly_t res, const ca_poly_t f, slong len, ca_ctx_t ctx)

Sets res to the power series exponential of f truncated to length len.

void _ca_poly_log_series(ca_ptr res, ca_srcptr f, slong flen, slong len, ca_ctx_t ctx)
void ca_poly_log_series(ca_poly_t res, const ca_poly_t f, slong len, ca_ctx_t ctx)

Sets res to the power series logarithm of f truncated to length len.

10.6.13 Greatest common divisor

slong _ca_poly_gcd_euclidean(ca_ptr res, ca_srcptr A, slong lenA, ca_srcptr B, slong lenB, ca_ctx_t ctx)
int ca_poly_gcd_euclidean(ca_poly_t res, const ca_poly_t A, const ca_poly_t B, ca_ctx_t ctx)
slong _ca_poly_gcd(ca_ptr res, ca_srcptr A, slong lenA, ca_srcptr B, slong lenB, ca_ctx_t ctx)
int ca_poly_gcd(ca_poly_t res, const ca_poly_t A, const ca_poly_t B, ca_ctx_t ctx)

Sets res to the GCD of A and B and returns 1 on success. On failure, returns 0 leaving the value of res arbitrary. The computation can fail if testing a leading coefficient for zero fails in the execution of the GCD algorithm. The output is normalized to be monic if it is not the zero polynomial.

The underscore methods assume lenA ≥ lenB ≥ 1, and that both A and B have nonzero leading coefficient. They return the length of the GCD, or 0 if the computation fails.

The euclidean version implements the standard Euclidean algorithm. The default version first checks for rational polynomials or attempts to certify numerically that the polynomials are coprime and otherwise falls back to an automatic choice of algorithm (currently only the Euclidean algorithm).
10.6.14 Roots and factorization

```c
int ca_poly_factor_squarefree(ca_t c, ca_poly_vec_t fac, ulong *exp, const ca_poly_t F, ca_ctx_t ctx)
```

Computes the squarefree factorization of $F$, giving a product $F = c f_1 f_2^2 \cdots f_n^n$ where all $f_i$ with $f_i \neq 1$ are squarefree and pairwise coprime. The nontrivial factors $f_i$ are written to `fac` and the corresponding exponents are written to `exp`. This algorithm can fail if GCD computation fails internally. Returns 1 on success and 0 on failure.

```c
int ca_poly_squarefree_part(ca_poly_t res, const ca_poly_t poly, ca_ctx_t ctx)
```

Sets `res` to the squarefree part of `poly`, normalized to be monic. This algorithm can fail if GCD computation fails internally. Returns 1 on success and 0 on failure.

```c
void _ca_poly_set_roots(ca_ptr poly, ca_srcptr roots, const ulong *exp, slong n, ca_ctx_t ctx)
void ca_poly_set_roots(ca_poly_t poly, ca_vec_t roots, const ulong *exp, ca_ctx_t ctx)
```

Sets `poly` to the monic polynomial with the $n$ roots given in the vector `roots`, with multiplicities given in the vector `exp`. In other words, this constructs the polynomial $(x - r_0)^e_0 (x - r_1)^e_1 \cdots (x - r_{n-1})^{e_{n-1}}$. Uses binary splitting.

```c
int _ca_poly_roots(ca_ptr roots, ca_srcptr poly, slong len, ca_ctx_t ctx)
int ca_poly_roots(ca_vec_t roots, ulong *exp, const ca_poly_t poly, ca_ctx_t ctx)
```

Attempts to compute all complex roots of the given polynomial `poly`. On success, returns 1 and sets `roots` to a vector containing all the distinct roots with corresponding multiplicities in `exp`. On failure, returns 0 and leaves the values in `roots` arbitrary. The roots are returned in arbitrary order. Failure will occur if the leading coefficient of `poly` cannot be proved to be nonzero, if determining the correct multiplicities fails, or if the built-in algorithms do not have a means to represent the roots symbolically.

The underscore method assumes that the polynomial is squarefree. The non-underscore method performs a squarefree factorization.

10.6.15 Vectors of polynomials

```c
type ca_poly_vec_struct

type ca_poly_vec_t
    Represents a vector of polynomials.

ca_poly_struct * _ca_poly_vec_init(slong len, ca_ctx_t ctx)
void ca_poly_vec_init(ca_poly_vec_t res, slong len, ca_ctx_t ctx)
    Initializes a vector with `len` polynomials.

void _ca_poly_vec_fit_length(ca_poly_vec_t vec, slong len, ca_ctx_t ctx)
    Allocates space for `len` polynomials in `vec`.

void ca_poly_vec_set_length(ca_poly_vec_t vec, slong len, ca_ctx_t ctx)
    Resizes `vec` to length `len`, zero-extending if needed.

void _ca_poly_vec_clear(ca_poly_vec_t *vec, slong len, ca_ctx_t ctx)
void ca_poly_vec_clear(ca_poly_vec_t vec, ca_ctx_t ctx)
    Clears the vector `vec`.

void ca_poly_vec_append(ca_poly_vec_t vec, const ca_poly_t poly, ca_ctx_t ctx)
    Appends `poly` to the end of the vector `vec`.
```
10.7 ca_mat.h – matrices over the real and complex numbers

A `ca_mat_t` represents a dense matrix over the real or complex numbers, implemented as an array of entries of type `ca_struct`. The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

### 10.7.1 Types, macros and constants

**type** `ca_mat_struct`  
`ca_mat_t`  
Contains a pointer to a flat array of the entries (`entries`), an array of pointers to the start of each row (`rows`), and the number of rows (`r`) and columns (`c`).

A `ca_mat_t` is defined as an array of length one of type `ca_mat_struct`, permitting a `ca_mat_t` to be passed by reference.

**ca_mat_entry(mat, i, j)**  
Macro giving a pointer to the entry at row `i` and column `j`.

**ca_mat_nrows(mat)**  
Returns the number of rows of the matrix.

**ca_mat_ncols(mat)**  
Returns the number of columns of the matrix.

**ca_ptr ca_mat_entry_ptr(ca_mat_t mat, slong i, slong j)**  
Returns a pointer to the entry at row `i` and column `j`. Equivalent to `ca_mat_entry` but implemented as a function.

### 10.7.2 Memory management

**void ca_mat_init(ca_mat_t mat, slong r, slong c, ca_ctx_t ctx)**  
Initializes the matrix, setting it to the zero matrix with `r` rows and `c` columns.

**void ca_mat_clear(ca_mat_t mat, ca_ctx_t ctx)**  
Clears the matrix, deallocating all entries.

**void ca_mat_swap(ca_mat_t mat1, ca_mat_t mat2, ca_ctx_t ctx)**  
Efficiently swaps `mat1` and `mat2`.

**void ca_mat_window_init(ca_mat_t window, const ca_mat_t mat, slong r1, slong c1, slong r2, slong c2, ca_ctx_t ctx)**  
Initializes `window` to a window matrix into the submatrix of `mat` starting at the corner at row `r1` and column `c1` (inclusive) and ending at row `r2` and column `c2` (exclusive).

**void ca_mat_window_clear(ca_mat_t window, ca_ctx_t ctx)**  
Frees the window matrix.
10.7.3 Assignment and conversions

void ca_mat_set (ca_mat_t dest, const ca_mat_t src, ca_ctx_t ctx)
void ca_mat_set_fmpz_mat (ca_mat_t dest, const fmpz_mat_t src, ca_ctx_t ctx)
void ca_mat_set_fmpq_mat (ca_mat_t dest, const fmpq_mat_t src, ca_ctx_t ctx)

Sets dest to src. The operands must have identical dimensions.

void ca_mat_set_ca (ca_mat_t mat, const ca_t c, ca_ctx_t ctx)

Sets mat to the matrix with the scalar c on the main diagonal and zeros elsewhere.

void ca_mat_transfer (ca_mat_t res, ca_ctx_t res_ctx, const ca_mat_t src, ca_ctx_t src_ctx)

Sets res to src where the corresponding context objects res_ctx and src_ctx may be different.

This operation preserves the mathematical value represented by src, but may result in a different internal representation depending on the settings of the context objects.

10.7.4 Random generation

void ca_mat_randtest (ca_mat_t mat, flint_rand_t state, slong depth, slong bits, ca_ctx_t ctx)

Sets mat to a random matrix with entries having complexity up to depth and bits (see ca_randtest ()).

void ca_mat_randtest_rational (ca_mat_t mat, flint_rand_t state, slong bits, ca_ctx_t ctx)

Sets mat to a random rational matrix with entries up to bits bits in size.

void ca_mat_randops (ca_mat_t mat, flint_rand_t state, slong count, ca_ctx_t ctx)

Randomizes mat in-place by performing elementary row or column operations. More precisely, at most count random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, the determinant) unchanged.

10.7.5 Input and output

void ca_mat_print (const ca_mat_t mat, ca_ctx_t ctx)

Prints mat to standard output. The entries are printed on separate lines.

void ca_mat_printn (const ca_mat_t mat, slong digits, ca_ctx_t ctx)

Prints a decimal representation of mat with precision specified by digits. The entries are comma-separated with square brackets and comma separation for the rows.

10.7.6 Special matrices

void ca_mat_zero (ca_mat_t mat, ca_ctx_t ctx)
Sets all entries in mat to zero.

void ca_mat_one (ca_mat_t mat, ca_ctx_t ctx)
Sets the entries on the main diagonal of mat to one, and all other entries to zero.

void ca_mat_ones (ca_mat_t mat, ca_ctx_t ctx)
Sets all entries in mat to one.

void ca_mat_pascal (ca_mat_t mat, int triangular, ca_ctx_t ctx)
Sets mat to a Pascal matrix, whose entries are binomial coefficients. If triangular is 0, constructs a full symmetric matrix with the rows of Pascal’s triangle as successive antidiagonals. If triangular is 1, constructs the upper triangular matrix with the rows of Pascal’s triangle as columns, and if triangular is -1, constructs the lower triangular matrix with the rows of Pascal’s triangle as rows.
void **ca_mat_stirling**(ca_mat_t mat, int kind, ca_ctx_t ctx)

Sets mat to a Stirling matrix, whose entries are Stirling numbers. If kind is 0, the entries are set to the unsigned Stirling numbers of the first kind. If kind is 1, the entries are set to the signed Stirling numbers of the first kind. If kind is 2, the entries are set to the Stirling numbers of the second kind.

void **ca_mat_hilbert**(ca_mat_t mat, ca_ctx_t ctx)

Sets mat to the Hilbert matrix, which has entries \( A_{ij} = 1/(i + j + 1) \).

void **ca_mat_dft**(ca_mat_t mat, int type, ca_ctx_t ctx)

Sets mat to the DFT (discrete Fourier transform) matrix of order \( n \) where \( n \) is the smallest dimension of mat (if mat is not square, the matrix is extended periodically along the larger dimension). The type parameter selects between four different versions of the DFT matrix (in which \( \omega = e^{2\pi i/n} \)):

- Type 0 – entries \( A_{jk} = \omega^{-jk} \)
- Type 1 – entries \( A_{jk} = \omega^{jk}/n \)
- Type 2 – entries \( A_{jk} = \omega^{-jk}/\sqrt{n} \)
- Type 3 – entries \( A_{jk} = \omega^{jk}/\sqrt{n} \)

The type 0 and 1 matrices are inverse pairs, and similarly for the type 2 and 3 matrices.

10.7.7 Comparisons and properties

**truth_t ca_mat_check_equal**(const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)

Compares A and B for equality.

**truth_t ca_mat_check_is_zero**(const ca_mat_t A, ca_ctx_t ctx)

Tests if A is the zero matrix.

**truth_t ca_mat_check_is_one**(const ca_mat_t A, ca_ctx_t ctx)

Tests if A has ones on the main diagonal and zeros elsewhere.

10.7.8 Conjugate and transpose

void **ca_mat_transpose**(ca_mat_t res, const ca_mat_t A, ca_ctx_t ctx)

Sets res to the transpose of A.

void **ca_mat_conj**(ca_mat_t res, const ca_mat_t A, ca_ctx_t ctx)

Sets res to the entrywise complex conjugate of A.

void **ca_mat_conj_transpose**(ca_mat_t res, const ca_mat_t A, ca_ctx_t ctx)

Sets res to the conjugate transpose (Hermitian transpose) of A.

10.7.9 Arithmetic

void **ca_mat_neg**(ca_mat_t res, const ca_mat_t A, ca_ctx_t ctx)

Sets res to the negation of A.

void **ca_mat_add**(ca_mat_t res, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)

Sets res to the sum of A and B.

void **ca_mat_sub**(ca_mat_t res, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)

Sets res to the difference of A and B.

void **ca_mat_mul_classical**(ca_mat_t res, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)
void ca_mat_mul_same_nf(ca_mat_t res, const ca_mat_t A, const ca_mat_t B, ca_field_t K, ca_ctx_t ctx)
void ca_mat_mul(ca_mat_t res, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)

Sets res to the matrix product of A and B. The classical version uses classical multiplication. The same_nf version assumes (not checked) that both A and B have coefficients in the same simple algebraic number field K or in Q. The default version chooses an algorithm automatically.

void ca_mat_mul_si(ca_mat_t B, const ca_mat_t A, slong c, ca_ctx_t ctx)
void ca_mat_mul_fmpz(ca_mat_t B, const ca_mat_t A, const fmpz_t c, ca_ctx_t ctx)
void ca_mat_mul_fmpq(ca_mat_t B, const ca_mat_t A, const fmpq_t c, ca_ctx_t ctx)
void ca_mat_mul_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)

Sets B to A multiplied by the scalar c.

void ca_mat_div_si(ca_mat_t B, const ca_mat_t A, slong c, ca_ctx_t ctx)
void ca_mat_div_fmpz(ca_mat_t B, const ca_mat_t A, const fmpz_t c, ca_ctx_t ctx)
void ca_mat_div_fmpq(ca_mat_t B, const ca_mat_t A, const fmpq_t c, ca_ctx_t ctx)
void ca_mat_div_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)

Sets B to A divided by the scalar c.

void ca_mat_add_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)
void ca_mat_sub_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)

Sets B to A plus or minus the scalar c (interpreted as a diagonal matrix).

void ca_mat_addmul_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)
void ca_mat_submul_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)

Sets the matrix B to B plus (or minus) the matrix A multiplied by the scalar c.

10.7.10 Powers

void ca_mat_sqr(ca_mat_t B, const ca_mat_t A, ca_ctx_t ctx)

Sets B to the square of A.

void ca_mat_pow_ui_binexp(ca_mat_t B, const ca_mat_t A, ulong exp, ca_ctx_t ctx)

Sets B to A raised to the power exp, evaluated using binary exponentiation.

10.7.11 Polynomial evaluation

void ca_mat_ca_poly_evaluate(ca_mat_t res, const ca_poly_t poly, slong len, const ca_mat_t A, ca_ctx_t ctx)
void ca_mat_ca_poly_evaluate(ca_mat_t res, const ca_poly_t poly, const ca_mat_t A, ca_ctx_t ctx)

Sets res to \(f(A)\) where \(f\) is the polynomial given by poly and A is a square matrix. Uses the Paterson-Stockmeyer algorithm.
10.7.12 Gaussian elimination and LU decomposition

truth_t ca_mat_find_pivot(slong *pivot_row, ca_mat_t mat, slong start_row, slong end_row, slong column, ca_ctx_t ctx)

Attempts to find a nonzero entry in mat with column index column and row index between start_row (inclusive) and end_row (exclusive).

If the return value is T_TRUE, such an element exists, and pivot_row is set to the row index. If the return value is T_FALSE, no such element exists (all entries in this part of the column are zero).

If the return value is T_UNKNOWN, it is unknown whether such an element exists (zero certification failed).

This function is destructive: any elements that are nontrivially zero but can be certified zero will be overwritten by exact zeros.

int ca_mat_lu_classical(slong *rank, slong *P, ca_mat_t LU, const ca_mat_t A, int rank_check, ca_ctx_t ctx)

int ca_mat_lu_recursive(slong *rank, slong *P, ca_mat_t LU, const ca_mat_t A, int rank_check, ca_ctx_t ctx)

int ca_mat_lu(slong *rank, slong *P, ca_mat_t LU, const ca_mat_t A, int rank_check, ca_ctx_t ctx)

Computes a generalized LU decomposition $A = PLU$ of a given matrix $A$, writing the rank of $A$ to rank.

If $A$ is a nonsingular square matrix, $LU$ will be set to a unit diagonal lower triangular matrix $L$ and an upper triangular matrix $U$ (the diagonal of $L$ will not be stored explicitly).

If $A$ is an arbitrary matrix of rank $r$, $U$ will be in row echelon form having $r$ nonzero rows, and $L$ will be lower triangular but truncated to $r$ columns, having implicit ones on the $r$ first entries of the main diagonal. All other entries will be zero.

If a nonzero value for rank_check is passed, the function will abandon the output matrix in an undefined state and set the rank to 0 if $A$ is detected to be rank-deficient.

The algorithm can fail if it fails to certify that a pivot element is zero or nonzero, in which case the correct rank cannot be determined. The return value is 1 on success and 0 on failure. On failure, the data in the output variables rank, P and LU will be meaningless.

The classical version uses iterative Gaussian elimination. The recursive version uses a block recursive algorithm to take advantage of fast matrix multiplication.

int ca_mat_fflu(slong *rank, slong *P, ca_mat_t LU, ca_t den, const ca_mat_t A, int rank_check, ca_ctx_t ctx)

Similar to ca_mat_lu(), but computes a fraction-free LU decomposition using the Bareiss algorithm. The denominator is written to den. Note that despite being “fraction-free”, this algorithm may introduce fractions due to incomplete symbolic simplifications.

truth_t ca_mat_nonsingular_lu(slong *P, ca_mat_t LU, const ca_mat_t A, ca_ctx_t ctx)

Wrapper for ca_mat_lu(). If $A$ can be proved to be invertible/nonsingular, returns T_TRUE and sets $P$ and $LU$ to a LU decomposition $A = PLU$. If $A$ can be proved to be singular, returns T_FALSE. If $A$ cannot be proved to be either singular or nonsingular, returns T_UNKNOWN. When the return value is T_FALSE or T_UNKNOWN, the LU factorization is not completed and the values of $P$ and $LU$ are arbitrary.

truth_t ca_mat_nonsingular_fflu(slong *P, ca_mat_t LU, ca_t den, const ca_mat_t A, ca_ctx_t ctx)

Wrapper for ca_mat_fflu(). Similar to ca_mat_nonsingular_lu(), but computes a fraction-free LU decomposition using the Bareiss algorithm. The denominator is written to den. Note that despite being “fraction-free”, this algorithm may introduce fractions due to incomplete symbolic simplifications.
10.7.13 Solving and inverse

truth_t ca_mat_inv(ca_mat_t X, const ca_mat_t A, ca_ctx_t ctx)
Determine if the square matrix $A$ is nonsingular, and if successful, sets $X = A^{-1}$ and returns $T\_TRUE$. Returns $T\_FALSE$ if $A$ is singular, and $T\_UNKNOWN$ if the rank of $A$ cannot be determined.

truth_t ca_mat_nonsingular_solve_adjugate(ca_mat_t X, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)
truth_t ca_mat_nonsingular_solve_fflu(ca_mat_t X, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)
truth_t ca_mat_nonsingular_solve_lu(ca_mat_t X, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)
Determine if the square matrix $A$ is nonsingular, and if successful, solves $AX = B$ and returns $T\_TRUE$. Returns $T\_FALSE$ if $A$ is singular, and $T\_UNKNOWN$ if the rank of $A$ cannot be determined.

void ca_mat_solve_tril_classical(ca_mat_t X, const ca_mat_t L, const ca_mat_t B, int unit, ca_ctx_t ctx)
void ca_mat_solve_tril_recursive(ca_mat_t X, const ca_mat_t L, const ca_mat_t B, int unit, ca_ctx_t ctx)
void ca_mat_solve_tril(ca_mat_t X, const ca_mat_t L, const ca_mat_t B, int unit, ca_ctx_t ctx)
void ca_mat_solve_triu_classical(ca_mat_t X, const ca_mat_t U, const ca_mat_t B, int unit, ca_ctx_t ctx)
void ca_mat_solve_triu_recursive(ca_mat_t X, const ca_mat_t U, const ca_mat_t B, int unit, ca_ctx_t ctx)
void ca_mat_solve_triu(ca_mat_t X, const ca_mat_t U, const ca_mat_t B, int unit, ca_ctx_t ctx)
Solves the lower triangular system $LX = B$ or the upper triangular system $UX = B$, respectively. It is assumed (not checked) that the diagonal entries are nonzero. If unit is set, the main diagonal of $L$ or $U$ is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The classical versions perform the computations iteratively while the recursive versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

void ca_mat_solve_fflu_precomp(ca_mat_t X, const slong *perm, const ca_mat_t A, const ca_t den, const ca_mat_t B, ca_ctx_t ctx)
void ca_mat_solve_lu_precomp(ca_mat_t X, const slong *P, const ca_mat_t LU, const ca_mat_t B, ca_ctx_t ctx)
Solves $AX = B$ given the precomputed nonsingular LU decomposition $A = PLU$ or fraction-free LU decomposition with denominator den. The matrices $X$ and $B$ are allowed to be aliased with each other, but $X$ is not allowed to be aliased with $LU$.

10.7.14 Rank and echelon form

int ca_mat_rank(slong *rank, const ca_mat_t A, ca_ctx_t ctx)
Computes the rank of the matrix $A$. If successful, returns 1 and writes the rank to rank. If unsuccessful, returns 0.

int ca_mat_rref_fflu(slong *rank, ca_mat_t R, const ca_mat_t A, ca_ctx_t ctx)
int ca_mat_rref_lu(slong *rank, ca_mat_t R, const ca_mat_t A, ca_ctx_t ctx)
int ca_mat_rref (slong *rank, ca_mat_t R, const ca_mat_t A, ca_ctx_t ctx)

Computes the reduced row echelon form (rref) of a given matrix. On success, sets R to the rref of A, writes the rank to rank, and returns 1. On failure to certify the correct rank, returns 0, leaving the data in rank and R meaningless.

The pdf version computes a fraction-free LU decomposition and then converts the output to rref form. The lu version computes a regular LU decomposition and then converts the output to rref form. The default version uses an automatic algorithm choice and may implement additional methods for special cases.

int ca_mat_right_kernel (ca_mat_t X, const ca_mat_t A, ca_ctx_t ctx)

Sets X to a basis of the right kernel (nullspace) of A. The output matrix X will be resized in-place to have a number of columns equal to the nullity of A. Returns 1 on success. On failure, returns 0 and leaves the data in X meaningless.

### 10.7.15 Determinant and trace

void ca_mat_trace (ca_t trace, const ca_mat_t mat, ca_ctx_t ctx)

Sets trace to the sum of the entries on the main diagonal of mat.

void ca_mat_det_berkowitz (ca_t det, const ca_mat_t A, ca_ctx_t ctx)

int ca_mat_det_lu (ca_t det, const ca_mat_t A, ca_ctx_t ctx)

int ca_mat_det_bareiss (ca_t det, const ca_mat_t A, ca_ctx_t ctx)

void ca_mat_det_cofactor (ca_t det, const ca_mat_t A, ca_ctx_t ctx)

void ca_mat_det (ca_t det, const ca_mat_t A, ca_ctx_t ctx)

Sets det to the determinant of the square matrix A. Various algorithms are available:

- The berkowitz version uses the division-free Berkowitz algorithm performing \(O(n^4)\) operations. Since no zero tests are required, it is guaranteed to succeed.
- The cofactor version performs cofactor expansion. This is currently only supported for matrices up to size 4.
- The lu and bareiss versions use rational LU decomposition and fraction-free LU decomposition (Bareiss algorithm) respectively, requiring \(O(n^3)\) operations. These algorithms can fail if zero certification fails (see ca_mat_nonsingular_lu()); they return 1 for success and 0 for failure. Note that the Bareiss algorithm, despite being “fraction-free”, may introduce fractions due to incomplete symbolic simplifications.

The default function chooses an algorithm automatically. It will, in addition, recognize trivially rational and integer matrices and evaluate those determinants using fmpq_mat_t or fmpz_mat_t.

The various algorithms can produce different symbolic forms of the same determinant. Which algorithm performs better depends strongly and sometimes unpredictably on the structure of the matrix.

void ca_mat_adjugate_cofactor (ca_mat_t adj, ca_t det, const ca_mat_t A, ca_ctx_t ctx)

void ca_mat_adjugate_charpoly (ca_mat_t adj, ca_t det, const ca_mat_t A, ca_ctx_t ctx)

void ca_mat_adjugate (ca_mat_t adj, ca_t det, const ca_mat_t A, ca_ctx_t ctx)

Sets adj to the adjugate matrix of A and det to the determinant of A, both computed simultaneously. The cofactor version uses cofactor expansion. The charpoly version computes and evaluates the characteristic polynomial. The default version uses an automatic algorithm choice.
10.7.16 Characteristic polynomial

void _ca_mat_charpoly_berkowitz(ca_ptr cp, const ca_mat_t mat, ca_ctx_t ctx)
void ca_mat_charpoly_berkowitz(ca_poly_t cp, const ca_mat_t mat, ca_ctx_t ctx)
int _ca_mat_charpoly_danilevsky(ca_ptr cp, const ca_mat_t mat, ca_ctx_t ctx)
int ca_mat_charpoly_danilevsky(ca_poly_t cp, const ca_mat_t mat, ca_ctx_t ctx)
void _ca_mat_charpoly(ca_ptr cp, const ca_mat_t mat, ca_ctx_t ctx)
void ca_mat_charpoly(ca_poly_t cp, const ca_mat_t mat, ca_ctx_t ctx)

Sets poly to the characteristic polynomial of mat which must be a square matrix. If the matrix has \( n \) rows, the underscore method requires space for \( n + 1 \) output coefficients.

The berkowitz version uses a division-free algorithm requiring \( O(n^4) \) operations. The danilevsky version only performs \( O(n^3) \) operations, but performs divisions and needs to check for zero which can fail. This version returns 1 on success and 0 on failure. The default version chooses an algorithm automatically.

int ca_mat_companion(ca_mat_t mat, const ca_poly_t poly, ca_ctx_t ctx)

Sets mat to the companion matrix of poly. This function verifies that the leading coefficient of poly is provably nonzero and that the output matrix has the right size, returning 1 on success. It returns 0 if the leading coefficient of poly cannot be proved nonzero or if the size of the output matrix does not match.

10.7.17 Eigenvalues and eigenvectors

int ca_mat_eigenvalues(ca_vec_t lambda, ulong *exp, const ca_mat_t mat, ca_ctx_t ctx)

Attempts to compute all complex eigenvalues of the given matrix mat. On success, returns 1 and sets lambda to the distinct eigenvalues with corresponding multiplicities in exp. The eigenvalues are returned in arbitrary order. On failure, returns 0 and leaves the values in lambda and exp arbitrary.

This function effectively computes the characteristic polynomial and then calls ca_poly_roots.

truth_t ca_mat_diagonalization(ca_mat_t D, ca_mat_t P, const ca_mat_t A, ca_ctx_t ctx)

Matrix diagonalization: attempts to compute a diagonal matrix \( D \) and an invertible matrix \( P \) such that \( A = PDP^{-1} \). Returns T_TRUE if \( A \) is diagonalizable and the computation succeeds, T_FALSE if \( A \) is provably not diagonalizable, and T_UNKNOWN if it is unknown whether \( A \) is diagonalizable. If the return value is not T_TRUE, the values in \( D \) and \( P \) are arbitrary.

10.7.18 Jordan canonical form

int ca_mat_jordan_blocks(ca_vec_t lambda, slong *num_blocks, slong *block_lambda, slong *block_size, const ca_mat_t A, ca_ctx_t ctx)

Computes the blocks of the Jordan canonical form of \( A \). On success, returns 1 and sets lambda to the unique eigenvalues of \( A \), sets num_blocks to the number of Jordan blocks, entry \( i \) of block_lambda to the index of the eigenvalue in Jordan block \( i \), and entry \( i \) of block_size to the size of Jordan block \( i \). On failure, returns 0, leaving arbitrary values in the output variables. The user should allocate space in block_lambda and block_size for up to \( n \) entries where \( n \) is the size of the matrix.

The Jordan form is unique up to the ordering of blocks, which is arbitrary.

void ca_mat_set_jordan_blocks(ca_mat_t mat, const ca_vec_t lambda, slong num_blocks, slong *block_lambda, slong *block_size, ca_ctx_t ctx)

Sets mat to the concatenation of the Jordan blocks given in lambda, num_blocks, block_lambda and block_size. See ca_mat_jordan_blocks() for an explanation of these variables.
int ca_mat_jordan_transformation(ca_mat_t mat, const ca_vec_t lambda, slong num_blocks, slong *block_lambda, slong *block_size, const ca_mat_t A, ca_ctx_t ctx)

Given the precomputed Jordan block decomposition \((\lambda, \text{num\_blocks}, \text{block\_lambda}, \text{block\_size})\) of the square matrix \(A\), computes the corresponding transformation matrix \(P\) such that \(A = PJP^{-1}\). On success, writes \(P\) to \(mat\) and returns 1. On failure, returns 0, leaving the value of \(mat\) arbitrary.

int ca_mat_jordan_form(ca_mat_t J, ca_mat_t P, const ca_mat_t A, ca_ctx_t ctx)

Computes the Jordan decomposition \(A = PJP^{-1}\) of the given square matrix \(A\). The user can pass \(NULL\) for the output variable \(P\), in which case only \(J\) is computed. On success, returns 1. On failure, returns 0, leaving the values of \(J\) and \(P\) arbitrary.

This function is a convenience wrapper around \(\text{ca\_mat\_jordan\_blocks()}\), \(\text{ca\_mat\_set\_jordan\_blocks()}\) and \(\text{ca\_mat\_jordan\_transformation()}\). For computations with the Jordan decomposition, it is often better to use those methods directly since they give direct access to the spectrum and block structure.

10.7.19 Matrix functions

int ca_mat_exp(ca_mat_t res, const ca_mat_t A, ca_ctx_t ctx)

Matrix exponential: given a square matrix \(A\), sets \(res\) to \(e^A\) and returns 1 on success. If unsuccessful, returns 0, leaving the values in \(res\) arbitrary.

This function uses Jordan decomposition. The matrix exponential always exists, but computation can fail if computing the Jordan decomposition fails.

truth_t ca_mat_log(ca_mat_t res, const ca_mat_t A, ca_ctx_t ctx)

Matrix logarithm: given a square matrix \(A\), sets \(res\) to a logarithm \(\log(A)\) and returns \(T\_TRUE\) on success. If \(A\) can be proved to have no logarithm, returns \(T\_FALSE\). If the existence of a logarithm cannot be proved, returns \(T\_UNKNOWN\).

This function uses the Jordan decomposition, and the branch of the matrix logarithm is defined by taking the principal values of the logarithms of all eigenvalues.

10.8 \texttt{ca\_ext.h} – real and complex extension numbers

A \texttt{ca\_ext\_t} represents a fixed real or complex number \(a\). The content of a \texttt{ca\_ext\_t} can be one of the following:

- An algebraic constant represented in canonical form by a \texttt{qqbar\_t} instance (example: \(i\), represented as the root of \(x^2 + 1\) with positive imaginary part).
- A constant of the form \(f(x_1, \ldots, x_n)\) where \(f\) is a builtin symbolic function and \(x_1, \ldots, x_n\) are given \texttt{ca\_t} instances.
- A builtin symbolic constant such as \(\pi\). (This is just a special case of the above with a zero-length argument list.)
- (Not implemented): a user-defined constant or function defined by suppling a function pointer for Arb numerical evaluation to specified precision.

The \texttt{ca\_ext\_t} structure is heavy-weight object, not just meant to act as a node in a symbolic expression. It will cache various data to support repeated computation with this particular number, including its numerical enclosure and number field data in the case of algebraic numbers.

Extension numbers are used internally by the \texttt{ca\_t} type to define the embeddings \(\mathbb{Q}(a) \rightarrow \mathbb{C}\) of formal fields. The user does not normally need to create \texttt{ca\_ext\_t} instances directly; the intended way for the user to work with the extension number \(a\) is to create a \texttt{ca\_t} representing the field element \(1 \cdot a\). The underlying \texttt{ca\_ext\_t} may be accessed to determine symbolic and numerical properties of this number.
Since extension numbers may depend recursively on nontrivial fields for function arguments, `ca_ext_t` operations require a `ca_ctx_t` context object.

### 10.8.1 Type and macros

For all types, a `type_t` is defined as an array of length one of type `type_struct`, permitting a `type_t` to be passed by reference.

**type** `ca_ext_struct`

**type** `ca_ext_t`

An extension number object contains a header, a hash value, data (a `qqbar_t` instance and an Antic `nf_t` in the case of algebraic numbers, and a pointer to arguments in the case of a symbolic function), and a cached `acb_t` enclosure (in the case of a `qqbar_t`, the enclosure internal to that structure is used).

**type** `ca_ext_ptr`

Alias for `ca_ext_struct *`.

**type** `ca_ext_srcptr`

Alias for `const ca_ext_struct *`.

**CA_EXT_HEAD(x)**

Accesses the head (a `calcium_func_code`) of `x`. This is `CA_QQBar` if `x` represents an algebraic constant in canonical form, and `CA_Exp`, `CA_Pi`, etc. for symbolic functions and constants.

**CA_EXT_HASH(x)**

Accesses the hash value of `x`.

**CA_EXT_QQBAR(x)**

Assuming that `x` represents an algebraic constant in canonical form, accesses this `qqbar_t` object.

**CA_EXT_QQBAR_NF(x)**

Assuming that `x` represents an algebraic constant in canonical form, accesses the corresponding Antic number field `nf_t` object.

**CA_EXT_FUNC_ARGS(x)**

Assuming that `x` represents a symbolic constant or function, accesses the argument list (as a `ca_ptr`).

**CA_EXT_FUNC_NARGS(x)**

Assuming that `x` represents a symbolic constant or function, accesses the number of function arguments.

**CA_EXT_FUNC_ENCLOSURE(x)**

Assuming that `x` represents a symbolic constant or function, accesses the cached `acb_t` numerical enclosure.

**CA_EXT_FUNC_PREC(x)**

Assuming that `x` represents a symbolic constant or function, accesses the working precision of the cached numerical enclosure.
10.8.2 Memory management

```c
void ca_ext_init_qqbar(ca_ext_t res, const qqbar_t x, ca_ctx_t ctx)
    Initializes res and sets it to the algebraic constant x.

void ca_ext_init_const(ca_ext_t res, calcium_func_code func, ca_ctx_t ctx)
    Initializes res and sets it to the constant defined by func (example: func = CA_Pi for \( x = \pi \)).

void ca_ext_init_fx(ca_ext_t res, calcium_func_code func, const ca_t x, ca_ctx_t ctx)
    Initializes res and sets it to the univariate function value \( f(x) \) where \( f \) is defined by func (example: func = CA_Exp for \( e^x \)).

void ca_ext_init_fxy(ca_ext_t res, calcium_func_code func, const ca_t x, const ca_t y, ca_ctx_t ctx)
    Initializes res and sets it to the bivariate function value \( f(x, y) \) where \( f \) is defined by func (example: func = CA_Pow for \( x^y \)).

void ca_ext_init_fxn(ca_ext_t res, calcium_func_code func, ca_srcptr x, slong nargs, ca_ctx_t ctx)
    Initializes res and sets it to the multivariate function value \( f(x_1, \ldots, x_n) \) where \( f \) is defined by func and \( n \) is given by nargs.

void ca_ext_init_set(ca_ext_t res, const ca_ext_t x, ca_ctx_t ctx)
    Initializes res and sets it to a copy of x.

void ca_ext_clear(ca_ext_t res, ca_ctx_t ctx)
    Clears res.
```

10.8.3 Structure

```c
slong ca_ext_nargs(const ca_ext_t x, ca_ctx_t ctx)
    Returns the number of function arguments of x. The return value is 0 for any algebraic constant and for any built-in symbolic constant such as \( \pi \).

void ca_ext_get_arg(ca_t res, const ca_ext_t x, slong i, ca_ctx_t ctx)
    Sets res to argument \( i \) (indexed from zero) of x. This calls flint_abort if \( i \) is out of range.

ulong ca_ext_hash(const ca_ext_t x, ca_ctx_t ctx)
    Returns a hash of the structural representation of x.

int ca_ext_equal_repr(const ca_ext_t x, const ca_ext_t y, ca_ctx_t ctx)
    Tests x and y for structural equality, returning 0 (false) or 1 (true).

int ca_ext_cmp_repr(const ca_ext_t x, const ca_ext_t y, ca_ctx_t ctx)
    Compares the representations of x and y in a canonical sort order, returning -1, 0 or 1. This only performs a structural comparison of the symbolic representations; the return value does not say anything meaningful about the numbers represented by x and y.
```

10.8.4 Input and output

```c
void ca_ext_print(const ca_ext_t x, ca_ctx_t ctx)
    Prints a description of x to standard output.
```
10.8.5 Numerical evaluation

void ca_ext_get_acb_raw(acb_t res, ca_ext_t x, slong prec, ca_ctx_t ctx)

Sets res to an enclosure of the numerical value of x. A working precision of prec bits is used for the evaluation, without adaptive refinement.

10.8.6 Cache

type ca_ext_cache_struct

type ca_ext_cache_t

Represents a set of structurally distinct ca_ext_t instances. This object contains an array of pointers to individual heap-allocated ca_ext_struct objects as well as a hash table for quick lookup.

void ca_ext_cache_init(ca_ext_cache_t cache, ca_ctx_t ctx)

Initializes cache for use.

void ca_ext_cache_clear(ca_ext_cache_t cache, ca_ctx_t ctx)

Clears cache, freeing the memory allocated internally.

ca_ext_ptr ca_ext_cache_insert(ca_ext_cache_t cache, const ca_ext_t x, ca_ctx_t ctx)

Adds x to cache without duplication. If a structurally identical instance already exists in cache, a pointer to that instance is returned. Otherwise, a copy of x is inserted into cache and a pointer to that new instance is returned.
10.9 ca_field.h – extension fields

A `ca_field_t` represents the parent field $K = \mathbb{Q}(a_1, \ldots, a_n)$ of a `ca_t` element. A `ca_field_t` contains a list of pointers to `ca_ext_t` objects as well as a reduction ideal.

The user does not normally need to create `ca_field_t` objects manually: a `ca_ctx_t` context object manages a cache of fields automatically.

Internally, three types of field representation are used:

- The trivial field $\mathbb{Q}$.
- An Antic number field $\mathbb{Q}(a)$ where $a$ is defined by a `qqbar_t`
- A generic field $\mathbb{Q}(a_1, \ldots, a_n)$ where $n \geq 1$, and $a_1$ is not defined by a `qqbar_t` if $n = 1$.

The field type mainly affects the internal storage of the field elements; the distinction is mostly transparent to the external interface.

10.9.1 Type and macros

For all types, a `type_t` is defined as an array of length one of type `type_struct`, permitting a `type_t` to be passed by reference.

`type ca_field_struct`

`type ca_field_t`

Represents a formal field.

`type ca_field_ptr`

Alias for `ca_field_struct *`.

`type ca_field_srcptr`

Alias for `const ca_field_struct *`.

`CA_FIELD_LENGTH(K)`

Accesses the number $n$ of extension numbers of $K$. This is 0 if $K = \mathbb{Q}$.

`CA_FIELD_EXT(K)`

Accesses the array of extension numbers as a `ca_ext_ptr`.

`CA_FIELD_EXT_ELEM(K, i)`

Accesses the extension number at position $i$ (indexed from zero) as a `ca_ext_t`.

`CA_FIELD_HASH(K)`

Accesses the hash value of $K$.

`CA_FIELD_IS_QQ(K)`

Returns whether $K$ is the trivial field $\mathbb{Q}$.

`CA_FIELD_IS_NF(K)`

Returns whether $K$ represents an Antic number field $K = \mathbb{Q}(a)$ where $a$ is represented by a `qqbar_t`.

`CA_FIELD_IS_GENERIC(K)`

Returns whether $K$ represents a generic field.

`CA_FIELD_NF(K)`

Assuming that $K$ represents an Antic number field $K = \mathbb{Q}(a)$, accesses the `nf_t` object representing this field.
CA_Field_Nf_Qqbar(K)

Assuming that K represents an Antic number field K = \mathbb{Q}(a), accesses the \texttt{qqbar\_t} object representing a.

CA_Field_Ideal(K)

Assuming that K represents a multivariate field, accesses the reduction ideal as a \texttt{fmpz\_mpoly\_t} array.

CA_Field_Ideal_Elem(K, i)

Assuming that K represents a multivariate field, accesses element i (indexed from zero) of the reduction ideal as a \texttt{fmpz\_mpoly\_t}.

CA_Field_Ideal_Length(K)

Assuming that K represents a multivariate field, accesses the number of polynomials in the reduction ideal.

CA_Field_Mctx(K, ctx)

Assuming that K represents a multivariate field, accesses the \texttt{fmpz\_mpoly\_ctx\_t} context object for multivariate polynomial arithmetic on the internal representation of elements in this field.

10.9.2 Memory management

void \textbf{ca_field_init_qq} (ca_field_t K, ca_ctx_t ctx)

Initializes K to represent the trivial field \mathbb{Q}.

void \textbf{ca_field_init_nf} (ca_field_t K, const qqbar_t x, ca_ctx_t ctx)

Initializes K to represent the algebraic number field \mathbb{Q}(x).

void \textbf{ca_field_init_const} (ca_field_t K, calcium_func_code func, ca_ctx_t ctx)

Initializes K to represent the field \mathbb{Q}(x) where x is a built-in constant defined by \texttt{func} (example: \texttt{func = CA_Pi} for x = \pi).

void \textbf{ca_field_init_fx} (ca_field_t K, calcium_func_code func, const ca_t x, ca_ctx_t ctx)

Initializes K to represent the field \mathbb{Q}(a) where a = f(x), given a number x and a built-in univariate function \texttt{func} (example: \texttt{func = CA_Exp} for e^x).

void \textbf{ca_field_init_fxy} (ca_field_t K, calcium_func_code func, const ca_t x, const ca_t y, ca_ctx_t ctx)

Initializes K to represent the field \mathbb{Q}(a,b) where a = f(x,y).

void \textbf{ca_field_init_multi} (ca_field_t K, slong len, ca_ctx_t ctx)

Initializes K to represent a multivariate field \mathbb{Q}(a_1, \ldots, a_n) in n extension numbers. The extension numbers must subsequently be assigned one by one using \texttt{ca_field_set_ext()}.  

void \textbf{ca_field_set_ext} (ca_field_t K, slong i, ca_ext_srcptr x\_index, ca_ctx_t ctx)

Sets the extension number at position i (here indexed from 0) of K to the generator of the field with index \texttt{x\_index} in \texttt{ctx}. (It is assumed that the generating field is a univariate field.)  

This only inserts a shallow reference: the field at index \texttt{x\_index} must be kept alive until K has been cleared.

void \textbf{ca_field_clear} (ca_field_t K, ca_ctx_t ctx)

Clears the field K. This does not clear the individual extension numbers, which are only held as references.
10.9.3 Input and output

```c
void ca_field_print(const ca_field_t K, ca_ctx_t ctx)
    Prints a description of the field K to standard output.
```

10.9.4 Ideal

```c
void ca_field_build_ideal(ca_field_t K, ca_ctx_t ctx)
    Given K with assigned extension numbers, builds the reduction ideal in-place.

void ca_field_build_ideal_erf(ca_field_t K, ca_ctx_t ctx)
    Builds relations for error functions present among the extension numbers in K. This heuristic adds
    relations that are consequences of the functional equations erf(x) = −erf(−x), erfc(x) = 1 − erf(x),
    erfi(x) = −i erf(ix).
```

10.9.5 Structure operations

```c
int ca_field_cmp(const ca_field_t K1, const ca_field_t K2, ca_ctx_t ctx)
    Compares the field objects K1 and K2 in a canonical sort order, returning -1, 0 or 1. This only
    performs a lexicographic comparison of the representations of K1 and K2; the return value does
    not say anything meaningful about the relative structures of K1 and K2 as mathematical fields.
```

10.9.6 Cache

```c
struct ca_field_cache_struct

typedef ca_field_cache_t
    Represents a set of distinct ca_field_t instances. This object contains an array of pointers to
    individual heap-allocated ca_field_struct objects as well as a hash table for quick lookup.

void ca_field_cache_init(ca_field_cache_t cache, ca_ctx_t ctx)
    Initializes cache for use.

void ca_field_cache_clear(ca_field_cache_t cache, ca_ctx_t ctx)
    Clears cache, freeing the memory allocated internally. This does not clear the individual extension
    numbers, which are only held as references.

c_field_ptr ca_field_cache_insert_ext(ca_field_cache_t cache, ca_ext_struct **x, slong len,
     ca_ctx_t ctx)
    Adds the field defined by the length-len list of extension numbers x to cache without duplication.
    If such a field already exists in cache, a pointer to that instance is returned. Otherwise, a field with
    extension numbers x is inserted into cache and a pointer to that new instance is returned. Upon
    insertion of a new field, the reduction ideal is constructed via ca_field_build_ideal().
```
10.10 fexpr.h – flat-packed symbolic expressions

This module supports working with symbolic expressions.

10.10.1 Introduction

Formally, a symbolic expression is either:

- An atom, being one of the following:
  - An integer, for example 0 or -34.
  - A symbol, for example $x$, $\text{Mul}$, $\text{SomeUserNamedSymbol}$. Symbols should be valid C identifiers (containing only the characters A-Z, a-z, 0-9, _, and not starting with a digit).
  - A string, for example "Hello, world!". For the moment, we only consider ASCII strings, but there is no obstacle in principle to supporting UTF-8.
- A non-atomic expression, representing a function call $e_0(e_1, \ldots, e_n)$ where $e_0, \ldots, e_n$ are symbolic expressions.

The meaning of an expression depends on the interpretation of symbols in a given context. For example, with a standard interpretation (used within Calcium) of the symbols $\text{Mul}$, $\text{Add}$ and $\text{Neg}$, the expression $\text{Mul}(3, \text{Add}(\text{Neg}(x), y))$ encodes the formula $3 \cdot (-(x) + y)$ where $x$ and $y$ are symbolic variables. See fexpr_builtin.h – builtin symbols for documentation of builtin symbols.

Computing and embedding data

Symbolic expressions are usually not the best data structure to use directly for heavy-duty computations. Functions acting on symbolic expressions will typically convert to a dedicated data structure (e.g. polynomials) internally and (optionally) convert the final result back to a symbolic expression.

Symbolic expressions do not allow embedding arbitrary binary objects such as Flint/Arb/Antic/Calcium types as atoms. This is done on purpose to make symbolic expressions easy to use as a data exchange format. To embed an object in an expression, one has the following options:

- Represent the object structurally using atoms supported natively by symbolic expressions (for example, an integer polynomial can be represented as a list of coefficients or as an arithmetic expression tree).
- Introduce a dummy symbol to represent the object, maintaining an external translation table mapping this symbol to the intended value.
- Encode the object using a string or symbol name. This is generally not recommended, as it requires parsing; properly used, symbolic expressions have the benefit of being able to represent the parsed structure.

Flat-packed representation

Symbolic expressions are often implemented using trees of pointers (often together with hash tables for uniqueness), requiring some form of memory management. The fexpr_t type, by contrast, stores a symbolic expression using a “flat-packed” representation without internal pointers. The expression data is just an array of words (ulong). The first word is a header encoding type information (whether the expression is a function call or an atom, and the type of the atom) and the total number of words in the expression. For atoms, the data is stored either in the header word itself (small integers and short symbols/strings) or in the following words. For function calls, the header is followed by the expressions $e_0, \ldots, e_n$ packed continguously in memory.

Pros:

- Memory management is trivial.
• Copying an expression is just copying an array of words.
• Comparing expressions for equality is just comparing arrays of words.
• Merging expressions is basically just concatenating arrays of words.
• Expression data can be shared freely in binary form between threads and even between machines (as long as all machines have the same word size and endianness).

Cons:
• Repeated instances of the same subexpression cannot share memory (a workaround is to introduce local dummy symbols for repeated subexpressions).
• Extracting a subexpression for modification generally requires making a complete copy of that subexpression (however, for read-only access to subexpressions, one can use “view” expressions which have zero overhead).
• Manipulating a part of an expression generally requires rebuilding the whole expression.
• Building an expression incrementally is typically $O(n^2)$. As a workaround, it is a good idea to work with balanced (low-depth) expressions and try to construct an expression in one go (for example, to create a sum, create a single `Add` expression with many arguments instead of chaining binary `Add` operations).

10.10.2 Types and macros

type fexpr_struct

type fexpr_t
    An fexpr_struct consists of a pointer to an array of words along with a record of the number of allocated words.
    An fexpr_t is defined as an array of length one of type fexpr_struct, permitting an fexpr_t to be passed by reference.

type fexpr_ptr
    Alias for fexpr_struct *, used for arrays of expressions.

type fexpr_srcptr
    Alias for const fexpr_struct *, used for arrays of expressions when passed as constant input to functions.

type fexpr_vec_struct

type fexpr_vec_t
    A type representing a vector of expressions with managed length. The structure contains an fexpr_ptr entries for the entries, an integer length (the size of the vector), and an integer alloc (the number of allocated entries).

fexpr_vec_entry(vec, i)
    Returns a pointer to entry i in the vector vec.
10.10.3 Memory management

void fexpr_init(fexpr_t expr)
    Initializes expr for use. Its value is set to the atomic integer 0.

void fexpr_clear(fexpr_t expr)
    Clears expr, freeing its allocated memory.

fexpr_ptr _fexpr_vec_init(slong len)
    Returns a heap-allocated vector of len initialized expressions.

void _fexpr_vec_clear(fexpr_ptr vec, slong len)
    Clears the len expressions in vec and frees vec itself.

void fexpr_fit_size(fexpr_t expr, slong size)
    Ensures that expr has room for size words.

void fexpr_set(fexpr_t res, const fexpr_t expr)
    Sets res to the a copy of expr.

void fexpr_swap(fexpr_t a, fexpr_t b)
    Swaps a and b efficiently.

10.10.4 Size information

slong fexpr_depth(const fexpr_t expr)
    Returns the depth of expr as a symbolic expression tree.

slong fexpr_num_leaves(const fexpr_t expr)
    Returns the number of leaves (atoms, counted with repetition) in the expression expr.

slong fexpr_size(const fexpr_t expr)
    Returns the number of words in the internal representation of expr.

slong fexpr_size_bytes(const fexpr_t expr)
    Returns the number of bytes in the internal representation of expr. The count excludes the size of the structure itself. Add sizeof(fexpr_struct) to get the size of the object as a whole.

slong fexpr_allocated_bytes(const fexpr_t expr)
    Returns the number of allocated bytes in the internal representation of expr. The count excludes the size of the structure itself. Add sizeof(fexpr_struct) to get the size of the object as a whole.

10.10.5 Comparisons

int fexpr_equal(const fexpr_t a, const fexpr_t b)
    Checks if a and b are exactly equal as expressions.

int fexpr_equal_si(const fexpr_t expr, slong c)

int fexpr_equal_ui(const fexpr_t expr, ulong c)
    Checks if expr is an atomic integer exactly equal to c.

ulong fexpr_hash(const fexpr_t expr)
    Returns a hash of the expression expr.

int fexpr_cmp_fast(const fexpr_t a, const fexpr_t b)
    Compares a and b using an ordering based on the internal representation, returning -1, 0 or 1. This can be used, for instance, to maintain sorted arrays of expressions for binary search; the sort order has no mathematical significance.
10.10.6 Atoms

```c
int fexpr_is_integer(const fexpr_t expr)
    Returns whether expr is an atomic integer.

int fexpr_is_symbol(const fexpr_t expr)
    Returns whether expr is an atomic symbol.

int fexpr_is_string(const fexpr_t expr)
    Returns whether expr is an atomic string.

int fexpr_is_atom(const fexpr_t expr)
    Returns whether expr is any atom.

void fexpr_zero(fexpr_t res)
    Sets res to the atomic integer 0.

int fexpr_is_zero(const fexpr_t expr)
    Returns whether expr is the atomic integer 0.

int fexpr_is_neg_integer(const fexpr_t expr)
    Returns whether expr is any negative atomic integer.

void fexpr_set_si(fexpr_t res, slong c)
void fexpr_set_ui(f expr_t res, ulong c)
void fexpr_set_fmpz(fexpr_t res, const fmpz_t c)
    Sets res to the atomic integer c.

int fexpr_get_fmpz(fmpz_t res, const fexpr_t expr)
    Sets res to the atomic integer in expr. This aborts if expr is not an atomic integer.

void fexpr_set_symbol_builtin(fexpr_t res, slong id)
    Sets res to the builtin symbol with internal index id (see fexpr_builtin.h - builtin symbols).

int fexpr_is_builtin_symbol(const fexpr_t expr, slong id)
    Returns whether expr is the builtin symbol with index id (see fexpr_builtin.h - builtin symbols).

int fexpr_is_any_builtin_symbol(const fexpr_t expr)
    Returns whether expr is any builtin symbol (see fexpr_builtin.h - builtin symbols).

void fexpr_set_symbol_str(fexpr_t res, const char *s)
    Sets res to the symbol given by s.

char *fexpr_get_symbol_str(const fexpr_t expr)
    Returns the symbol in expr as a string. The string must be freed with flint_free(). This aborts if expr is not an atomic symbol.

void fexpr_set_string(fexpr_t res, const char *s)
    Sets res to the atomic string s.

char *fexpr_get_string(const fexpr_t expr)
    Assuming that expr is an atomic string, returns a copy of this string. The string must be freed with flint_free().
```
10.10.7 Input and output

void \texttt{fexpr\_write}(\texttt{calcium\_stream\_t} \texttt{stream}, \texttt{const fexpr\_t} \texttt{expr})
    \begin{quote}
    Writes \texttt{expr} to \texttt{stream}.
    \end{quote}

void \texttt{fexpr\_print}(\texttt{const fexpr\_t} \texttt{expr})
    \begin{quote}
    Prints \texttt{expr} to standard output.
    \end{quote}

char *\texttt{fexpr\_get\_str}(\texttt{const fexpr\_t} \texttt{expr})
    \begin{quote}
    Returns a string representation of \texttt{expr}. The string must be freed with \texttt{flint\_free()}.
    Warning: string literals appearing in expressions are currently not escaped.
    \end{quote}

10.10.8 LaTeX output

void \texttt{fexpr\_write\_latex}(\texttt{calcium\_stream\_t} \texttt{stream}, \texttt{const fexpr\_t} \texttt{expr}, \texttt{ulong} \texttt{flags})
    \begin{quote}
    Writes the LaTeX representation of \texttt{expr} to \texttt{stream}.
    \end{quote}

void \texttt{fexpr\_print\_latex}(\texttt{const fexpr\_t} \texttt{expr}, \texttt{ulong} \texttt{flags})
    \begin{quote}
    Prints the LaTeX representation of \texttt{expr} to standard output.
    \end{quote}

char *\texttt{fexpr\_get\_str\_latex}(\texttt{const fexpr\_t} \texttt{expr}, \texttt{ulong} \texttt{flags})
    \begin{quote}
    Returns a string of the LaTeX representation of \texttt{expr}. The string must be freed with \texttt{flint\_free()}.
    Warning: string literals appearing in expressions are currently not escaped.
    \end{quote}

The \texttt{flags} parameter allows specifying options for LaTeX output. The following flags are supported:

\texttt{FEXPR\_LATEX\_SMALL}
    \begin{quote}
    Generate more compact formulas, most importantly by printing fractions inline as \(\frac{p}{q}\) instead of as \(\frac{p}{q}\). This flag is automatically activated within subscripts and superscripts and in certain other parts of formulas.
    \end{quote}

\texttt{FEXPR\_LATEX\_LOGIC}
    \begin{quote}
    Use symbols for logical operators such as Not, And, Or, which by default are rendered as words for legibility.
    \end{quote}

10.10.9 Function call structure

\texttt{slong fexpr\_nargs}(\texttt{const fexpr\_t} \texttt{expr})
    \begin{quote}
    Returns the number of arguments \(n\) in the function call \(f(e_1,\ldots,e_n)\) represented by \texttt{expr}. If \texttt{expr} is an atom, returns -1.
    \end{quote}

void \texttt{fexpr\_func}(\texttt{fexpr\_t} \texttt{res}, \texttt{const fexpr\_t} \texttt{expr})
    \begin{quote}
    Assuming that \texttt{expr} represents a function call \(f(e_1,\ldots,e_n)\), sets \texttt{res} to the function expression \(f\).
    \end{quote}

void \texttt{fexpr\_view\_func}(\texttt{fexpr\_t} \texttt{view}, \texttt{const fexpr\_t} \texttt{expr})
    \begin{quote}
    As \texttt{fexpr\_func()}, but sets \texttt{view} to a shallow view instead of copying the expression. The variable \texttt{view} must not be initialized before use or cleared after use, and \texttt{expr} must not be modified or cleared as long as \texttt{view} is in use.
    \end{quote}

void \texttt{fexpr\_arg}(\texttt{fexpr\_t} \texttt{res}, \texttt{const fexpr\_t} \texttt{expr}, \texttt{slong} \texttt{i})
    \begin{quote}
    Assuming that \texttt{expr} represents a function call \(f(e_1,\ldots,e_n)\), sets \texttt{res} to the argument \(e_{i+1}\). Note that indexing starts from 0. The index must be in bounds, with \(0 \leq i < n\).
    \end{quote}

void \texttt{fexpr\_view\_arg}(\texttt{fexpr\_t} \texttt{view}, \texttt{const fexpr\_t} \texttt{expr}, \texttt{slong} \texttt{i})
    \begin{quote}
    As \texttt{fexpr\_arg()}, but sets \texttt{view} to a shallow view instead of copying the expression. The variable \texttt{view} must not be initialized before use or cleared after use, and \texttt{expr} must not be modified or cleared as long as \texttt{view} is in use.
void fexpr_view_next(fexpr_t view)

    Assuming that view is a shallow view of a function argument \( e_i \) in a function call \( f(e_1, \ldots, e_n) \), sets view to a view of the next argument \( e_{i+1} \). This function can be called when view refers to the last argument \( e_n \), provided that view is not used afterwards. This function can also be called when view refers to the function \( f \), in which case it will make view point to \( e_1 \).

int fexpr_is_builtin_call(const fexpr_t expr, slong id)

    Returns whether expr has the form \( f(\ldots) \) where \( f \) is a builtin function defined by id (see fexpr_builtin.h – builtin symbols).

int fexpr_is_any_builtin_call(const fexpr_t expr)

    Returns whether expr has the form \( f(\ldots) \) where \( f \) is any builtin function (see fexpr_builtin.h – builtin symbols).

10.10.10 Composition

void fexpr_call0(fexpr_t res, const fexpr_t f)
void fexpr_call1(fexpr_t res, const fexpr_t f, const fexpr_t x1)
void fexpr_call2(fexpr_t res, const fexpr_t f, const fexpr_t x1, const fexpr_t x2)
void fexpr_call3(fexpr_t res, const fexpr_t f, const fexpr_t x1, const fexpr_t x2, const fexpr_t x3)
void fexpr_call4(fexpr_t res, const fexpr_t f, const fexpr_t x1, const fexpr_t x2, const fexpr_t x3, const fexpr_t x4)
void fexpr_call_vec(fexpr_t res, const fexpr_t f, fexpr_srcptr args, slong len)

    Creates the function call \( f(x_1, \ldots, x_n) \). The vec version takes the arguments as an array args and \( n \) is given by len. Warning: aliasing between inputs and outputs is not implemented.

void fexpr_call_builtin1(fexpr_t res, slong f, const fexpr_t x1)
void fexpr_call_builtin2(fexpr_t res, slong f, const fexpr_t x1, const fexpr_t x2)

    Creates the function call \( f(x_1, \ldots, x_n) \), where \( f \) defines a builtin symbol.

10.10.11 Subexpressions and replacement

int fexpr_contains(const fexpr_t expr, const fexpr_t x)

    Returns whether expr contains the expression \( x \) as a subexpression (this includes the case where expr and \( x \) are equal).

int fexpr_replace(fexpr_t res, const fexpr_t expr, const fexpr_t x, const fexpr_t y)

    Sets res to the expression expr with all occurrences of the subexpression \( x \) replaced by the expression \( y \). Returns a boolean value indicating whether any replacements have been performed. Aliasing is allowed between res and expr but not between res and \( x \) or \( y \).

int fexpr_replace2(fexpr_t res, const fexpr_t expr, const fexpr_t x1, const fexpr_t y1, const fexpr_t x2, const fexpr_t y2)

    Like fexpr_replace(), but simultaneously replaces \( x1 \) by \( y1 \) and \( x2 \) by \( y2 \).

int fexpr_replace_vec(fexpr_t res, const fexpr_t expr, const fexpr_vec_t xs, const fexpr_vec_t ys)

    Sets res to the expression expr with all occurrences of the subexpressions given by entries in \( x_s \) replaced by the corresponding expressions in ys. It is required that \( x_s \) and \( y_s \) have the same length. Returns a boolean value indicating whether any replacements have been performed. Aliasing is allowed between res and expr but not between res and the entries of xs or ys.
10.10.12 Arithmetic expressions

void **fexpr_set_fmpq** (*fexpr_t res*, const *fmpq_t x)

Sets res to the rational number x. This creates an atomic integer if the denominator of x is one, and otherwise creates a division expression.

void **fexpr_set_arf** (*fexpr_t res*, const *arf_t x)

void **fexpr_set_d** (*fexpr_t res*, double x)

Sets res to an expression for the value of the floating-point number x. NaN is represented as Undefined. For a regular value, this creates an atomic integer or a rational fraction if the exponent is small, and otherwise creates an expression of the form $\text{Mul}(m, \text{Pow}(2, e))$.

void **fexpr_set_re_im_d** (*fexpr_t res*, double x, double y)

Sets res to an expression for the complex number with real part x and imaginary part y.

void **fexpr_neg** (*fexpr_t res*, const *fexpr_t a)

void **fexpr_add** (*fexpr_t res*, const *fexpr_t a, const *fexpr_t b)

void **fexpr_sub** (*fexpr_t res*, const *fexpr_t a, const *fexpr_t b)

void **fexpr_mul** (*fexpr_t res*, const *fexpr_t a, const *fexpr_t b)

void **fexpr_div** (*fexpr_t res*, const *fexpr_t a, const *fexpr_t b)

void **fexpr_pow** (*fexpr_t res*, const *fexpr_t a, const *fexpr_t b)

Constructs an arithmetic expression with given arguments. No simplifications whatsoever are performed.

int **fexpr_is_arithmetic_operation** (const *fexpr_t expr)

Returns whether expr is of the form $f(e_1, \ldots, e_n)$ where f is one of the arithmetic operators Pos, Neg, Add, Sub, Mul, Div.

void **fexpr_arithmetic_nodes** (*fexpr_vec_t nodes*, const *fexpr_t expr)

Sets nodes to a vector of subexpressions of expr such that expr is an arithmetic expression with nodes as leaves. More precisely, expr will be constructed out of nested application of the arithmetic operators Pos, Neg, Add, Sub, Mul, Div with integers and expressions in nodes as leaves. Powers $\text{Pow}$ with an atomic integer exponent are also allowed. The nodes are output without repetition but are not automatically sorted in a canonical order.

int **fexpr_get_fmpz_mpoly_q** (*fmpz_mpoly_q_t res*, const *fexpr_t expr*, const *fexpr_vec_t vars*, const *fmpz_mpoly_ctx_t ctx*)

Sets res to the expression expr as a formal rational function of the subexpressions in vars. The vector vars must have the same length as the number of variables specified in ctx. To build vars automatically for a given expression, fexpr_arithmetic_nodes() may be used.

Returns 1 on success and 0 on failure. Failure can occur for the following reasons:

- A subexpression is encountered that cannot be interpreted as an arithmetic operation and does not appear (exactly) in vars.
- Overflow (too many terms or too large exponent).
- Division by zero (a zero denominator is encountered).

It is important to note that this function views expr as a formal rational function with vars as formal indeterminates. It does thus not check for algebraic relations between vars and can implicitly divide by zero if vars are not algebraically independent.

void **fexpr_set_fmpz_mpoly** (*fexpr_t res*, const *fmpz_mpoly_t poly*, const *fexpr_vec_t vars*, const *fmpz_mpoly_ctx_t ctx*)

void **fexpr_set_fmpz_mpoly_q** (*fexpr_t res*, const *fmpz_mpoly_q_t frac*, const *fexpr_vec_t vars*, const *fmpz_mpoly_ctx_t ctx*)

Sets res to an expression for the multivariate polynomial poly (or rational function frac), using the expressions in vars as the variables. The length of vars must agree with the number of variables in ctx. If NULL is passed for vars, a default choice of symbols is used.
int fexpr_expanded_normal_form(fexpr_t res, const fexpr_t expr, ulong flags)

Sets res to expr converted to expanded normal form viewed as a formal rational function with its non-arithmetic subexpressions as terminal nodes. This function first computes nodes with fexpr_arithmetic_nodes(), sorts the nodes, evaluates to a rational function with fexpr_get_fmpz_mpoly_q(), and then converts back to an expression with fexpr_set_fmpz_mpoly_q(). Optional flags are reserved for future use.

10.10.13 Vectors

void fexpr_vec_init(fexpr_vec_t vec, slong len)

Initializes vec to a vector of length len. All entries are set to the atomic integer 0.

void fexpr_vec_clear(fexpr_vec_t vec)

Clears the vector vec.

void fexpr_vec_print(const fexpr_vec_t vec)

Prints vec to standard output.

void fexpr_vec_swap(fexpr_vec_t x, fexpr_vec_t y)

Swaps x and y efficiently.

void fexpr_vec_fit_length(fexpr_vec_t vec, slong len)

Ensures that vec has space for len entries.

void fexpr_vec_set(fexpr_vec_t dest, const fexpr_vec_t src)

Sets dest to a copy of src.

void fexpr_vec_append(fexpr_vec_t vec, const fexpr_t expr)

Appends expr to the end of the vector vec.

slong fexpr_vec_insert_unique(fexpr_vec_t vec, const fexpr_t expr)

Inserts expr without duplication into vec, returning its position. If this expression already exists, vec is unchanged. If this expression does not exist in vec, it is appended.

void fexpr_vec_set_length(fexpr_vec_t vec, slong len)

Sets the length of vec to len, truncating or zero-extending as needed.

void _fexpr_vec_sort_fast(fexpr_ptr vec, slong len)

Sorts the len entries in vec using the comparison function fexpr_cmp_fast().
10.11 fexpr_builtin.h – builtin symbols

This module defines symbol names with a predefined meaning for use in symbolic expressions. These symbols will eventually all support LaTeX rendering as well as symbolic and numerical evaluation (where applicable).

By convention, all builtin symbol names are at least two characters long and start with an uppercase letter. Single-letter symbol names and symbol names beginning with a lowercase letter are reserved for variables.

For any builtin symbol name Symbol, the header file fexpr_builtin.h defines a C constant FEXPR_Symbol as an index to a builtin symbol table. The symbol will be documented as Symbol below.

10.11.1 C helper functions

slong fexpr_builtin_lookup(const char *s)
    Returns the internal index used to encode the builtin symbol with name s in expressions. If s is not the name of a builtin symbol, returns -1.

const char *fexpr_builtin_name(slong n)
    Returns a read-only pointer for a string giving the name of the builtin symbol with index n.

slong fexpr_builtin_length(void)
    Returns the number of builtin symbols.

10.11.2 Variables and iteration

Expressions involving the following symbols have a special role in binding variables.

For
    Generator expression. This is a syntactical construct which does not represent a mathematical object on its own. In general, For(x, ...) defines the symbol x as a locally bound variable in the scope of the parent expression. The following arguments ... specify an evaluation range, set or point. Their interpretation depends on the parent operator. The following cases are possible.

Case 1: For(x, S) specifies iteration or comprehension for x ranging over the values of the set S. This interpretation is used in operators that aggregate values over a set. The For expression may be followed by a filter predicate P(x) restricting the range to a subset of S. Examples:

    Set(f(x), For(x, S)) denotes \{f(x) : x \in S\}.
    Set(f(x), For(x, S), P(x)) denotes \{f(x) : x \in S \text{ and } P(x)\}.
    Sum(f(x), For(x, S)) denotes \sum_{x \in S} f(x).
    Sum(f(x), For(x, S), P(x)) denotes \sum_{x \in S, P(x)} f(x).

Case 2: For(x, a, b) specifies that x ranges between the endpoints a and b in the context of Sum, Product, Integral, and similar operators. Examples:

    Sum(f(n), For(n, a, b)) denotes \sum_{n=a}^{b} f(n). The iteration is empty if b < a.
    Integral(f(x), For(x, a, b)) denotes \int_{a}^{b} f(x)dx, where the integral follows a straight-line path from a to b. Swapping a and b negates the value.

Case 3: For(x, a) specifies that x approaches the point a in the context of Limit-type operator, or differentiation with respect to x at the point a in the context of a Derivative-type operator. Examples:

    Derivative(f(x), For(x, a)) denotes f'(a).
    Limit(f(x), For(x, a)) denotes \lim_{x \to a} f(x).
Case 4: \(\text{For}(x, a, n)\) specifies differentiation with respect to \(x\) at the point \(a\) to order \(n\) in the context of a Derivative-type operator. Examples:

\[
\text{Derivative}(f(x), \text{For}(x, a, n)) \text{ denotes } f^{(n)}(a).
\]

Where

Where\((f(x), \text{Def}(x, a))\) defines the symbol \(x\) as an alias for the expression \(a\) and evaluates the expression \(f(x)\) with this bound value of \(x\). This is equivalent to \(f(a)\). This may be rendered as \(f(x)\) where \(x = a\).

Where\((f(x), \text{Def}(f(t), a))\) defines the symbol \(f\) as a function mapping the dummy variable \(t\) to \(a\).

Where\((\text{Add}(a, b), \text{Def}(\text{Tuple}(a, b), T))\) is a destructuring assignment.

Def

Definition expression. This is a syntactical construct which does not represent a mathematical object on its own. The Def expression is used only within a Where-expression; see that documentation of that symbol for more examples.

Def\((x, a)\) defines the symbol \(x\) as an alias for the expression \(a\).

Def\((f(x, y, z), a)\) defines the symbol \(f\) as a function of three variables. The dummy variables \(x, y\) and \(z\) may appear within the expression \(a\).

Fun

Fun\((x, \text{expr})\) defines an anonymous univariate function mapping the symbol \(x\) to the expression \(\text{expr}\). The symbol \(x\) becomes locally bound within this Fun expression.

Step

Repeat

10.11.3 Booleans and logic

Equal

Equal\((a, b)\), signifying \(a = b\), is True if \(a\) and \(b\) represent the same object, and False otherwise. This operator can be called with any number of arguments, in which case it evaluates whether all arguments are equal.

NotEqual

NotEqual\((a, b)\), signifying \(a \neq b\), is equivalent to \(\text{Not} (\text{Equal}(a, b))\).

Same

Same\((a, b)\) gives a (or equivalently \(b\)) if \(a\) and \(b\) represent the same object, and Undefined otherwise. This can be used to assert or emphasize that two expressions represent the same value within a formula. This operator can be called with any number of arguments, in which case it asserts that all arguments are equal.

True

True is a logical constant.

False

False is a logical constant.

Not

Not\((x)\) is the logical negation of \(x\).

And

And\((x, y)\) is the logical AND of \(x\) and \(y\). This function can be called with any number of arguments.
Or

\( \text{Or}(x, y) \) is the logical OR of \( x \) and \( y \). This function can be called with any number of arguments.

Equivalent

\( \text{Equivalent}(x, y) \) denotes the logical equivalence \( x \Leftrightarrow y \). Semantically, this is the same as \( \text{Equal} \) called with logical arguments.

Implies

\( \text{Implies}(x, y) \) denotes the logical implication \( x \implies y \).

Exists

Existence quantifier.

\( \text{Exists}(f(x), \text{For}(x, S)) \) denotes \( f(x) \) for some \( x \in S \).

\( \text{Exists}(f(x), \text{For}(x, S), P(x)) \) denotes \( f(x) \) for some \( x \in S \) with \( P(x) \).

All

Universal quantifier.

\( \text{All}(f(x), \text{For}(x, S)) \) denotes \( f(x) \) for all \( x \in S \).

\( \text{All}(f(x), \text{For}(x, S), P(x)) \) denotes \( f(x) \) for all \( x \in S \) with \( P(x) \).

Cases

\( \text{Cases}(\text{Case}(f(x), P(x)), \text{Case}(g(x), \text{Otherwise}))) \) denotes:

\[
\begin{align*}
  \{ & f(x), \ P(x) \\
  & g(x), \ \text{otherwise} \}
\end{align*}
\]

\( \text{Cases}(\text{Case}(f(x), P(x)), \text{Case}(g(x), Q(x)), \text{Case}(h(x), \text{Otherwise}))) \) denotes:

\[
\begin{align*}
  \{ & f(x), \ P(x) \\
  & g(x), \ Q(x) \\
  & h(x), \ \text{otherwise} \}
\end{align*}
\]

If both \( P(x) \) and \( Q(x) \) are true simultaneously, no ordering is implied; it is assumed that \( f(x) \) and \( g(x) \) give the same value for any such \( x \). More generally, this operator can be called with any number of case distinctions.

If the \( \text{Otherwise} \) case is omitted, the result is undefined if neither predicate is true.

Case

See Cases.

Otherwise

See Cases.

10.11.4 Tuples, lists and sets

Tuple

List

Set

Item

Element

NotElement
EqualAndElement
Length
Cardinality
Concatenation
Union
Intersection
SetMinus
Subset
SubsetEqual
CartesianProduct
CartesianPower
Subsets

Subsets(S) is the power set $\mathcal{P}(S)$ comprising all subsets of the set $S$.

Sets
Sets is the class Sets of all sets.

Tuples
Tuples is the class of all tuples.
Tuples(S) is the set of all tuples with elements in the set $S$.
Tuples(S, n) is the set of all length-n tuples with elements in the set $S$.

10.11.5 Numbers and arithmetic

Undefined

Undefined
Undefined is the special value u (undefined).

Particular numbers

Pi
Pi is the constant $\pi$.

NumberI
NumberI is the imaginary unit $i$. The verbose name leaves i and I to be used as a variable names.

NumberE
NumberE is the base of the natural logarithm $e$. The verbose name leaves e and E to be used as a variable names.

GoldenRatio
GoldenRatio is the golden ratio $\varphi$.

Euler
Euler is Euler’s constant $\gamma$. 
CatalanConstant

CatalanConstant is Catalan’s constant $G$.

KhinchinConstant

KhinchinConstant is Khinchin’s constant $K$.

GlaisherConstant

GlaisherConstant is Glaisher’s constant $A$.

RootOfUnity

RootOfUnity(n) is the principal complex $n$-th root of unity $\zeta_n = e^{2\pi i/n}$.

RootOfUnity(n, k) is the complex $n$-th root of unity $\zeta_n^k$.

Number constructors

Remark: the rational number with numerator $p$ and denominator $q$ can be constructed as Div(p, q).

Decimal

Decimal(str) gives the rational number specified by the string str in ordinary decimal floating-point notation (for example -3.25e-725).

Arithmetic operations

Pos
Neg
Add
Sub
Mul
Div
Pow
Sqrt
Root
Inequalities

Less
LessEqual
Greater
GreaterEqual
EqualNearestDecimal

Sets of numbers

NN
NN is the set of natural numbers (including 0), N.

ZZ
ZZ is the set of integers, Z.

QQ
QQ is the set of rational numbers, Q.

RR
RR is the set of real numbers, R.

CC
CC is the set of complex numbers, C.

Primes
Primes is the set of positive prime numbers, P

IntegersGreaterEqual
IntegersGreaterEqual(x), given an extended real number x, gives the set \( \mathbb{Z}_{\geq x} \) of integers greater than or equal to x.

IntegersLessEqual
IntegersLessEqual(x), given an extended real number x, gives the set \( \mathbb{Z}_{\leq x} \) of integers less than or equal to x.

Range
Range(a, b), given integers a and b, gives the set \{a, a+1, ..., b\} of integers between a and b. This is the empty set if a is greater than b.

AlgebraicNumbers
The set of complex algebraic numbers \( \overline{\mathbb{Q}} \).

RealAlgebraicNumbers
The set of real algebraic numbers \( \overline{\mathbb{Q}}_R \).

Interval
Interval(a, b), given extended real numbers a and b, gives the closed interval \([a, b]\).

OpenInterval
OpenInterval(a, b), given extended real numbers a and b, gives the open interval \((a, b)\).

ClosedOpenInterval
ClosedOpenInterval(a, b), given extended real numbers a and b, gives the closed-open interval \([a, b)\).
**OpenClosedInterval**

$\texttt{OpenClosedInterval}(a, b)$, given extended real numbers $a$ and $b$, gives the closed-open interval $(a, b]$.

**RealBall**

$\texttt{RealBall}(m, r)$, given a real number $m$ and an extended real number $r$, gives the the closed real ball $[m \pm r]$ with center $m$ and radius $r$.

**OpenRealBall**

$\texttt{OpenRealBall}(m, r)$, given a real number $m$ and an extended real number $r$, gives the the open real ball $(m \pm r)$ with center $m$ and radius $r$.

**OpenComplexDisk**

$\texttt{OpenComplexDisk}(m, r)$, given a complex number $m$ and an extended real number $r$, gives the open complex disk $D(m, r)$ with center $m$ and radius $r$.

**ClosedComplexDisk**

$\texttt{ClosedComplexDisk}(m, r)$, given a complex number $m$ and a real number $r$, gives the closed complex disk $\overline{D}(m, r)$ with center $m$ and radius $r$.

**UpperHalfPlane**

$\texttt{UpperHalfPlane}$ is the set $\mathbb{H}$ of complex numbers with positive imaginary part.

**UnitCircle**

$\texttt{BernsteinEllipse}$

$\texttt{Lattice}$

**Infinities and extended numbers**

**Infinity**

$\texttt{Infinity}$ is the positive signed infinity $\infty$.

**UnsignedInfinity**

$\texttt{UnsignedInfinity}$ is the unsigned infinity $\tilde{\infty}$.

**RealSignedInfinities**

$\texttt{RealSignedInfinities}$ is the set of real signed infinities $\{+\infty, -\infty\}$.

**ComplexSignedInfinities**

$\texttt{ComplexSignedInfinities}$ is the set of complex signed infinities $\{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\}$.

**RealInfinities**

$\texttt{RealInfinities}$ is the set of real infinities (signed and unsigned) $\{+\infty, -\infty\} \cup \{\infty\}$.

**ComplexInfinities**

$\texttt{ComplexInfinities}$ is the set of complex infinities (signed and unsigned) $\{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\} \cup \{\infty\}$.

**ExtendedRealNumbers**

$\texttt{ExtendedRealNumbers}$ is the set of extended real numbers $\mathbb{R} \cup \{+\infty, -\infty\}$.

**ProjectiveRealNumbers**

$\texttt{ProjectiveRealNumbers}$ is the set of projectively extended real numbers $\mathbb{R} \cup \{\tilde{\infty}\}$.

**SignExtendedComplexNumbers**

$\texttt{SignExtendedComplexNumbers}$ is the set of complex numbers extended with signed infinities $\mathbb{C} \cup \{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\}$.
ProjectiveComplexNumbers

ProjectiveComplexNumbers is the set of projectively extended complex numbers (also known as the Riemann sphere) $\mathbb{C} \cup \{\tilde{\infty}\}$.

RealSingularityClosure

RealSingularityClosure is the Calcium singularity closure for real functions, encompassing real numbers, signed infinities, unsigned infinity, and *undefined* ($\text{u}$). This set is defined as $\mathbb{R}^{\text{Sing}} = \mathbb{R} \cup \{+\infty, -\infty\} \cup \{\tilde{\infty}\} \cup \{\text{u}\}$.

ComplexSingularityClosure

ComplexSingularityClosure is the Calcium singularity closure for complex functions, encompassing complex numbers, signed infinities, unsigned infinity, and *undefined* ($\text{u}$). This set is defined as $\mathbb{C}^{\text{Sing}} = \mathbb{C} \cup \{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\} \cup \{\tilde{\infty}\} \cup \{\text{u}\}$.

10.11.6 Operators and calculus

Sums and products

Sum
Product
PrimeSum
PrimeProduct
DivisorSum
DivisorProduct

Solutions and zeros

Zeros
UniqueZero
Solutions
UniqueSolution

Extreme values

Supremum
Infimum
Minimum
Maximum
ArgMin
ArgMax
ArgMinUnique
ArgMaxUnique
Limits
Limit
SequenceLimit
RealLimit
LeftLimit
RightLimit
ComplexLimit
MeromorphicLimit
SequenceLimitInferior
SequenceLimitSuperior
AsymptoticTo

Derivatives
Derivative
RealDerivative
ComplexDerivative
ComplexBranchDerivative
MeromorphicDerivative

Integrals
Integral

Complex analysis
Path
CurvePath
Poles
IsHolomorphicOn
IsMeromorphicOn
Residue
ComplexZeroMultiplicity
AnalyticContinuation
10.11.7 Matrices and linear algebra

Matrix
Row
Column
RowMatrix
ColumnMatrix
DiagonalMatrix
Matrix2x2
ZeroMatrix
IdentityMatrix
Det
Spectrum
SingularValues
Matrices
SL2Z
PSL2Z
SpecialLinearGroup
GeneralLinearGroup
HilbertMatrix

10.11.8 Polynomials, series and rings

Pol
Ser
Polynomial
Coefficient
PolynomialDegree
Polynomials
PolynomialFractions
FormalPowerSeries
FormalLaurentSeries
FormalPuiseuxSeries
Zero
One
Characteristic
Rings
CommutativeRings
Fields
QuotientRing
FiniteField
EqualQSeriesEllipsis
IndefiniteIntegralEqual
QSeriesCoefficient
Call
CallIndeterminate

10.11.9 Special functions

Number parts and step functions
Abs
Sign
Re
Im
Arg
Conjugate
Csgn
RealAbs
Max
Min
Floor
Ceil
KroneckerDelta

Primes and divisibility
IsOdd
IsEven
CongruentMod
Divides
Mod
GCD
LCM
XGCD
IsPrime
Prime
PrimePi
DivisorSigma
MoebiusMu
EulerPhi
DiscreteLog
LegendreSymbol
JacobiSymbol
KroneckerSymbol
SquaresR
LiouvilleLambda

Elementary functions
Exp
Log
Sin
Cos
Tan
Cot
Sec
Csc
Sinh
Cosh
Tanh
Coth
Sech
Csch
Asin
Acos
Atan
Acot
Asec
Acsc
Asinh
Acosh
Atanh
Acoth
Asech
Acsch
Atan2
Sinc
LambertW

Combinatorial functions
SloaneA
SymmetricPolynomial
Cyclotomic
Fibonacci
BernoulliB
BernoulliPolynomial
StirlingCycle
StirlingS1
StirlingS2
EulerE
EulerPolynomial
BellNumber
PartitionsP
LandauG
Gamma function and factorials

Factorial
Binomial
Gamma
LogGamma
DoubleFactorial
RisingFactorial
FallingFactorial
HarmonicNumber
DigammaFunction
DigammaFunctionZero
BetaFunction
BarnesG
LogBarnesG
StirlingSeriesRemainder
LogBarnesGRemainder

Orthogonal polynomials

ChebyshevT
ChebyshevU
LegendreP
JacobiP
HermiteH
LaguerreL
GegenbauerC
SphericalHarmonicY
LegendrePolynomialZero
GaussLegendreWeight
Exponential integrals

Erf
Erfc
Erfi
UpperGamma
LowerGamma
IncompleteBeta
IncompleteBetaRegularized
LogIntegral
ExpIntegralE
ExpIntegralEi
SinIntegral
SinhIntegral
CosIntegral
CoshIntegral
FresnelC
FresnelS

Bessel and Airy functions

AiryAi
AiryBi
AiryAiZero
AiryBiZero
BesselJ
BesselI
BesselY
BesselK
HankelH1
HankelH2
BesselJZero
BesselYZero
CoulombF
CoulombG
CoulombH
CoulombC
CoulombSigma

Hypergeometric functions
Hypergeometric0F1
Hypergeometric1F1
Hypergeometric1F2
Hypergeometric2F1
Hypergeometric2F2
Hypergeometric2F0
Hypergeometric3F2
HypergeometricU
HypergeometricUStar
HypergeometricUStarRemainder
Hypergeometric0F1Regularized
Hypergeometric1F1Regularized
Hypergeometric1F2Regularized
Hypergeometric2F1Regularized
Hypergeometric2F2Regularized
Hypergeometric3F2Regularized

Zeta and L-functions
RiemannZeta
RiemannZetaZero
RiemannHypothesis
RiemannXi
HurwitzZeta
LerchPhi
PolyLog
MultiZetaValue
DirichletL
DirichletLZero
DirichletLambda
DirichletCharacter
DirichletGroup
PrimitiveDirichletCharacters
GeneralizedRiemannHypothesis
ConreyGenerator
GeneralizedBernoulliB
StieltjesGamma
KeiperLiLambda
GaussSum

Elliptic integrals
AGM
AGMSequence
EllipticK
EllipticE
EllipticPi
IncompleteEllipticF
IncompleteEllipticE
IncompleteEllipticPi
CarlsonRF
CarlsonRG
CarlsonRJ
CarlsonRD
CarlsonRC
CarlsonHypergeometricR
CarlsonHypergeometricT

Elliptic, theta and modular functions
JacobiTheta
JacobiThetaQ
DedekindEta
ModularJ
ModularLambda
EisensteinG
EisensteinE
DedekindSum
WeierstrassP
WeierstrassZeta
WeierstrassSigma
EllipticRootE
HilbertClassPolynomial
EulerQSeries
DedekindEtaEpsilon
ModularGroupAction
ModularGroupFundamentalDomain
ModularLambdaFundamentalDomain
PrimitiveReducedPositiveIntegralBinaryQuadraticForms
JacobiThetaEpsilon
JacobiThetaPermutation

Nonsemantic markup

Ellipsis

Ellipsis renders as ... in LaTeX. It can be used to indicate missing function arguments for display purposes, but it has no predefined builtin semantics.

Parentheses

Parentheses(x) semantically represents x, but renders with parentheses ((x)) when converted to LaTeX.

Brackets

Brackets(x) semantically represents x, but renders with brackets ([x]) when converted to LaTeX.

Braces

Braces(x) semantically represents x, but renders with braces ({x}) when converted to LaTeX.

AngleBrackets

AngleBrackets(x) semantically represents x, but renders with angle brackets ⟨x⟩ when converted to LaTeX.

Logic

Logic(x) semantically represents x, but forces logical expressions within x to be rendered using symbols instead of text.

ShowExpandedNormalForm

ShowExpandedNormalForm(x) semantically represents x, but displays the expanded normal form of the expression instead of rendering the expression verbatim. Warning: this triggers a nontrivial (potentially very expensive) computation.

Subscript
11.1 fq.h – finite fields

We represent an element of the finite field $\mathbb{F}_p^n \cong \mathbb{F}_p[X]/(f(X))$, where $f(X) \in \mathbb{F}_p[X]$ is a monic, irreducible polynomial of degree $n$, as a polynomial in $\mathbb{F}_p[X]$ of degree less than $n$. The underlying data structure is an $\texttt{fmpz}_\text{poly}_\texttt{t}$.

The default choice for $f(X)$ is the Conway polynomial for the pair $(p, n)$, enabled by Frank Lübeck’s database of Conway polynomials using the $\texttt{_nmod\_poly\_conway()}$ function. If a Conway polynomial is not available, then a random irreducible polynomial will be chosen for $f(X)$. Additionally, the user is able to supply their own $f(X)$.

11.1.1 Types, macros and constants

- type $\texttt{fq\_ctx\_struct}$
- type $\texttt{fq\_ctx\_t}$
- type $\texttt{fq\_struct}$
- type $\texttt{fq\_t}$

11.1.2 Context Management

- void $\texttt{fq\_ctx\_init()}$ ($\texttt{fq\_ctx\_t}$ ctx, const $\texttt{fmpz\_t}$ p, $\texttt{slong}$ d, const char *var)
  - Initialises the context for prime $p$ and extension degree $d$, with name var for the generator. By default, it will try use a Conway polynomial; if one is not available, a random irreducible polynomial will be used.
  - Assumes that $p$ is a prime.
  - Assumes that the string var is a null-terminated string of length at least one.

- int $\texttt{fq\_ctx\_init\_conway()}$ ($\texttt{fq\_ctx\_t}$ ctx, const $\texttt{fmpz\_t}$ p, $\texttt{slong}$ d, const char *var)
  - Attempts to initialise the context for prime $p$ and extension degree $d$, with name var for the generator using a Conway polynomial for the modulus.
  - Returns 1 if the Conway polynomial is in the database for the given size and the initialization is successful; otherwise, returns 0.
  - Assumes that $p$ is a prime.
  - Assumes that the string var is a null-terminated string of length at least one.
void fq_ctx_init_conway(fq_ctx_t ctx, const fmpz_t p, slong d, const char *var)
    Initialises the context for prime \( p \) and extension degree \( d \), with name \( \text{var} \) for the generator using a Conway polynomial for the modulus.
    Assumes that \( p \) is a prime.
    Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

void fq_ctx_init_modulus(fq_ctx_t ctx, const fmpz_mod_poly_t modulus, const fmpz_mod_ctx_t ctxp, const char *var)
    Initialises the context for given modulus with name \( \text{var} \) for the generator.
    Assumes that \( \text{modulus} \) is an irreducible polynomial over the finite field \( \mathbb{F}_p \) in \( \text{ctxp} \).
    Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

void fq_ctx_init_randtest(fq_ctx_t ctx, flint_rand_t state, int type)
    Initialises \( \text{ctx} \) to a random finite field, where the prime and degree is set according to \( \text{type} \). To see what prime and degrees may be output, see \( \text{type} \) in \_nmod_poly_conway_rand().

void fq_ctx_init_randtest_reducible(fq_ctx_t ctx, flint_rand_t state, int type)
    Initializes \( \text{ctx} \) to a random extension of a prime field, where the prime and degree is set according to \( \text{type} \). If \( \text{type} \) is 0 the prime and degree may be large, else if \( \text{type} \) is 1 the degree is small but the prime may be large, else if \( \text{type} \) is 2 the prime is small but the degree may be large, else if \( \text{type} \) is 3 both prime and degree are small.
    The modulus may or may not be irreducible.

void fq_ctx_clear(fq_ctx_t ctx)
    Clears all memory that has been allocated as part of the context.

const fmpz_mod_poly_struct *fq_ctx_modulus(const fq_ctx_t ctx)
    Returns a pointer to the modulus in the context.

slong fq_ctx_degree(constfq_ctx_t ctx)
    Returns the degree of the field extension \( [\mathbb{F}_q : \mathbb{F}_p] \), which is equal to \( \log_p q \).

const fmpz *fq_ctx_prime(const fq_ctx_t ctx)
    Returns a pointer to the prime \( p \) in the context.

void fq_ctx_order(fmpz_t f, const fq_ctx_t ctx)
    Sets \( f \) to be the size of the finite field.

int fq_ctx_fprint(FILE *file, const fq_ctx_t ctx)
    Prints the context information to \text{file}. Returns 1 for a success and a negative number for an error.

void fq_ctx_print(const fq_ctx_t ctx)
    Prints the context information to \text{stdout}.

11.1.3 Memory management

void fq_init(fq_t rop, const fq_ctx_t ctx)
    Initialises the element \( \text{rop} \), setting its value to 0.

void fq_init2(fq_t rop, const fq_ctx_t ctx)
    Initialises \( \text{poly} \) with at least enough space for it to be an element of \( \text{ctx} \) and sets it to 0.

void fq_clear(fq_t rop, const fq_ctx_t ctx)
    Clears the element \( \text{rop} \).

void _fq_sparse_reduce(fmpz *R, slong lenR, const fq_ctx_t ctx)
    Reduces \( (R, \text{lenR}) \) modulo the polynomial \( f \) given by the modulus of \( \text{ctx} \).
void _fq_dense_reduce(fmpz *R, slong lenR, const fq_ctx_t ctx)
    Reduces (R, lenR) modulo the polynomial f given by the modulus of ctx using Newton division.

void _fq_reduce(fmpz *r, slong lenR, const fq_ctx_t ctx)
    Reduces (R, lenR) modulo the polynomial f given by the modulus of ctx. Does either sparse or dense reduction based on ctx->sparse_modulus.

void fq_reduce(fq_t rop, const fq_ctx_t ctx)
    Reduces the polynomial rop as an element of $\mathbb{F}_p[X]/(f(X))$.

11.1.4 Basic arithmetic

void fq_add(fq_t rop, const fq_t op1, const fq_t op2, const fq_ctx_t ctx)
    Sets rop to the sum of op1 and op2.

void fq_sub(fq_t rop, const fq_t op1, const fq_t op2, const fq_ctx_t ctx)
    Sets rop to the difference of op1 and op2.

void fq_sub_one(fq_t rop, const fq_t op1, const fq_ctx_t ctx)
    Sets rop to the difference of op1 and 1.

void fq_neg(fq_t rop, const fq_t op, const fq_ctx_t ctx)
    Sets rop to the negative of op.

void fq_mul(fq_t rop, const fq_t op1, const fq_t op2, const fq_ctx_t ctx)
    Sets rop to the product of op1 and op2, reducing the output in the given context.

void fq_mul_fmpz(fq_t rop, const fq_t op, const fmpz_t x, const fq_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_mul_si(fq_t rop, const fq_t op, slong x, const fq_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_mul_ui(fq_t rop, const fq_t op, ulong x, const fq_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_sqr(fq_t rop, const fq_t op, const fq_ctx_t ctx)
    Sets rop to the square of op, reducing the output in the given context.

void fq_div(fq_t rop, const fq_t op1, const fq_t op2, const fq_ctx_t ctx)
    Sets rop to the quotient of op1 and op2, reducing the output in the given context.

void _fq_inv(fmpz *rop, const fmpz *op, slong len, const fq_ctx_t ctx)
    Sets (rop, d) to the inverse of the non-zero element (op, len). Assumes that d ≥ 0 and that len is positive and at most d.
    Does not support aliasing.

11.1. fq.h – finite fields
void fq_pow(fq_t rop, const fq_t op, const fmpz_t e, const fq_ctx_t ctx)
    Sets rop the op raised to the power e.
    Currently assumes that e ≥ 0.
    Note that for any input op, rop is set to 1 whenever e = 0.

void fq_pow_ui(fq_t rop, const fq_t op, const ulong e, const fq_ctx_t ctx)
    Sets rop the op raised to the power e.
    Currently assumes that e ≥ 0.
    Note that for any input op, rop is set to 1 whenever e = 0.

11.1.5 Roots

int fq_sqrt(fq_t rop, const fq_t op1, const fq_ctx_t ctx)
    Sets rop to the square root of op1 if it is a square, and return 1, otherwise return 0.

void fq_pth_root(fq_t rop, const fq_t op1, const fq_ctx_t ctx)
    Sets rop to a p\text{th} root root of op1. Currently, this computes the root by raising op1 to p^{d-1} where d is the degree of the extension.

int fq_is_square(const fq_t op, const fq_ctx_t ctx)
    Return 1 if op is a square.

11.1.6 Output

int fq_fprint_pretty(FILE *file, const fq_t op, const fq_ctx_t ctx)
    Prints a pretty representation of op to file.
    In the current implementation, always returns 1. The return code is part of the function’s signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

int fq_print_pretty(const fq_t op, const fq_ctx_t ctx)
    Prints a pretty representation of op to stdout.
    In the current implementation, always returns 1. The return code is part of the function’s signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

int fq_fprint(FILE *file, const fq_t op, const fq_ctx_t ctx)
    Prints a representation of op to file.
    For further details on the representation used, see fmpz_mod_poly_fprint().

void fq_print(const fq_t op, const fq_ctx_t ctx)
    Prints a representation of op to stdout.
    For further details on the representation used, see fmpz_mod_poly_print().

char *fq_get_str(const fq_t op, const fq_ctx_t ctx)
    Returns the plain FLINT string representation of the element op.

char *fq_get_str_pretty(const fq_t op, const fq_ctx_t ctx)
    Returns a pretty representation of the element op using the null-terminated string x as the variable name.
11.1.7 Randomisation

void \texttt{fq\_randtest}(\texttt{fq\_t rop}, \texttt{flint\_rand\_t state}, const \texttt{fq\_ctx\_t ctx})
Generates a random element of $\mathbb{F}_q$.

void \texttt{fq\_randtest\_not\_zero}(\texttt{fq\_t rop}, \texttt{flint\_rand\_t state}, const \texttt{fq\_ctx\_t ctx})
Generates a random non-zero element of $\mathbb{F}_q$.

void \texttt{fq\_randtest\_dense}(\texttt{fq\_t rop}, \texttt{flint\_rand\_t state}, const \texttt{fq\_ctx\_t ctx})
Generates a random element of $\mathbb{F}_q$ which has an underlying polynomial with dense coefficients.

void \texttt{fq\_rand}(\texttt{fq\_t rop}, \texttt{flint\_rand\_t state}, const \texttt{fq\_ctx\_t ctx})
Generates a high quality random element of $\mathbb{F}_q$.

void \texttt{fq\_rand\_not\_zero}(\texttt{fq\_t rop}, \texttt{flint\_rand\_t state}, const \texttt{fq\_ctx\_t ctx})
Generates a high quality non-zero random element of $\mathbb{F}_q$.

11.1.8 Assignments and conversions

void \texttt{fq\_set}(\texttt{fq\_t rop}, const \texttt{fq\_t op}, const \texttt{fq\_ctx\_t ctx})
Sets \texttt{rop} to \texttt{op}.

void \texttt{fq\_set\_si}(\texttt{fq\_t rop}, const \texttt{slong x}, const \texttt{fq\_ctx\_t ctx})
Sets \texttt{rop} to \texttt{x}, considered as an element of $\mathbb{F}_p$.

void \texttt{fq\_set\_ui}(\texttt{fq\_t rop}, const \texttt{ulong x}, const \texttt{fq\_ctx\_t ctx})
Sets \texttt{rop} to \texttt{x}, considered as an element of $\mathbb{F}_p$.

void \texttt{fq\_set\_fmpz}(\texttt{fq\_t rop}, const \texttt{fmpz\_t x}, const \texttt{fq\_ctx\_t ctx})
Sets \texttt{rop} to \texttt{x}, considered as an element of $\mathbb{F}_p$.

void \texttt{fq\_swap}(\texttt{fq\_t op1}, \texttt{fq\_t op2}, const \texttt{fq\_ctx\_t ctx})
Swaps the two elements \texttt{op1} and \texttt{op2}.

void \texttt{fq\_zero}(\texttt{fq\_t rop}, const \texttt{fq\_ctx\_t ctx})
Sets \texttt{rop} to zero.

void \texttt{fq\_one}(\texttt{fq\_t rop}, const \texttt{fq\_ctx\_t ctx})
Sets \texttt{rop} to one, reduced in the given context.

void \texttt{fq\_gen}(\texttt{fq\_t rop}, const \texttt{fq\_ctx\_t ctx})
Sets \texttt{rop} to a generator for the finite field. There is no guarantee this is a multiplicative generator of the finite field.

int \texttt{fq\_get\_fmpz}(\texttt{fmpz\_t rop}, const \texttt{fq\_t op}, const \texttt{fq\_ctx\_t ctx})
If \texttt{op} has a lift to the integers, return 1 and set \texttt{rop} to the lift in $[0,p)$. Otherwise, return 0 and leave \texttt{rop} undefined.

void \texttt{fq\_get\_fmpz\_poly}(\texttt{fmpz\_poly\_t a}, const \texttt{fq\_t b}, const \texttt{fq\_ctx\_t ctx})
Set \texttt{a} to a representative of \texttt{b} in \texttt{ctx}. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in \texttt{ctx}.

void \texttt{fq\_set\_fmpz\_poly}(\texttt{fq\_t a}, const \texttt{fmpz\_poly\_t b}, const \texttt{fq\_ctx\_t ctx})
Set \texttt{a} to the element in \texttt{ctx} with representative \texttt{b}. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in \texttt{ctx}.
void fq_get_fmpz_mod_mat (fmpz_mod_mat_t col, const fq_t a, const fq_ctx_t ctx)
    Convert a to a column vector of length degree(ctx).

void fq_set_fmpz_mod_mat (fq_t a, const fmpz_mod_mat_t col, const fq_ctx_t ctx)
    Convert a column vector col of length degree(ctx) to an element of ctx.

11.1.9 Comparison

int fq_is_zero (const fq_t op, const fq_ctx_t ctx)
    Returns whether op is equal to zero.

int fq_is_one (const fq_t op, const fq_ctx_t ctx)
    Returns whether op is equal to one.

int fq_equal (const fq_t op1, const fq_t op2, const fq_ctx_t ctx)
    Returns whether op1 and op2 are equal.

int fq_is_invertible (const fq_t op, const fq_ctx_t ctx)
    Returns whether op is an invertible element.

int fq_is_invertible_f (fq_t f, const fq_t op, const fq_ctx_t ctx)
    Returns whether op is an invertible element. If it is not, then f is set of a factor of the modulus.

11.1.10 Special functions

void _fq_trace (fmpz_t rop, const fmpz *op, slong len, const fq_ctx_t ctx)
    Sets rop to the trace of the non-zero element (op, len) in \( \mathbb{F}_q \).

void fq_trace (fmpz_t rop, const fq_t op, const fq_ctx_t ctx)
    Sets rop to the trace of op.
    For an element \( a \in \mathbb{F}_q \), multiplication by \( a \) defines a \( \mathbb{F}_p \)-linear map on \( \mathbb{F}_q \). We define the trace of \( a \) as the trace of this map. Equivalently, if \( \Sigma \) generates \( \text{Gal}(\mathbb{F}_q/\mathbb{F}_p) \) then the trace of \( a \) is equal to \( \sum_{i=0}^{d-1} \Sigma^i(a) \), where \( d = \log_p q \).

void _fq_norm (fmpz_t rop, const fmpz *op, slong len, const fq_ctx_t ctx)
    Sets rop to the norm of the non-zero element (op, len) in \( \mathbb{F}_q \).

void fq_norm (fmpz_t rop, const fq_t op, const fq_ctx_t ctx)
    Computes the norm of op.
    For an element \( a \in \mathbb{F}_q \), multiplication by \( a \) defines a \( \mathbb{F}_p \)-linear map on \( \mathbb{F}_q \). We define the norm of \( a \) as the determinant of this map. Equivalently, if \( \Sigma \) generates \( \text{Gal}(\mathbb{F}_q/\mathbb{F}_p) \) then the trace of \( a \) is equal to \( \prod_{i=0}^{d-1} \Sigma^i(a) \), where \( d = \dim_{\mathbb{F}_p}(\mathbb{F}_q) \).
    Algorithm selection is automatic depending on the input.

void _fq_frobenius (fmpz_t rop, const fmpz *op, slong len, const fq_ctx_t ctx)
    Sets (rop, 2d-1) to the image of (op, len) under the Frobenius operator raised to the e-th power, assuming that neither op nor e are zero.

void fq_frobenius (fq_t rop, const fq_t op, slong e, const fq_ctx_t ctx)
    Evaluates the homomorphism \( \sigma^e \) at op.
    Recall that \( \mathbb{F}_q/\mathbb{F}_p \) is Galois with Galois group \( \langle \sigma \rangle \), which is also isomorphic to \( \mathbb{Z}/d\mathbb{Z} \), where \( \sigma \in \text{Gal}(\mathbb{F}_q/\mathbb{F}_p) \) is the Frobenius element \( \sigma : x \mapsto x^p \).
int fq_multiplicative_order(fmpz *ord, const fq_t op, const fq_ctx_t ctx)

Computes the order of op as an element of the multiplicative group of ctx.

Returns 0 if op is 0, otherwise it returns 1 if op is a generator of the multiplicative group, and -1 if it is not.

This function can also be used to check primitivity of a generator of a finite field whose defining polynomial is not primitive.

int fq_is_primitive(const fq_t op, const fq_ctx_t ctx)

Returns whether op is primitive, i.e., whether it is a generator of the multiplicative group of ctx.

### 11.1.11 Bit packing

void fq_bit_pack(fmpz_t f, const fq_t op, flint_bitcnt_t bit_size, const fq_ctx_t ctx)

Packs op into bitfields of size bit_size, writing the result to f.

void fq_bit_unpack(fq_t rop, const fmpz_t f, flint_bitcnt_t bit_size, const fq_ctx_t ctx)

Unpacks into rop the element with coefficients packed into fields of size bit_size as represented by the integer f.

### 11.2 fq_default_default.h – unified finite fields

#### 11.2.1 Types, macros and constants

type fq_default_default_ctx_t

type fq_default_default_t

#### 11.2.2 Context Management

void fq_default_ctx_init(fq_default_ctx_t ctx, const fmpz_t p, slong d, const char *var)

Initialises the context for prime p and extension degree d, with name var for the generator. By default, it will try use a Conway polynomial; if one is not available, a random irreducible polynomial will be used.

Assumes that p is a prime.

Assumes that the string var is a null-terminated string of length at least one.

void fq_default_ctx_init_type(fq_default_ctx_t ctx, const fmpz_t p, slong d, const char *var, int type)

As per the previous function except that if type == 1 an fq_zech context is created, if type == 2 an fq_nmod and if type == 3 an fq. If type == 0 the functionality is as per the previous function.

void fq_default_ctx_init_modulus(fq_default_ctx_t ctx, const fmpz_mod_poly_t modulus, const fmpz_mod_ctx_t mod_ctx, const char *var)

Initialises the context for the finite field defined by the given polynomial modulus. The characteristic will be the modulus of the polynomial and the degree equal to its degree.

Assumes that the characteristic is prime and the polynomial irreducible.

Assumes that the string var is a null-terminated string of length at least one.
void fq_default_ctx_init_modulus_type(fq_default_ctx_t ctx, const fmpz_mod_poly_t modulus, fmpz_mod_ctx_t mod_ctx, const char *var, int type)

As per the previous function except that if type == 1 an fq_zech context is created, if type == 2 an fq_nmod and if type == 3 an fq. If type == 0 the functionality is as per the previous function.

void fq_default_ctx_init_modulus_nmod(fq_default_ctx_t ctx, const nmod_poly_t modulus, const char *var)

Initialises the context for the finite field defined by the given polynomial modulus. The characteristic will be the modulus of the polynomial and the degree equal to its degree.

Assumes that the characteristic is prime and the polynomial irreducible.

Assumes that the string var is a null-terminated string of length at least one.

void fq_default_ctx_init_modulus_nmod_type(fq_default_ctx_t ctx, const nmod_poly_t modulus, const char *var, int type)

As per the previous function except that if type == 1 an fq_zech context is created, if type == 2 an fq_nmod and if type == 3 an fq. If type == 0 the functionality is as per the previous function.

void fq_default_ctx_clear(fq_default_ctx_t ctx)

Clears all memory that has been allocated as part of the context.

int fq_default_ctx_type(const fq_default_ctx_t ctx)

Returns 1 if the context contains an fq_zech context, 2 if it contains an fq_mod context and 3 if it contains an fq context.

slong fq_default_ctx_degree(const fq_default_ctx_t ctx)

Returns the degree of the field extension \([\mathbb{F}_q : \mathbb{F}_p]\), which is equal to \( \log_p q \).

void fq_default_ctx_prime(fmpz_t prime, const fq_default_ctx_t ctx)

Sets prime to the prime \( p \) in the context.

void fq_default_ctx_order(fmpz_t f, const fq_default_ctx_t ctx)

Sets \( f \) to be the size of the finite field.

void fq_default_ctx_modulus(fmpz_mod_poly_t p, const fq_default_ctx_t ctx)

Sets \( p \) to the defining polynomial of the finite field.

int fq_default_ctx_fprint(FILE *file, const fq_default_ctx_t ctx)

Prints the context information to file. Returns 1 for a success and a negative number for an error.

void fq_default_ctx_print(const fq_default_ctx_t ctx)

Prints the context information to stdout.

void fq_default_ctx_randtest(fq_default_ctx_t ctx)

Initializes \( \text{ctx} \) to a random finite field. Assumes that \( \text{fq_default_ctx_init} \) has not been called on \( \text{ctx} \) already.

void fq_default_ctx_randtest(fq_default_ctx_t ctx)

Initializes \( \text{ctx} \) to a random finite field. Assumes that \( \text{fq_default_ctx_init} \) has not been called on \( \text{ctx} \) already.

11.2.3 Memory management

void fq_default_init(fq_default_t rop, const fq_default_ctx_t ctx)

Initialises the element rop, setting its value to 0.

void fq_default_init2(fq_default_t rop, const fq_default_ctx_t ctx)

Initialises poly with at least enough space for it to be an element of ctx and sets it to 0.

void fq_default_clear(fq_default_t rop, constfq_default_ctx_t ctx)

Clears the element rop.
11.2.4 Predicates

```c
int fq_default_is_invertible(const fq_default_t op, const fq_default_ctx_t ctx)
    Return 1 if op is an invertible element.
```

11.2.5 Basic arithmetic

```c
void fq_default_add(fq_default_t rop, const fq_default_t op1, const fq_default_t op2, const fq_default_ctx_t ctx)
    Sets rop to the sum of op1 and op2.

void fq_default_sub(fq_default_t rop, const fq_default_t op1, const fq_default_t op2, const fq_default_ctx_t ctx)
    Sets rop to the difference of op1 and op2.

void fq_default_sub_one(fq_default_t rop, const fq_default_t op1, const fq_default_ctx_t ctx)
    Sets rop to the difference of op1 and 1.

void fq_default_neg(fq_default_t rop, const fq_default_t op, const fq_default_ctx_t ctx)
    Sets rop to the negative of op.

void fq_default_mul(fq_default_t rop, const fq_default_t op1, const fq_default_t op2, const fq_default_ctx_t ctx)
    Sets rop to the product of op1 and op2, reducing the output in the given context.

void fq_default_mul_fmpz(fq_default_t rop, const fq_default_t op, const fmpz_t x, const fq_default_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_default_mul_si(fq_default_t rop, const fq_default_t op, slong x, const fq_default_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_default_mul_ui(fq_default_t rop, const fq_default_t op, ulong x, const fq_default_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_default_sqr(fq_default_t rop, const fq_default_t op, const fq_default_ctx_t ctx)
    Sets rop to the square of op, reducing the output in the given context.

void fq_default_div(fq_default_t rop, fq_default_t op1, fq_default_t op2, const fq_default_ctx_t ctx)
    Sets rop to the quotient of op1 and op2, reducing the output in the given context.

void fq_default_inv(fq_default_t rop, const fq_default_t op, const fq_default_ctx_t ctx)
    Sets rop to the inverse of the non-zero element op.

void fq_default_pow(fq_default_t rop, const fq_default_t op, const fmpz_t e, const fq_default_ctx_t ctx)
    Sets rop the op raised to the power e.
    Currently assumes that e ≥ 0.
    Note that for any input op, rop is set to 1 whenever e = 0.

void fq_default_pow_ui(fq_default_t rop, const fq_default_t op, const ulong e, const fq_default_ctx_t ctx)
    Sets rop the op raised to the power e.
    Currently assumes that e ≥ 0.
    Note that for any input op, rop is set to 1 whenever e = 0.
```
11.2.6 Roots

```c
int fq_default_sqrt(fq_default_t rop, const fq_default_t op1, const fq_default_ctx_t ctx)
    Sets rop to the square root of op1 if it is a square, and return 1, otherwise return 0.

void fq_default_pth_root(fq_default_t rop, const fq_default_t op1, const fq_default_ctx_t ctx)
    Sets rop to a \( p \)th root root of op1. Currently, this computes the root by raising op1 to \( p^{d-1} \) where \( d \) is the degree of the extension.
```

```c
int fq_default_is_square(const fq_default_t op, const fq_default_ctx_t ctx)
    Return 1 if op is a square.
```

11.2.7 Output

```c
int fq_default_fprint_pretty(FILE *file, const fq_default_t op, const fq_default_ctx_t ctx)
    Prints a pretty representation of op to file.
    In the current implementation, always returns 1. The return code is part of the function’s signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

void fq_default_print_pretty(const fq_default_t op, const fq_default_ctx_t ctx)
    Prints a pretty representation of op to stdout.
    In the current implementation, always returns 1. The return code is part of the function’s signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

int fq_default_fprint(FILE *file, const fq_default_t op, const fq_default_ctx_t ctx)
    Prints a representation of op to file.

void fq_default_print(const fq_default_t op, const fq_default_ctx_t ctx)
    Prints a representation of op to stdout.

char *fq_default_get_str(const fq_default_t op, const fq_default_ctx_t ctx)
    Returns the plain FLINT string representation of the element op.

char *fq_default_get_str_pretty(const fq_default_t op, const fq_default_ctx_t ctx)
    Returns a pretty representation of the element op using the null-terminated string x as the variable name.
```

11.2.8 Randomisation

```c
void fq_default_randtest(fq_default_t rop, flint_rand_t state, const fq_default_ctx_t ctx)
    Generates a random element of \( \mathbb{F}_q \).

void fq_default_randtest_not_zero(fq_default_t rop, flint_rand_t state, const fq_default_ctx_t ctx)
    Generates a random non-zero element of \( \mathbb{F}_q \).

void fq_default_rand(fq_default_t rop, flint_rand_t state, const fq_default_ctx_t ctx)
    Generates a high quality random element of \( \mathbb{F}_q \).

void fq_default_rand_not_zero(fq_default_t rop, flint_rand_t state, const fq_default_ctx_t ctx)
    Generates a high quality non-zero random element of \( \mathbb{F}_q \).
```
11.2.9 Assignments and conversions

```c
void fq_default_set(fq_default_t rop, const fq_default_t op, const fq_default_ctx_t ctx)
    Sets rop to op.

void fq_default_set_si(fq_default_t rop, const slong x, const fq_default_ctx_t ctx)
    Sets rop to x, considered as an element of $\mathbb{F}_p$.

void fq_default_set_ui(fq_default_t rop, const ulong x, const fq_default_ctx_t ctx)
    Sets rop to x, considered as an element of $\mathbb{F}_p$.

void fq_default_set_fmpz(fq_default_t rop, const fmpz_t x, const fq_default_ctx_t ctx)
    Sets rop to x, considered as an element of $\mathbb{F}_p$.

void fq_default_swap(fq_default_t op1, fq_default_t op2, const fq_default_ctx_t ctx)
    Swaps the two elements op1 and op2.

void fq_default_zero(fq_default_t rop, const fq_default_ctx_t ctx)
    Sets rop to zero.

void fq_default_one(fq_default_t rop, const fq_default_ctx_t ctx)
    Sets rop to one, reduced in the given context.

void fq_default_gen(fq_default_t rop, const fq_default_ctx_t ctx)
    Sets rop to a generator for the finite field. There is no guarantee this is a multiplicative generator of the finite field.

int fq_default_get_fmpz(fmpz_t rop, const fq_default_t op, const fq_default_ctx_t ctx)
    If op has a lift to the integers, return 1 and set rop to the lift in $[0, p)$. Otherwise, return 0 and leave rop undefined.

void fq_default_get_nmod_poly(nmod_poly_t poly, const fq_default_t op, const fq_default_ctx_t ctx)
    Sets poly to the polynomial representation of op. Assumes the characteristic of the field and the modulus of the polynomial are the same. No checking of this occurs.

void fq_default_set_nmod_poly(fq_default_t op, const nmod_poly_t poly, const fq_default_ctx_t ctx)
    Sets op to the finite field element represented by the polynomial poly. Assumes the characteristic of the field and the modulus of the polynomial are the same. No checking of this occurs.

void fq_default_get_fmpz_mod_poly(fmpz_mod_poly_t poly, const fq_default_t op, const fq_default_ctx_t ctx)
    Sets poly to the polynomial representation of op. Assumes the characteristic of the field and the modulus of the polynomial are the same. No checking of this occurs.

void fq_default_set_fmpz_mod_poly(fq_default_t op, const fmpz_mod_poly_t poly, const fq_default_ctx_t ctx)
    Sets op to the finite field element represented by the polynomial poly. Assumes the characteristic of the field and the modulus of the polynomial are the same. No checking of this occurs.

void fq_default_get_fmpz_poly(fmpz_poly_t a, const fq_default_t b, const fq_default_ctx_t ctx)
    Set a to a representative of b in ctx. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in ctx.

void fq_default_set_fmpz_poly(fq_default_t a, const fmpz_poly_t b, const fq_default_ctx_t ctx)
    Set a to the element in ctx with representative b. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in ctx.
```
11.2.10 Comparison

```c
int fq_default_is_zero(const fq_default_t op, const fq_default_ctx_t ctx)
    Returns whether op is equal to zero.
```

```c
int fq_default_is_one(const fq_default_t op, const fq_default_ctx_t ctx)
    Returns whether op is equal to one.
```

```c
int fq_default_equal(const fq_default_t op1, const fq_default_t op2, const fq_default_ctx_t ctx)
    Returns whether op1 and op2 are equal.
```

11.2.11 Special functions

```c
void fq_default_trace(fmpz_t rop, const fq_default_t op, const fq_default_ctx_t ctx)
    Sets rop to the trace of op. For an element \(a \in F_q\), multiplication by \(a\) defines a \(F_p\)-linear map on \(F_q\). We define the trace of \(a\) as the trace of this map. Equivalently, if \(\Sigma\) generates \(\text{Gal}(F_q/F_p)\) then the trace of \(a\) is equal to \(\sum_{i=0}^{d-1} \Sigma^i(a)\), where \(d = \log_p q\).
```

```c
void fq_default_norm(fmpz_t rop, const fq_default_t op, const fq_default_ctx_t ctx)
    Computes the norm of op. For an element \(a \in F_q\), multiplication by \(a\) defines a \(F_p\)-linear map on \(F_q\). We define the norm of \(a\) as the determinant of this map. Equivalently, if \(\Sigma\) generates \(\text{Gal}(F_q/F_p)\) then the trace of \(a\) is equal to \(\prod_{i=0}^{d-1} \Sigma^i(a)\), where \(d = \dim_{F_p}(F_q)\).
```

Algorithm selection is automatic depending on the input.

```c
void fq_default_frobenius(fq_default_t rop, const fq_default_t op, slong e, const fq_default_ctx_t ctx)
    Evaluates the homomorphism \(\Sigma^e\) at op. Recall that \(F_q/F_p\) is Galois with Galois group \(\langle \sigma \rangle\), which is also isomorphic to \(Z/dZ\), where \(\sigma \in \text{Gal}(F_q/F_p)\) is the Frobenius element \(\sigma: x \mapsto x^p\).
```

11.3 *fq_vec.h* – vectors over finite fields

11.3.1 Memory management

```c
fq_struct *_fq_vec_init(slong len, const fq_ctx_t ctx)
    Returns an initialised vector of fq’s of given length.
```

```c
void _fq_vec_clear(fq_struct *vec, slong len, const fq_ctx_t ctx)
    Clears the entries of (vec, len) and frees the space allocated for vec.
```

11.3.2 Randomisation

```c
void _fq_vec_randtest(fq_struct *f, flint_rand_t state, slong len, const fq_ctx_t ctx)
    Sets the entries of a vector of the given length to elements of the finite field.
```
11.3.3 Input and output

```
int _fq_vec_fprint(FILE *file, const fq_struct *vec, slong len, const fq_ctx_t ctx)
Prints the vector of given length to the stream file. The format is the length followed by two
spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.
In case of success, returns a positive value. In case of failure, returns a non-positive value.
```

```
int _fq_vec_print(const fq_struct *vec, slong len, const fq_ctx_t ctx)
Prints the vector of given length to stdout.
For further details, see _fq_vec_fprint().
```

11.3.4 Assignment and basic manipulation

```
void _fq_vec_set(fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_ctx_t ctx)
Makes a copy of (vec2, len2) into vec1.
```

```
void _fq_vec_swap(fq_struct *vec1, fq_struct *vec2, slong len2, const fq_ctx_t ctx)
Swaps the elements in (vec1, len2) and (vec2, len2).
```

```
void _fq_vec_zero(fq_struct *vec, slong len, const fq_ctx_t ctx)
Zeros the entries of (vec, len).
```

```
void _fq_vec_neg(fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_ctx_t ctx)
Negates (vec2, len2) and places it into vec1.
```

11.3.5 Comparison

```
int _fq_vec_equal(const fq_struct *vec1, const fq_struct *vec2, slong len, const fq_ctx_t ctx)
Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.
```

```
int _fq_vec_is_zero(const fq_struct *vec, slong len, const fq_ctx_t ctx)
Returns 1 if (vec, len) is zero, and 0 otherwise.
```

11.3.6 Addition and subtraction

```
void _fq_vec_add(fq_struct *res, const fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_ctx_t ctx)
Sets (res, len2) to the sum of (vec1, len2) and (vec2, len2).
```

```
void _fq_vec_sub(fq_struct *res, const fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_ctx_t ctx)
Sets (res, len2) to (vec1, len2) minus (vec2, len2).
```

11.3.7 Scalar multiplication and division

```
void _fq_vec_scalar_addmul_fq(fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_t c, const fq_ctx_t ctx)
Adds (vec2, len2) times c to (vec1, len2), where c is a fq_t.
```

```
void _fq_vec_scalar_submul_fq(fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_t c, const fq_ctx_t ctx)
Subtracts (vec2, len2) times c from (vec1, len2), where c is a fq_t.
```
11.3.8 Dot products

```c
void _fq_vec_dot(fq_t res, const fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_ctx_t ctx)
    Sets res to the dot product of (vec1, len) and (vec2, len).
```

11.4 fq_mat.h – matrices over finite fields

11.4.1 Types, macros and constants

```c
type fq_mat_struct

type fq_mat_t
```

11.4.2 Memory management

```c
void fq_mat_init(fq_mat_t mat, slong rows, slong cols, const fq_ctx_t ctx)
    Initialises mat to a rows-by-cols matrix with coefficients in \( \mathbf{F}_q \) given by ctx. All elements are set to zero.

void fq_mat_init_set(fq_mat_t mat, const fq_mat_t src, const fq_ctx_t ctx)
    Initialises mat and sets its dimensions and elements to those of src.

void fq_mat_clear(fq_mat_t mat, const fq_ctx_t ctx)
    Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an fq_mat_t object.

void fq_mat_set(fq_mat_t mat, const fq_mat_t src, const fq_ctx_t ctx)
    Sets mat to a copy of src. It is assumed that mat and src have identical dimensions.
```

11.4.3 Basic properties and manipulation

```c
fq_struct *fq_mat_entry(const fq_mat_t mat, slong i, slong j)
    Directly accesses the entry in mat in row i and column j, indexed from zero. No bounds checking is performed.

void fq_mat_entry_set(fq_mat_t mat, slong i, slong j, const fq_t x, const fq_ctx_t ctx)
    Sets the entry in mat in row i and column j to x.

slong fq_mat_nrows(const fq_mat_t mat, const fq_ctx_t ctx)
    Returns the number of rows in mat.

slong fq_mat_ncols(const fq_mat_t mat, const fq_ctx_t ctx)
    Returns the number of columns in mat.

void fq_mat_swap(fq_mat_t mat1, fq_mat_t mat2, const fq_ctx_t ctx)
    Swaps two matrices. The dimensions of mat1 and mat2 are allowed to be different.

void fq_mat_swap_entrywise(fq_mat_t mat1, fq_mat_t mat2, const fq_ctx_t ctx)
    Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

void fq_mat_zero(fq_mat_t mat, const fq_ctx_t ctx)
    Sets all entries of mat to 0.
```
void fq_mat_one(fq_mat_t mat, const fq_ctx_t ctx)
    Sets all the diagonal entries of mat to 1 and all other entries to 0.

void fq_mat_swap_rows(fq_mat_t mat, slong *perm, slong r, slong s, const fq_ctx_t ctx)
    Swaps rows r and s of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

void fq_mat_swap_cols(fq_mat_t mat, slong *perm, slong r, slong s, const fq_ctx_t ctx)
    Swaps columns r and s of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.

void fq_mat_invert_rows(fq_mat_t mat, slong *perm, const fq_ctx_t ctx)
    Swaps rows i and r - i of mat for 0 <= i < r/2, where r is the number of rows of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

void fq_mat_invert_cols(fq_mat_t mat, slong *perm, const fq_ctx_t ctx)
    Swaps columns i and c - i of mat for 0 <= i < c/2, where c is the number of columns of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.

11.4.4 Conversions

void fq_mat_set_nmod_mat(fq_mat_t mat1, const nmod_mat_t mat2, const fq_ctx_t ctx)
    Sets the matrix mat1 to the matrix mat2.

void fq_mat_set_fmpz_mod_mat(fq_mat_t mat1, const fmpz_mod_mat_t mat2, const fq_ctx_t ctx)
    Sets the matrix mat1 to the matrix mat2.

11.4.5 Concatenate

void fq_mat_concat_vertical(fq_mat_t res, const fq_mat_t mat1, const fq_mat_t mat2, const fq_ctx_t ctx)
    Sets res to vertical concatenation of (mat1, mat2) in that order. Matrix dimensions : mat1 : m \times n, mat2 : k \times n, res : (m + k) \times n.

void fq_mat_concat_horizontal(fq_mat_t res, const fq_mat_t mat1, const fq_mat_t mat2, const fq_ctx_t ctx)
    Sets res to horizontal concatenation of (mat1, mat2) in that order. Matrix dimensions : mat1 : m \times n, mat2 : m \times k, res : m \times (n + k).

11.4.6 Printing

int fq_mat_print_pretty(const fq_mat_t mat, const fq_ctx_t ctx)
    Pretty-prints mat to stdout. A header is printed followed by the rows enclosed in brackets.

int fq_mat_fprint_pretty(FILE *file, const fq_mat_t mat, const fq_ctx_t ctx)
    Pretty-prints mat to file. A header is printed followed by the rows enclosed in brackets.

    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_mat_print(const fq_mat_t mat, const fq_ctx_t ctx)
    Prints mat to stdout. A header is printed followed by the rows enclosed in brackets.

int fq_mat_fprint(FILE *file, const fq_mat_t mat, const fq_ctx_t ctx)
    Prints mat to file. A header is printed followed by the rows enclosed in brackets.

    In case of success, returns a positive value. In case of failure, returns a non-positive value.
### 11.4.7 Window

```c
void fq_mat_window_init(fq_mat_t window, const fq_mat_t mat, slong r1, slong c1, slong r2, slong c2, const fq_ctx_t ctx)
```

Initializes the matrix `window` to be an \( r2 - r1 \) by \( c2 - c1 \) submatrix of `mat` whose \((0,0)\) entry is the \((r1, c1)\) entry of `mat`. The memory for the elements of `window` is shared with `mat`.

```c
void fq_mat_window_clear(fq_mat_t window, const fq_ctx_t ctx)
```

Clears the matrix `window` and releases any memory that it uses. Note that the memory to the underlying matrix that `window` points to is not freed.

### 11.4.8 Random matrix generation

```c
void fq_mat_randtest(fq_mat_t mat, flint_rand_t state, const fq_ctx_t ctx)
```

Sets the elements of `mat` to random elements of \( \mathbb{F}_q \), given by `ctx`.

```c
int fq_mat_randpermdiag(fq_mat_t mat, flint_rand_t state, fq_struct *diag, slong n, const fq_ctx_t ctx)
```

Sets `mat` to a random permutation of the diagonal matrix with \( n \) leading entries given by the vector `diag`. It is assumed that the main diagonal of `mat` has room for at least \( n \) entries. Returns 0 or 1, depending on whether the permutation is even or odd respectively.

```c
void fq_mat_randrank(fq_mat_t mat, flint_rand_t state, slong rank, const fq_ctx_t ctx)
```

Sets `mat` to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of \( \mathbb{F}_q \).

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling `fq_mat_randops()`.

```c
void fq_mat_randops(fq_mat_t mat, slong count, flint_rand_t state, const fq_ctx_t ctx)
```

Randomises `mat` by performing elementary row or column operations. More precisely, at most `count` random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

```c
void fq_mat_randtril(fq_mat_t mat, flint_rand_t state, int unit, const fq_ctx_t ctx)
```

Sets `mat` to a random lower triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

```c
void fq_mat_randtriu(fq_mat_t mat, flint_rand_t state, int unit, const fq_ctx_t ctx)
```

Sets `mat` to a random upper triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

### 11.4.9 Comparison

```c
int fq_mat_equal(const fq_mat_t mat1, const fq_mat_t mat2, const fq_ctx_t ctx)
```

Returns nonzero if `mat1` and `mat2` have the same dimensions and elements, and zero otherwise.

```c
int fq_mat_is_zero(const fq_mat_t mat, const fq_ctx_t ctx)
```

Returns a non-zero value if all entries of `mat` are zero, and otherwise returns zero.

```c
int fq_mat_is_one(const fq_mat_t mat, const fq_ctx_t ctx)
```

Returns a non-zero value if all entries `mat` are zero except the diagonal entries which must be one, otherwise returns zero.

```c
int fq_mat_is_empty(const fq_mat_t mat, const fq_ctx_t ctx)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.
int fq_mat_is_square(const fq_mat_t mat, const fq_ctx_t ctx)

Returns a non-zero value if the number of rows is equal to the number of columns in mat, and otherwise returns zero.

11.4.10 Addition and subtraction

void fq_mat_add(fq_mat_t C, const fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)

Computes $C = A + B$. Dimensions must be identical.

void fq_mat_sub(fq_mat_t C, const fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)

Computes $C = A - B$. Dimensions must be identical.

void fq_mat_neg(fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)

Sets $B = -A$. Dimensions must be identical.

11.4.11 Matrix multiplication

void fq_mat_mul(fq_mat_t C, const fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and KS multiplication.

void fq_mat_mul_classical(fq_mat_t C, const fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. $C$ is not allowed to be aliased with $A$ or $B$. Uses classical matrix multiplication.

void fq_mat_mul_KS(fq_mat_t C, const fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. $C$ is not allowed to be aliased with $A$ or $B$. Uses Kronecker substitution to perform the multiplication over the integers.

void fq_mat_submul(fq_mat_t D, const fq_mat_t C, const fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)

Sets $D = C + AB$. $C$ and $D$ may be aliased with each other but not with $A$ or $B$.

void fq_mat_mul_vec(fq_struct *c, const fq_mat_t A, const fq_struct *b, slong blen, const fq_ctx_t ctx)

void fq_mat_mul_vec_ptr(fq_struct *const *c, const fq_mat_t A, const fq_struct *const *b, slong blen, const fq_ctx_t ctx)

Compute a matrix-vector product of $A$ and $(b, blen)$ and store the result in $c$. The vector $(b, blen)$ is either truncated or zero-extended to the number of columns of $A$. The number entries written to $c$ is always equal to the number of rows of $A$.

void fq_mat_vec_mul(fq_struct *c, const fq_struct *a, slong alen, const fq_mat_t B, const fq_ctx_t ctx)

void fq_mat_vec_mul_ptr(fq_struct *const *c, const fq_struct *const *a, slong alen, const fq_mat_t B, const fq_ctx_t ctx)

Compute a vector-matrix product of $(a, alen)$ and $B$ and store the result in $c$. The vector $(a, alen)$ is either truncated or zero-extended to the number of rows of $B$. The number entries written to $c$ is always equal to the number of columns of $B$. 

11.4. $fq\_mat.t$ – matrices over finite fields

851
11.4.12 Inverse

```c
int fq_mat_inv(fq_mat_t B, fq_mat_t A, const fq_ctx_t ctx)
```

Sets $B = A^{-1}$ and returns 1 if $A$ is invertible. If $A$ is singular, returns 0 and sets the elements of $B$ to undefined values.

$A$ and $B$ must be square matrices with the same dimensions.

11.4.13 LU decomposition

```c
slong fq_mat_lu(slong *P, fq_mat_t A, int rank_check, const fq_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix $A$, returning the rank of $A$.

If $A$ is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix $L$ and an upper triangular matrix $U$ (the diagonal of $L$ will not be stored explicitly).

If $A$ is an arbitrary matrix of rank $r$, $U$ will be in row echelon form having $r$ nonzero rows, and $L$ will be lower triangular but truncated to $r$ columns, having implicit ones on the $r$ first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if $A$ is detected to be rank-deficient.

This function calls `fq_mat_lu_recursive`.

```c
slong fq_mat_lu_classical(slong *P, fq_mat_t A, int rank_check, const fq_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix $A$, returning the rank of $A$. The behavior of this function is identical to that of `fq_mat_lu`. Uses Gaussian elimination.

```c
slong fq_mat_lu_recursive(slong *P, fq_mat_t A, int rank_check, const fq_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix $A$, returning the rank of $A$. The behavior of this function is identical to that of `fq_mat_lu`. Uses recursive block decomposition, switching to classical Gaussian elimination for sufficiently small blocks.

11.4.14 Reduced row echelon form

```c
slong fq_mat_rref(fq_mat_t A, const fq_ctx_t ctx)
```

Puts $A$ in reduced row echelon form and returns the rank of $A$.

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

```c
slong fq_mat_reduce_row(fq_mat_t A, slong *P, slong *L, slong n, const fq_ctx_t ctx)
```

Reduce row $n$ of the matrix $A$, assuming the prior rows are in Gauss form. However those rows may not be in order. The entry $i$ of the array $P$ is the row of $A$ which has a pivot in the $i$-th column. If no such row exists, the entry of $P$ will be $-1$. The function returns the column in which the $n$-th row has a pivot after reduction. This will always be chosen to be the first available column for a pivot from the left. This information is also updated in $P$. Entry $i$ of the array $L$ contains the number of possibly nonzero columns of $A$ row $i$. This speeds up reduction in the case that $A$ is chambered on the right. Otherwise the entries of $L$ can all be set to the number of columns of $A$. We require the entries of $L$ to be monotonic increasing.
11.4.15 Triangular solving

**void fq_mat_solve_tril(fq_mat_t X, const fq_mat_t L, const fq_mat_t B, int unit, const fq_ctx_t ctx)**

Sets $X = L^{-1}B$ where $L$ is a full rank lower triangular square matrix. If $\text{unit} = 1$, $L$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

**void fq_mat_solve_tril_classical(fq_mat_t X, const fq_mat_t L, const fq_mat_t B, int unit, const fq_ctx_t ctx)**

Sets $X = L^{-1}B$ where $L$ is a full rank lower triangular square matrix. If $\text{unit} = 1$, $L$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

**void fq_mat_solve_tril_recursive(fq_mat_t X, const fq_mat_t L, const fq_mat_t B, int unit, const fq_ctx_t ctx)**

Sets $X = L^{-1}B$ where $L$ is a full rank lower triangular square matrix. If $\text{unit} = 1$, $L$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$
\begin{pmatrix} A & 0 \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}X \\ D^{-1}(Y - CA^{-1}X) \end{pmatrix}
$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

**void fq_mat_solve_triu(fq_mat_t X, const fq_mat_t U, const fq_mat_t B, int unit, const fq_ctx_t ctx)**

Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If $\text{unit} = 1$, $U$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

**void fq_mat_solve_triu_classical(fq_mat_t X, const fq_mat_t U, const fq_mat_t B, int unit, const fq_ctx_t ctx)**

Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If $\text{unit} = 1$, $U$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

**void fq_mat_solve_triu_recursive(fq_mat_t X, const fq_mat_t U, const fq_mat_t B, int unit, const fq_ctx_t ctx)**

Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If $\text{unit} = 1$, $U$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$
\begin{pmatrix} A & B \\ 0 & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}(X - BD^{-1}Y) \\ D^{-1}Y \end{pmatrix}
$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.
11.4.16 Solving

int fq_mat_solve(fq_mat_t X, const fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)

Solves the matrix-matrix equation \( AX = B \).

Returns 1 if \( A \) has full rank; otherwise returns 0 and sets the elements of \( X \) to undefined values.

The matrix \( A \) must be square.

int fq_mat_can_solve(fq_mat_t X, const fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)

Solves the matrix-matrix equation \( AX = B \) over \( F_q \).

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of \( X \) to zero. If more than one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of \( A \) and it may be singular.

11.4.17 Transforms

void fq_mat_similarity(fq_mat_t M, slong r, fq_t d, const fq_ctx_t ctx)

Applies a similarity transform to the \( n \times n \) matrix \( M \) in-place.

If \( P \) is the \( n \times n \) identity matrix the zero entries of whose row \( r \) (0-indexed) have been replaced by \( d \), this transform is equivalent to \( M = P^{-1}MP \).

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

The value \( d \) is required to be reduced modulo the modulus of the entries in the matrix.

11.4.18 Characteristic polynomial

void fq_mat_charpoly_danilevsky(fq_poly_t p, const fq_mat_t M, const fq_ctx_t ctx)

Compute the characteristic polynomial \( p \) of the matrix \( M \). The matrix is assumed to be square.

void fq_mat_charpoly(fq_poly_t p, const fq_mat_t M, const fq_ctx_t ctx)

Compute the characteristic polynomial \( p \) of the matrix \( M \). The matrix is required to be square, otherwise an exception is raised.

11.4.19 Minimal polynomial

void fq_mat_minpoly(fq_poly_t p, const fq_mat_t M, const fq_ctx_t ctx)

Compute the minimal polynomial \( p \) of the matrix \( M \). The matrix is required to be square, otherwise an exception is raised.

11.5 fq_default_mat.h – matrices over finite fields

11.5.1 Types, macros and constants

type fq_default_mat_t
11.5.2 Memory management

```c
void fq_default_mat_init(fq_default_mat_t mat, slong rows, slong cols, const fq_default_ctx_t ctx)
```
Initialises `mat` to a `rows`-by-`cols` matrix with coefficients in \( \mathbb{F}_q \) given by `ctx`. All elements are set to zero.

```c
void fq_default_mat_init_set(fq_default_mat_t mat, fq_default_mat_t src, const fq_default_ctx_t ctx)
```
Initialises `mat` and sets its dimensions and elements to those of `src`.

```c
void fq_default_mat_clear(fq_default_mat_t mat, const fq_default_ctx_t ctx)
```
Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an `fq_default_mat_t` object.

```c
void fq_default_mat_set(fq_default_mat_t mat, const fq_default_mat_t src, const fq_default_ctx_t ctx)
```
Sets `mat` to a copy of `src`. It is assumed that `mat` and `src` have identical dimensions.

11.5.3 Basic properties and manipulation

```c
void fq_default_mat_entry(fq_default_t val, const fq_default_mat_t mat, slong i, slong j, const fq_default_ctx_t ctx)
```
Directly accesses the entry in `mat` in row `i` and column `j`, indexed from zero by setting `val` to the value of that entry. No bounds checking is performed.

```c
void fq_default_mat_entry_set(fq_default_mat_t mat, slong i, slong j, const fq_default_t x, const fq_default_ctx_t ctx)
```
Sets the entry in `mat` in row `i` and column `j` to `x`.

```c
void fq_default_mat_entry_set_fmpz(fq_default_mat_t mat, slong i, slong j, const fmpz_t x, const fq_default_ctx_t ctx)
```
Sets the entry in `mat` in row `i` and column `j` to `x`.

```c
slong fq_default_mat_nrows(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
```
Returns the number of rows in `mat`.

```c
slong fq_default_mat_ncols(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
```
Returns the number of columns in `mat`.

```c
void fq_default_mat_swap(fq_default_mat_t mat1, fq_default_mat_t mat2, const fq_default_ctx_t ctx)
```
Swaps two matrices. The dimensions of `mat1` and `mat2` are allowed to be different.

```c
void fq_default_mat_zero(fq_default_mat_t mat, const fq_default_ctx_t ctx)
```
Sets all entries of `mat` to 0.

```c
void fq_default_mat_one(fq_default_mat_t mat, const fq_default_ctx_t ctx)
```
Sets the diagonal entries of `mat` to 1 and all other entries to 0.

```c
void fq_default_mat_swap_rows(fq_default_mat_t mat, slong *perm, slong r, slong s, const fq_default_ctx_t ctx)
```
Swaps rows `r` and `s` of `mat`. If `perm` is non-NULL, the permutation of the rows will also be applied to `perm`. 

11.5. `fq_default_mat.h` — matrices over finite fields
void fq_default_mat_swap_cols(fq_default_mat_t mat, slong *perm, slong r, slong s, const fq_default_ctx_t ctx)
    Swaps columns r and s of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.

void fq_default_mat_invert_rows(fq_default_mat_t mat, slong *perm, const fq_default_ctx_t ctx)
    Swaps rows i and r - i of mat for 0 <= i < r/2, where r is the number of rows of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

void fq_default_mat_invert_cols(fq_default_mat_t mat, slong *perm, const fq_default_ctx_t ctx)
    Swaps columns i and c - i of mat for 0 <= i < c/2, where c is the number of columns of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.

11.5.4 Conversions

void fq_default_mat_set_nmod_mat(fq_default_mat_t mat1, const nmod_mat_t mat2, const fq_default_ctx_t ctx)
    Sets the matrix mat1 to the matrix mat2.

void fq_default_mat_set_fmpz_mod_mat(fq_default_mat_t mat1, const fmpz_mod_mat_t mat2, const fq_default_ctx_t ctx)
    Sets the matrix mat1 to the matrix mat2.

void fq_default_mat_set_fmpz_mat(fq_default_mat_t mat1, const fmpz_mat_t mat2, const fq_default_ctx_t ctx)
    Sets the matrix mat1 to the matrix mat2, reducing the entries modulo the characteristic of the finite field.

11.5.5 Concatenate

void fq_default_mat_concat_vertical(fq_default_mat_t res, const fq_default_mat_t mat1, const fq_default_mat_t mat2, const fq_default_ctx_t ctx)
    Sets res to vertical concatenation of (mat1, mat2) in that order. Matrix dimensions : mat1 : m \times n, mat2 : k \times n, res : (m + k) \times n.

void fq_default_mat_concat_horizontal(fq_default_mat_t res, const fq_default_mat_t mat1, const fq_default_mat_t mat2, const fq_default_ctx_t ctx)
    Sets res to horizontal concatenation of (mat1, mat2) in that order. Matrix dimensions : mat1 : m \times n, mat2 : m \times k, res : m \times (n + k).

11.5.6 Printing

int fq_default_mat_print_pretty(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
    Pretty-prints mat to stdout. A header is printed followed by the rows enclosed in brackets.

int fq_default_mat_fprint_pretty(FILE *file, const fq_default_mat_t mat, const fq_default_ctx_t ctx)
    Pretty-prints mat to file. A header is printed followed by the rows enclosed in brackets.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.
int fq_default_mat_print(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
    Prints mat to stdout. A header is printed followed by the rows enclosed in brackets.

int fq_default_mat_fprint(FILE *file, const fq_default_mat_t mat, const fq_default_ctx_t ctx)
    Prints mat to file. A header is printed followed by the rows enclosed in brackets.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

11.5.7 Window

void fq_default_mat_window_init(fq_default_mat_t window, const fq_default_mat_t mat, slong r1, slong c1, slong r2, slong c2, const fq_default_ctx_t ctx)
    Initializes the matrix window to be an r2 - r1 by c2 - c1 submatrix of mat whose (0,0) entry is the (r1, c1) entry of mat. The memory for the elements of window is shared with mat.

void fq_default_mat_window_clear(fq_default_mat_t window, const fq_default_ctx_t ctx)
    Clears the matrix window and releases any memory that it uses. Note that the memory to the underlying matrix that window points to is not freed.

11.5.8 Random matrix generation

void fq_default_mat_randtest(fq_default_mat_t mat, flint_rand_t state, const fq_default_ctx_t ctx)
    Sets the elements of mat to random elements of F_q, given by ctx.

int fq_default_mat_randpermdiag(fq_mat_t mat, flint_rand_t state, fq_struct *diag, slong n, const fq_ctx_t ctx)
    Sets mat to a random permutation of the diagonal matrix with n leading entries given by the vector diag. It is assumed that the main diagonal of mat has room for at least n entries.
    Returns 0 or 1, depending on whether the permutation is even or odd respectively.

void fq_default_mat_randrank(fq_default_mat_t mat, flint_rand_t state, slong rank, const fq_default_ctx_t ctx)
    Sets mat to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of F_q.
    The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling fq_default_mat_randops() .

void fq_default_mat_randops(fq_default_mat_t mat, slong count, flint_rand_t state, const fq_default_ctx_t ctx)
    Randomises mat by performing elementary row or column operations. More precisely, at most count random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

void fq_default_mat_randtril(fq_default_mat_t mat, flint_rand_t state, int unit, const fq_default_ctx_t ctx)
    Sets mat to a random lower triangular matrix. If unit is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

void fq_default_mat_randtriu(fq_default_mat_t mat, flint_rand_t state, int unit, const fq_default_ctx_t ctx)
    Sets mat to a random upper triangular matrix. If unit is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.
11.5.9 Comparison

```c
int fq_default_mat_equal(const fq_default_mat_t mat1, const fq_default_mat_t mat2, const fq_default_ctx_t ctx)
```

Returns nonzero if mat1 and mat2 have the same dimensions and elements, and zero otherwise.

```c
int fq_default_mat_is_zero(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
```

Returns a non-zero value if all entries of mat are zero, and otherwise returns zero.

```c
int fq_default_mat_is_one(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
```

Returns a non-zero value if all diagonal entries of mat are one and all other entries are zero, and otherwise returns zero.

```c
int fq_default_mat_is_empty(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
```

Returns a non-zero value if the number of rows or the number of columns in mat is zero, and otherwise returns zero.

```c
int fq_default_mat_is_square(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
```

Returns a non-zero value if the number of rows is equal to the number of columns in mat, and otherwise returns zero.

11.5.10 Addition and subtraction

```c
void fq_default_mat_add(fq_default_mat_t C, const fq_default_mat_t A, const fq_default_mat_t B, const fq_default_ctx_t ctx)
```

Computes $C = A + B$. Dimensions must be identical.

```c
void fq_default_mat_sub(fq_default_mat_t C, const fq_default_mat_t A, const fq_default_mat_t B, const fq_default_ctx_t ctx)
```

Computes $C = A - B$. Dimensions must be identical.

```c
void fq_default_mat_neg(fq_default_mat_t A, const fq_default_mat_t B, const fq_default_ctx_t ctx)
```

Sets $B = -A$. Dimensions must be identical.

11.5.11 Matrix multiplication

```c
void fq_default_mat_mul(fq_default_mat_t C, const fq_default_mat_t A, const fq_default_mat_t B, const fq_default_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and KS multiplication.

```c
void fq_default_mat_submul(fq_default_mat_t D, const fq_default_mat_t C, const fq_default_mat_t A, const fq_default_mat_t B, const fq_default_ctx_t ctx)
```

Sets $D = C + AB$. $C$ and $D$ may be aliased with each other but not with $A$ or $B$. 858 Chapter 11. Finite fields
11.5.12 Inverse

```c
int fq_default_mat_inv(fq_default_mat_t B, fq_default_mat_t A, const fq_default_ctx_t ctx)
```

Sets $B = A^{-1}$ and returns 1 if $A$ is invertible. If $A$ is singular, returns 0 and sets the elements of $B$ to undefined values.

$A$ and $B$ must be square matrices with the same dimensions.

11.5.13 LU decomposition

```c
slong fq_default_mat_lu(slong *P, fq_default_mat_t A, int rank_check, const fq_default_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix $A$, returning the rank of $A$.

If $A$ is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix $L$ and an upper triangular matrix $U$ (the diagonal of $L$ will not be stored explicitly).

If $A$ is an arbitrary matrix of rank $r$, $U$ will be in row echelon form having $r$ nonzero rows, and $L$ will be lower triangular but truncated to $r$ columns, having implicit ones on the $r$ first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if $A$ is detected to be rank-deficient.

This function calls `fq_default_mat_lu_recursive`.

11.5.14 Reduced row echelon form

```c
slong fq_default_mat_rref(fq_default_mat_t A, const fq_default_ctx_t ctx)
```

Puts $A$ in reduced row echelon form and returns the rank of $A$.

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

11.5.15 Triangular solving

```c
void fq_default_mat_solve_tril(fq_default_mat_t X, const fq_default_mat_t L, const fq_default_mat_t B, int unit, const fq_default_ctx_t ctx)
```

Sets $X = L^{-1}B$ where $L$ is a full rank lower triangular square matrix. If `unit` = 1, $L$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```c
void fq_default_mat_solve_triu(fq_default_mat_t X, const fq_default_mat_t U, const fq_default_mat_t B, int unit, const fq_default_ctx_t ctx)
```

Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If `unit` = 1, $U$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.
11.5.16 Solving

```c
int fq_default_mat_solve(fq_default_mat_t X, const fq_default_mat_t A, const fq_default_mat_t B, const fq_default_ctx_t ctx)
```

Solves the matrix-matrix equation $AX = B$.

- Returns 1 if $A$ has full rank; otherwise returns 0 and sets the elements of $X$ to undefined values.
- The matrix $A$ must be square.

```c
int fq_default_mat_can_solve(fq_default_mat_t X, const fq_default_mat_t A, const fq_default_mat_t B, const fq_default_ctx_t ctx)
```

Solves the matrix-matrix equation $AX = B$ over $F_q$.

- Returns 1 if a solution exists; otherwise returns 0 and sets the elements of $X$ to zero. If more than one solution exists, one of the valid solutions is given.
- There are no restrictions on the shape of $A$ and it may be singular.

11.5.17 Transforms

```c
void fq_default_mat_similarity(fq_default_mat_t M, slong r, fq_default_t d, const fq_default_ctx_t ctx)
```

Applies a similarity transform to the $n \times n$ matrix $M$ in-place.

- If $P$ is the $n \times n$ identity matrix the zero entries of whose row $r$ (0-indexed) have been replaced by $d$, this transform is equivalent to $M = P^{-1}MP$.
- Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.
- The value $d$ is required to be reduced modulo the determinant, characteristic polynomial and minimal polynomial.

11.5.18 Characteristic polynomial

```c
void fq_default_mat_charpoly(fq_default_poly_t p, const fq_default_mat_t M, const fq_default_ctx_t ctx)
```

Compute the characteristic polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised.

11.5.19 Minimal polynomial

```c
void fq_default_mat_minpoly(fq_default_poly_t p, const fq_default_mat_t M, const fq_default_ctx_t ctx)
```

Compute the minimal polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised.

11.6 fq_poly.h – univariate polynomials over finite fields

We represent a polynomial in $F_q[X]$ as a `struct` which includes an array `coeffs` with the coefficients, as well as the length `length` and the number `alloc` of coefficients for which memory has been allocated.

As a data structure, we call this polynomial `normalised` if the top coefficient is non-zero.

Unless otherwise stated here, all functions that deal with polynomials assume that the $F_q$ context of said polynomials are compatible, i.e., it assumes that the fields are generated by the same polynomial.
11.6.1 Types, macros and constants

type fq_poly_struct

type fq_poly_t

11.6.2 Memory management

void fq_poly_init(fq_poly_t poly, const fq_ctx_t ctx)

Initialises poly for use, with context ctx, and setting its length to zero. A corresponding call to fq_poly_clear() must be made after finishing with the fq_poly_t to free the memory used by the polynomial.

void fq_poly_init2(fq_poly_t poly, slong alloc, const fq_ctx_t ctx)

Initialises poly with space for at least alloc coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to fq_poly_clear() must be made after finishing with the fq_poly_t to free the memory used by the polynomial.

void fq_poly_realloc(fq_poly_t poly, slong alloc, const fq_ctx_t ctx)

Reallocates the given polynomial to have space for alloc coefficients. If alloc is zero the polynomial is cleared and then reinitialised. If the current length is greater than alloc the polynomial is first truncated to length alloc.

void fq_poly_fit_length(fq_poly_t poly, slong len, const fq_ctx_t ctx)

If len is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least len coefficients. No data is lost when calling this function.

The function efficiently deals with the case where fit_length is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void _fq_poly_set_length(fq_poly_t poly, slong newlen, const fq_ctx_t ctx)

Sets the coefficients of poly beyond len to zero and sets the length of poly to len.

void fq_poly_clear(fq_poly_t poly, const fq_ctx_t ctx)

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

void _fq_poly_normalise(fq_poly_t poly, const fq_ctx_t ctx)

Sets the length of poly so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void _fq_poly_normalise2(const fq_struct *poly, slong *length, const fq_ctx_t ctx)

Sets the length length of (poly,length) so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void fq_poly_truncate(fq_poly_t poly, slong newlen, const fq_ctx_t ctx)

Truncates the polynomial to length at most n.

void fq_poly_set_trunc(fq_poly_t poly1, fq_poly_t poly2, slong newlen, const fq_ctx_t ctx)

Sets poly1 to poly2 truncated to length n.

void _fq_poly_reverse(fq_struct *output, const fq_struct *input, slong len, slong m, const fq_ctx_t ctx)

Sets output to the reverse of input, which is of length len, but thinking of it as a polynomial of length m, notionally zero-padded if necessary. The length m must be non-negative, but there are no other restrictions. The polynomial output must have space for m coefficients.
Sets output to the reverse of input, thinking of it as a polynomial of length \( m \), notionally zero-padded if necessary). The length \( m \) must be non-negative, but there are no other restrictions. The output polynomial will be set to length \( m \) and then normalised.

### 11.6.3 Polynomial parameters

- **slong** `fq_poly_degree(const fq_poly_t poly, const fq_ctx_t ctx)`
  - Returns the degree of the polynomial `poly`.

- **slong** `fq_poly_length(const fq_poly_t poly, const fq_ctx_t ctx)`
  - Returns the length of the polynomial `poly`.

- **fq_struct** `*fq_poly_lead(const fq_poly_t poly, const fq_ctx_t ctx)`
  - Returns a pointer to the leading coefficient of `poly`, or `NULL` if `poly` is the zero polynomial.

### 11.6.4 Randomisation

- **void** `fq_poly_randtest(fq_poly_t f, flint_rand_t state, slong len, const fq_ctx_t ctx)`
  - Sets \( f \) to a random polynomial of length at most \( \text{len} \) with entries in the field described by \( \text{ctx} \).

- **void** `fq_poly_randtest_not_zero(fq_poly_t f, flint_rand_t state, slong len, const fq_ctx_t ctx)`
  - Same as `fq_poly_randtest` but guarantees that the polynomial is not zero.

- **void** `fq_poly_randtest_monic(fq_poly_t f, flint_rand_t state, slong len, const fq_ctx_t ctx)`
  - Sets \( f \) to a random monic polynomial of length \( \text{len} \) with entries in the field described by \( \text{ctx} \).

- **void** `fq_poly_randtest_irreducible(fq_poly_t f, flint_rand_t state, slong len, const fq_ctx_t ctx)`
  - Sets \( f \) to a random monic, irreducible polynomial of length \( \text{len} \) with entries in the field described by \( \text{ctx} \).

### 11.6.5 Assignment and basic manipulation

- **void** `_fq_poly_set(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)`
  - Sets \((\text{rop}, \text{len})\) to \((\text{op}, \text{len})\).

- **void** `fq_poly_set(fq_poly_t poly1, const fq_poly_t poly2, const fq_ctx_t ctx)`
  - Sets the polynomial `poly1` to the polynomial `poly2`.

- **void** `fq_poly_set_fq(fq_poly_t poly, const fq_t c, const fq_ctx_t ctx)`
  - Sets the polynomial `poly` to `c`.

- **void** `fq_poly_set_fmpz_mod_poly(fq_poly_t rop, const fmpz_mod_poly_t op, const fq_ctx_t ctx)`
  - Sets the polynomial `rop` to the polynomial `op`.

- **void** `fq_poly_set_nthmpoly(fq_poly_t rop, const nmpoly_t op, const fq_ctx_t ctx)`
  - Sets the polynomial `rop` to the polynomial `op`.

- **void** `fq_poly_swap(fq_poly_t op1, fq_poly_t op2, const fq_ctx_t ctx)`
  - Swaps the two polynomials `op1` and `op2`.

- **void** `_fq_poly_zero(fq_struct *rop, slong len, const fq_ctx_t ctx)`
  - Sets \((\text{rop}, \text{len})\) to the zero polynomial.

- **void** `fq_poly_zero(fq_poly_t poly, const fq_ctx_t ctx)`
  - Sets `poly` to the zero polynomial.
void fq_poly_one(fq_poly_t poly, const fq_ctx_t ctx)
    Sets poly to the constant polynomial 1.

don t

void fq_poly_gen(fq_poly_t poly, const fq_ctx_t ctx)
    Sets poly to the polynomial x.

void fq_poly_make_monic(fq_poly_t rop, const fq_poly_t op, const fq_ctx_t ctx)
    Sets rop to op, normed to have leading coefficient 1.

void _fq_poly_make_monic(fq_struct *rop, const fq_struct *op, slong length, const fq_ctx_t ctx)
    Sets rop to (op,length), normed to have leading coefficient 1. Assumes that rop has enough space for the polynomial, assumes that op is not zero (and thus has an invertible leading coefficient).

11.6.6 Getting and setting coefficients

void fq_poly_get_coeff(fq_t x, const fq_poly_t poly, slong n, const fq_ctx_t ctx)
    Sets x to the coefficient of $X^n$ in poly.

void fq_poly_set_coeff(fq_poly_t poly, slong n, const fq_t x, const fq_ctx_t ctx)
    Sets the coefficient of $X^n$ in poly to x.

void fq_poly_set_coeff_fmpz(fq_poly_t poly, slong n, const fmpz_t x, const fq_ctx_t ctx)
    Sets the coefficient of $X^n$ in the polynomial to x, assuming $n \geq 0$.

11.6.7 Comparison

int fq_poly_equal(const fq_poly_t poly1, const fq_poly_t poly2, const fq_ctx_t ctx)
    Returns nonzero if the two polynomials poly1 and poly2 are equal, otherwise returns zero.

int fq_poly_equal_trunc(const fq_poly_t poly1, const fq_poly_t poly2, slong n, const fq_ctx_t ctx)
    Notionally truncate poly1 and poly2 to length n and return nonzero if they are equal, otherwise return zero.

int fq_poly_is_zero(const fq_poly_t poly, const fq_ctx_t ctx)
    Returns whether the polynomial poly is the zero polynomial.

int fq_poly_is_one(const fq_poly_t op, const fq_ctx_t ctx)
    Returns whether the polynomial poly is equal to the constant polynomial 1.

int fq_poly_is_gen(const fq_poly_t op, const fq_ctx_t ctx)
    Returns whether the polynomial poly is equal to the polynomial x.

int fq_poly_is_unit(const fq_poly_t op, const fq_ctx_t ctx)
    Returns whether the polynomial poly is a unit in the polynomial ring $\mathbb{F}_q[X]$, i.e. if it has degree 0 and is non-zero.

int fq_poly_equal_fq(const fq_poly_t poly, const fq_t c, const fq_ctx_t ctx)
    Returns whether the polynomial poly is equal the (constant) $\mathbb{F}_q$ element c.

11.6. fq_poly.h – univariate polynomials over finite fields
**11.6.8 Addition and subtraction**

void \_fq\_poly\_add (fq\_struct \*res, const fq\_struct \*poly1, slong len1, const fq\_struct \*poly2, slong len2, const fq\_ctx\_t ctx)

Sets res to the sum of (poly1,len1) and (poly2,len2).

void fq\_poly\_add (fq\_poly\_t res, const fq\_poly\_t poly1, const fq\_poly\_t poly2, const fq\_ctx\_t ctx)

Sets res to the sum of poly1 and poly2.

void fq\_poly\_add\_si (fq\_poly\_t res, const fq\_poly\_t poly1, slong c, const fq\_ctx\_t ctx)

Sets res to the sum of poly1 and c.

void fq\_poly\_add\_series (fq\_poly\_t res, const fq\_poly\_t poly1, const fq\_poly\_t poly2, slong n, const fq\_ctx\_t ctx)

Notionally truncate poly1 and poly2 to length n and set res to the sum.

void \_fq\_poly\_sub (fq\_struct \*res, const fq\_struct \*poly1, slong len1, const fq\_struct \*poly2, slong len2, const fq\_ctx\_t ctx)

Sets res to the difference of (poly1,len1) and (poly2,len2).

void fq\_poly\_sub (fq\_poly\_t res, const fq\_poly\_t poly1, const fq\_poly\_t poly2, const fq\_ctx\_t ctx)

Sets res to the difference of poly1 and poly2.

void fq\_poly\_sub\_series (fq\_poly\_t res, const fq\_poly\_t poly1, const fq\_poly\_t poly2, slong n, const fq\_ctx\_t ctx)

Notionally truncate poly1 and poly2 to length n and set res to the difference.

void \_fq\_poly\_neg (fq\_struct \*rop, const fq\_struct \*op, slong len, const fq\_ctx\_t ctx)

Sets rop to the additive inverse of (poly,len).

void fq\_poly\_neg (fq\_poly\_t res, const fq\_poly\_t poly, const fq\_ctx\_t ctx)

Sets res to the additive inverse of poly.

**11.6.9 Scalar multiplication and division**

void \_fq\_poly\_scalar\_mul\_fq (fq\_struct \*rop, const fq\_struct \*op, slong len, const fq\_t x, const fq\_ctx\_t ctx)

Sets (rop,len) to the product of (op,len) by the scalar x, in the context defined by ctx.

void fq\_poly\_scalar\_mul\_fq (fq\_poly\_t rop, const fq\_poly\_t op, const fq\_t x, const fq\_ctx\_t ctx)

Sets rop to the product of op by the scalar x, in the context defined by ctx.

void \_fq\_poly\_scalar\_addmul\_fq (fq\_struct \*rop, const fq\_struct \*op, slong len, const fq\_t x, const fq\_ctx\_t ctx)

Adds to (rop,len) the product of (op,len) by the scalar x, in the context defined by ctx. In particular, assumes the same length for op and rop.

void fq\_poly\_scalar\_addmul\_fq (fq\_poly\_t rop, const fq\_poly\_t op, const fq\_t x, const fq\_ctx\_t ctx)

Adds to rop the product of op by the scalar x, in the context defined by ctx.

void \_fq\_poly\_scalar\_submul\_fq (fq\_struct \*rop, const fq\_struct \*op, slong len, const fq\_t x, const fq\_ctx\_t ctx)

Subtracts from (rop,len) the product of (op,len) by the scalar x, in the context defined by ctx. In particular, assumes the same length for op and rop.

void fq\_poly\_scalar\_submul\_fq (fq\_poly\_t rop, const fq\_poly\_t op, const fq\_t x, const fq\_ctx\_t ctx)

Subtracts from rop the product of op by the scalar x, in the context defined by ctx.
void _fq_poly_scalar_div_fq(fq_struct *rop, const fq_struct *op, slong len, const fq_t x, const fq_ctx_t ctx)

Sets (rop, len) to the quotient of (op, len) by the scalar x, in the context defined by ctx. An exception is raised if x is zero.

void fq_poly_scalar_div_fq fq_poly_t rop, const fq_poly_t op, const fq_t x, const fq_ctx_t ctx)

Sets rop to the quotient of op by the scalar x, in the context defined by ctx. An exception is raised if x is zero.

11.6.10 Multiplication

void _fq_poly_mul_classical(fq_struct *rop, const fq_struct *op1, slong len1, const fq_t x, const fq_ctx_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), assuming that len1 is at least len2 and neither is zero.

Permits zero padding. Does not support aliasing of rop with either op1 or op2.

void fq_poly_mul_classical(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)

Sets rop to the product of op1 and op2 using classical polynomial multiplication.

void _fq_poly_mul_reorder(fq_struct *rop, const fq_struct *op1, slong len1, const fq_t x, const fq_ctx_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), assuming that len1 and len2 are non-zero.

Permits zero padding. Supports aliasing.

void fq_poly_mul_reorder(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)

Sets rop to the product of op1 and op2, reordering the two indeterminates 𝑋 and 𝑌 when viewing the polynomials as elements of 𝐹ₚ[𝑋,𝑌].

Suppose 𝐹ₗ = 𝐹ₚ[𝑋]/(f(𝑋)) and recall that elements of 𝐹ₗ are internally represented by elements of type fmpz_poly. For small degree extensions but polynomials in 𝐹ₗ[𝑌] of large degree ℎ, we change the representation to

\[ g(𝑌) = \sum_{i=0}^{n} a_i(𝑋)𝑌^i = \sum_{j=0}^{d} \sum_{i=0}^{n} \text{Coeff}(a_i(𝑋), j)𝑌^i. \]

This allows us to use a poor algorithm (such as classical multiplication) in the 𝑋-direction and leverage the existing fast integer multiplication routines in the 𝑌-direction where the polynomial degree ℎ is large.

void _fq_poly_mul_univariate(fq_struct *rop, const fq_struct *op1, slong len1, const fq_t x, const fq_ctx_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2).

Permits zero padding and places no assumptions on the lengths len1 and len2. Supports aliasing.

void fq_poly_mul_univariate(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)

Sets rop to the product of op1 and op2 using a bivariate to univariate transformation and reducing this problem to multiplying two univariate polynomials.

11.6. fq_poly.h – univariate polynomials over finite fields
void _fq_poly_mul(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2, slong len2, const fq_ctx_t ctx)

Sets \((\text{rop}, \text{len1} + \text{len2} - 1)\) to the product of \((\text{op1}, \text{len1})\) and \((\text{op2}, \text{len2})\).
Permits zero padding and places no assumptions on the lengths \(\text{len1}\) and \(\text{len2}\). Supports aliasing.

void _fq_poly_mul_KS(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)

Sets \(\text{rop}\) to the product of \(\text{op1}\) and \(\text{op2}\) using Kronecker substitution, that is, by encoding each coefficient in \(F_q\) as an integer and reducing this problem to multiplying two polynomials over the integers.

void _fq_poly_mullow_classical(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2, slong len2, const fq_ctx_t ctx)

Sets \((\text{rop}, n)\) to the first \(n\) coefficients of the product of \((\text{op1}, \text{len1})\) multiplied by \((\text{op2}, \text{len2})\).
Assumes \(0 < n \leq \text{len1} + \text{len2} - 1\). Assumes neither \(\text{len1}\) nor \(\text{len2}\) is zero.

void _fq_poly_mullow_univariate(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2, slong len2, slong n, const fq_ctx_t ctx)

Sets \((\text{rop}, n)\) to the lowest \(n\) coefficients of the product of \((\text{op1}, \text{len1})\) and \((\text{op2}, \text{len2})\), computed using a bivariate to univariate transformation.
Assumes that \(\text{len1}\) and \(\text{len2}\) are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes \(n\) is positive. Supports aliasing between \text{res}, \text{poly1} and \text{poly2}.

void _fq_poly_mullow_KS(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2, slong len2, slong n, const fq_ctx_t ctx)

Sets \((\text{rop}, n)\) to the lowest \(n\) coefficients of the product of \((\text{op1}, \text{len1})\) and \((\text{op2}, \text{len2})\). Assumens that \(\text{len1}\) and \(\text{len2}\) are positive, but does allow for the polynomials to be zero-padded.
The polynomials may be zero, too. Assumes \(n\) is positive. Supports aliasing between \text{rop}, \text{op1} and \text{op2}.

void _fq_poly_mullow(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, slong len2, slong n, const fq_ctx_t ctx)

Sets \((\text{rop}, n)\) to the lowest \(n\) coefficients of the product of \((\text{op1}, \text{len1})\) and \((\text{op2}, \text{len2})\).
Assumes \(0 < n \leq \text{len1} + \text{len2} - 1\). Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.
void fq_poly_mullow(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, slong n, const fq_ctx_t ctx)

Sets rop to the lowest n coefficients of the product of op1 and op2.

void _fq_poly_mulhigh_classical(fq_struct* res, const fq_struct* poly1, slong len1, const fq_struct* poly2, slong len2, slong start, const fq_ctx_t ctx)

Computes the product of (poly1, len1) and (poly2, len2) and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Assumes that len1 ≥ len2 > 0. Aliasing of inputs and output is not permitted. Algorithm is classical multiplication.

void fq_poly_mullow(fq_poly_t res, const fq_poly_t poly1, const fq_poly_t poly2, slong start, const fq_ctx_t ctx)

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Algorithm is classical multiplication.

void _fq_poly_mulhigh_classical(fq_struct* res, const fq_struct* poly1, slong len1, const fq_struct* poly2, slong len2, slong start, const fq_ctx_t ctx)

Computes the product of (poly1, len1) and (poly2, len2) and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Assumes that len1 ≥ len2 > 0. Aliasing of inputs and output is not permitted.

void fq_poly_mullow(fq_poly_t res, const fq_poly_t poly1, const fq_poly_t poly2, slong start, const fq_ctx_t ctx)

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced.

void _fq_poly_mulmod(fq_struct* res, const fq_struct* poly1, slong len1, const fq_struct* poly2, slong len2, const fq_struct* f, slong lenf, const fq_ctx_t ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. It is required that len1 + len2 - lenf > 0, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use _fq_poly_mul instead. Aliasing of f and res is not permitted.

void fq_poly_mulmod(fq_poly_t res, const fq_poly_t poly1, const fq_poly_t poly2, const fq_poly_t f, const fq_ctx_t ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. f is the inverse of the reverse of f mod x^lenf. Aliasing of res with any of the inputs is not permitted.

void _fq_poly_mulmod_preinv(fq_struct* res, const fq_struct* poly1, slong len1, const fq_struct* poly2, slong len2, const fq_struct* f, slong lenf, const fq_struct* finv, slong lenfinv, const fq_ctx_t ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. finv is the inverse of the reverse of f mod x^-lenf. Aliasing of res with any of the inputs is not permitted.

void fq_poly_mulmod_preinv(fq_poly_t res, const fq_poly_t poly1, const fq_poly_t poly2, const fq_poly_t f, const fq_poly_t finv, const fq_ctx_t ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. finv is the inverse of the reverse of f.
### 11.6.11 Squaring

```c
void _fq_poly_sqr_classical(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
Sets (rop, 2*len - 1) to the square of (op, len), assuming that (op,len) is not zero and using classical polynomial multiplication.
Permits zero padding. Does not support aliasing of rop with either op1 or op2.
```

```c
void fq_poly_sqr_classical(fq_poly_t rop, const fq_poly_t op, const fq_ctx_t ctx)
Sets rop to the square of op using classical polynomial multiplication.
```

```c
void _fq_poly_sqr_reorder(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
Sets (rop, 2*len- 1) to the square of (op, len), assuming that len is not zero reordering the two indeterminates X and Y when viewing the polynomials as elements of \( \mathbb{F}_p[X,Y] \).
Permits zero padding. Supports aliasing.
```

```c
void fq_poly_sqr_reorder(fq_poly_t rop, const fq_poly_t op, const fq_ctx_t ctx)
Sets rop to the square of op, assuming that len is not zero reordering the two indeterminates X and Y when viewing the polynomials as elements of \( \mathbb{F}_p[X,Y] \). See fq_poly_mul_reorder.
```

```c
void _fq_poly_sqr_KS(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
Sets (rop, 2*len - 1) to the square of (op, len).
Permits zero padding and places no assumptions on the lengths len1 and len2. Supports aliasing.
```

```c
void fq_poly_sqr_KS(fq_poly_t rop, const fq_poly_t op, const fq_ctx_t ctx)
Sets rop to the square of op using Kronecker substitution, that is, by encoding each coefficient in \( \mathbb{F}_q \) as an integer and reducing this problem to multiplying two polynomials over the integers.
```

```c
void _fq_poly_sqr(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
Sets (rop, 2*len - 1) to the square of (op, len), choosing an appropriate algorithm.
Permits zero padding. Does not support aliasing.
```

```c
void fq_poly_sqr(fq_poly_t rop, const fq_poly_t op, const fq_ctx_t ctx)
Sets rop to the square of op, choosing an appropriate algorithm.
```

### 11.6.12 Powering

```c
void _fq_poly_pow(fq_struct *rop, const fq_struct *op, slong len, ulong e, const fq_ctx_t ctx)
Sets rop = op^e, assuming that e, len > 0 and that rop has space for e*(len - 1) + 1 coefficients. Does not support aliasing.
```

```c
void fq_poly_pow(fq_poly_t rop, const fq_poly_t op, ulong e, const fq_ctx_t ctx)
Computes rop = op^e. If e is zero, returns one, so that in particular 0^0 = 1.
```

```c
void _fq_poly_powmod_ui_binexp(fq_struct *res, const fq_struct *poly, ulong e, const fq_struct *f, slong lenf, const fq_ctx_t ctx)
Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.
```

```c
void fq_poly_powmod_ui_binexp(fq_poly_t res, const fq_poly_t poly, ulong e, const fq_poly_t f, const fq_ctx_t ctx)
Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0.
```
void _fq_poly_powmod_ui_binexp_preinv(fq_struct *res, const fq_struct *poly, ulong e, const
fq_struct *f, slong lenf, const fq_struct *finv, slong lenfinv, const fq_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require finv to be the inverse of the reverse of f.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_poly_powmod_ui_binexp_preinv(fq_poly_t res, const fq_poly_t poly, ulong e, const
fq_poly_t f, const fq_poly_t finv, const fq_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require finv to be the inverse of the reverse of f.

void _fq_poly_powmod_fmpz_binexp(fq_struct *res, const fq_struct *poly, const fmpz_t e, const
fq_struct *f, slong lenf, const fq_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_poly_powmod_fmpz_binexp(fq_poly_t res, const fq_poly_t poly, const fmpz_t e, const
fq_poly_t f, const fq_poly_t finv, const fq_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require finv to be the inverse of the reverse of f.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void _fq_poly_powmod_fmpz_binexp_preinv(fq_struct *res, const fq_struct *poly, const fmpz_t e,
const fq_struct *f, slong lenf, const fq_struct *finv, slong lenfinv, const fq_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require finv to be the inverse of the reverse of f.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_poly_powmod_fmpz_binexp_preinv(fq_poly_t res, const fq_poly_t poly, const fmpz_t e,
const fq_poly_t f, const fq_poly_t finv, const fq_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require finv to be the inverse of the reverse of f.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void _fq_poly_powmod_fmpz_sliding_preinv(fq_struct *res, const fq_struct *poly, const fmpz_t e,
const fq_struct *f, slong lenf, const fq_struct *finv, slong lenfinv, const fq_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using sliding-window exponentiation with window size k. We require e > 0. We require finv to be the inverse of the reverse of f. If k is set to zero, then an “optimum” size will be selected automatically base on e.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_poly_powmod_fmpz_sliding_preinv(fq_poly_t res, const fq_poly_t poly, const fmpz_t e,
ulong k, const fq_poly_t f, const fq_poly_t finv, const
fq_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using sliding-window exponentiation with window size k. We require e >= 0. We require finv to be the inverse of the reverse of f. If k is set to zero, then an “optimum” size will be selected automatically base on e.
void _fq_poly_powmod_x_fmpz_preinv(fq_struct *res, const fmpz_t e, const fq_struct *f, slong lenf, const fq_struct *finv, slong lenfinv, const fq_ctx_t ctx)

Sets res to x raised to the power e modulo f, using sliding window exponentiation. We require e > 0. We require finv to be the inverse of the reverse of f.

We require lenf > 2. The output res must have room for lenf - 1 coefficients.

void fq_poly_powmod_x_fmpz(fq_poly_t res, const fmpz_t e, const fq_poly_t f, const fq_poly_t finv, const fq_ctx_t ctx)

Sets res to x raised to the power e modulo f, using sliding window exponentiation. We require e >= 0. We require finv to be the inverse of the reverse of f.

void _fq_poly_pow_trunc_binexp(fq_struct *res, const fq_struct *poly, ulong e, slong trunc, const fq_ctx_t ctx)

Sets res to the low trunc coefficients of poly (assumed to be zero padded if necessary to length trunc) to the power e. This is equivalent to doing a powering followed by a truncation. We require that res has enough space for trunc coefficients, that trunc > 0 and that e > 1. Aliasing is not permitted. Uses the binary exponentiation method.

void fq_poly_pow_trunc_binexp(fq_poly_t res, const fq_poly_t poly, ulong e, slong trunc, const fq_ctx_t ctx)

Sets res to the low trunc coefficients of poly to the power e. This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.

void _fq_poly_pow_trunc(fq_struct *res, const fq_struct *poly, ulong e, slong trunc, const fq_ctx_t ctx)

Sets res to the low trunc coefficients of poly (assumed to be zero padded if necessary to length trunc) to the power e. This is equivalent to doing a powering followed by a truncation. We require that res has enough space for trunc coefficients, that trunc > 0 and that e > 1. Aliasing is not permitted.

void fq_poly_pow_trunc(fq_poly_t res, const fq_poly_t poly, ulong e, slong trunc, const fq_ctx_t ctx)

Sets res to the low trunc coefficients of poly to the power e. This is equivalent to doing a powering followed by a truncation.

11.6.13 Shifting

void _fq_poly_shift_left(fq_struct *rop, const fq_struct *op, slong len, slong n, const fq_ctx_t ctx)

Sets (rop, len + n) to (op, len) shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that len and n are positive, and that rop fits len + n elements. Supports aliasing between rop and op.

void fq_poly_shift_left(fq_poly_t rop, const fq_poly_t op, slong n, const fq_ctx_t ctx)

Sets rop to op shifted left by n coeffs. Zero coefficients are inserted.

void _fq_poly_shift_right(fq_struct *rop, const fq_struct *op, slong len, slong n, const fq_ctx_t ctx)

Sets (rop, len - n) to (op, len) shifted right by n coefficients.

Assumes that len and n are positive, that len > n, and that rop fits len - n elements. Supports aliasing between rop and op, although in this case the top coefficients of op are not set to zero.

void fq_poly_shift_right(fq_poly_t rop, const fq_poly_t op, slong n, const fq_ctx_t ctx)

Sets rop to op shifted right by n coefficients. If n is equal to or greater than the current length of op, rop is set to the zero polynomial.
11.6.14 Norms

\begin{verbatim}
slong _fq_poly_hamming_weight(const fq_struct *op, slong len, const fq_ctx_t ctx)
  Returns the number of non-zero entries in (op, len).
slong fq_poly_hamming_weight(const fq_poly_t op, const fq_ctx_t ctx)
  Returns the number of non-zero entries in the polynomial op.
\end{verbatim}

11.6.15 Euclidean division

\begin{verbatim}
void _fq_poly_divrem(fq_struct *Q, fq_struct *R, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB, const fq_t invB, const fq_ctx_t ctx)
  Computes (Q, lenA - lenB + 1), (R, lenA) such that A = BQ + R with 0 \leq len(R) < len(B).
  Assumes that the leading coefficient of B is invertible and that invB is its inverse.
  Assumes that len(A), len(B) > 0. Allows zero-padding in (A, lenA). R and A may be aliased, but
  apart from this no aliasing of input and output operands is allowed.

void fq_poly_divrem(fq_poly_t Q, fq_poly_t R, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
  Computes Q, R such that A = BQ + R with 0 \leq len(R) < len(B).
  Assumes that the leading coefficient of B is invertible. This can be taken for granted the context
  is for a finite field, that is, when p is prime and f(X) is irreducible.

void fq_poly_divrem_f(fq_t f, fq_poly_t Q, fq_poly_t R, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
  Either finds a non-trivial factor f of the modulus of ctx, or computes Q, R such that A = BQ + R
  and 0 \leq len(R) < len(B).
  If the leading coefficient of B is invertible, the division with remainder operation is carried out, Q
  and R are computed correctly, and f is set to 1. Otherwise, f is set to a non-trivial factor of the
  modulus and Q and R are not touched.
  Assumes that B is non-zero.

void _fq_poly_rem(fq_struct *R, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB, const fq_t invB, const fq_ctx_t ctx)
  Sets R to the remainder of the division of (A, lenA) by (B, lenB). Assumes that the leading coefficient
  of (B, lenB) is invertible and that invB is its inverse.

void fq_poly_rem(fq_poly_t R, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
  Sets R to the remainder of the division of A by B in the context described by ctx.

void _fq_poly_div(fq_struct *Q, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB, const fq_t invB, const fq_ctx_t ctx)
  Notationally, computes Q, R such that A = BQ + R with 0 \leq len(R) < len(B) but only sets (Q, lenA - lenB + 1). Allows zero-padding in A but not in B. Assumes that the leading coefficient
  of B is a unit.

void fq_poly_div(fq_poly_t Q, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
  Notationally finds polynomials Q and R such that A = BQ + R with len(R) < len(B), but returns
  only Q. If len(B) = 0 an exception is raised.

void _fq_poly_div_newton_n_preinv(fq_struct *Q, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB, const fq_struct *Binv, slong lenBinv, const fq_ctx_t ctx)
  Notationally computes polynomials Q and R such that A = BQ + R with len(R) less than lenB,
  where A is of length lenA and B is of length lenB, but return only Q.
\end{verbatim}
We require that $Q$ have space for $\text{len}A - \text{len}B + 1$ coefficients and assume that the leading coefficient of $B$ is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of $B \mod x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```c
void fq_poly_div_newton_n_preinv(fq_poly_t Q, const fq_poly_t A, const fq_poly_t B, const fq_poly_t Binv, const fq_ctx_t ctx)
```

Notionally computes $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only $Q$.

We assume that the leading coefficient of $B$ is a unit and that $Binv$ is the inverse of the reverse of $B \mod x^{\text{len}(B)}$.

It is required that the length of $A$ is less than or equal to $2^*\text{the length of} \ B - 2$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```c
void _fq_poly_divrem_newton_n_preinv(fq_struct *Q, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB, const fq_struct *Binv, slong lenBinv, const fq_ctx_t ctx)
```

Computes $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, where $A$ is of length $\text{len}A$ and $B$ is of length $\text{len}B$. We require that $Q$ have space for $\text{len}A - \text{len}B + 1$ coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of $B \mod x^{\text{len}(B)}$. The algorithm used is to call `div_newton_n_preinv()` and then multiply out and compute the remainder.

```c
void fq_poly_divrem_newton_n_preinv(fq_poly_t Q, fq_poly_t R, const fq_poly_t A, const fq_poly_t B, const fq_poly_t Binv, const fq_ctx_t ctx)
```

Computes $Q$ and $R$ such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume $Binv$ is the inverse of the reverse of $B \mod x^{\text{len}(B)}$.

It is required that the length of $A$ is less than or equal to $2^*\text{the length of} \ B - 2$.

The algorithm used is to call `div_newton_n()` and then multiply out and compute the remainder.

```c
void _fq_poly_inv_series_newton(fq_struct *Qinv, const fq_struct *Q, slong n, const fq_t cinv, const fq_ctx_t ctx)
```

Given $Q$ of length $n$ whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length $n$ such that $Q \ast Qinv$ is 1 modulo $x^n$. Requires $n > 0$. This function can be viewed as inverting a power series via Newton iteration.

```c
void fq_poly_inv_series_newton(fq_poly_t Qinv, const fq_poly_t Q, slong n, const fq_ctx_t ctx)
```

Given $Q$ find $Qinv$ such that $Q \ast Qinv$ is 1 modulo $x^n$. The constant coefficient of $Q$ must be invertible modulo the modulus of $Q$. An exception is raised if this is not the case or if $n = 0$. This function can be viewed as inverting a power series via Newton iteration.

```c
void _fq_poly_inv_series(fq_struct *Qinv, const fq_struct *Q, slong n, const fq_t cinv, const fq_ctx_t ctx)
```

Given $Q$ of length $n$ whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length $n$ such that $Q \ast Qinv$ is 1 modulo $x^n$. Requires $n > 0$.

```c
void fq_poly_inv_series(fq_poly_t Qinv, const fq_poly_t Q, slong n, const fq_ctx_t ctx)
```

Given $Q$ find $Qinv$ such that $Q \ast Qinv$ is 1 modulo $x^n$. The constant coefficient of $Q$ must be invertible modulo the modulus of $Q$. An exception is raised if this is not the case or if $n = 0$.

```c
void _fq_poly_div_series(fq_struct *Q, const fq_struct *A, slong Alen, const fq_struct *B, slong Blen, slong n, const fq_ctx_t ctx)
```

Set $(Q, n)$ to the quotient of the series $(A, Alen)$ and $(B, Blen)$ assuming $Alen, Blen <= n$. We assume the bottom coefficient of $B$ is invertible.
void \texttt{fq\_poly\_div\_series}(\texttt{fq\_poly\_t} Q, \texttt{const} \texttt{fq\_poly\_t} A, \texttt{const} \texttt{fq\_poly\_t} B, \texttt{slong} n, \texttt{const} \texttt{fq\_ctx\_t} ctx)

Set $Q$ to the quotient of the series $A$ by $B$, thinking of the series as though they were of length $n$.
We assume that the bottom coefficient of $B$ is invertible.

### 11.6.16 Greatest common divisor

void \texttt{fq\_poly\_gcd}(\texttt{fq\_poly\_t} rop, \texttt{const} \texttt{fq\_poly\_t} op1, \texttt{const} \texttt{fq\_poly\_t} op2, \texttt{const} \texttt{fq\_ctx\_t} ctx)

Sets \texttt{rop} to the greatest common divisor of \texttt{op1} and \texttt{op2}, using either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial $P$ is defined to be $P$. Except in the case where the GCD is zero, the GCD $G$ is made monic.

\texttt{slong \_fq\_poly\_gcd} \texttt{(fq\_struct \*G, const} \texttt{fq\_struct \*A, slong} lenA, \texttt{const} \texttt{fq\_struct \*B, slong} lenB, \texttt{const} \texttt{fq\_ctx\_t} ctx)

Computes the GCD of $A$ of length $\texttt{lenA}$ and $B$ of length $\texttt{lenB}$, where $\texttt{lenA} \geq \texttt{lenB} > 0$ and sets $G$ to it. The length of the GCD $G$ is returned by the function. No attempt is made to make the GCD monic. It is required that $G$ have space for $\texttt{lenB}$ coefficients.

\texttt{slong \_fq\_poly\_gcd\_euclidean\_f} \texttt{(\texttt{fq\_t} f, \texttt{fq\_struct \*G, const} \texttt{fq\_struct \*A, slong} lenA, \texttt{const} \texttt{fq\_struct \*B, slong} lenB, \texttt{const} \texttt{fq\_ctx\_t} ctx)}

Either sets $f = 1$ and $G$ to the greatest common divisor of $(A, \texttt{len}(A))$ and $(B, \texttt{len}(B))$ and returns its length, or sets $f$ to a non-trivial factor of the modulus of \texttt{ctx} and leaves the contents of the vector $(G, \texttt{len}(B))$ undefined.

Assumes that $\texttt{len}(A) \geq \texttt{len}(B) > 0$ and that the vector $G$ has space for sufficiently many coefficients.

void \texttt{fq\_poly\_gcd\_euclidean\_f} \texttt{(\texttt{fq\_t} f, \texttt{fq\_poly\_t} G, \texttt{const} \texttt{fq\_poly\_t} A, \texttt{const} \texttt{fq\_poly\_t} B, \texttt{const} \texttt{fq\_ctx\_t} ctx)}

Either sets $f = 1$ and $G$ to the greatest common divisor of $A$ and $B$ or sets $f$ to a factor of the modulus of \texttt{ctx}.

\texttt{slong \_fq\_poly\_xgcd} \texttt{(\texttt{fq\_struct \*G, \texttt{fq\_struct \*S, fq\_struct \*T}, const} \texttt{fq\_struct \*A, slong} lenA, \texttt{const} \texttt{fq\_struct \*B, slong} lenB, \texttt{const} \texttt{fq\_ctx\_t} ctx)}

Computes the GCD of $A$ and $B$ together with cofactors $S$ and $T$ such that $SA + TB = G$. Returns the length of $G$.

Assumes that $\texttt{len}(A) \geq \texttt{len}(B) \geq 1$ and $(\texttt{len}(A), \texttt{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that $G$ have space for $\texttt{len}(B)$ coefficients. Writes $\texttt{len}(B) - 1$ and $\texttt{len}(A) - 1$ coefficients to $S$ and $T$, respectively. Note that, in fact, $\texttt{len}(S) \leq \max(\texttt{len}(B) - \texttt{len}(G), 1)$ and $\texttt{len}(T) \leq \max(\texttt{len}(A) - \texttt{len}(G), 1)$.

No aliasing of input and output operands is permitted.

void \texttt{fq\_poly\_gcd} \texttt{(fq\_poly\_t} G, \texttt{fq\_poly\_t} S, \texttt{fq\_poly\_t} T, \texttt{const} \texttt{fq\_poly\_t} A, \texttt{const} \texttt{fq\_poly\_t} B, \texttt{const} \texttt{fq\_ctx\_t} ctx)

Computes the GCD of $A$ and $B$. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial $P$ is defined to be $P$. Except in the case where the GCD is zero, the GCD $G$ is made monic.

Polynomials $S$ and $T$ are computed such that $S \cdot A + T \cdot B = G$. The length of $S$ will be at most $\texttt{len}(B)$ and the length of $T$ will be at most $\texttt{len}(A)$.

\texttt{slong \_fq\_poly\_xgcd\_euclidean\_f} \texttt{(\texttt{fq\_t} f, \texttt{fq\_struct \*G, \texttt{fq\_struct \*S, fq\_struct \*T, const} \texttt{fq\_struct \*A, slong} lenA, \texttt{const} \texttt{fq\_struct \*B, slong} lenB, \texttt{const} \texttt{fq\_ctx\_t} ctx)}

Either sets $f = 1$ and computes the GCD of $A$ and $B$ together with cofactors $S$ and $T$ such that
\[ SA + TB = G; \text{ otherwise, sets } f \text{ to a non-trivial factor of the modulus of } \text{ctx} \text{ and leaves } G, S, \text{ and } T \text{ undefined. Returns the length of } G. \]

Assumes that \( \text{len}(A) \geq \text{len}(B) \geq 1 \) and \( (\text{len}(A), \text{len}(B)) \neq (1,1) \).

No attempt is made to make the GCD monic.

Requires that \( G \) have space for \( \text{len}(B) \) coefficients. Writes \( \text{len}(B) - 1 \) and \( \text{len}(A) - 1 \) coefficients to \( S \) and \( T \), respectively. Note that, in fact, \( \text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1) \) and \( \text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1) \).

No aliasing of input and output operands is permitted.

```c
void fq_poly_xgcd_euclidean_f(fq_t f, fq_poly_t G, fq_poly_t S, fq_poly_t T, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
```

Either sets \( f = 1 \) and computes the GCD of \( A \) and \( B \) or sets \( f \) to a non-trivial factor of the modulus of \( \text{ctx} \).

If the GCD is computed, polynomials \( S \) and \( T \) are computed such that \( S*A + T*B = G \); otherwise, they are undefined. The length of \( S \) will be at most \( \text{len}(B) \) and the length of \( T \) will be at most \( \text{len}(A) \).

The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \( P \) is defined to be \( P \). Except in the case where the GCD is zero, the GCD \( G \) is made monic.

### 11.6.17 Divisibility testing

```c
int _fq_poly_divides(fq_struct *Q, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB, const fq_t invB, const fq_ctx_t ctx)
```

Returns 1 if \( (B, \text{len}(B)) \) divides \( (A, \text{len}(A)) \) exactly and sets \( Q \) to the quotient, otherwise returns 0.

It is assumed that \( \text{len}(A) \geq \text{len}(B) > 0 \) and that \( Q \) has space for \( \text{len}(A) - \text{len}(B) + 1 \) coefficients.

Aliasing of \( Q \) with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```c
int fq_poly_divides(fq_poly_t Q, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
```

Returns 1 if \( B \) divides \( A \) exactly and sets \( Q \) to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

### 11.6.18 Derivative

```c
void _fq_poly_derivative(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
```

Sets \( (\text{rop}, \text{len} - 1) \) to the derivative of \( (\text{op}, \text{len}) \). Also handles the cases where \( \text{len} \) is 0 or 1 correctly. Supports aliasing of \( \text{rop} \) and \( \text{op} \).

```c
void fq_poly_derivative(fq_poly_t rop, const fq_poly_t op, const fq_ctx_t ctx)
```

Sets \( \text{rop} \) to the derivative of \( \text{op} \).
11.6.19 Square root

void _fq_poly_invsqrt_series(fq_struct *g, const fq_struct *h, slong n, fq_ctx_t mod)
  Set the first $n$ terms of $g$ to the series expansion of $1/\sqrt{h}$. It is assumed that $n > 0$, that $h$ has constant term 1 and that $h$ is zero-padded as necessary to length $n$. Aliasing is not permitted.

void fq_poly_invsqrt_series(fq_poly_t g, const fq_poly_t h, slong n, fq_ctx_t ctx)
  Set $g$ to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that $h$ has constant term 1.

void _fq_poly_sqrt_series(fq_struct *g, const fq_struct *h, slong n, fq_ctx_t ctx)
  Set the first $n$ terms of $g$ to the series expansion of $\sqrt{h}$. It is assumed that $n > 0$, that $h$ has constant term 1 and that $h$ is zero-padded as necessary to length $n$. Aliasing is not permitted.

void fq_poly_sqrt_series(fq_poly_t g, const fq_poly_t h, slong n, fq_ctx_t ctx)
  Set $g$ to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that $h$ has constant term 1.

int _fq_poly_sqrt(fq_struct *s, const fq_struct *p, slong n, fq_ctx_t mod)
  If $(p, n)$ is a perfect square, sets $(s, n/2 + 1)$ to a square root of $p$ and returns 1. Otherwise returns 0.

int fq_poly_sqrt(fq_poly_t s, const fq_poly_t p, fq_ctx_t mod)
  If $p$ is a perfect square, sets $s$ to a square root of $p$ and returns 1. Otherwise returns 0.

11.6.20 Evaluation

void _fq_poly_evaluate_fq(fq_t rop, const fq_struct *op, slong len, const fq_t a, const fq_ctx_t ctx)
  Sets $rop$ to $(op, len)$ evaluated at $a$.
  Supports zero padding. There are no restrictions on $len$, that is, $len$ is allowed to be zero, too.

void fq_poly_evaluate_fq(fq_t rop, const fq_poly_t f, const fq_t a, const fq_ctx_t ctx)
  Sets $rop$ to the value of $f(a)$.
  As the coefficient ring $\mathbb{F}_q$ is finite, Horner’s method is sufficient.

11.6.21 Composition

void _fq_poly_compose(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2, slong len2, const fq_ctx_t ctx)
  Sets $rop$ to the composition of $(op1, len1)$ and $(op2, len2)$.
  Assumes that $rop$ has space for $(len1-1)*(len2-1) + 1$ coefficients. Assumes that $op1$ and $op2$ are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

void fq_poly_compose(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)
  Sets $rop$ to the composition of $op1$ and $op2$. To be precise about the order of composition, denoting $rop$, $op1$, and $op2$ by $f$, $g$, and $h$, respectively, sets $f(t) = g(h(t))$.

void _fq_poly_compose_mod_horner(fq_struct *res, const fq_struct *f, slong lenf, const fq_struct *g, const fq_struct *h, slong lenh, const fq_ctx_t ctx)
  Sets $res$ to the composition $f(g)$ modulo $h$. We require that $h$ is nonzero and that the length of $g$ is one less than the length of $h$ (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.
  The algorithm used is Horner’s rule.
The algorithm used is Horner's rule.

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero. The algorithm used is Horner's rule.

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). Furthermore, we require \( \texttt{hinv} \) to be the inverse of the reverse of \( h \). The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). The algorithm used is the Brent-Kung matrix algorithm.

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). The algorithm used is the Brent-Kung matrix algorithm.

Sets \( \texttt{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). Furthermore, we require \( \texttt{hinv} \) to be the inverse of the reverse of \( h \). The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.
Sets res to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). We also require that the length of \( f \) is less than the length of \( h \). Furthermore, we require \texttt{hinv} to be the inverse of the reverse of \( h \). The output is not allowed to be aliased with any of the inputs.

Sets res to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). Furthermore, we require \texttt{hinv} to be the inverse of the reverse of \( h \).

Sets the ith row of \( A \) to the composition \( f(g) \) modulo \( h \). We require \( h \) to be of the reverse of \( g \) and \( g \) to be nonzero.

Sets the ith row of \( A \) to the composition \( f(g) \) modulo \( h \). We require \( h \) to be of the reverse of \( g \) and \( g \) to be nonzero.

Sets the ith row of \( A \) to the composition \( f(g) \) modulo \( h \). We require \( h \) to be of the reverse of \( g \). The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.
11.6.22 Output

int _fq_poly_fprint_pretty(FILE *file, const fq_struct *poly, slong len, const char *x, const fq_ctx_t ctx)
Prints the pretty representation of (poly, len) to the stream file, using the string x to represent the indeterminate.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_poly_fprint_pretty(FILE *file, const fq_poly_t poly, const char *x, const fq_ctx_t ctx)
Prints the pretty representation of poly to the stream file, using the string x to represent the indeterminate.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fq_poly_print_pretty(const fq_struct *poly, slong len, const char *x, const fq_ctx_t ctx)
Prints the pretty representation of (poly, len) to stdout, using the string x to represent the indeterminate.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_poly_print_pretty(const fq_poly_t poly, const char *x, const fq_ctx_t ctx)
Prints the pretty representation of poly to stdout, using the string x to represent the indeterminate.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fq_poly_fprint(FILE *file, const fq_struct *poly, slong len, const fq_ctx_t ctx)
Prints the pretty representation of (poly, len) to the stream file.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_poly_fprint(FILE *file, const fq_poly_t poly, const fq_ctx_t ctx)
Prints the pretty representation of poly to the stream file.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fq_poly_print(FILE *file, const fq_struct *poly, slong len, const fq_ctx_t ctx)
Prints the representation of poly to stdout.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_poly_print(const fq_poly_t poly, const fq_ctx_t ctx)
Prints the representation of poly to stdout.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

char *_fq_poly_get_str(const fq_struct *poly, slong len, const fq_ctx_t ctx)
Returns the plain FLINT string representation of the polynomial (poly, len).

char *fq_poly_get_str(const fq_poly_t poly, const fq_ctx_t ctx)
Returns the plain FLINT string representation of the polynomial poly.

char *_fq_poly_get_str_pretty(const fq_struct *poly, slong len, const char *x, const fq_ctx_t ctx)
Returns a pretty representation of the polynomial (poly, len) using the null-terminated string x as the variable name.

char *fq_poly_get_str_pretty(const fq_poly_t poly, const char *x, const fq_ctx_t ctx)
Returns a pretty representation of the polynomial poly using the null-terminated string x as the variable name.
11.6.23 Inflation and deflation

```c
void fq_poly_inflate(fq_poly_t result, const fq_poly_t input, ulong inflation, const fq_ctx_t ctx)
Sets result to the inflated polynomial \( p(x^n) \) where \( p \) is given by input and \( n \) is given by inflation.
```

```c
void fq_poly_deflate(fq_poly_t result, const fq_poly_t input, ulong deflation, const fq_ctx_t ctx)
Sets result to the deflated polynomial \( p(x^{1/n}) \) where \( p \) is given by input and \( n \) is given by deflation. Requires \( n > 0 \).
```

```c
ulong fq_poly_deflation(const fq_poly_t input, const fq_ctx_t ctx)
Returns the largest integer by which input can be deflated. As special cases, returns 0 if input is the zero polynomial and 1 of input is a constant polynomial.
```

11.7 \_fq\_default\_poly.h – univariate polynomials over finite fields

11.7.1 Types, macros and constants

type fq\_default\_poly\_t

11.7.2 Memory management

```c
void fq_default_poly_init(fq_default_poly_t poly, const fq_default_ctx_t ctx)
Initialises poly for use, with context ctx, and setting its length to zero. A corresponding call to
fq_default_poly_clear() must be made after finishing with the fq_default_poly_t to free the
memory used by the polynomial.
```

```c
void fq_default_poly_init2(fq_default_poly_t poly, slong alloc, const fq_default_ctx_t ctx)
Initialises poly with space for at least alloc coefficients and sets the length to zero. The allocated
coefficients are all set to zero. A corresponding call to fq_default_poly_clear() must be made
after finishing with the fq_default_poly_t to free the memory used by the polynomial.
```

```c
void fq_default_poly_realloc(fq_default_poly_t poly, slong alloc, const fq_default_ctx_t ctx)
Reallocates the given polynomial to have space for alloc coefficients. If alloc is zero the polynomial
is cleared and then reinitialised. If the current length is greater than alloc the polynomial is
first truncated to length alloc.
```

```c
void fq_default_poly_fit_length(fq_default_poly_t poly, slong len, const fq_default_ctx_t ctx)
If len is greater than the number of coefficients currently allocated, then the polynomial is reallocated
for at least len coefficients. No data is lost when calling this function.

The function efficiently deals with the case where fit_length is called many times in small incre-
ments by at least doubling the number of allocated coefficients when length is larger than the
number of coefficients currently allocated.
```

```c
void fq_default_poly_clear(fq_default_poly_t poly, const fq_default_ctx_t ctx)
Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be
used again.
```

```c
void _fq_default_poly_set_length(fq_default_poly_t poly, slong len, const fq_default_ctx_t ctx)
Set the length of poly to len.
```

```c
void fq_default_poly_truncate(fq_default_poly_t poly, slong newlen, const fq_default_ctx_t ctx)
Truncates the polynomial to length at most newlen.
```

```c
void fq_default_poly_set_trunc(fq_default_poly_t poly1, fq_default_poly_t poly2, slong newlen, const fq_default_ctx_t ctx)
Sets poly1 to poly2 truncated to length newlen.
```
void fq_default_poly_reverse(fq_default_poly_t output, const fq_default_poly_t input, slong m, const fq_default_ctx_t ctx)

Sets output to the reverse of input, thinking of it as a polynomial of length m, notionally zero-padded if necessary. The length m must be non-negative, but there are no other restrictions. The output polynomial will be set to length m and then normalised.

11.7.3 Polynomial parameters

slong fq_default_poly_degree(const fq_default_poly_t poly, const fq_default_ctx_t ctx)
Returns the degree of the polynomial poly.

slong fq_default_poly_length(const fq_default_poly_t poly, const fq_default_ctx_t ctx)
Returns the length of the polynomial poly.

11.7.4 Randomisation

void fq_default_poly_randtest(fq_default_poly_t f, flint_rand_t state, slong len, const fq_default_ctx_t ctx)
Sets f to a random polynomial of length at most len with entries in the field described by ctx.

void fq_default_poly_randtest_not_zero(fq_default_poly_t f, flint_rand_t state, slong len, const fq_default_ctx_t ctx)
Same as fq_default_poly_randtest but guarantees that the polynomial is not zero.

void fq_default_poly_randtest_monic(fq_default_poly_t f, flint_rand_t state, slong len, const fq_default_ctx_t ctx)
Sets f to a random monic polynomial of length len with entries in the field described by ctx.

void fq_default_poly_randtest_irreducible(fq_default_poly_t f, flint_rand_t state, slong len, const fq_default_ctx_t ctx)
Sets f to a random monic, irreducible polynomial of length len with entries in the field described by ctx.

11.7.5 Assignment and basic manipulation

void fq_default_poly_set(fq_default_poly_t poly1, const fq_default_poly_t poly2, const fq_default_ctx_t ctx)
Sets the polynomial poly1 to the polynomial poly2.

void fq_default_poly_set_fq_default(fq_default_poly_t poly, const fq_default_t c, const fq_default_ctx_t ctx)
Sets the polynomial poly to c.

void fq_default_poly_swap(fq_default_poly_t op1, fq_default_poly_t op2, const fq_default_ctx_t ctx)
Swaps the two polynomials op1 and op2.

void fq_default_poly_zero(fq_default_poly_t poly, const fq_default_ctx_t ctx)
Sets poly to the zero polynomial.

void fq_default_poly_one(fq_default_poly_t poly, const fq_default_ctx_t ctx)
Sets poly to the constant polynomial 1.

void fq_default_poly_gen(fq_default_poly_t poly, const fq_default_ctx_t ctx)
Sets poly to the polynomial x.
void fq_default_poly_make_monic(fq_default_poly_t rop, const fq_default_poly_t op, const fq_default_ctx_t ctx)
    Sets rop to op, normed to have leading coefficient 1.

void fq_default_poly_set_nmod_poly(fq_default_poly_t rop, const nmod_poly_t op, const fq_default_ctx_t ctx)
    Sets the polynomial rop to the polynomial op.

void fq_default_poly_set_fmpz_mod_poly(fq_default_poly_t rop, const fmpz_mod_poly_t op, const fq_default_ctx_t ctx)
    Sets the polynomial rop to the polynomial op.

void fq_default_poly_set_fmpz_poly(fq_default_poly_t rop, const fmpz_poly_t op, const fq_default_ctx_t ctx)
    Sets the polynomial rop to the polynomial op.

11.7.6 Getting and setting coefficients

void fq_default_poly_get_coeff(fq_t x, const fq_default_poly_t poly, slong n, const fq_default_ctx_t ctx)
    Sets x to the coefficient of X^n in poly.

void fq_default_poly_set_coeff(fq_default_poly_t poly, slong n, const fq_default_t x, const fq_default_ctx_t ctx)
    Sets the coefficient of X^n in poly to x.

void fq_default_poly_set_coeff_fmpz(fq_default_poly_t poly, slong n, const fmpz_t x, const fq_default_ctx_t ctx)
    Sets the coefficient of X^n in the polynomial to x, assuming n ≥ 0.

11.7.7 Comparison

int fq_default_poly_equal(const fq_default_poly_t poly1, const fq_default_poly_t poly2, const fq_default_ctx_t ctx)
    Returns nonzero if the two polynomials poly1 and poly2 are equal, otherwise returns zero.

int fq_default_poly_equal_trunc(const fq_default_poly_t poly1, const fq_default_poly_t poly2, slong n, const fq_default_ctx_t ctx)
    Notionally truncate poly1 and poly2 to length n and return nonzero if they are equal, otherwise return zero.

int fq_default_poly_is_zero(const fq_default_poly_t poly, const fq_default_ctx_t ctx)
    Returns whether the polynomial poly is the zero polynomial.

int fq_default_poly_is_one(const fq_default_poly_t op, const fq_default_ctx_t ctx)
    Returns whether the polynomial poly is equal to the constant polynomial 1.

int fq_default_poly_is_gen(const fq_default_poly_t op, const fq_default_ctx_t ctx)
    Returns whether the polynomial poly is equal to the polynomial x.

int fq_default_poly_is_unit(const fq_default_poly_t op, const fq_default_ctx_t ctx)
    Returns whether the polynomial poly is a unit in the polynomial ring \( \mathbb{F}_q[X] \), i.e. if it has degree 0 and is non-zero.

int fq_default_poly_equal_fq_default(const fq_default_poly_t poly, const fq_default_t c, const fq_default_ctx_t ctx)
    Returns whether the polynomial poly is equal to the (constant) \( \mathbb{F}_q \) element c.
11.7.8 Addition and subtraction

void fq_default_poly_add(fq_default_poly_t res, const fq_default_poly_t poly1, const fq_default_poly_t poly2, const fq_default_ctx_t ctx)

Sets res to the sum of poly1 and poly2.

void fq_default_poly_add_si(fq_default_poly_t res, const fq_default_poly_t poly1, slong c, const fq_default_ctx_t ctx)

Sets res to the sum of poly1 and c.

void fq_default_poly_add_series(fq_default_poly_t res, const fq_default_poly_t poly1, const fq_default_poly_t poly2, slong n, const fq_default_ctx_t ctx)

Notionally truncate poly1 and poly2 to length n and set res to the sum.

void fq_default_poly_sub(fq_default_poly_t res, const fq_default_poly_t poly1, const fq_default_poly_t poly2, const fq_default_ctx_t ctx)

Sets res to the difference of poly1 and poly2.

void fq_default_poly_sub_series(fq_default_poly_t res, const fq_default_poly_t poly1, const fq_default_poly_t poly2, slong n, const fq_default_ctx_t ctx)

Notionally truncate poly1 and poly2 to length n and set res to the difference.

void fq_default_poly_neg(fq_default_poly_t res, const fq_default_poly_t poly, const fq_default_ctx_t ctx)

Sets res to the additive inverse of poly.

11.7.9 Scalar multiplication and division

void fq_default_poly_scalar_mul_fq_default(fq_default_poly_t rop, const fq_default_poly_t op, const fq_default_t x, const fq_default_ctx_t ctx)

Sets rop to the product of op by the scalar x, in the context defined by ctx.

void fq_default_poly_scalar_addmul_fq_default(fq_default_poly_t rop, const fq_default_poly_t op, const fq_default_t x, const fq_default_ctx_t ctx)

Adds to rop the product of op by the scalar x, in the context defined by ctx.

void fq_default_poly_scalar_submul_fq_default(fq_default_poly_t rop, const fq_default_poly_t op, const fq_default_t x, const fq_default_ctx_t ctx)

Subtracts from rop the product of op by the scalar x, in the context defined by ctx.

void fq_default_poly_scalar_div_fq_default(fq_default_poly_t rop, const fq_default_poly_t op, const fq_default_t x, const fq_default_ctx_t ctx)

Sets rop to the quotient of op by the scalar x, in the context defined by ctx. An exception is raised if x is zero.

11.7.10 Multiplication

void fq_default_poly_mul(fq_default_poly_t rop, const fq_default_poly_t op1, const fq_default_poly_t op2, const fq_default_ctx_t ctx)

Sets rop to the product of op1 and op2, choosing an appropriate algorithm.

void fq_default_poly_mullow(fq_default_poly_t rop, const fq_default_poly_t op1, const fq_default_poly_t op2, slong n, const fq_default_ctx_t ctx)

Sets rop to the lowest n coefficients of the product of op1 and op2.
void fq_default_poly_mulhigh(fq_default_poly_t res, const fq_default_poly_t poly1, const fq_default_poly_t poly2, slong start, const fq_default_ctx_t ctx)

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced.

void fq_default_poly_mulmod(fq_default_poly_t res, const fq_default_poly_t poly1, const fq_default_poly_t poly2, const fq_default_poly_t f, const fq_default_ctx_t ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f.

11.7.11 Squaring

void fq_default_poly_sqr(fq_default_poly_t rop, const fq_default_poly_t op, const fq_default_ctx_t ctx)

Sets rop to the square of op, choosing an appropriate algorithm.

11.7.12 Powering

void fq_default_poly_pow(fq_default_poly_t rop, const fq_default_poly_t op, ulong e, const fq_default_ctx_t ctx)

Computes rop = op^e. If e is zero, returns one, so that in particular 0^0 = 1.

void fq_default_poly_powmod_ui_binexp(fq_default_poly_t res, const fq_default_poly_t poly, ulong e, const fq_default_poly_t f, const fq_default_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0.

void fq_default_poly_powmod_fmpz_binexp(fq_default_poly_t res, const fq_default_poly_t poly, const fmpz_t e, const fq_default_poly_t f, const fq_default_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0.

void fq_default_poly_pow_trunc(fq_default_poly_t res, const fq_default_poly_t poly, ulong e, slong trunc, const fq_default_ctx_t ctx)

Sets res to the low trunc coefficients of poly to the power e. This is equivalent to doing a powering followed by a truncation.

11.7.13 Shifting

void fq_default_poly_shift_left(fq_default_poly_t rop, const fq_default_poly_t op, slong n, const fq_default_ctx_t ctx)

Sets rop to op shifted left by n coeffs. Zero coefficients are inserted.

void fq_default_poly_shift_right(fq_default_poly_t rop, const fq_default_poly_t op, slong n, const fq_default_ctx_t ctx)

Sets rop to op shifted right by n coefficients. If n is equal to or greater than the current length of op, rop is set to the zero polynomial.
11.7.14 Norms

```c
slong fq_default_poly_hamming_weight(const fq_default_poly_t op, const fq_default_ctx_t ctx)
```

Returns the number of non-zero entries in the polynomial op.

11.7.15 Euclidean division

```c
void fq_default_poly_divrem(fq_default_poly_t Q, fq_default_poly_t R, const fq_default_poly_t A, const fq_default_poly_t B, const fq_default_ctx_t ctx)
```

Computes \( Q, R \) such that \( A = BQ + R \) with \( 0 \leq \text{len}(R) < \text{len}(B) \).

Assumes that the leading coefficient of \( B \) is invertible. This can be taken for granted the context is for a finite field, that is, when \( p \) is prime and \( f(X) \) is irreducible.

```c
void fq_default_poly_rem(fq_default_poly_t R, const fq_default_poly_t A, const fq_default_poly_t B, const fq_default_ctx_t ctx)
```

Sets \( R \) to the remainder of the division of \( A \) by \( B \) in the context described by \( ctx \).

```c
void fq_default_poly_inv_series(fq_default_poly_t Qinv, const fq_default_poly_t Q, slong n, const fq_default_ctx_t ctx)
```

Given \( Q \) find \( Qinv \) such that \( Q \cdot Qinv \equiv 1 \mod x^n \). The constant coefficient of \( Q \) must be invertible modulo the modulus of \( Q \). An exception is raised if this is not the case or if \( n = 0 \).

```c
void fq_default_poly_div_series(fq_default_poly_t Q, const fq_default_poly_t A, const fq_default_poly_t B, slong n, const fq_default_ctx_t ctx)
```

Set \( Q \) to the quotient of the series \( A \) by \( B \), thinking of the series as though they were of length \( n \). We assume that the bottom coefficient of \( B \) is invertible.

11.7.16 Greatest common divisor

```c
void fq_default_poly_gcd(fq_default_poly_t rop, const fq_default_poly_t op1, const fq_default_poly_t op2, const fq_default_ctx_t ctx)
```

Sets \( rop \) to the greatest common divisor of \( op1 \) and \( op2 \), using the either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \( P \) is defined to be \( P \). Except in the case where the GCD is zero, the GCD \( G \) is made monic.

```c
void fq_default_poly_xgcd(fq_default_poly_t G, fq_default_poly_t S, fq_default_poly_t T, const fq_default_poly_t A, const fq_default_poly_t B, const fq_default_ctx_t ctx)
```

Computes the GCD of \( A \) and \( B \). The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \( P \) is defined to be \( P \). Except in the case where the GCD is zero, the GCD \( G \) is made monic.

Polynomials \( S \) and \( T \) are computed such that \( S \cdot A + T \cdot B = G \). The length of \( S \) will be at most \( \text{len}B \) and the length of \( T \) will be at most \( \text{len}A \).
11.7.17 Divisibility testing

```c
int fq_default_poly_divides(fq_default_poly_t Q, const fq_default_poly_t A, const fq_default_poly_t B, const fq_default_ctx_t ctx)
```

Returns 1 if \( B \) divides \( A \) exactly and sets \( Q \) to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

11.7.18 Derivative

```c
void fq_default_poly_derivative(fq_default_poly_t rop, const fq_default_poly_t op, const fq_default_ctx_t ctx)
```

Sets \( rop \) to the derivative of \( op \).

11.7.19 Square root

```c
void fq_default_poly_invsqrt_series(fq_default_poly_t g, const fq_default_poly_t h, slong n, fq_default_ctx_t ctx)
```

Set \( g \) to the series expansion of \( 1/\sqrt{h} \) to order \( O(x^n) \). It is assumed that \( h \) has constant term 1.

```c
void fq_default_poly_sqrt_series(fq_default_poly_t g, const fq_default_poly_t h, slong n, fq_default_ctx_t ctx)
```

Set \( g \) to the series expansion of \( \sqrt{h} \) to order \( O(x^n) \). It is assumed that \( h \) has constant term 1.

```c
int fq_default_poly_sqrt(fq_default_poly_t s, const fq_default_poly_t p, fq_default_ctx_t mod)
```

If \( p \) is a perfect square, sets \( s \) to a square root of \( p \) and returns 1. Otherwise returns 0.

11.7.20 Evaluation

```c
void fq_default_poly_evaluate_fq_default(fq_t rop, const fq_default_poly_t f, const fq_t a, const fq_default_ctx_t ctx)
```

Sets \( rop \) to the value of \( f(a) \).

As the coefficient ring \( \mathbb{F}_q \) is finite, Horner’s method is sufficient.

11.7.21 Composition

```c
void fq_default_poly_compose(fq_default_poly_t rop, const fq_default_poly_t op1, const fq_default_poly_t op2, const fq_default_ctx_t ctx)
```

Sets \( rop \) to the composition of \( op1 \) and \( op2 \). To be precise about the order of composition, denoting \( rop \), \( op1 \), and \( op2 \) by \( f \), \( g \), and \( h \), respectively, sets \( f(t) = g(h(t)) \).

```c
void fq_default_poly_compose_mod(fq_default_poly_t res, const fq_default_poly_t f, const fq_default_poly_t g, const fq_default_poly_t h, const fq_default_ctx_t ctx)
```

Sets \( res \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero.
11.7.22 Output

int fq_default_poly_fprint_pretty(FILE *file, const fq_default_poly_t poly, const char *x, const fq_default_ctx_t ctx)

Prints the pretty representation of poly to the stream file, using the string x to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_default_poly_print_pretty(const fq_default_poly_t poly, const char *x, const fq_default_ctx_t ctx)

Prints the pretty representation of poly to stdout, using the string x to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_default_poly_fprint(FILE *file, const fq_default_poly_t poly, const fq_default_ctx_t ctx)

Prints the pretty representation of poly to the stream file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_default_poly_print(const fq_default_poly_t poly, const fq_default_ctx_t ctx)

Prints the representation of poly to stdout.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

char *fq_default_poly_get_str(const fq_default_poly_t poly, const fq_default_ctx_t ctx)

Returns the plain FLINT string representation of the polynomial.

char *fq_default_poly_get_str_pretty(const fq_default_poly_t poly, const char *x, const fq_default_ctx_t ctx)

Returns a pretty representation of the polynomial poly using the null-terminated string x as the variable name.

11.7.23 Inflation and deflation

void fq_default_poly_inflate(fq_default_poly_t result, const fq_default_poly_t input, ulong inflation, const fq_default_ctx_t ctx)

Sets result to the inflated polynomial $p(x^n)$ where $p$ is given by input and $n$ is given by inflation.

void fq_default_poly_deflate(fq_default_poly_t result, const fq_default_poly_t input, ulong deflation, const fq_default_ctx_t ctx)

Sets result to the deflated polynomial $p(x^{1/n})$ where $p$ is given by input and $n$ is given by deflation. Requires $n > 0$.

ulong fq_default_poly_deflation(const fq_default_poly_t input, const fq_default_ctx_t ctx)

Returns the largest integer by which input can be deflated. As special cases, returns 0 if input is the zero polynomial and 1 if input is a constant polynomial.
11.8 fq_poly_factor.h – factorisation of univariate polynomials over finite fields

11.8.1 Types, macros and constants

type fq_poly_factor_struct

type fq_poly_factor_t

11.8.2 Memory Management

void fq_poly_factor_init(fq_poly_factor_t fac, const fq_ctx_t ctx)
   
   Initialises fac for use. An fq_poly_factor_t represents a polynomial in factorised form as a product of polynomials with associated exponents.

void fq_poly_factor_clear(fq_poly_factor_t fac, const fq_ctx_t ctx)

   Frees all memory associated with fac.

void fq_poly_factor_realloc(fq_poly_factor_t fac, slong alloc, const fq_ctx_t ctx)

   Reallocates the factor structure to provide space for precisely alloc factors.

void fq_poly_factor_fit_length(fq_poly_factor_t fac, slong len, const fq_ctx_t ctx)

   Ensures that the factor structure has space for at least len factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

11.8.3 Basic Operations

void fq_poly_factor_set(fq_poly_factor_t res, const fq_poly_factor_t fac, const fq_ctx_t ctx)

   Sets res to the same factorisation as fac.

void fq_poly_factor_print_pretty(const fq_poly_factor_t fac, const char *var, const fq_ctx_t ctx)

   Pretty-prints the entries of fac to standard output.

void fq_poly_factor_print(const fq_poly_factor_t fac, const fq_ctx_t ctx)

   Prints the entries of fac to standard output.

void fq_poly_factor_insert(fq_poly_factor_t fac, const fq_poly_t poly, slong exp, const fq_ctx_t ctx)

   Inserts the factor poly with multiplicity exp into the factorisation fac.

   If fac already contains poly, then exp simply gets added to the exponent of the existing entry.

void fq_poly_factor_concat(fq_poly_factor_t res, const fq_poly_factor_t fac, const fq_ctx_t ctx)

   Concatenates two factorisations.

   This is equivalent to calling fq_poly_factor_insert() repeatedly with the individual factors of fac.

   Does not support aliasing between res and fac.

void fq_poly_factor_pow(fq_poly_factor_t fac, slong exp, const fq_ctx_t ctx)

   Raises fac to the power exp.

ulong fq_poly_remove(fq_poly_t f, const fq_poly_t p, const fq_ctx_t ctx)

   Removes the highest possible power of p from f and returns the exponent.
11.8.4 Irreducibility Testing

int fq_poly_is_irreducible(const fq_poly_t f, const fq_ctx_t ctx)
Returns 1 if the polynomial f is irreducible, otherwise returns 0.

int fq_poly_is_irreducible_ddf(const fq_poly_t f, const fq_ctx_t ctx)
Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

int fq_poly_is_irreducible_ben_or(const fq_poly_t f, const fq_ctx_t ctx)
Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses Ben-Or's irreducibility test.

int _fq_poly_is_squarefree(const fq_struct *f, slong len, const fq_ctx_t ctx)
Returns 1 if (f, len) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

int fq_poly_is_squarefree(const fq_poly_t f, const fq_ctx_t ctx)
Returns 1 if f is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

11.8.5 Factorisation

int fq_poly_factor_equal_deg_prob(fq_poly_t factor, flint_rand_t state, const fq_poly_t pol, slong d, const fq_ctx_t ctx)
Probabilistic equal degree factorisation of pol into irreducible factors of degree d. If it passes, a factor is placed in factor and 1 is returned, otherwise 0 is returned and the value of factor is undetermined.

Requires that pol be monic, non-constant and squarefree.

void fq_poly_factor_equal_deg(fq_poly_factor_t factors, const fq_poly_t pol, slong d, const fq_ctx_t ctx)
Assuming pol is a product of irreducible factors all of degree d, finds all those factors and places them in factors. Requires that pol be monic, non-constant and squarefree.

void fq_poly_factor_split_single(fq_poly_t linfactor, const fq_poly_t input, const fq_ctx_t ctx)
Assuming input is a product of factors all of degree 1, finds a single linear factor of input and places it in linfactor. Requires that input be monic and non-constant.

void fq_poly_factor_distinct_deg(fq_poly_factor_t res, const fq_poly_t poly, slong *const *degs, const fq_ctx_t ctx)
Factorises a monic non-constant squarefree polynomial poly of degree n into factors f[d] such that for 1 ≤ d ≤ n f[d] is the product of the monic irreducible factors of poly of degree d. Factors are stored in res, associated powers of irreducible polynomials are stored in degs in the same order as factors.

Requires that degs have enough space for irreducible polynomials' powers (maximum space required is n * sizeof(slong)).

void fq_poly_factor_squarefree(fq_poly_factor_t res, const fq_poly_t f, const fq_ctx_t ctx)
Sets res to a squarefree factorization of f.

void fq_poly_factor(fq_poly_factor_t res, fq_t lead, const fq_poly_t f, const fq_ctx_t ctx)
Factorises a non-constant polynomial f into monic irreducible factors choosing the best algorithm for given modulo and degree. The output lead is set to the leading coefficient of f upon return. Choice of algorithm is based on heuristic measurements.
void \texttt{fq\_poly\_factor\_cantor\_zassenhaus}(\texttt{fq\_poly\_factor\_t} \texttt{res}, \texttt{const fq\_poly\_t} \texttt{f}, \texttt{const fq\_ctx\_t} \texttt{ctx})

Factorises a non-constant polynomial \texttt{f} into monic irreducible factors using the Cantor-Zassenhaus algorithm.

void \texttt{fq\_poly\_factor\_kaltofen\_shoup}(\texttt{fq\_poly\_factor\_t} \texttt{res}, \texttt{const fq\_poly\_t} \texttt{poly}, \texttt{const fq\_ctx\_t} \texttt{ctx})

Factorises a non-constant polynomial \texttt{f} into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a “baby step/giant step” strategy for the distinct-degree factorization step.

void \texttt{fq\_poly\_factor\_berlekamp}(\texttt{fq\_poly\_factor\_t} \texttt{factors}, \texttt{const fq\_poly\_t} \texttt{f}, \texttt{const fq\_ctx\_t} \texttt{ctx})

Factorises a non-constant polynomial \texttt{f} into monic irreducible factors using the Berlekamp algorithm.

void \texttt{fq\_poly\_factor\_with\_berlekamp}(\texttt{fq\_poly\_factor\_t} \texttt{res}, \texttt{fq\_t} \texttt{leading\_coeff}, \texttt{const fq\_poly\_t} \texttt{f}, \texttt{const fq\_ctx\_t} \texttt{ctx})

Factorises a general polynomial \texttt{f} into monic irreducible factors and sets \texttt{leading\_coeff} to the leading coefficient of \texttt{f}, or 0 if \texttt{f} is the zero polynomial.

This function first checks for small special cases, deflates \texttt{f} if it is of the form \(p(x^m)\) for some \(m > 1\), then performs a square-free factorisation, and finally runs Berlekamp factorisation on all the individual square-free factors.

void \texttt{fq\_poly\_factor\_with\_cantor\_zassenhaus}(\texttt{fq\_poly\_factor\_t} \texttt{res}, \texttt{fq\_t} \texttt{leading\_coeff}, \texttt{const fq\_poly\_t} \texttt{f}, \texttt{const fq\_ctx\_t} \texttt{ctx})

Factorises a general polynomial \texttt{f} into monic irreducible factors and sets \texttt{leading\_coeff} to the leading coefficient of \texttt{f}, or 0 if \texttt{f} is the zero polynomial.

This function first checks for small special cases, deflates \texttt{f} if it is of the form \(p(x^m)\) for some \(m > 1\), then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

void \texttt{fq\_poly\_factor\_with\_kaltofen\_shoup}(\texttt{fq\_poly\_factor\_t} \texttt{res}, \texttt{fq\_t} \texttt{leading\_coeff}, \texttt{const fq\_poly\_t} \texttt{f}, \texttt{const fq\_ctx\_t} \texttt{ctx})

Factorises a general polynomial \texttt{f} into monic irreducible factors and sets \texttt{leading\_coeff} to the leading coefficient of \texttt{f}, or 0 if \texttt{f} is the zero polynomial.

This function first checks for small special cases, deflates \texttt{f} if it is of the form \(p(x^m)\) for some \(m > 1\), then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

void \texttt{fq\_poly\_iterated\_frobenius\_preinv}(\texttt{fq\_poly\_t} *\texttt{rop}, \texttt{slong} \texttt{n}, \texttt{const fq\_poly\_t} \texttt{v}, \texttt{const fq\_poly\_t} \texttt{vinv}, \texttt{const fq\_ctx\_t} \texttt{ctx})

Sets \texttt{rop[i]} to be \(x^{q^i} \mod v\) for \(0 \leq i < n\).

It is required that \texttt{vinv} is the inverse of the reverse of \texttt{v} mod \texttt{x}^{\texttt{lenv}}.

### 11.8.6 Root Finding

void \texttt{fq\_poly\_roots}(\texttt{fq\_poly\_factor\_t} \texttt{r}, \texttt{const fq\_poly\_t} \texttt{f}, \texttt{int} with\_multiplicity, \texttt{const fq\_ctx\_t} \texttt{ctx})

Fill \texttt{r} with factors of the form \(x - r_i\) where the \(r_i\) are the distinct roots of a nonzero \(f\) in \(F_q\). If \texttt{with\_multiplicity} is zero, the exponent \(e_i\) of the factor \(x - r_i\) is 1. Otherwise, it is the largest \(e_i\) such that \((x - r_i)^{e_i}\) divides \(f\). This function throws if \(f\) is zero, but is otherwise always successful.

11.8. \texttt{fq\_poly\_factor\_h} – factorisation of univariate polynomials over finite fields
11.9 fq_default_poly_factor.h – factorisation of univariate polynomials over finite fields

11.9.1 Types, macros and constants

typedef fq_default_poly_factor_t

11.9.2 Memory Management

void fq_default_poly_factor_init(fq_default_poly_factor_t fac, const fq_default_ctx_t ctx)

Initialises fac for use. An fq_default_poly_factor_t represents a polynomial in factorised form as a product of polynomials with associated exponents.

void fq_default_poly_factor_clear(fq_default_poly_factor_t fac, const fq_default_ctx_t ctx)

Frees all memory associated with fac.

void fq_default_poly_factor_realloc(fq_default_poly_factor_t fac, slong alloc, const fq_default_ctx_t ctx)

Reallocates the factor structure to provide space for precisely alloc factors.

void fq_default_poly_factor_fit_length(fq_default_poly_factor_t fac, slong len, const fq_default_ctx_t ctx)

Ensures that the factor structure has space for at least len factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

11.9.3 Basic Operations

void fq_default_poly_factor_set(fq_default_poly_factor_t res, const fq_default_poly_factor_t fac, const fq_default_ctx_t ctx)

Sets res to the same factorisation as fac.

void fq_default_poly_factor_print_pretty(const fq_default_poly_factor_t fac, char *var, const fq_default_ctx_t ctx)

Pretty-prints the entries of fac to standard output.

void fq_default_poly_factor_print(const fq_default_poly_factor_t fac, const fq_default_ctx_t ctx)

Prints the entries of fac to standard output.

void fq_default_poly_factor_insert(fq_default_poly_factor_t fac, const fq_default_poly_t poly, slong exp, const fq_default_ctx_t ctx)

Inserts the factor poly with multiplicity exp into the factorisation fac.

If fac already contains poly, then exp simply gets added to the exponent of the existing entry.

void fq_default_poly_factor_concat(fq_default_poly_factor_t res, const fq_default_poly_factor_t fac, const fq_default_ctx_t ctx)

Concatenates two factorisations.

This is equivalent to calling fq_default_poly_factor_insert() repeatedly with the individual factors of fac.

Does not support aliasing between res and fac.

void fq_default_poly_factor_pow(fq_default_poly_factor_t fac, slong exp, const fq_default_ctx_t ctx)

Raises fac to the power exp.
ulong fq_default_poly_remove(fq_default_poly_t f, const fq_default_poly_t p, const fq_default_ctx_t ctx)

Removes the highest possible power of p from f and returns the exponent.

slong fq_default_poly_factor_length(fq_default_poly_factor_t fac, const fq_default_ctx_t ctx)

Return the number of factors, not including the unit.

void fq_default_poly_factor_get_poly(fq_default_poly_t poly, const fq_default_poly_factor_t fac, slong i, const fq_default_ctx_t ctx)

Set poly to factor i of fac (numbering starts at zero).

slong fq_default_poly_factor_exp(fq_default_poly_factor_t fac, slong i, const fq_default_ctx_t ctx)

Return the exponent of factor i of fac.

### 11.9.4 Irreducibility Testing

int fq_default_poly_is_irreducible(const fq_default_poly_t f, const fq_default_ctx_t ctx)

Returns 1 if the polynomial f is irreducible, otherwise returns 0.

int fq_default_poly_is_squarefree(const fq_default_poly_t f, const fq_default_ctx_t ctx)

Returns 1 if f is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

### 11.9.5 Factorisation

void fq_default_poly_factor_equal_deg(fq_default_poly_factor_t factors, const fq_default_poly_t pol, slong d, const fq_default_ctx_t ctx)

Assuming pol is a product of irreducible factors all of degree d, finds all those factors and places them in factors. Requires that pol be monic, non-constant and squarefree.

void fq_default_poly_factor_split_single(fq_default_poly_t linfactor, const fq_default_poly_t input, const fq_default_ctx_t ctx)

Assuming input is a product of factors all of degree 1, finds a single linear factor of input and places it in linfactor. Requires that input be monic and non-constant.

void fq_default_poly_factor_distinct_deg(fq_default_poly_factor_t res, const fq_default_poly_t poly, slong *const *degs, const fq_default_ctx_t ctx)

Factorises a monic non-constant squarefree polynomial poly of degree n into factors f[d] such that for 1 ≤ d ≤ n f[d] is the product of the monic irreducible factors of poly of degree d. Factors are stored in res, associated powers of irreducible polynomials are stored in degs in the same order as factors.

Requires that degs have enough space for irreducible polynomials’ powers (maximum space required is n * sizeof(slong)).

void fq_default_poly_factor_squarefree(fq_default_poly_factor_t res, const fq_default_poly_t f, const fq_default_ctx_t ctx)

Sets res to a squarefree factorization of f.

void fq_default_poly_factor(fq_default_poly_factor_t res, fq_default_t lead, const fq_default_poly_t f, const fq_default_ctx_t ctx)

Factorises a non-constant polynomial f into monic irreducible factors choosing the best algorithm for given modulo and degree. The output lead is set to the leading coefficient of f upon return. Choice of algorithm is based on heuristic measurements.
11.9.6 Root Finding

void \texttt{fq\_default\_poly\_roots}(\texttt{fq\_default\_poly\_factor\_t} r, const \texttt{fq\_default\_poly\_t} f, int with\_multiplicity, const \texttt{fq\_default\_ctx\_t} ctx)

Fill \(r\) with factors of the form \(x - r_i\), where the \(r_i\) are the distinct roots of a nonzero \(f\) in \(F_q\). If \texttt{with\_multiplicity} is zero, the exponent \(e_i\) of the factor \(x - r_i\) is 1. Otherwise, it is the largest \(e_i\) such that \((x - r_i)^{e_i}\) divides \(f\). This function throws if \(f\) is zero, but is otherwise always successful.

11.10 \texttt{fq\_embed.h} – Computing isomorphisms and embeddings of finite fields

void \texttt{fq\_embed\_gens}(\texttt{fq\_t} gen\_sub, \texttt{fq\_t} gen\_sup, \texttt{fmpz\_mod\_poly\_t} minpoly, const \texttt{fq\_ctx\_t} sub\_ctx, const \texttt{fq\_ctx\_t} sup\_ctx)

Given two contexts \texttt{sub\_ctx} and \texttt{sup\_ctx}, such that \texttt{degree}(\texttt{sub\_ctx}) divides \texttt{degree}(\texttt{sup\_ctx}), compute:

- an element \texttt{gen\_sub} in \texttt{sub\_ctx} such that \texttt{gen\_sub} generates the finite field defined by \texttt{sub\_ctx},
- its minimal polynomial \texttt{minpoly},
- a root \texttt{gen\_sup} of \texttt{minpoly} inside the field defined by \texttt{sup\_ctx}.

These data uniquely define an embedding of \texttt{sub\_ctx} into \texttt{sup\_ctx}.

void \texttt{\_fq\_embed\_gens\_naive}(\texttt{fq\_t} gen\_sub, \texttt{fq\_t} gen\_sup, \texttt{fmpz\_mod\_poly\_t} minpoly, const \texttt{fq\_ctx\_t} sub\_ctx, const \texttt{fq\_ctx\_t} sup\_ctx)

Given two contexts \texttt{sub\_ctx} and \texttt{sup\_ctx}, such that \texttt{degree}(\texttt{sub\_ctx}) divides \texttt{degree}(\texttt{sup\_ctx}), compute an embedding of \texttt{sub\_ctx} into \texttt{sup\_ctx} defined as follows:

- \texttt{gen\_sub} is the canonical generator of \texttt{sup\_ctx} (i.e., the class of \(X\)),
- \texttt{minpoly} is the defining polynomial of \texttt{sub\_ctx},
- \texttt{gen\_sup} is a root of \texttt{minpoly} inside the field defined by \texttt{sup\_ctx}.

void \texttt{fq\_embed\_matrices}(\texttt{fmpz\_mod\_mat\_t} embed, \texttt{fmpz\_mod\_mat\_t} project, const \texttt{fq\_t} gen\_sub, const \texttt{fq\_ctx\_t} sub\_ctx, const \texttt{fq\_ctx\_t} sup\_ctx, const \texttt{fmpz\_mod\_poly\_t} gen\_minpoly)

Given:

- two contexts \texttt{sub\_ctx} and \texttt{sup\_ctx}, of respective degrees \(m\) and \(n\), such that \(m\) divides \(n\);
- a generator \texttt{gen\_sub} of \texttt{sub\_ctx}, its minimal polynomial \texttt{gen\_minpoly}, and a root \texttt{gen\_sup} of \texttt{gen\_minpoly} in \texttt{sup\_ctx}, as returned by \texttt{fq\_embed\_gens()};

Compute:

- the \(n \times m\) matrix \texttt{embed} mapping \texttt{gen\_sub} to \texttt{gen\_sup}, and all their powers accordingly;
- an \(m \times n\) matrix \texttt{project} such that \texttt{project} \times \texttt{embed} is the \(m \times m\) identity matrix.

void \texttt{fq\_embed\_trace\_matrix}(\texttt{fmpz\_mod\_mat\_t} res, const \texttt{fmpz\_mod\_mat\_t} basis, const \texttt{fq\_ctx\_t} sub\_ctx, const \texttt{fq\_ctx\_t} sup\_ctx)

Given:

- two contexts \texttt{sub\_ctx} and \texttt{sup\_ctx}, of degrees \(m\) and \(n\), such that \(m\) divides \(n\);
- an \(n \times m\) matrix \texttt{basis} that maps \texttt{sub\_ctx} to an isomorphic subfield in \texttt{sup\_ctx};

Compute the \(m \times n\) matrix of the trace from \texttt{sup\_ctx} to \texttt{sub\_ctx}.

This matrix is computed as
embed_dual_to_mono_matrix(_, sub_ctx) × basis^t × embedMono_to_dual_matrix(_, sup_ctx).

Note: if $m = n$, basis represents a Frobenius, and the result is its inverse matrix.

void fq_embed_composition_matrix(fmpz_mod_mat_t matrix, const fq_t gen, const fq_ctx_t ctx)
Compute the composition matrix of gen.
For an element $a \in F_p^n$, its composition matrix is the matrix whose columns are $a^0, a^1, \ldots, a^{n-1}$.

void fq_embed_composition_matrix_sub(fmpz_mod_mat_t matrix, const fq_t gen, const fq_ctx_t ctx, slong trunc)
Compute the composition matrix of gen, truncated to trunc columns.

void fq_embed_mul_matrix(fmpz_mod_mat_t matrix, const fq_t gen, const fq_ctx_t ctx)
Compute the multiplication matrix of gen.
For an element $a$ in $F_p^n = F_p[x]/(f(X))$, its multiplication matrix is the matrix whose columns are $a, ax, \ldots, ax^{n-1}$.

void fq_embed_mono_to_dual_matrix(fmpz_mod_mat_t res, const fq_ctx_t ctx)
Compute the change of basis matrix from the monomial basis of ctx to its dual basis.

void fq_embed_dual_to_mono_matrix(fmpz_mod_mat_t res, const fq_ctx_t ctx)
Compute the change of basis matrix from the dual basis of ctx to its monomial basis.

void fq_modulus_pow_series_inv(fmpz_mod_poly_t res, const fq_ctx_t ctx, slong trunc)
Compute the power series inverse of the reverse of the modulus of ctx up to $O(x^{\text{trunc}})$.

void fq_modulus_derivative_inv(fq_t m_prime, fq_t m_prime_inv, const fq_ctx_t ctx)
Compute the derivative $m_{\text{prime}}$ of the modulus of ctx as an element of ctx, and its inverse $m_{\text{prime}_{\text{inv}}}$.

11.11 fq_nmod.h – finite fields (word-size characteristic)

We represent an element of the finite field $F_{p^n} \cong F_p[X]/(f(X))$, where $f(X) \in F_p[X]$ is a monic, irreducible polynomial of degree $n$, as a polynomial in $F_p[X]$ of degree less than $n$. The underlying data structure is an nmod_poly_t.

The default choice for $f(X)$ is the Conway polynomial for the pair $(p, n)$, enabled by Frank Lübeck’s data base of Conway polynomials using the _nmod_poly_conway() function. If a Conway polynomial is not available, then a random irreducible polynomial will be chosen for $f(X)$. Additionally, the user is able to supply their own $f(X)$.

11.11.1 Types, macros and constants

type fq_nmod_ctx_struct

type fq_nmod_ctx_t

type fq_nmod_struct

type fq_nmod_t
11.11.2 Context Management

```c
void fq_nmod_ctx_init_ui(fq_nmod_ctx_t ctx, ulong p, slong d, const char *var)
    Initialises the context for prime \( p \) and extension degree \( d \), with name \( \text{var} \) for the generator. By default, it will try use a Conway polynomial; if one is not available, a random irreducible polynomial will be used.
    Assumes that \( p \) is a prime.
    Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

int _fq_nmod_ctx_init_conway_ui(fq_nmod_ctx_t ctx, ulong p, slong d, const char *var)
    Attempts to initialise the context for prime \( p \) and extension degree \( d \), with name \( \text{var} \) for the generator using a Conway polynomial for the modulus.
    Returns 1 if the Conway polynomial is in the database for the given size and the initialization is successful; otherwise, returns 0.
    Assumes that \( p \) is a prime.
    Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

void fq_nmod_ctx_init_conway_ui(fq_nmod_ctx_t ctx, ulong p, slong d, const char *var)
    Initialises the context for prime \( p \) and extension degree \( d \), with name \( \text{var} \) for the generator using a Conway polynomial for the modulus.
    Assumes that \( p \) is a prime.
    Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

void fq_nmod_ctx_init_modulus(fq_nmod_ctx_t ctx, const nmod_poly_t modulus, const char *var)
    Initialises the context for given modulus with name \( \text{var} \) for the generator.
    Assumes that modulus is an irreducible polynomial over \( \mathbb{F}_p \).
    Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

void fq_nmod_ctx_init_randtest(fq_nmod_ctx_t ctx, flint_rand_t state, int type)
    Initialises \( \text{ctx} \) to a random finite field, where the prime and degree is set according to \( \text{type} \). To see what prime and degrees may be output, see \( \text{type} \) in \( \_nmod_poly_conway_rand() \).

void fq_nmod_ctx_init_randtest_reducible(fq_nmod_ctx_t ctx, flint_rand_t state, int type)
    Initializes \( \text{ctx} \) to a random extension of a word-sized prime field, where the prime and degree is set according to \( \text{type} \). If \( \text{type} \) is 0 the prime and degree may be large, else if \( \text{type} \) is 1 the degree is small but the prime may be large, else if \( \text{type} \) is 2 the prime is small but the degree may be large, else if \( \text{type} \) is 3 both prime and degree are small.
    The modulus may or may not be irreducible.

void fq_nmod_ctx_clear(fq_nmod_ctx_t ctx)
    Clears all memory that has been allocated as part of the context.

const nmod_poly_struct *fq_nmod_ctx_modulus(const fq_nmod_ctx_t ctx)
    Returns a pointer to the modulus in the context.

slong fq_nmod_ctx_degree(const fq_nmod_ctx_t ctx)
    Returns the degree of the field extension \( [\mathbb{F}_q : \mathbb{F}_p] \), which is equal to \( \log_p q \).

ulong fq_nmod_ctx_prime(const fq_nmod_ctx_t ctx)
    Returns the prime \( p \) of the context.

void fq_nmod_ctx_order(fmpz_t f, const fq_nmod_ctx_t ctx)
    Sets \( f \) to be the size of the finite field.
```
int fq_nmod_ctx_fprint(FILE *file, const fq_nmod_ctx_t ctx)
    Prints the context information to file. Returns 1 for a success and a negative number for an error.

void fq_nmod_ctx_print(const fq_nmod_ctx_t ctx)
    Prints the context information to stdout.

11.11.3 Memory management

void fq_nmod_init(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
    Initialises the element rop, setting its value to 0. Currently, the behaviour is identical to
    fq_nmod_init2, as it also ensures rop has enough space for it to be an element of ctx, this
    may change in the future.

void fq_nmod_init2(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
    Initialises rop with at least enough space for it to be an element of ctx and sets it to 0.

void fq_nmod_clear(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
    Clears the element rop.

void _fq_nmod_sparse_reduce(mp_limb_t *R, slong lenR, const fq_nmod_ctx_t ctx)
    Reduces (R, lenR) modulo the polynomial \( f \) given by the modulus of ctx.

void _fq_nmod_dense_reduce(mp_limb_t *R, slong lenR, const fq_nmod_ctx_t ctx)
    Reduces (R, lenR) modulo the polynomial \( f \) given by the modulus of ctx using Newton division.

void _fq_nmod_reduce(mp_limb_t *R, slong lenR, const fq_nmod_ctx_t ctx)
    Reduces (R, lenR) modulo the polynomial \( f \) given by the modulus of ctx. Does either sparse or
    dense reduction based on ctx->sparse_modulus.

void fq_nmod_reduce(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
    Reduces the polynomial rop as an element of \( \mathbb{F}_p[X]/(f(X)) \).

11.11.4 Basic arithmetic

void fq_nmod_add(fq_nmod_t rop, const fq_nmod_t op1, const fq_nmod_t op2, const fq_nmod_ctx_t ctx)
    Sets rop to the sum of op1 and op2.

void fq_nmod_sub(fq_nmod_t rop, const fq_nmod_t op1, const fq_nmod_t op2, const fq_nmod_ctx_t ctx)
    Sets rop to the difference of op1 and op2.

void fq_nmod_sub_one(fq_nmod_t rop, const fq_nmod_t op1, const fq_nmod_ctx_t ctx)
    Sets rop to the difference of op1 and 1.

void fq_nmod_neg(fq_nmod_t rop, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Sets rop to the negative of op.

void fq_nmod_mul(fq_nmod_t rop, const fq_nmod_t op1, const fq_nmod_t op2, const fq_nmod_ctx_t ctx)
    Sets rop to the product of op1 and op2, reducing the output in the given context.

void fq_nmod_mul_fmpz(fq_nmod_t rop, const fq_nmod_t op, const fmpz_t x, const fq_nmod_ctx_t ctx)
    Sets rop to the product of op and \( x \), reducing the output in the given context.

void fq_nmod_mul_si(fq_nmod_t rop, const fq_nmod_t op, slong x, const fq_nmod_ctx_t ctx)
    Sets rop to the product of op and \( x \), reducing the output in the given context.

11.11. fq_nmod.h – finite fields (word-size characteristic)
void fq_nmod_mul_ui(fq_nmod_t rop, const fq_nmod_t op, ulong x, const fq_nmod_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_nmod_sqr(fq_nmod_t rop, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Sets rop to the square of op, reducing the output in the given context.

void fq_nmod_inv(mp_ptr *rop, mp_srcptr *op, slong len, const fq_nmod_ctx_t ctx)
    Sets (rop, d) to the inverse of the non-zero element (op, len).

void fq_nmod_inv(fq_nmod_t rop, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Sets rop to the inverse of the non-zero element op.

void fq_nmod_gcdinv(fq_nmod_t f, fq_nmod_t inv, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Sets inv to be the inverse of op modulo the modulus of ctx. If op is not invertible, then f is set to a factor of the modulus; otherwise, it is set to one.

void _fq_nmod_pow(mp_limb_t *rop, const mp_limb_t *op, slong len, const fmpz_t e, const fq_nmod_ctx_t ctx)
    Sets (rop, 2*d-1) to (op,len) raised to the power e, reduced modulo \( f(X) \), the modulus of ctx.
    Assumes that \( e \geq 0 \) and that len is positive and at most \( d \).
    Although we require that rop provides space for \( 2d - 1 \) coefficients, the output will be reduced modulo \( f(X) \), which is a polynomial of degree \( d \).
    Does not support aliasing.

void fq_nmod_pow(fq_nmod_t rop, const fq_nmod_t op, const fmpz_t e, const fq_nmod_ctx_t ctx)
    Sets rop to op raised to the power e.
    Currently assumes that \( e \geq 0 \).
    Note that for any input op, rop is set to 1 whenever \( e = 0 \).

void fq_nmod_pow_ui(fq_nmod_t rop, const fq_nmod_t op, const ulong e, const fq_nmod_ctx_t ctx)
    Sets rop to op raised to the power e.
    Currently assumes that \( e \geq 0 \).
    Note that for any input op, rop is set to 1 whenever \( e = 0 \).

11.11.5 Roots

int fq_nmod_sqrt(fq_nmod_t rop, const fq_nmod_t op1, const fq_nmod_ctx_t ctx)
    Sets rop to the square root of op1 if it is a square, and return 1, otherwise return 0.

void fq_nmod_pth_root(fq_nmod_t rop, const fq_nmod_t op1, const fq_nmod_ctx_t ctx)
    Sets rop to a \( p^{th} \) root of op1. Currently, this computes the root by raising op1 to \( p^{d-1} \) where \( d \) is the degree of the extension.

int fq_nmod_is_square(const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Return 1 if op is a square.
11.11.6 Output

```c
int fq_nmod_fprint_pretty(FILE *file, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Prints a pretty representation of op to file.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

void fq_nmod_print_pretty(const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Prints a pretty representation of op to stdout.
    In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_nmod_fprint(FILE *file, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Prints a representation of op to file.
    For further details on the representation used, see nmod_poly_fprint().

void fq_nmod_print(const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Prints a representation of op to stdout.
    For further details on the representation used, see nmod_poly_print().

char *fq_nmod_get_str(const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Returns the plain FLINT string representation of the element op.

char *fq_nmod_get_str_pretty(const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Returns a pretty representation of the element op using the null-terminated string x as the variable name.
```

11.11.7 Randomisation

```c
void fq_nmod_randtest(fq_nmod_t rop, flint_rand_t state, const fq_nmod_ctx_t ctx)
    Generates a random element of \( F_q \).

void fq_nmod_randtest_not_zero(fq_nmod_t rop, flint_rand_t state, const fq_nmod_ctx_t ctx)
    Generates a random non-zero element of \( F_q \).

void fq_nmod_randtest_dense(fq_nmod_t rop, flint_rand_t state, const fq_nmod_ctx_t ctx)
    Generates a random element of \( F_q \) which has an underlying polynomial with dense coefficients.

void fq_nmod_rand(fq_nmod_t rop, flint_rand_t state, const fq_nmod_ctx_t ctx)
    Generates a high quality random element of \( F_q \).

void fq_nmod_rand_not_zero(fq_nmod_t rop, flint_rand_t state, const fq_nmod_ctx_t ctx)
    Generates a high quality non-zero random element of \( F_q \).
```

11.11.8 Assignments and conversions

```c
void fq_nmod_set(fq_nmod_t rop, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Sets rop to op.

void fq_nmod_set_si(fq_nmod_t rop, const slong x, const fq_nmod_ctx_t ctx)
    Sets rop to x, considered as an element of \( F_p \).

void fq_nmod_set_ui(fq_nmod_t rop, const ulong x, const fq_nmod_ctx_t ctx)
    Sets rop to x, considered as an element of \( F_p \).

void fq_nmod_set_fmpz(fq_nmod_t rop, const fmpz_t x, const fq_nmod_ctx_t ctx)
    Sets rop to x, considered as an element of \( F_p \).
```
Flint Documentation, Release 3.1.0-dev

```c
void fq_nmod_swap(fq_nmod_t op1, fq_nmod_t op2, const fq_nmod_ctx_t ctx)
    Swaps the two elements op1 and op2.

void fq_nmod_zero(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
    Sets rop to zero.

void fq_nmod_one(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
    Sets rop to one, reduced in the given context.

void fq_nmod_gen(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
    Sets rop to a generator for the finite field. There is no guarantee this is a multiplicative generator of the finite field.

int fq_nmod_get_fmpz(fmpz_t rop, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    If op has a lift to the integers, return 1 and set rop to the lift in \([0, p)\). Otherwise, return 0 and leave rop undefined.

void fq_nmod_get_nmod_poly(nmod_poly_t a, const fq_nmod_t b, const fq_nmod_ctx_t ctx)
    Set a to a representative of b in ctx. The representatives are taken in \((\mathbb{Z}/p\mathbb{Z})[x]/h(x)\) where \(h(x)\) is the defining polynomial in ctx.

void fq_nmod_set_nmod_poly(fq_nmod_t a, const nmod_poly_t b, const fq_nmod_ctx_t ctx)
    Set a to the element in ctx with representative b. The representatives are taken in \((\mathbb{Z}/p\mathbb{Z})[x]/h(x)\) where \(h(x)\) is the defining polynomial in ctx.

void fq_nmod_get_nmod_mat(nmod_mat_t col, const fq_nmod_t a, const fq_nmod_ctx_t ctx)
    Convert a to a column vector of length \(\deg(ctx)\).

void fq_nmod_set_nmod_mat(fq_nmod_t a, const nmod_mat_t col, const fq_nmod_ctx_t ctx)
    Convert a column vector col of length \(\deg(ctx)\) to an element of ctx.

11.11.9 Comparison

int fq_nmod_is_zero(const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Returns whether op is equal to zero.

int fq_nmod_is_one(const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Returns whether op is equal to one.

int fq_nmod_equal(const fq_nmod_t op1, const fq_nmod_t op2, const fq_nmod_ctx_t ctx)
    Returns whether op1 and op2 are equal.

int fq_nmod_is_invertible(const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Returns whether op is an invertible element.

int fq_nmod_is_invertible_f(fq_nmod_t f, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Returns whether op is an invertible element. If it is not, then f is set to a factor of the modulus.

int fq_nmod_cmp(const fq_nmod_t a, const fq_nmod_t b, const fq_nmod_ctx_t ctx)
    Return 1 (resp. -1, or 0) if a is after (resp. before, same as) b in some arbitrary but fixed total ordering of the elements.
```
11.11.10 Special functions

void _fq_nmod_trace(fmpz_t rop, const mp_limb_t *op, slong len, const fq_nmod_ctx_t ctx)
    Sets rop to the trace of the non-zero element (op, len) in \( F_q \).

void fq_nmod_trace(fmpz_t rop, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Sets rop to the trace of op.

For an element \( a \in F_q \), multiplication by \( a \) defines a \( F_p \)-linear map on \( F_q \). We define the trace of \( a \) as the trace of this map. Equivalently, if \( \Sigma \) generates \( \text{Gal}(F_q/F_p) \) then the trace of \( a \) is equal to \( \sum_{i=0}^{d-1} \Sigma^i(a) \), where \( d = \log_p q \).

int fq_nmod_multiplicative_order(fmpz *ord, const fq_nmod_t op, const fq_nmod_ctx_t ctx)
    Computes the order of op as an element of the multiplicative group of ctx.

Returns 0 if op is 0, otherwise it returns 1 if op is a generator of the multiplicative group, and -1 if it is not.

This function can also be used to check primitivity of a generator of a finite field whose defining polynomial is not primitive.

void fq_nmod_bit_pack(fmpz_t f, const fq_nmod_t op, flint_bitcnt_t bit_size, const fq_nmod_ctx_t ctx)
    Packs op into bitfields of size bit_size, writing the result to f.

void fq_nmod_bit_unpack(fq_nmod_t rop, const fmpz_t f, flint_bitcnt_t bit_size, const fq_nmod_ctx_t ctx)
    Unpacks into rop the element with coefficients packed into fields of size bit_size as represented by the integer f.

11.11. Bit packing
11.12 fq_nmod_vec.h – vectors over finite fields (word-size characteristic)

11.12.1 Memory management

\texttt{fq_nmod_struct *\_fq\_nmod\_vec\_init(slong len, const fq\_nmod\_ctx\_t ctx)}

Returns an initialised vector of \texttt{fq\_nmod}'s of given length.

\texttt{void \_fq\_nmod\_vec\_clear(fq\_nmod\_struct *vec, slong len, const fq\_nmod\_ctx\_t ctx)}

Clears the entries of \texttt{(vec, len)} and frees the space allocated for \texttt{vec}.

11.12.2 Randomisation

\texttt{void \_fq\_nmod\_vec\_randtest(fq\_nmod\_struct *f, flint\_rand\_t state, slong len, const fq\_nmod\_ctx\_t ctx)}

Sets the entries of a vector of the given length to elements of the finite field.

11.12.3 Input and output

\texttt{int \_fq\_nmod\_vec\_fprint(FILE *file, const fq\_nmod\_struct *vec, slong len, const fq\_nmod\_ctx\_t ctx)}

Prints the vector of given length to the stream \texttt{file}. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

\texttt{int \_fq\_nmod\_vec\_print(const fq\_nmod\_struct *vec, slong len, const fq\_nmod\_ctx\_t ctx)}

Prints the vector of given length to \texttt{stdout}.

For further details, see \texttt{\_fq\_nmod\_vec\_fprint()}.  

11.12.4 Assignment and basic manipulation

\texttt{void \_fq\_nmod\_vec\_set(fq\_nmod\_struct *vec1, const fq\_nmod\_struct *vec2, slong len2, const fq\_nmod\_ctx\_t ctx)}

Makes a copy of \texttt{(vec2, len2)} into \texttt{vec1}.

\texttt{void \_fq\_nmod\_vec\_swap(fq\_nmod\_struct *vec1, fq\_nmod\_struct *vec2, slong len2, const fq\_nmod\_ctx\_t ctx)}

Swaps the elements in \texttt{(vec1, len2)} and \texttt{(vec2, len2)}.

\texttt{void \_fq\_nmod\_vec\_zero(fq\_nmod\_struct *vec, slong len, const fq\_nmod\_ctx\_t ctx)}

Zeros the entries of \texttt{(vec, len)}.

\texttt{void \_fq\_nmod\_vec\_neg(fq\_nmod\_struct *vec1, const fq\_nmod\_struct *vec2, slong len2, const fq\_nmod\_ctx\_t ctx)}

Negates \texttt{(vec2, len2)} and places it into \texttt{vec1}. 


11.12.5 Comparison

```c
int _fq_nmod_vec_equal(const fq_nmod_struct *vec1, const fq_nmod_struct *vec2, slong len, const fq_nmod_ctx_t ctx)
```

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

```c
int _fq_nmod_vec_is_zero(const fq_nmod_struct *vec, slong len, const fq_nmod_ctx_t ctx)
```

Returns 1 if (vec, len) is zero, and 0 otherwise.

11.12.6 Addition and subtraction

```c
void _fq_nmod_vec_add(fq_nmod_struct *res, const fq_nmod_struct *vec1, const fq_nmod_struct *vec2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets (res, len2) to the sum of (vec1, len2) and (vec2, len2).

```c
void _fq_nmod_vec_sub(fq_nmod_struct *res, const fq_nmod_struct *vec1, const fq_nmod_struct *vec2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets (res, len2) to (vec1, len2) minus (vec2, len2).

11.12.7 Scalar multiplication and division

```c
void _fq_nmod_vec_scalar_addmul_fq_nmod(fq_nmod_struct *vec1, const fq_nmod_struct *vec2, slong len2, const fq_nmod_t c, const fq_nmod_ctx_t ctx)
```

Adds (vec2, len2) times c to (vec1, len2), where c is a fq_nmod_t.

```c
void _fq_nmod_vec_scalar_submul_fq_nmod(fq_nmod_struct *vec1, const fq_nmod_struct *vec2, slong len2, const fq_nmod_t c, const fq_nmod_ctx_t ctx)
```

Subtracts (vec2, len2) times c from (vec1, len2), where c is a fq_nmod_t.

11.12.8 Dot products

```c
void _fq_nmod_vec_dot(fq_nmod_t res, const fq_nmod_struct *vec1, const fq_nmod_struct *vec2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets res to the dot product of (vec1, len) and (vec2, len).

11.13 fq_nmod_mat.h – matrices over finite fields (word-size characteristic)

11.13.1 Types, macros and constants

```c
type fq_nmod_mat_struct

type fq_nmod_mat_t
```
11.13.2 Memory management

void fq_nmod_mat_init(fq_nmod_mat_t mat, slong rows, slong cols, const fq_nmod_ctx_t ctx)
    Initialises mat to a rows-by-cols matrix with coefficients in \( \mathbb{F}_q \) given by ctx. All elements are set to zero.

void fq_nmod_mat_init_set(fq_nmod_mat_t mat, const fq_nmod_mat_t src, const fq_nmod_ctx_t ctx)
    Initialises mat and sets its dimensions and elements to those of src.

void fq_nmod_mat_clear(fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
    Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an fq_nmod_mat_t object.

void fq_nmod_mat_set(fq_nmod_mat_t mat, const fq_nmod_mat_t src, const fq_nmod_ctx_t ctx)
    Sets mat to a copy of src. It is assumed that mat and src have identical dimensions.

11.13.3 Basic properties and manipulation

fq_nmod_struct *fq_nmod_mat_entry(const fq_nmod_mat_t mat, slong i, slong j)
    Directly accesses the entry in mat in row i and column j, indexed from zero. No bounds checking is performed.

void fq_nmod_mat_entry_set(fq_nmod_mat_t mat, slong i, slong j, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Sets the entry in mat in row i and column j to x.

slong fq_nmod_mat_nrows(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
    Returns the number of rows in mat.

slong fq_nmod_mat_ncols(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
    Returns the number of columns in mat.

void fq_nmod_mat_swap(fq_nmod_mat_t mat1, fq_nmod_mat_t mat2, const fq_nmod_ctx_t ctx)
    Swaps two matrices. The dimensions of mat1 and mat2 are allowed to be different.

void fq_nmod_mat_swap_entrywise(fq_nmod_mat_t mat1, fq_nmod_mat_t mat2, const fq_nmod_ctx_t ctx)
    Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

void fq_nmod_mat_zero(fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
    Sets all entries of mat to 0.

void fq_nmod_mat_one(fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
    Sets all diagonal entries of mat to 1 and all other entries to 0.

void fq_nmod_mat_swap_rows(fq_nmod_mat_t mat, slong *perm, slong r, slong s, const fq_nmod_ctx_t ctx)
    Swaps rows r and s of mat. If perm is non-NULL, the permutation of the rows will also be applied to perm.

void fq_nmod_mat_swap_cols(fq_nmod_mat_t mat, slong *perm, slong r, slong s, const fq_nmod_ctx_t ctx)
    Swaps columns r and s of mat. If perm is non-NULL, the permutation of the columns will also be applied to perm.
void fq_nmod_mat_invert_rows(fq_nmod_mat_t mat, slong *perm, const fq_nmod_ctx_t ctx)
   Swaps rows $i$ and $r - i$ of $\text{mat}$ for $0 \leq i < r/2$, where $r$ is the number of rows of $\text{mat}$. If $\text{perm}$ is non-NULL, the permutation of the rows will also be applied to $\text{perm}$.

void fq_nmod_mat_invert_cols(fq_nmod_mat_t mat, slong *perm, const fq_nmod_ctx_t ctx)
   Swaps columns $i$ and $c - i$ of $\text{mat}$ for $0 \leq i < c/2$, where $c$ is the number of columns of $\text{mat}$. If $\text{perm}$ is non-NULL, the permutation of the columns will also be applied to $\text{perm}$. 

11.13.4 Conversions

void fq_nmod_mat_set_nmod_mat(fq_nmod_mat_t mat1, const nmod_mat_t mat2, const fq_nmod_ctx_t ctx)
   Sets the matrix $\text{mat1}$ to the matrix $\text{mat2}$.

void fq_nmod_mat_set_fmpz_mod_mat(fq_nmod_mat_t mat1, const fmpz_mod_mat_t mat2, const fq_nmod_ctx_t ctx)
   Sets the matrix $\text{mat1}$ to the matrix $\text{mat2}$.

11.13.5 Concatenate

void fq_nmod_mat_concat_vertical(fq_nmod_mat_t res, const fq_nmod_mat_t mat1, const fq_nmod_mat_t mat2, const fq_nmod_ctx_t ctx)
   Sets $\text{res}$ to vertical concatenation of $(\text{mat1}, \text{mat2})$ in that order. Matrix dimensions: $\text{mat1} : m \times n$, $\text{mat2} : k \times n$, $\text{res} : (m + k) \times n$.

void fq_nmod_mat_concat_horizontal(fq_nmod_mat_t res, const fq_nmod_mat_t mat1, const fq_nmod_mat_t mat2, const fq_nmod_ctx_t ctx)
   Sets $\text{res}$ to horizontal concatenation of $(\text{mat1}, \text{mat2})$ in that order. Matrix dimensions: $\text{mat1} : m \times n$, $\text{mat2} : m \times k$, $\text{res} : m \times (n + k)$.

11.13.6 Printing

int fq_nmod_mat_print_pretty(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
   Pretty-prints $\text{mat}$ to stdout. A header is printed followed by the rows enclosed in brackets.

int fq_nmod_mat_fprint_pretty(FILE *file, const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
   Pretty-prints $\text{mat}$ to $\text{file}$. A header is printed followed by the rows enclosed in brackets.
   In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_nmod_mat_print(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
   Prints $\text{mat}$ to stdout. A header is printed followed by the rows enclosed in brackets.

int fq_nmod_mat_fprint(FILE *file, const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
   Prints $\text{mat}$ to $\text{file}$. A header is printed followed by the rows enclosed in brackets.
   In case of success, returns a positive value. In case of failure, returns a non-positive value.
11.13.7 Window

```c
void fq_nmod_mat_window_init(fq_nmod_mat_t window, const fq_nmod_mat_t mat, slong r1, slong c1, slong r2, slong c2, const fq_nmod_ctx_t ctx)
```

Initializes the matrix `window` to be an `r2 - r1` by `c2 - c1` submatrix of `mat` whose `(0,0)` entry is the `(r1, c1)` entry of `mat`. The memory for the elements of `window` is shared with `mat`.

```c
void fq_nmod_mat_window_clear(fq_nmod_mat_t window, const fq_nmod_ctx_t ctx)
```

Clears the matrix `window` and releases any memory that it uses. Note that the memory to the underlying matrix that `window` points to is not freed.

11.13.8 Random matrix generation

```c
void fq_nmod_mat_randtest(fq_nmod_mat_t mat, flint_rand_t state, const fq_nmod_ctx_t ctx)
```

Sets the elements of `mat` to random elements of \( F_q \), given by `ctx`.

```c
int fq_nmod_mat_randpermdiag(fq_nmod_mat_t mat, flint_rand_t state, fq_nmod_struct *diag, slong n, const fq_nmod_ctx_t ctx)
```

Sets `mat` to a random permutation of the diagonal matrix with `n` leading entries given by the vector `diag`. It is assumed that the main diagonal of `mat` has room for at least `n` entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

```c
void fq_nmod_mat_randrank(fq_nmod_mat_t mat, flint_rand_t state, slong rank, const fq_nmod_ctx_t ctx)
```

Sets `mat` to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of \( F_q \).

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling `fq_nmod_mat_randops()`.

```c
void fq_nmod_mat_randops(fq_nmod_mat_t mat, slong count, flint_rand_t state, const fq_nmod_ctx_t ctx)
```

Randomises `mat` by performing elementary row or column operations. More precisely, at most `count` random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

```c
void fq_nmod_mat_randtril(fq_nmod_mat_t mat, flint_rand_t state, int unit, const fq_nmod_ctx_t ctx)
```

Sets `mat` to a random lower triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

```c
void fq_nmod_mat_randtriu(fq_nmod_mat_t mat, flint_rand_t state, int unit, const fq_nmod_ctx_t ctx)
```

Sets `mat` to a random upper triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

11.13.9 Comparison

```c
int fq_nmod_mat_equal(const fq_nmod_mat_t mat1, const fq_nmod_mat_t mat2, const fq_nmod_ctx_t ctx)
```

Returns nonzero if `mat1` and `mat2` have the same dimensions and elements, and zero otherwise.

```c
int fq_nmod_mat_is_zero(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Returns a non-zero value if all entries `mat` are zero, and otherwise returns zero.
int fq_nmod_mat_is_one(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)

Returns a non-zero value if all entries mat are zero except the diagonal entries which must be one, otherwise returns zero.

int fq_nmod_mat_is_empty(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)

Returns a non-zero value if the number of rows or the number of columns in mat is zero, and otherwise returns zero.

int fq_nmod_mat_is_square(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)

Returns a non-zero value if the number of rows is equal to the number of columns in mat, and otherwise returns zero.

11.13.10 Addition and subtraction

void fq_nmod_mat_add(fq_nmod_mat_t C, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Computes $C = A + B$. Dimensions must be identical.

void fq_nmod_mat_sub(fq_nmod_mat_t C, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Computes $C = A - B$. Dimensions must be identical.

void fq_nmod_mat_neg(fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Sets $B = -A$. Dimensions must be identical.

11.13.11 Matrix multiplication

void fq_nmod_mat_mul(fq_nmod_mat_t C, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and KS multiplication.

void fq_nmod_mat_mul_classical(fq_nmod_mat_t C, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. $C$ is not allowed to be aliased with $A$ or $B$. Uses classical matrix multiplication.

void fq_nmod_mat_mul_KS(fq_nmod_mat_t C, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. $C$ is not allowed to be aliased with $A$ or $B$. Uses Kronecker substitution to perform the multiplication over the integers.

void fq_nmod_mat_submul(fq_nmod_mat_t D, const fq_nmod_mat_t C, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Sets $D = C + AB$. $C$ and $D$ may be aliased with each other but not with $A$ or $B$.

void fq_nmod_mat_mul_vec(fq_nmod_struct *c, const fq_nmod_mat_t A, const fq_nmod_struct *b, slong blen, const fq_nmod_ctx_t ctx)

void fq_nmod_mat_mul_vec_ptr(fq_nmod_struct *const *c, const fq_nmod_mat_t A, const fq_nmod_struct *const *b, slong blen, const fq_nmod_ctx_t ctx)

Compute a matrix-vector product of $A$ and $(b, blen)$ and store the result in $c$. The vector $(b, blen)$ is either truncated or zero-extended to the number of columns of $A$. The number entries written to $c$ is always equal to the number of rows of $A$.

void fq_nmod_mat_vec_mul(fq_nmod_struct *c, const fq_nmod_struct *a, slong alen, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)
void fq_nmod_mat_vec_mul_ptr(fq_nmod_struct *const *c, const fq_nmod_struct *const *a, slong alen, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Compute a vector-matrix product of (a, alen) and B and and store the result in c. The vector (a, alen) is either truncated or zero-extended to the number of rows of B. The number entries written to c is always equal to the number of columns of B.

11.13.12 Inverse

int fq_nmod_mat_inv(fq_nmod_mat_t B, fq_nmod_mat_t A, const fq_nmod_ctx_t ctx)

Sets $B = A^{-1}$ and returns 1 if A is invertible. If A is singular, returns 0 and sets the elements of $B$ to undefined values.

A and B must be square matrices with the same dimensions.

11.13.13 LU decomposition

slong fq_nmod_mat_lu(slong *P, fq_nmod_mat_t A, int rank_check, const fq_nmod_ctx_t ctx)

Computes a generalised LU decomposition $LU = PA$ of a given matrix $A$, returning the rank of $A$.

If $A$ is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix $L$ and an upper triangular matrix $U$ (the diagonal of $L$ will not be stored explicitly).

If $A$ is an arbitrary matrix of rank $r$, $U$ will be in row echelon form having $r$ nonzero rows, and $L$ will be lower triangular but truncated to $r$ columns, having implicit ones on the $r$ first entries of the main diagonal. All other entries will be zero.

If a nonzero value for rank_check is passed, the function will abandon the output matrix in an undefined state and return 0 if $A$ is detected to be rank-deficient.

This function calls fq_nmod_mat_lu_recursive.

slong fq_nmod_mat_lu_classical(slong *P, fq_nmod_mat_t A, int rank_check, const fq_nmod_ctx_t ctx)

Computes a generalised LU decomposition $LU = PA$ of a given matrix $A$, returning the rank of $A$. The behavior of this function is identical to that of fq_nmod_mat_lu. Uses Gaussian elimination.

slong fq_nmod_mat_lu_recursive(slong *P, fq_nmod_mat_t A, int rank_check, const fq_nmod_ctx_t ctx)

Computes a generalised LU decomposition $LU = PA$ of a given matrix $A$, returning the rank of $A$. The behavior of this function is identical to that of fq_nmod_mat_lu. Uses recursive block decomposition, switching to classical Gaussian elimination for sufficiently small blocks.

11.13.14 Reduced row echelon form

slong fq_nmod_mat_rref(fq_nmod_mat_t A, const fq_nmod_ctx_t ctx)

Puts $A$ in reduced row echelon form and returns the rank of $A$.

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

slong fq_nmod_mat_reduce_row(fq_nmod_mat_t A, slong *P, slong *L, slong n, const fq_nmod_ctx_t ctx)

Reduce row $n$ of the matrix $A$, assuming the prior rows are in Gauss form. However those rows may not be in order. The entry $i$ of the array $P$ is the row of $A$ which has a pivot in the $i$-th column. If no such row exists, the entry of $P$ will be $-1$. The function returns the column in which the $n$-th row has a pivot after reduction. This will always be chosen to be the first available column for a pivot from the left. This information is also updated in $P$. Entry $i$ of the array $L$ contains
the number of possibly nonzero columns of $A$ row $i$. This speeds up reduction in the case that $A$ is chambered on the right. Otherwise the entries of $L$ can all be set to the number of columns of $A$. We require the entries of $L$ to be monotonic increasing.

11.13.15 Triangular solving

void fq_nmod_mat_solve_tril(fq_nmod_mat_t X, const fq_nmod_mat_t L, const fq_nmod_mat_t B, int unit, const fq_nmod_ctx_t ctx)

Sets $X = L^{-1}B$ where $L$ is a full rank lower triangular square matrix. If $unit = 1$, $L$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

void fq_nmod_mat_solve_tril_classical(fq_nmod_mat_t X, const fq_nmod_mat_t L, const fq_nmod_mat_t B, int unit, const fq_nmod_ctx_t ctx)

Sets $X = L^{-1}B$ where $L$ is a full rank lower triangular square matrix. If $unit = 1$, $L$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

void fq_nmod_mat_solve_tril_recursive(fq_nmod_mat_t X, const fq_nmod_mat_t L, const fq_nmod_mat_t B, int unit, const fq_nmod_ctx_t ctx)

Sets $X = L^{-1}B$ where $L$ is a full rank lower triangular square matrix. If $unit = 1$, $L$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Uses the block inversion formula $$(A 0 \ 0 D)^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}X \\ D^{-1}(Y - CA^{-1}X) \end{pmatrix}$$ to reduce the problem to matrix multiplication and triangular solving of smaller systems.

void fq_nmod_mat_solve_triu(fq_nmod_mat_t X, const fq_nmod_mat_t U, const fq_nmod_mat_t B, int unit, const fq_nmod_ctx_t ctx)

Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If $unit = 1$, $U$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

void fq_nmod_mat_solve_triu_classical(fq_nmod_mat_t X, const fq_nmod_mat_t U, const fq_nmod_mat_t B, int unit, const fq_nmod_ctx_t ctx)

Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If $unit = 1$, $U$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

void fq_nmod_mat_solve_triu_recursive(fq_nmod_mat_t X, const fq_nmod_mat_t U, const fq_nmod_mat_t B, int unit, const fq_nmod_ctx_t ctx)

Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If $unit = 1$, $U$ is assumed to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed to be the same matrix, but no other aliasing is allowed. Uses the block inversion formula $$(A 0 B' \ 0 D)^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}(X - BD^{-1}Y) \\ D^{-1}Y \end{pmatrix}$$ to reduce the problem to matrix multiplication and triangular solving of smaller systems.
11.13.16 Solving

int fq_nmod_mat_solve(fq_nmod_mat_t X, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Solves the matrix-matrix equation $AX = B$.
Returns 1 if $A$ has full rank; otherwise returns 0 and sets the elements of $X$ to undefined values.
The matrix $A$ must be square.

int fq_nmod_mat_can_solve(fq_nmod_mat_t X, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)

Solves the matrix-matrix equation $AX = B$ over $F_q$.
Returns 1 if a solution exists; otherwise returns 0 and sets the elements of $X$ to zero. If more than
one solution exists, one of the valid solutions is given.
There are no restrictions on the shape of $A$ and it may be singular.

11.13.17 Transforms

void fq_nmod_mat_similarity(fq_nmod_mat_t M, slong r, fq_nmod_t d, const fq_nmod_ctx_t ctx)

Applies a similarity transform to the $n \times n$ matrix $M$ in-place.
If $P$ is the $n \times n$ identity matrix the zero entries of whose row $r$ (0-indexed) have been replaced by $d$, this transform is equivalent to $M = P^{-1}MP$.
Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.
The value $d$ is required to be reduced modulo the modulus of the entries in the matrix.

11.13.18 Characteristic polynomial

void fq_nmod_mat_charpoly_danilevsky(fq_nmod_poly_t p, const fq_nmod_mat_t M, const fq_nmod_ctx_t ctx)

Compute the characteristic polynomial $p$ of the matrix $M$. The matrix is assumed to be square.

void fq_nmod_mat_charpoly(fq_nmod_poly_t p, const fq_nmod_mat_t M, const fq_nmod_ctx_t ctx)

Compute the characteristic polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised.

11.13.19 Minimal polynomial

void fq_nmod_mat_minpoly(fq_nmod_poly_t p, const fq_nmod_mat_t M, const fq_nmod_ctx_t ctx)

Compute the minimal polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised.
11.14 fq_nmod_poly.h – univariate polynomials over finite fields (word-size characteristic)

We represent a polynomial in \( \mathbb{F}_q[X] \) as a *struct* which includes an array *coeffs* with the coefficients, as well as the length *length* and the number *alloc* of coefficients for which memory has been allocated. As a data structure, we call this polynomial *normalised* if the top coefficient is non-zero.

Unless otherwise stated here, all functions that deal with polynomials assume that the \( \mathbb{F}_q \) context of said polynomials are compatible, i.e., it assumes that the fields are generated by the same polynomial.

11.14.1 Types, macros and constants

type fq_nmod_poly_struct

type fq_nmod_poly_t

11.14.2 Memory management

```c
void fq_nmod_poly_init(fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
Initialises poly for use, with context ctx, and setting its length to zero. A corresponding call to fq_nmod_poly_clear() must be made after finishing with the fq_nmod_poly_t to free the memory used by the polynomial.
```

```c
void fq_nmod_poly_init2(fq_nmod_poly_t poly, slong alloc, const fq_nmod_ctx_t ctx)
Initialises poly with space for at least alloc coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to fq_nmod_poly_clear() must be made after finishing with the fq_nmod_poly_t to free the memory used by the polynomial.
```

```c
void fq_nmod_poly_realloc(fq_nmod_poly_t poly, slong alloc, const fq_nmod_ctx_t ctx)
Reallocates the given polynomial to have space for alloc coefficients. If alloc is zero the polynomial is cleared and then reinitialised. If the current length is greater than alloc the polynomial is first truncated to length alloc.
```

```c
void fq_nmod_poly_fit_length(fq_nmod_poly_t poly, slong len, const fq_nmod_ctx_t ctx)
If len is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least len coefficients. No data is lost when calling this function.
```

The function efficiently deals with the case where fit_length is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

```c
void _fq_nmod_poly_set_length(fq_nmod_poly_t poly, slong newlen, const fq_nmod_ctx_t ctx)
Sets the coefficients of poly beyond len to zero and sets the length of poly to len.
```

```c
void fq_nmod_poly_clear(fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.
```

```c
void _fq_nmod_poly_normalise(fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
Sets the length of poly so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.
```

```c
void _fq_nmod_poly_normalise2(const fq_nmod_struct *poly, slong *length, const fq_nmod_ctx_t ctx)
Sets the length length of (poly,length) so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.
```
void fq_nmod_poly_truncate(fq_nmod_poly_t poly, slong newlen, const fq_nmod_ctx_t ctx)
    Truncates the polynomial to length at most n.

void fq_nmod_poly_set_trunc(fq_nmod_poly_t poly1, fq_nmod_poly_t poly2, slong newlen, const fq_nmod_ctx_t ctx)
    Sets poly1 to poly2 truncated to length n.

void _fq_nmod_poly_reverse(fq_nmod_struct *output, const fq_nmod_struct *input, slong len, const fq_nmod_ctx_t ctx)
    Sets output to the reverse of input, which is of length len, but thinking of it as a polynomial of length m, notionally zero-padded if necessary. The length m must be non-negative, but there are no other restrictions. The polynomial output must have space for m coefficients.

void fq_nmod_poly_reverse(fq_nmod_poly_t output, const fq_nmod_poly_t input, slong m, const fq_nmod_ctx_t ctx)
    Sets output to the reverse of input, thinking of it as a polynomial of length m, notionally zero-padded if necessary). The length m must be non-negative, but there are no other restrictions. The output polynomial will be set to length m and then normalised.

11.14.3 Polynomial parameters

slong fq_nmod_poly_degree(const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
    Returns the degree of the polynomial poly.

slong fq_nmod_poly_length(const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
    Returns the length of the polynomial poly.

fq_nmod_struct *fq_nmod_poly_lead(const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
    Returns a pointer to the leading coefficient of poly, or NULL if poly is the zero polynomial.

11.14.4 Randomisation

void fq_nmod_poly_randtest(fq_nmod_poly_t f, flint_rand_t state, slong len, const fq_nmod_ctx_t ctx)
    Sets f to a random polynomial of length at most len with entries in the field described by ctx.

void fq_nmod_poly_randtest_not_zero(fq_nmod_poly_t f, flint_rand_t state, slong len, const fq_nmod_ctx_t ctx)
    Same as fq_nmod_poly_randtest but guarantees that the polynomial is not zero.

void fq_nmod_poly_randtest_monic(fq_nmod_poly_t f, flint_rand_t state, slong len, const fq_nmod_ctx_t ctx)
    Sets f to a random monic polynomial of length len with entries in the field described by ctx.

void fq_nmod_poly_randtest_irreducible(fq_nmod_poly_t f, flint_rand_t state, slong len, const fq_nmod_ctx_t ctx)
    Sets f to a random monic, irreducible polynomial of length len with entries in the field described by ctx.
11.14.5 Assignment and basic manipulation

```c
void _fq_nmod_poly_set(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
    Sets (rop, len) to (op, len).
void fq_nmod_poly_set(fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const fq_nmod_ctx_t ctx)
    Sets the polynomial poly1 to the polynomial poly2.
void fq_nmod_poly_set_fq_nmod(fq_nmod_poly_t poly, const fq_nmod_t c, const fq_nmod_ctx_t ctx)
    Sets the polynomial poly to c.
void fq_nmod_poly_set_fmpz_mod_poly(fq_nmod_poly_t rop, const fmpz_mod_poly_t op, const fq_nmod_ctx_t ctx)
    Sets the polynomial rop to the polynomial op.
void fq_nmod_poly_set_nmod_poly(fq_nmod_poly_t rop, const nmod_poly_t op, const fq_nmod_ctx_t ctx)
    Sets the polynomial rop to the polynomial op.
void fq_nmod_poly_swap(fq_nmod_poly_t op1, fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)
    Swaps the two polynomials op1 and op2.
void _fq_nmod_poly_zero(fq_nmod_struct *rop, slong len, const fq_nmod_ctx_t ctx)
    Sets (rop, len) to the zero polynomial.
void fq_nmod_poly_zero(fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
    Sets poly to the zero polynomial.
void fq_nmod_poly_one(fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
    Sets poly to the constant polynomial 1.
void fq_nmod_poly_gen(fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
    Sets poly to the polynomial x.
void fq_nmod_poly_make_monic(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
    Sets rop to op, normed to have leading coefficient 1.
void _fq_nmod_poly_make_monic(fq_nmod_struct *rop, const fq_nmod_struct *op, slong length, const fq_nmod_ctx_t ctx)
    Sets rop to (op, length), normed to have leading coefficient 1. Assumes that rop has enough space for the polynomial, assumes that op is not zero (and thus has an invertible leading coefficient).
```

11.14.6 Getting and setting coefficients

```c
void fq_nmod_poly_get_coeff(fq_nmod_t x, const fq_nmod_poly_t poly, slong n, const fq_nmod_ctx_t ctx)
    Sets x to the coefficient of X^n in poly.
void fq_nmod_poly_set_coeff(fq_nmod_poly_t poly, slong n, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Sets the coefficient of X^n in poly to x.
void fq_nmod_poly_set_coeff_fmpz(fq_nmod_poly_t poly, slong n, const fmpz_t x, const fq_nmod_ctx_t ctx)
    Sets the coefficient of X^n in the polynomial to x, assuming n ≥ 0.
```
11.14.7 Comparison

```c
int fq_nmod_poly_equal(const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const fq_nmod_ctx_t ctx)
```

Returns nonzero if the two polynomials \( poly1 \) and \( poly2 \) are equal, otherwise return zero.

```c
int fq_nmod_poly_equal_trunc(const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, slong n, const fq_nmod_ctx_t ctx)
```

Notionally truncate \( poly1 \) and \( poly2 \) to length \( n \) and return nonzero if they are equal, otherwise return zero.

```c
int fq_nmod_poly_is_zero(const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial \( poly \) is the zero polynomial.

```c
int fq_nmod_poly_is_one(const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial \( poly \) is equal to the constant polynomial 1.

```c
int fq_nmod_poly_is_gen(const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial \( poly \) is equal to the polynomial \( x \).

```c
int fq_nmod_poly_is_unit(const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial \( poly \) is a unit in the polynomial ring \( \mathbb{F}_q[X] \), i.e. if it has degree 0 and is non-zero.

```c
int fq_nmod_poly_equal_ffq(const fq_nmod_poly_t poly, const fq_nmod_t c, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial \( poly \) is equal the (constant) \( \mathbb{F}_q \) element \( c \).

11.14.8 Addition and subtraction

```c
void _fq_nmod_poly_add(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const fq_nmod_struct *poly2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets \( res \) to the sum of \( (poly1,len1) \) and \( (poly2,len2) \).

```c
void fq_nmod_poly_add(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const fq_nmod_ctx_t ctx)
```

Sets \( res \) to the sum of \( poly1 \) and \( poly2 \).

```c
void fq_nmod_poly_add_si(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, slong c, const fq_nmod_ctx_t ctx)
```

Sets \( res \) to the sum of \( poly1 \) and \( c \).

```c
void fq_nmod_poly_add_series(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, slong n, const fq_nmod_ctx_t ctx)
```

Notionally truncate \( poly1 \) and \( poly2 \) to length \( n \) and set \( res \) to the sum.

```c
void _fq_nmod_poly_sub(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const fq_nmod_struct *poly2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets \( res \) to the difference of \( (poly1,len1) \) and \( (poly2,len2) \).

```c
void fq_nmod_poly_sub(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const fq_nmod_ctx_t ctx)
```

Sets \( res \) to the difference of \( poly1 \) and \( poly2 \).

```c
void fq_nmod_poly_sub_series(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, slong n, const fq_nmod_ctx_t ctx)
```

Notionally truncate \( poly1 \) and \( poly2 \) to length \( n \) and set \( res \) to the difference.
void _fq_nmod_poly_neg(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
    Sets rop to the additive inverse of (poly,len).
void fq_nmod_poly_neg(fq_nmod_poly_t res, const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
    Sets res to the additive inverse of poly.

11.14.9 Scalar multiplication and division

void _fq_nmod_poly_scalar_mul_fq_nmod(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Sets (rop,len) to the product of (op,len) by the scalar x, in the context defined by ctx.
void fq_nmod_poly_scalar_mul_fq_nmod(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Sets rop to the product of op by the scalar x, in the context defined by ctx.
void _fq_nmod_poly_scalar_addmul_fq_nmod(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Adds to (rop,len) the product of (op,len) by the scalar x, in the context defined by ctx. In particular, assumes the same length for op and rop.
void fq_nmod_poly_scalar_addmul_fq_nmod(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Adds to rop the product of op by the scalar x, in the context defined by ctx.
void _fq_nmod_poly_scalar_submul_fq_nmod(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Subtracts from (rop,len) the product of (op,len) by the scalar x, in the context defined by ctx. In particular, assumes the same length for op and rop.
void fq_nmod_poly_scalar_submul_fq_nmod(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Subtracts from rop the product of op by the scalar x, in the context defined by ctx.
void _fq_nmod_poly_scalar_div_fq(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Sets (rop,len) to the quotient of (op,len) by the scalar x, in the context defined by ctx. An exception is raised if x is zero.
void fq_nmod_poly_scalar_div_fq(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
    Sets rop to the quotient of op by the scalar x, in the context defined by ctx. An exception is raised if x is zero.

11.14.10 Multiplication

void _fq_nmod_poly_mul_classical(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)
    Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), assuming that len1 is at least len2 and neither is zero.
    Permits zero padding. Does not support aliasing of rop with either op1 or op2.
void fq_nmod_poly_mul_classical(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)

Sets rop to the product of op1 and op2 using classical polynomial multiplication.

void _fq_nmod_poly_mul_reorder(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), assuming that len1 and len2 are non-zero.

Permits zero padding. Supports aliasing.

void fq_nmod_poly_mul_reorder(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)

Sets rop to the product of op1 and op2, reordering the two indeterminates $X$ and $Y$ when viewing the polynomials as elements of $\mathbb{F}_p[X,Y]$.

Suppose $\mathbb{F}_q = \mathbb{F}_p[X]/(f(X))$ and recall that elements of $\mathbb{F}_q$ are internally represented by elements of type fmpz_poly. For small degree extensions but polynomials in $\mathbb{F}_q[Y]$ of large degree $n$, we change the representation to

$$g(Y) = \sum_{i=0}^{n} a_i(X)Y^i = \sum_{j=0}^{d} \sum_{i=0}^{n} \text{Coeff}(a_i(X),j)Y^i.$$

This allows us to use a poor algorithm (such as classical multiplication) in the $X$-direction and leverage the existing fast integer multiplication routines in the $Y$-direction where the polynomial degree $n$ is large.

void _fq_nmod_poly_mul_univariate(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2).

Permits zero padding and makes no assumptions on len1 and len2. Supports aliasing.

void fq_nmod_poly_mul_univariate(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)

Sets rop to the product of op1 and op2 using a bivariate to univariate transformation and reducing this problem to multiplying two univariate polynomials.

void _fq_nmod_poly_mul_KS(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2).

Permits zero padding and places no assumptions on the lengths len1 and len2. Supports aliasing.

void fq_nmod_poly_mul_KS(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)

Sets rop to the product of op1 and op2 using Kronecker substitution, that is, by encoding each coefficient in $\mathbb{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

void _fq_nmod_poly_mul(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.
void fq_nmod_poly_mul(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)

Sets rop to the product of op1 and op2, choosing an appropriate algorithm.

void _fq_nmod_poly_mullow_classical(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, slong n, const fq_nmod_ctx_t ctx)

Sets (rop, n) to the first n coefficients of (op1, len1) multiplied by (op2, len2).

Assumes 0 < n <= len1 + len2 - 1. Assumes neither len1 nor len2 is zero.

void fq_nmod_poly_mullow_classical(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, slong n, const fq_nmod_ctx_t ctx)

Sets rop to the product of op1 and op2, computed using the classical or schoolbook method.

void fq_nmod_poly_mullow_univariate(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, slong n, const fq_nmod_ctx_t ctx)

Sets (rop, n) to the lowest n coefficients of the product of (op1, len1) multiplied by (op2, len2), computed using a bivariate to univariate transformation.

Assumes that len1 and len2 are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between rop, op1 and op2.

void fq_nmod_poly_mullow_univariate(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, slong n, const fq_nmod_ctx_t ctx)

Sets rop to the lowest n coefficients of the product of op1 and op2, computed using a bivariate to univariate transformation.

void _fq_nmod_poly_mullow_KS(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, slong n, const fq_nmod_ctx_t ctx)

Sets (rop, n) to the lowest n coefficients of the product of (op1, len1) multiplied by (op2, len2).

Assumes that len1 and len2 are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between rop, op1 and op2.

void fq_nmod_poly_mullow_KS(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, slong n, const fq_nmod_ctx_t ctx)

Sets rop to the product of op1 and op2.

void _fq_nmod_poly_mullow(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, slong n, const fq_nmod_ctx_t ctx)

Sets (rop, n) to the lowest n coefficients of the product of (op1, len1) multiplied by (op2, len2).

Assumes 0 < n <= len1 + len2 - 1. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

void fq_nmod_poly_mullow(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, slong n, const fq_nmod_ctx_t ctx)

Sets rop to the lowest n coefficients of the product of op1 and op2.

void _fq_nmod_poly_mulhigh_classical(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const fq_nmod_struct *poly2, slong len2, slong start, const fq_nmod_ctx_t ctx)

Computes the product of (poly1, len1) and (poly2, len2) and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Assumes that len1 >= len2 > 0. Aliasing of inputs and output is not permitted. Algorithm is classical multiplication.
void fq_nmod_poly_mulhigh_classical(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, slong start, const fq_nmod_ctx_t ctx)

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Algorithm is classical multiplication.

void _fq_nmod_poly_mulhigh(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const fq_nmod_struct *poly2, slong len2, slong start, const fq_nmod_ctx_t ctx)

Computes the product of (poly1, len1) and (poly2, len2) and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Assumes that len1 >= len2 > 0. Aliasing of inputs and output is not permitted.

void fq_nmod_poly_mulhigh(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, slong start, const fq_nmod_ctx_t ctx)

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced.

void _fq_nmod_poly_mulmod(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const fq_nmod_struct *poly2, slong len2, constfq_nmod_struct *f, slong lenf, const fq_nmod_ctx_t ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. It is required that len1 + len2 - lenf > 0, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use _fq_nmod_poly_mul instead. Aliasing of f and res is not permitted.

void fq_nmod_poly_mulmod(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. finv is the inverse of the reverse of f mod x^lenf. Aliasing of res with any of the inputs is not permitted.

void _fq_nmod_poly_mulmod_preinv(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const fq_nmod_struct *polyn2, slong len2, const fq_nmod_struct *f, slong lenf, const fq_nmod_struct *finv, slong lenfinv, const fq_nmod_ctx_t ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. It is required that finv is the inverse of the reverse of f mod x^lenf. Aliasing of res with any of the inputs is not permitted.

void fq_nmod_poly_mulmod_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const fq_nmod_poly_t f, const fq_nmod_poly_t finv, const fq_nmod_ctx_t ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f. finv is the inverse of the reverse of f.

11.14.11 Squaring

void _fq_nmod_poly_sqr_classical(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)

Sets (rop, 2*len - 1) to the square of (op, len), assuming that (op, len) is not zero and using classical polynomial multiplication. Permits zero padding. Does not support aliasing of rop with either op1 or op2.
**void fq_nmod_poly_sqr_classical**(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)

Sets rop to the square of op using classical polynomial multiplication.

**void _fq_nmod_poly_sqr_KS**(fq_nmod_struct* rop, const fq_nmod_struct* op, slong len, const fq_nmod_ctx_t ctx)

Sets (rop, 2*len - 1) to the square of (op, len).

Permits zero padding and places no assumptions on the lengths len1 and len2. Supports aliasing.

**void fq_nmod_poly_sqr_KS**(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)

Sets rop to the square of op using Kronecker substitution, that is, by encoding each coefficient in \( \mathbb{F}_q \) as an integer and reducing this problem to multiplying two polynomials over the integers.

**void _fq_nmod_poly_sqr**(fq_nmod_struct* rop, const fq_nmod_struct* op, slong len, const fq_nmod_ctx_t ctx)

Sets (rop, 2 * len - 1) to the square of (op, len), choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

**void fq_nmod_poly_sqr**(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)

Sets rop to the square of op, choosing an appropriate algorithm.

### 11.14.12 Powering

**void _fq_nmod_poly_pow**(fq_nmod_struct* rop, const fq_nmod_struct* op, slong len, ulong e, const fq_nmod_ctx_t ctx)

Sets rop = op^e, assuming that e, len > 0 and that rop has space for e*(len - 1) + 1 coefficients. Does not support aliasing.

**void fq_nmod_poly_pow**(fq_nmod_poly_t rop, const fq_nmod_poly_t op, ulong e, const fq_nmod_ctx_t ctx)

Computes rop = op^e. If e is zero, returns one, so that in particular 0^0 = 1.

**void _fq_nmod_poly_powmod_ui_binexp**(fq_nmod_struct* res, const fq_nmod_struct* poly, ulong e, const fq_nmod_struct* f, slong lenf, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

**void fq_nmod_poly_powmod_ui_binexp**(fq_nmod_poly_t res, const fq_nmod_poly_t poly, ulong e, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0.

**void _fq_nmod_poly_powmod_ui_binexp_preinv**(fq_nmod_struct* res, const fq_nmod_struct* poly, ulong e, const fq_nmod_struct* f, slong lenf, const fq_nmod_struct* finv, slong lenfinv, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require finv to be the inverse of the reverse of f.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.
void fq_nmod_poly_powmod_ui_binexp_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t poly, ulong e, const fq_nmod_poly_t f, const fq_nmod_poly_t finv, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require finv to be the inverse of the reverse of f.

void _fq_nmod_poly_powmod_fmpz_binexp(fq_nmod_struct *res, const fq_nmod_struct *poly, const fmpz_t e, const fq_nmod_struct *f, slong lenf, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_nmod_poly_powmod_fmpz_binexp(fq_nmod_poly_t res, const fq_nmod_poly_t poly, const fmpz_t e, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0.

void _fq_nmod_poly_powmod_fmpz_binexp_preinv(fq_nmod_struct *res, const fq_nmod_struct *poly, const fmpz_t e, const fq_nmod_struct *f, slong lenf, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_nmod_poly_powmod_fmpz_binexp_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t poly, const fmpz_t e, const fq_nmod_poly_t f, const fq_nmod_poly_t finv, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require finv to be the inverse of the reverse of f.

void _fq_nmod_poly_powmod_fmpz_sliding_preinv(fq_nmod_struct *res, const fq_nmod_struct *poly, const fmpz_t e, ulong k, const fq_nmod_struct *f, slong lenf, const fq_nmod_struct *finv, slong lenfinv, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using sliding-window exponentiation with window size k. We require e > 0. We require finv to be the inverse of the reverse of f. If k is set to zero, then an “optimum” size will be selected automatically base on e. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_nmod_poly_powmod_fmpz_sliding_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t poly, const fmpz_t e, ulong k, const fq_nmod_poly_t f, const fq_nmod_poly_t finv, const fq_nmod_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using sliding-window exponentiation with window size k. We require e >= 0. We require finv to be the inverse of the reverse of f. If k is set to zero, then an “optimum” size will be selected automatically base on e.

void _fq_nmod_poly_powmod_x_fmpz_preinv(fq_nmod_struct *res, const fmpz_t e, const fq_nmod_struct *f, slong lenf, const fq_nmod_struct *finv, slong lenfinv, const fq_nmod_ctx_t ctx)
Sets \( res \) to \( x \) raised to the power \( e \) modulo \( f \), using sliding window exponentiation. We require \( e > 0 \). We require \( \text{finv} \) to be the inverse of the reverse of \( f \).

We require \( \text{lenf} > 2 \). The output \( res \) must have room for \( \text{lenf} - 1 \) coefficients.

```c
void fq_nmod_poly_powmod_x_fmpz_preinv(fq_nmod_poly_t res, const fmpz_t e, const fq_nmod_poly_t f, const fq_nmod_poly_t finv, const fq_nmod_ctx_t ctx)
```
Sets \( res \) to \( x \) raised to the power \( e \) modulo \( f \), using sliding window exponentiation. We require \( \text{finv} \) to be the inverse of the reverse of \( f \).

```c
void _fq_nmod_poly_pow_trunc_binexp(fq_nmod_struct *res, const fq_nmod_struct *poly, ulong e, slong trunc, const fq_nmod_ctx_t ctx)
```
Sets \( res \) to the low \( \text{trunc} \) coefficients of \( \text{poly} \) (assumed to be zero padded if necessary to length \( \text{trunc} \)) to the power \( e \). This is equivalent to doing a powering followed by a truncation. We require that \( \text{res} \) has enough space for \( \text{trunc} \) coefficients, that \( \text{trunc} > 0 \) and that \( e > 1 \). Aliasing is not permitted. Uses the binary exponentiation method.

```c
void fq_nmod_poly_pow_trunc_binexp(fq_nmod_poly_t res, const fq_nmod_poly_t poly, ulong e, slong trunc, const fq_nmod_ctx_t ctx)
```
Sets \( res \) to the low \( \text{trunc} \) coefficients of \( \text{poly} \) to the power \( e \). This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.

```c
void _fq_nmod_poly_pow_trunc(fq_nmod_struct *res, const fq_nmod_struct *poly, ulong e, slong trunc, const fq_nmod_ctx_t mod)
```
Sets \( res \) to the low \( \text{trunc} \) coefficients of \( \text{poly} \) (assumed to be zero padded if necessary to length \( \text{trunc} \)) to the power \( e \). This is equivalent to doing a powering followed by a truncation. We require that \( \text{res} \) has enough space for \( \text{trunc} \) coefficients, that \( \text{trunc} > 0 \) and that \( e > 1 \). Aliasing is not permitted.

```c
void fq_nmod_poly_pow_trunc(fq_nmod_poly_t res, const fq_nmod_poly_t poly, ulong e, slong trunc, const fq_nmod_ctx_t ctx)
```
Sets \( res \) to the low \( \text{trunc} \) coefficients of \( \text{poly} \) to the power \( e \). This is equivalent to doing a powering followed by a truncation.

### 11.14.13 Shifting

```c
void _fq_nmod_poly_shift_left(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, slong n, const fq_nmod_ctx_t ctx)
```
Sets \( (\text{rop}, \text{len} + n) \) to \( (\text{op}, \text{len}) \) shifted left by \( n \) coefficients.

Inserts zero coefficients at the lower end. Assumes that \( \text{len} \) and \( n \) are positive, and that \( \text{rop} \) fits \( \text{len} + n \) elements. Supports aliasing between \( \text{rop} \) and \( \text{op} \).

```c
void fq_nmod_poly_shift_left(fq_nmod_poly_t rop, const fq_nmod_poly_t op, slong len, const fq_nmod_ctx_t ctx)
```
Sets \( \text{rop} \) to \( \text{op} \) shifted left by \( n \) coeff. Zero coefficients are inserted.

```c
void _fq_nmod_poly_shift_right(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, slong n, const fq_nmod_ctx_t ctx)
```
Sets \( (\text{rop}, \text{len} - n) \) to \( (\text{op}, \text{len}) \) shifted right by \( n \) coefficients.

Assumes that \( \text{len} \) and \( n \) are positive, that \( \text{len} > n \), and that \( \text{rop} \) fits \( \text{len} - n \) elements. Supports aliasing between \( \text{rop} \) and \( \text{op} \), although in this case the top coefficients of \( \text{op} \) are not set to zero.

```c
void fq_nmod_poly_shift_right(fq_nmod_poly_t rop, const fq_nmod_poly_t op, slong n, const fq_nmod_ctx_t ctx)
```
Sets \( \text{rop} \) to \( \text{op} \) shifted right by \( n \) coefficients. If \( n \) is equal to or greater than the current length of \( \text{op} \), \( \text{rop} \) is set to the zero polynomial.
11.14.14 Norms

\texttt{slong \_fq\_nmod\_poly\_hamming\_weight(const \textit{fq\_nmod\_struct} \*op, \textit{slong} len, const \textit{fq\_nmod\_ctx\_t} ctx)}

Returns the number of non-zero entries in (\textit{op}, \textit{len}).

\texttt{slong \textit{fq\_nmod\_poly\_hamming\_weight(const \textit{fq\_nmod\_poly\_t} op, const \textit{fq\_nmod\_ctx\_t} ctx)}}

Returns the number of non-zero entries in the polynomial \textit{op}.

11.14.15 Euclidean division

\texttt{void \_fq\_nmod\_poly\_divrem}\(\texttt{(fq\_nmod\_struct} \*Q, \texttt{fq\_nmod\_struct} \*R, \texttt{const \textit{fq\_nmod\_struct}} \*A, \textit{slong} lenA, \texttt{const \textit{fq\_nmod\_struct}} \*B, \textit{slong} lenB, \texttt{const \textit{fq\_nmod\_t}} invB, \texttt{const \textit{fq\_nmod\_ctx\_t} ctx})\)

Computes (\textit{Q}, \textit{lenA} - \textit{lenB} + 1), (\textit{R}, \textit{lenA}) such that \(A = BQ + R\) with \(0 \leq \text{len}(R) < \text{len}(B)\).

Assumes that the leading coefficient of \(B\) is invertible and that \textit{invB} is its inverse.

Assumes that \textit{len}(\textit{A}), \textit{len}(\textit{B}) > 0. Allows zero-padding in (\textit{A}, \textit{lenA}). \textit{R} and \textit{A} may be aliased, but apart from this no aliasing of input and output operands is allowed.

\texttt{void \textit{fq\_nmod\_poly\_divrem(const \textit{fq\_nmod\_poly\_t} A, const \textit{fq\_nmod\_poly\_t} B, const \textit{fq\_nmod\_ctx\_t} ctx)}}

Computes \textit{Q}, \textit{R} such that \(A = BQ + R\) with \(0 \leq \text{len}(R) < \text{len}(B)\).

Assumes that the leading coefficient of \(B\) is invertible. This can be taken for granted the context is for a finite field, that is, when \(p\) is prime and \(f(X)\) is irreducible.

\texttt{void \textit{fq\_nmod\_poly\_divrem\_f(const \textit{fq\_nmod\_t} f, const \textit{fq\_nmod\_poly\_t} A, const \textit{fq\_nmod\_poly\_t} B, const \textit{fq\_nmod\_ctx\_t} ctx)}}

Either finds a non-trivial factor \textit{f} of the modulus of \textit{ctx}, or computes \textit{Q}, \textit{R} such that \(A = BQ + R\) and \(0 \leq \text{len}(R) < \text{len}(B)\).

If the leading coefficient of \(B\) is invertible, the division with remainder operation is carried out, \textit{Q} and \textit{R} are computed correctly, and \textit{f} is set to 1. Otherwise, \textit{f} is set to a non-trivial factor of the modulus and \textit{Q} and \textit{R} are not touched.

Assumes that \(B\) is non-zero.

\texttt{void \_fq\_nmod\_poly\_rem(const \textit{fq\_nmod\_struct} \*A, const \textit{fq\_nmod\_struct} \*B, \textit{slong} lenA, \textit{slong} lenB, \texttt{const \textit{fq\_nmod\_t} invB, const \textit{fq\_nmod\_ctx\_t} ctx})}

Sets \textit{R} to the remainder of the division of (\textit{A}, \textit{lenA}) by (\textit{B}, \textit{lenB}). Assumes that the leading coefficient of (\textit{B}, \textit{lenB}) is invertible and that \textit{invB} is its inverse.

\texttt{void \textit{fq\_nmod\_poly\_rem(const \textit{fq\_nmod\_poly\_t} A, const \textit{fq\_nmod\_poly\_t} B, const \textit{fq\_nmod\_ctx\_t} ctx)}}

Sets \textit{R} to the remainder of the division of \textit{A} by \textit{B} in the context described by \textit{ctx}.

\texttt{void \_fq\_nmod\_poly\_div(const \textit{fq\_nmod\_struct} \*Q, const \textit{fq\_nmod\_struct} \*A, const \textit{fq\_nmod\_struct} \*B, \textit{slong} lenA, \textit{slong} lenB, \texttt{const \textit{fq\_nmod\_t} invB, const \textit{fq\_nmod\_ctx\_t} ctx})}

Notationally, computes \textit{Q}, \textit{R} such that \(A = BQ + R\) with \(0 \leq \text{len}(R) < \text{len}(B)\) but only sets (\textit{Q}, \textit{lenA} - \textit{lenB} + 1).

Allows zero-padding in \textit{A} but not in \textit{B}. Assumes that the leading coefficient of \(B\) is a unit.
void fq_nmod_poly_div(fq_nmod_poly_t Q, const fq_nmod_poly_t A, const fq_nmod_poly_t B, const fq_nmod_ctx_t ctx)

Notionally finds polynomials Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q. If $\text{len}(B) = 0$ an exception is raised.

void _fq_nmod_poly_div_newton_n_preinv(fq_nmod_struct *Q, const fq_nmod_struct *A, slong lenA, const fq_nmod_struct *B, slong lenB, const fq_nmod_struct *Binv, slong lenBinv, const fq_nmod_ctx_t ctx)

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, where A is of length lenA and B is of length lenB, but return only Q.

We require that Q have space for lenA - lenB + 1 coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that Binv is the inverse of the reverse of B modulo $x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

void fq_nmod_poly_divrem_newton_n_preinv(fq_nmod_poly_t Q, const fq_nmod_poly_t A, const fq_nmod_poly_t B, const fq_nmod_poly_t Binv, const fq_nmod_ctx_t ctx)

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q.

We assume that the leading coefficient of B is a unit and that Binv is the inverse of the reverse of B modulo $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

void _fq_nmod_poly_divrem_newton_n_preinv(fq_nmod_struct *Q, fq_nmod_struct *R, const fq_nmod_struct *A, slong lenA, const fq_nmod_struct *B, slong lenB, const fq_nmod_struct *Binv, slong lenBinv, const fq_nmod_ctx_t ctx)

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, where A is of length lenA and B is of length lenB. We require that Q have space for lenA - lenB + 1 coefficients. Furthermore, we assume that Binv is the inverse of the reverse of B modulo $x^{\text{len}(B)}$. The algorithm used is to call div_newton_preinv() and then multiply out and compute the remainder.

void fq_nmod_poly_divrem_newton_n_preinv(fq_nmod_poly_t Q, fq_nmod_poly_t R, const fq_nmod_poly_t A, const fq_nmod_poly_t B, const fq_nmod_poly_t Binv, const fq_nmod_ctx_t ctx)

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume Binv is the inverse of the reverse of B modulo $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to call div_newton() and then multiply out and compute the remainder.

void _fq_nmod_poly_inv_series_newton(fq_nmod_struct *Qinv, const fq_nmod_struct *Q, slong n, const fq_nmod_t cinv, const fq_nmod_ctx_t ctx)

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial Qinv of length n such that $Q \cdot Qinv$ is 1 modulo $x^n$. Requires n > 0. This function can be viewed as inverting a power series via Newton iteration.

void fq_nmod_poly_inv_series_newton(fq_nmod_poly_t Qinv, const fq_nmod_poly_t Q, slong n, const fq_nmod_ctx_t ctx)

Given Q find Qinv such that $Q \cdot Qinv$ is 1 modulo $x^n$. The constant coefficient of Q must be invertible modulo the modulus of Q. An exception is raised if this is not the case or if n = 0. This function can be viewed as inverting a power series via Newton iteration.
void fq_nmod_poly_inv_series(fq_nmod_struct *Qinv, const fq_nmod_struct *Q, slong n, const fq_nmod_ctx_t ctx)

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial Qinv of length n such that Q * Qinv is 1 modulo \( x^n \). Requires \( n > 0 \).

void fq_nmod_poly_inv_series(fq_nmod_poly_t Qinv, const fq_nmod_poly_t Q, slong n, const fq_nmod_ctx_t ctx)

Given Q find Qinv such that Q * Qinv is 1 modulo \( x^n \). The constant coefficient of Q must be invertible modulo the modulus of Q. An exception is raised if this is not the case or if \( n = 0 \).

void fq_nmod_poly_div_series(fq_nmod_struct *Q, const fq_nmod_struct *A, mp_limb_signed_t Alen, const fq_nmod_struct *B, mp_limb_signed_t Blen, mp_limb_signed_t n, const fq_nmod_ctx_t ctx)

Set \((Q, n)\) to the quotient of the series \((A, Alen)\) and \((B, Blen)\) assuming \( Alen, Blen <= n \). We assume the bottom coefficient of B is invertible.

void fq_nmod_poly_div_series(fq_nmod_poly_t Q, const fq_nmod_poly_t A, const fq_nmod_poly_t B, slong n, fq_nmod_ctx_t ctx)

Set \(Q\) to the quotient of the series \(A\) by \(B\), thinking of the series as though they were of length \(n\). We assume that the bottom coefficient of \(B\) is invertible.

11.14.16 Greatest common divisor

void fq_nmod_poly_gcd(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)

Sets rop to the greatest common divisor of op1 and op2, using either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial \(P\) is defined to be \(P\). Except in the case where the GCD is zero, the GCD \(G\) is made monic.

slong fq_nmod_poly_gcd(fq_nmod_struct *G, const fq_nmod_struct *A, slong lenA, const fq_nmod_struct *B, slong lenB, const fq_nmod_ctx_t ctx)

Computes the GCD of \(A\) of length \(lenA\) and \(B\) of length \(lenB\), where \(lenA >= lenB > 0\) and sets \(G\) to it. The length of the GCD \(G\) is returned by the function. No attempt is made to make the GCD monic. It is required that \(G\) have space for \(lenB\) coefficients.

slong fq_nmod_poly_gcd_euclidean_f(fq_nmod_t f, fq_nmod_struct *G, const fq_nmod_struct *A, slong lenA, const fq_nmod_struct *B, slong lenB, const fq_nmod_ctx_t ctx)

Either sets \(f = 1\) and \(G\) to the greatest common divisor of \((A, len(A))\) and \((B, len(B))\) and returns its length, or sets \(f\) to a non-trivial factor of the modulus of \(ctx\) and leaves the contents of the vector \((G, lenB)\) undefined.

Assumes that \(len(A) >= len(B) > 0\) and that the vector \(G\) has space for sufficiently many coefficients.

void fq_nmod_poly_gcd_euclidean_f(fq_nmod_t f, fq_nmod_poly_t G, const fq_nmod_poly_t A, const fq_nmod_poly_t B, const fq_nmod_ctx_t ctx)

Either sets \(f = 1\) and \(G\) to the greatest common divisor of \(A\) and \(B\) or sets \(f\) to a factor of the modulus of \(ctx\).

slong fq_nmod_poly_xgcd(fq_nmod_struct *G, fq_nmod_struct *S, fq_nmod_struct *T, const fq_nmod_struct *A, slong lenA, const fq_nmod_struct *B, slong lenB, const fq_nmod_ctx_t ctx)

Computes the GCD of \(A\) and \(B\) together with cofactors \(S\) and \(T\) such that \(SA + TB = G\). Returns the length of \(G\).

Assumes that \(len(A) >= len(B) >= 1\) and \((len(A), len(B)) \neq (1, 1)\).

No attempt is made to make the GCD monic.
Requires that $G$ have space for $\text{len}(B)$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to $S$ and $T$, respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```c
void fq_nmod_poly_xgcd(fq_nmod_poly_t G, fq_nmod_poly_t S, fq_nmod_poly_t T, const fq_nmod_poly_t A, const fq_nmod_poly_t B, const fq_nmod_ctx_t ctx)
```

Computes the GCD of $A$ and $B$. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial $P$ is defined to be $P$. Except in the case where the GCD is zero, the GCD $G$ is made monic.

Polynomials $S$ and $T$ are computed such that $S*A + T*B = G$. The length of $S$ will be at most $\text{len}B$ and the length of $T$ will be at most $\text{len}A$.

```c
slong fq_nmod_poly_xgcd_euclidean_f(fq_nmod_t f, fq_nmod_struct *G, fq_nmod_struct *S, fq_nmod_struct *T, const fq_nmod_struct *A, slong lenA, const fq_nmod_struct *B, slong lenB, const fq_nmod_ctx_t ctx)
```

Either sets $f = 1$ and computes the GCD of $A$ and $B$ together with cofactors $S$ and $T$ such that $SA + TB = G$; otherwise, sets $f$ to a non-trivial factor of the modulus of $\text{ctx}$ and leaves $G$, $S$, and $T$ undefined. Returns the length of $G$.

Assumes that $\text{len}(A) \geq \text{len}(B) \geq 1$ and $(\text{len}(A), \text{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that $G$ have space for $\text{len}(B)$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to $S$ and $T$, respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```c
void fq_nmod_poly_xgcd_euclidean_f(fq_nmod_t f, fq_nmod_poly_t G, fq_nmod_poly_t S, fq_nmod_poly_t T, const fq_nmod_poly_t A, const fq_nmod_poly_t B, const fq_nmod_ctx_t ctx)
```

Either sets $f = 1$ and computes the GCD of $A$ and $B$ or sets $f$ to a non-trivial factor of the modulus of $\text{ctx}$.

If the GCD is computed, polynomials $S$ and $T$ are computed such that $S*A + T*B = G$; otherwise, they are undefined. The length of $S$ will be at most $\text{len}B$ and the length of $T$ will be at most $\text{len}A$.

The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial $P$ is defined to be $P$. Except in the case where the GCD is zero, the GCD $G$ is made monic.

### 11.14.17 Divisibility testing

```c
int _fq_nmod_poly_divides(fq_nmod_struct *Q, const fq_nmod_struct *A, slong lenA, const fq_nmod_struct *B, slong lenB, const fq_nmod_ctx_t ctx)
```

Returns 1 if $(B, \text{len}B)$ divides $(A, \text{len}A)$ exactly and sets $Q$ to the quotient, otherwise returns 0.

It is assumed that $\text{len}(A) \geq \text{len}(B) > 0$ and that $Q$ has space for $\text{len}(A) - \text{len}(B) + 1$ coefficients.

Aliasing of $Q$ with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```c
int fq_nmod_poly_divides(fq_nmod_poly_t Q, const fq_nmod_poly_t A, const fq_nmod_poly_t B, const fq_nmod_ctx_t ctx)
```
Returns 1 if $B$ divides $A$ exactly and sets $Q$ to the quotient, otherwise returns 0.
This function is currently unoptimised and provided for convenience only.

### 11.14.18 Derivative

```c
void _fq_nmod_poly_derivative(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
```

Sets $(rop, len - 1)$ to the derivative of $(op, len)$. Also handles the cases where $len$ is 0 or 1 correctly. Supports aliasing of rop and op.

```c
void fq_nmod_poly_derivative(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Sets rop to the derivative of op.

### 11.14.19 Square root

```c
void _fq_nmod_poly_invsqrt_series(fq_nmod_struct *g, const fq_nmod_struct *h, slong n, const fq_nmod_ctx_t ctx)
```

Set the first $n$ terms of $g$ to the series expansion of $1/\sqrt{h}$. It is assumed that $n > 0$, that $h$ has constant term 1 and that $h$ is zero-padded as necessary to length $n$. Aliasing is not permitted.

```c
void fq_nmod_poly_invsqrt_series(fq_nmod_poly_t g, const fq_nmod_poly_t h, slong n, const fq_nmod_ctx_t ctx)
```

Set $g$ to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that $h$ has constant term 1.

```c
void _fq_nmod_poly_sqrt_series(fq_nmod_struct *g, const fq_nmod_struct *h, slong n, const fq_nmod_ctx_t ctx)
```

Set the first $n$ terms of $g$ to the series expansion of $\sqrt{h}$. It is assumed that $n > 0$, that $h$ has constant term 1 and that $h$ is zero-padded as necessary to length $n$. Aliasing is not permitted.

```c
void fq_nmod_poly_sqrt_series(fq_nmod_poly_t g, const fq_nmod_poly_t h, slong n, const fq_nmod_ctx_t ctx)
```

Set $g$ to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that $h$ has constant term 1.

```c
int _fq_nmod_poly_sqrt(fq_nmod_struct *s, const fq_nmod_struct *p, slong n, const fq_nmod_ctx_t mod)
```

If $(p, n)$ is a perfect square, sets $(s, n / 2 + 1)$ to a square root of $p$ and returns 1. Otherwise returns 0.

```c
int fq_nmod_poly_sqrt(fq_nmod_poly_t s, const fq_nmod_poly_t p, const fq_nmod_ctx_t mod)
```

If $p$ is a perfect square, sets $s$ to a square root of $p$ and returns 1. Otherwise returns 0.

### 11.14.20 Evaluation

```c
void _fq_nmod_poly_evaluate_fq_nmod(fq_nmod_t rop, const fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
```

Sets rop to $(op, len)$ evaluated at $a$.

Supports zero padding. There are no restrictions on $len$, that is, $len$ is allowed to be zero, too.

```c
void fq_nmod_poly_evaluate_fq_nmod(fq_nmod_t rop, const fq_nmod_poly_t f, const fq_nmod_t a, const fq_nmod_ctx_t ctx)
```

Sets rop to the value of $f(a)$.

As the coefficient ring $\mathbb{F}_q$ is finite, Horner’s method is sufficient.
11.14.21 Composition

void _fq_nmod_poly_compose(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)

Sets rop to the composition of (op1, len1) and (op2, len2).

Assumes that rop has space for (len1-1)*(len2-1) + 1 coefficients. Assumes that op1 and op2 are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

void fq_nmod_poly_compose(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)

Sets rop to the composition of op1 and op2. To be precise about the order of composition, denoting rop, op1, and op2 by \(f\), \(g\), and \(h\), respectively, sets \(f(t) = g(h(t))\).

void _fq_nmod_poly_compose_mod_horner(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf, const fq_nmod_struct *g, const fq_nmod_struct *h, slong lenh, const fq_nmod_ctx_t ctx)

Sets res to the composition \(f(g)\) modulo \(h\). We require that \(h\) is nonzero and that the length of \(g\) is one less than the length of \(h\) (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner’s rule.

void fq_nmod_poly_compose_mod_horner(fq_nmod_poly_t res, const fq_nmod_poly_t f, const fq_nmod_poly_t g, const fq_nmod_poly_t h, const fq_nmod_ctx_t ctx)

Sets res to the composition \(f(g)\) modulo \(h\). We require that \(h\) is nonzero. The algorithm used is Horner’s rule.

void _fq_nmod_poly_compose_mod_horner_preinv(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf, const fq_nmod_struct *g, const fq_nmod_struct *h, slong lenh, const fq_nmod_struct *hinv, slong lenhinv, const fq_nmod_ctx_t ctx)

Sets res to the composition \(f(g)\) modulo \(h\). We require that \(h\) is nonzero and that the length of \(f\) is less than the length of \(h\). Furthermore, we require \(hinv\) to be the inverse of the reverse of \(h\). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner’s rule.

void fq_nmod_poly_compose_mod_horner_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t f, const fq_nmod_poly_t g, const fq_nmod_poly_t hinv, const fq_nmod_ctx_t ctx)

Sets res to the composition \(f(g)\) modulo \(h\). We require that \(h\) is nonzero and that \(f\) has smaller degree than \(h\). Furthermore, we require \(hinv\) to be the inverse of the reverse of \(h\). The algorithm used is Horner’s rule.

void _fq_nmod_poly_compose_mod_brent_kung(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf, const fq_nmod_struct *g, const fq_nmod_struct *h, slong lenh, const fq_nmod_ctx_t ctx)

Sets res to the composition \(f(g)\) modulo \(h\). We require that \(h\) is nonzero and that the length of \(g\) is one less than the length of \(h\) (possibly with zero padding). We also require that the length of \(f\) is less than the length of \(h\). The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

void fq_nmod_poly_compose_mod_brent_kung(fq_nmod_poly_t res, const fq_nmod_poly_t f, const fq_nmod_poly_t g, const fq_nmod_poly_t h, const fq_nmod_ctx_t ctx)
Sets \( \text{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). The algorithm used is the Brent-Kung matrix algorithm.

```c
void _fq_nmod_poly_compose_mod_brent_kung_preinv(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf, const fq_nmod_struct *g, const fq_nmod_struct *h, slong lenh, const fq_nmod_struct *hinv, slong lenhinv, const fq_nmod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). We also require that \( f \) is less than the length of \( h \). Furthermore, we require \( \text{hinv} \) to be the inverse of the reverse of \( h \). The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```c
void fq_nmod_poly_compose_mod_brent_kung_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t f, const fq_nmod_poly_t g, const fq_nmod_poly_t h, const fq_nmod_poly_t hinv, const fq_nmod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). Furthermore, we require \( \text{hinv} \) to be the inverse of the reverse of \( h \). The algorithm used is the Brent-Kung matrix algorithm.

```c
void _fq_nmod_poly_compose_mod(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf, const fq_nmod_struct *g, const fq_nmod_struct *h, slong lenh, const fq_nmod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( f \) (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```c
void fq_nmod_poly_compose_mod(fq_nmod_poly_t res, const fq_nmod_poly_t f, const fq_nmod_poly_t g, const fq_nmod_poly_t h, const fq_nmod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero.

```c
void _fq_nmod_poly_compose_mod_preinv(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf, const fq_nmod_struct *g, const fq_nmod_struct *h, slong lenh, const fq_nmod_struct *hinv, slong lenhinv, const fq_nmod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( f \) (possibly with zero padding). We also require that the length of \( f \) is less than the length of \( h \). Furthermore, we require \( \text{hinv} \) to be the inverse of the reverse of \( h \). The output is not allowed to be aliased with any of the inputs.

```c
void fq_nmod_poly_compose_mod_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t f, const fq_nmod_poly_t g, const fq_nmod_poly_t h, const fq_nmod_poly_t hinv, const fq_nmod_ctx_t ctx)
```

Sets \( \text{res} \) to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). Furthermore, we require \( \text{hinv} \) to be the inverse of the reverse of \( h \).

```c
void _fq_nmod_poly_reduce_matrix_mod_poly(fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)
```

Sets the \( i \)th row of \( A \) to the reduction of the \( i \)th row of \( B \) modulo \( f \) for \( i = 1, \ldots, \sqrt{\deg(f)} \). We require \( B \) to be at least \( \sqrt{\deg(f)} \times \deg(f) \) matrix and \( f \) to be nonzero.

```c
void _fq_nmod_poly_precompute_matrix(fq_nmod_mat_t A, const fq_nmod_struct *f, const fq_nmod_struct *g, slong leng, const fq_nmod_struct *ginv, slong lenglinv, const fq_nmod_ctx_t ctx)
```
Sets the $i$th row of $A$ to $f^i$ modulo $g$ for $i = 1, \ldots, \sqrt{\deg(g)}$. We require $A$ to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require $\text{ginv}$ to be the inverse of the reverse of $g$ and $g$ to be nonzero.

```c
void fq_nmod_poly_precompute_matrix(fq_nmod_mat_t A, const fq_nmod_poly_t f, const fq_nmod_poly_t g, const fq_nmod_poly_t ginv, const fq_nmod_ctx_t ctx)
```

Sets the $i$th row of $A$ to $f^i$ modulo $g$ for $i = 1, \ldots, \sqrt{\deg(g)}$. We require $A$ to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require $\text{ginv}$ to be the inverse of the reverse of $g$.

```c
void _fq_nmod_poly_compose_mod_brent_kung_precomp_preinv(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf, const fq_nmod_mat_t A, const fq_nmod_struct *h, slong lenh, const fq_nmod_struct *hinv, slong lenhinv, const fq_nmod_ctx_t ctx)
```

Sets $\text{res}$ to the composition $f(g)$ modulo $h$. We require that $h$ is nonzero. We require that the $i$th row of $A$ contains $g^i$ for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. $A$ is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of $f$ is less than the length of $h$. Furthermore, we require $\text{hinv}$ to be the inverse of the reverse of $h$. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```c
void fq_nmod_poly_compose_mod_brent_kung_precomp_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t f, const fq_nmod_mat_t A, const fq_nmod_poly_t h, const fq_nmod_poly_t hinv, const fq_nmod_ctx_t ctx)
```

Sets $\text{res}$ to the composition $f(g)$ modulo $h$. We require that the $i$th row of $A$ contains $g^i$ for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. $A$ is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that $h$ has smaller degree than $h$. Furthermore, we require $\text{hinv}$ to be the inverse of the reverse of $h$. This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo $h$ for fixed $g$ and $h$.

### 11.14.22 Output

```c
int _fq_nmod_poly_fprint_pretty(FILE *file, const fq_nmod_struct *poly, slong len, const char *x, const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of $(\text{poly}, \text{len})$ to the stream $\text{file}$, using the string $x$ to represent the indeterminate. In case of success, returns a positive value. In case of failure, returns a non-positive value.

```c
int fq_nmod_poly_fprint_pretty(FILE *file, const fq_nmod_poly_t poly, const char *x, const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of $\text{poly}$ to the stream $\text{file}$, using the string $x$ to represent the indeterminate. In case of success, returns a positive value. In case of failure, returns a non-positive value.

```c
int _fq_nmod_poly_print_pretty(const fq_nmod_struct *poly, slong len, const char *x, const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of $(\text{poly}, \text{len})$ to $\text{stdout}$, using the string $x$ to represent the indeterminate. In case of success, returns a positive value. In case of failure, returns a non-positive value.
int fq_nmod_poly_print_pretty(const fq_nmod_poly_t poly, const char *x, const fq_nmod_ctx_t ctx)

Prints the pretty representation of poly to stdout, using the string x to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_nmod_poly_fprint(FILE *file, const fq_nmod_struct *poly, slong len, const fq_nmod_ctx_t ctx)

Prints the pretty representation of (poly, len) to the stream file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_nmod_poly_fprint(FILE *file, const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)

Prints the pretty representation of poly to the stream file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fq_nmod_poly_print(const fq_nmod_struct *poly, slong len, const fq_nmod_ctx_t ctx)

Prints the pretty representation of (poly, len) to stdout.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_nmod_poly_print(const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)

Prints the representation of poly to stdout.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

char *fq_nmod_poly_get_str(const fq_nmod_struct *poly, slong len, const fq_nmod_ctx_t ctx)

Returns the plain FLINT string representation of the polynomial (poly, len).

char *fq_nmod_poly_get_str(const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)

Returns the plain FLINT string representation of the polynomial poly.

char *fq_nmod_poly_get_str_pretty(const fq_nmod_struct *poly, slong len, const char *x, const fq_nmod_ctx_t ctx)

Returns a pretty representation of the polynomial (poly, len) using the null-terminated string x as the variable name.

char *fq_nmod_poly_get_str_pretty(const fq_nmod_poly_t poly, const char *x, const fq_nmod_ctx_t ctx)

Returns a pretty representation of the polynomial poly using the null-terminated string x as the variable name.

11.14.23 Inflation and deflation

void fq_nmod_poly_inflate(fq_nmod_poly_t result, const fq_nmod_poly_t input, ulong inflation, const fq_nmod_ctx_t ctx)

Sets result to the inflated polynomial \( p(x^n) \) where \( p \) is given by input and \( n \) is given by inflation.

void fq_nmod_poly_deflate(fq_nmod_poly_t result, const fq_nmod_poly_t input, ulong deflation, const fq_nmod_ctx_t ctx)

Sets result to the deflated polynomial \( p(x^{1/n}) \) where \( p \) is given by input and \( n \) is given by deflation. Requires \( n > 0 \).

ulong fq_nmod_poly_deflation(const fq_nmod_poly_t input, const fq_nmod_ctx_t ctx)

Returns the largest integer by which input can be deflated. As special cases, returns 0 if input is the zero polynomial and 1 of input is a constant polynomial.
11.15 fq_nmod_poly_factor.h – factorisation of univariate polynomials over finite fields (word-size characteristic)

11.15.1 Types, macros and constants

type fq_nmod_poly_factor_struct

type fq_nmod_poly_factor_t

11.15.2 Memory Management

void fq_nmod_poly_factor_init(fq_nmod_poly_factor_t fac, const fq_nmod_ctx_t ctx)
    Initialises fac for use. An fq_nmod_poly_factor_t represents a polynomial in factorised form as a product of polynomials with associated exponents.

void fq_nmod_poly_factor_clear(fq_nmod_poly_factor_t fac, const fq_nmod_ctx_t ctx)
    Frees all memory associated with fac.

void fq_nmod_poly_factor_realloc(fq_nmod_poly_factor_t fac, slong alloc, const fq_nmod_ctx_t ctx)
    Reallocates the factor structure to provide space for precisely alloc factors.

void fq_nmod_poly_factor_fit_length(fq_nmod_poly_factor_t fac, slong len, const fq_nmod_ctx_t ctx)
    Ensures that the factor structure has space for at least len factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

11.15.3 Basic Operations

void fq_nmod_poly_factor_set(fq_nmod_poly_factor_t res, const fq_nmod_poly_factor_t fac, const fq_nmod_ctx_t ctx)
    Sets res to the same factorisation as fac.

void fq_nmod_poly_factor_print_pretty(const fq_nmod_poly_factor_t fac, const char *var, const fq_nmod_ctx_t ctx)
    Pretty-prints the entries of fac to standard output.

void fq_nmod_poly_factor_print(const fq_nmod_poly_factor_t fac, const fq_nmod_ctx_t ctx)
    Prints the entries of fac to standard output.

void fq_nmod_poly_factor_insert(fq_nmod_poly_factor_t fac, const fq_nmod_poly_t poly, slong exp, const fq_nmod_ctx_t ctx)
    Inserts the factor poly with multiplicity exp into the factorisation fac.

    If fac already contains poly, then exp simply gets added to the exponent of the existing entry.

void fq_nmod_poly_factor_concat(fq_nmod_poly_factor_t res, const fq_nmod_poly_factor_t fac, const fq_nmod_ctx_t ctx)
    Concatenates two factorisations.

    This is equivalent to calling fq_nmod_poly_factor_insert() repeatedly with the individual factors of fac.

    Does not support aliasing between res and fac.

void fq_nmod_poly_factor_pow(fq_nmod_poly_factor_t fac, slong exp, const fq_nmod_ctx_t ctx)
    Raises fac to the power exp.
ulong fq_nmod_poly_remove(fq_nmod_poly_t f, const fq_nmod_poly_t p, const fq_nmod_ctx_t ctx)

Removes the highest possible power of p from f and returns the exponent.

11.15.4 Irreducibility Testing

int fq_nmod_poly_is_irreducible(const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Returns 1 if the polynomial f is irreducible, otherwise returns 0.

int fq_nmod_poly_is_irreducible_ddf(const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

int fq_nmod_poly_is_irreducible_ben_or(const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses Ben-Or’s irreducibility test.

int _fq_nmod_poly_is_squarefree(const fq_nmod_struct *f, slong len, const fq_nmod_ctx_t ctx)

Returns 1 if (f, len) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

int fq_nmod_poly_is_squarefree(const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Returns 1 if f is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

11.15.5 Factorisation

int fq_nmod_poly_factor_equal_deg_prob(fq_nmod_poly_t factor, flint_rand_t state, const fq_nmod_poly_t pol, slong d, const fq_nmod_ctx_t ctx)

Probabilistic equal degree factorisation of pol into irreducible factors of degree d. If it passes, a factor is placed in factor and 1 is returned, otherwise 0 is returned and the value of factor is undetermined.

Requires that pol be monic, non-constant and squarefree.

void fq_nmod_poly_factor_equal_deg(fq_nmod_poly_t_factor, flint_rand_t state, const fq_nmod_poly_t pol, slong d, const fq_nmod_ctx_t ctx)

Assuming pol is a product of irreducible factors all of degree d, finds all those factors and places them in factors. Requires that pol be monic, non-constant and squarefree.

void fq_nmod_poly_factor_split_single(fq_nmod_poly_t linfactor, const fq_nmod_poly_t input, const fq_nmod_ctx_t ctx)

Assuming input is a product of factors all of degree 1, finds a single linear factor of input and places it in linfactor. Requires that input be monic and non-constant.

void fq_nmod_poly_factor_distinct_deg(fq_nmod_poly_factor_t res, const fq_nmod_poly_t poly, slong *const *degs, const fq_nmod_ctx_t ctx)

Factorises a monic non-constant squarefree polynomial poly of degree n into factors \( f[d] \) such that for \( 1 \leq d \leq n \) \( f[d] \) is the product of the monic irreducible factors of poly of degree d. Factors are stored in res, associated powers of irreducible polynomials are stored in degs in the same order as factors.

Requires that degs have enough space for irreducible polynomials’ powers (maximum space required is \( n * \text{sizeof(slong)} \)).

void fq_nmod_poly_factor_squarefree(fq_nmod_poly_factor_t res, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Sets res to a squarefree factorization of f.
void fq_nmod_poly_factor(fq_nmod_poly_factor_t res, fq_nmod_t lead, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Factorises a non-constant polynomial \( f \) into monic irreducible factors choosing the best algorithm for given modulo and degree. The output lead is set to the leading coefficient of \( f \) upon return. Choice of algorithm is based on heuristic measurements.

void fq_nmod_poly_factor_cantor_zassenhaus(fq_nmod_poly_factor_t res, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Factorises a non-constant polynomial \( f \) into monic irreducible factors using the Cantor-Zassenhaus algorithm.

void fq_nmod_poly_factor_kaltofen_shoup(fq_nmod_poly_factor_t res, const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)

Factorises a non-constant polynomial \( f \) into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a “baby step/giant step” strategy for the distinct-degree factorization step.

void fq_nmod_poly_factor_berlekamp(fq_nmod_poly_factor_t factors, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Factorises a non-constant polynomial \( f \) into monic irreducible factors using the Berlekamp algorithm.

void fq_nmod_poly_factor_with_berlekamp(fq_nmod_poly_factor_t res, fq_nmod_t leading_coeff, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Factorises a general polynomial \( f \) into monic irreducible factors and sets leading_coeff to the leading coefficient of \( f \), or 0 if \( f \) is the zero polynomial.

This function first checks for small special cases, deflates \( f \) if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs Berlekamp on all the individual square-free factors.

void fq_nmod_poly_factor_with_cantor_zassenhaus(fq_nmod_poly_factor_t res, fq_nmod_t leading_coeff, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Factorises a general polynomial \( f \) into monic irreducible factors and sets leading_coeff to the leading coefficient of \( f \), or 0 if \( f \) is the zero polynomial.

This function first checks for small special cases, deflates \( f \) if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

void fq_nmod_poly_factor_with_kaltofen_shoup(fq_nmod_poly_factor_t res, fq_nmod_t leading_coeff, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)

Factorises a general polynomial \( f \) into monic irreducible factors and sets leading_coeff to the leading coefficient of \( f \), or 0 if \( f \) is the zero polynomial.

This function first checks for small special cases, deflates \( f \) if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

void fq_nmod_poly_iterated_frobenius_preinv(fq_nmod_poly_t *rop, slong n, const fq_nmod_poly_t v, const fq_nmod_poly_t vinv, const fq_nmod_ctx_t ctx)

Sets \( rop[1] \) to be \( x^d \) mod \( v \) for \( 0 \leq i < n \).

It is required that \( vinv \) is the inverse of the reverse of \( v \) mod \( x^{\text{lenv}} \).
11.15.6 Root Finding

```c
void fq_nmod_poly_roots(fq_nmod_poly_factor_t r, const fq_nmod_poly_t f, int with_multiplicity, const fq_nmod_ctx_t ctx)
```

Fill \( r \) with factors of the form \( x - r_i \) where the \( r_i \) are the distinct roots of a nonzero \( f \) in \( \mathbb{F}_q \). If \( \text{with\_multiplicity} \) is zero, the exponent \( e_i \) of the factor \( x - r_i \) is 1. Otherwise, it is the largest \( e_i \) such that \( (x - r_i)^e_i \) divides \( f \). This function throws if \( f \) is zero, but is otherwise always successful.

11.16 fq_nmod_embed.h – Computing isomorphisms and embeddings of finite fields

```c
void fq_nmod_embed_gens(fq_nmod_t gen_sub, fq_nmod_t gen_sup, nmod_poly_t minpoly, const fq_nmod_ctx_t sub_ctx, const fq_nmod_ctx_t sup_ctx)
```

Given two contexts \( \text{sub\_ctx} \) and \( \text{sup\_ctx} \), such that \( \text{degree}(\text{sub\_ctx}) \) divides \( \text{degree}(\text{sup\_ctx}) \), compute:

- an element \( \text{gen\_sub} \) in \( \text{sub\_ctx} \) such that \( \text{gen\_sub} \) generates the finite field defined by \( \text{sub\_ctx} \),
- its minimal polynomial \( \text{minpoly} \),
- a root \( \text{gen\_sup} \) of \( \text{minpoly} \) inside the field defined by \( \text{sup\_ctx} \).

These data uniquely define an embedding of \( \text{sub\_ctx} \) into \( \text{sup\_ctx} \).

```c
void _fq_nmod_embed_gens_naive(fq_nmod_t gen_sub, fq_nmod_t gen_sup, nmod_poly_t minpoly, const fq_nmod_ctx_t sub_ctx, const fq_nmod_ctx_t sup_ctx)
```

Given two contexts \( \text{sub\_ctx} \) and \( \text{sup\_ctx} \), such that \( \text{degree}(\text{sub\_ctx}) \) divides \( \text{degree}(\text{sup\_ctx}) \), compute an embedding of \( \text{sub\_ctx} \) into \( \text{sup\_ctx} \) defined as follows:

- \( \text{gen\_sub} \) is the canonical generator of \( \text{sup\_ctx} \) (i.e., the class of \( X \)),
- \( \text{minpoly} \) is the defining polynomial of \( \text{sub\_ctx} \),
- \( \text{gen\_sup} \) is a root of \( \text{minpoly} \) inside the field defined by \( \text{sup\_ctx} \).

```c
void fq_nmod_embed_matrices(nmod_mat_t embed, nmod_mat_t project, const fq_nmod_t gen_sub, const fq_nmod_ctx_t sub_ctx, const fq_nmod_t gen_sup, const fq_nmod_ctx_t sup_ctx, const nmod_poly_t gen_minpoly)
```

Given:

- two contexts \( \text{sub\_ctx} \) and \( \text{sup\_ctx} \), of respective degrees \( m \) and \( n \), such that \( m \) divides \( n \);
- a generator \( \text{gen\_sub} \) of \( \text{sub\_ctx} \), its minimal polynomial \( \text{gen\_minpoly} \), and a root \( \text{gen\_sup} \) of \( \text{gen\_minpoly} \) in \( \text{sup\_ctx} \), as returned by \( \text{fq\_nmod\_embed\_gens} \);

Compute:

- the \( n \times m \) matrix \( \text{embed} \) mapping \( \text{gen\_sub} \) to \( \text{gen\_sup} \), and all their powers accordingly;
- an \( m \times n \) matrix \( \text{project} \) such that \( \text{project} \times \text{embed} \) is the \( m \times m \) identity matrix.

```c
void fq_nmod_embed_trace_matrix(nmod_mat_t res, const nmod_mat_t basis, const fq_nmod_ctx_t sub_ctx, const fq_nmod_ctx_t sup_ctx)
```

Given:

- two contexts \( \text{sub\_ctx} \) and \( \text{sup\_ctx} \), of degrees \( m \) and \( n \), such that \( m \) divides \( n \);
- an \( n \times m \) matrix \( \text{basis} \) that maps \( \text{sub\_ctx} \) to an isomorphic subfield in \( \text{sup\_ctx} \);

Compute the \( m \times n \) matrix of the trace from \( \text{sup\_ctx} \) to \( \text{sub\_ctx} \).

This matrix is computed as
Note: if $m = n$, basis represents a Frobenius, and the result is its inverse matrix.

```c
void fq_nmod_embed_composition_matrix(nmod_mat_t matrix, const fq_nmod_t gen, const fq_nmod_ctx_t ctx)
```
Compute the composition matrix of gen.

For an element $a \in F_p^n$, its composition matrix is the matrix whose columns are $a^0, a^1, \ldots, a^{n-1}$.

```c
void fq_nmod_embed_composition_matrix_sub(nmod_mat_t matrix, const fq_nmod_t gen, const fq_nmod_ctx_t ctx, slong trunc)
```
Compute the composition matrix of gen, truncated to trunc columns.

```c
void fq_nmod_embed_mul_matrix(nmod_mat_t matrix, const fq_nmod_t gen, const fq_nmod_ctx_t ctx)
```
Compute the multiplication matrix of gen.

For an element $a$ in $F_p^n = F_p[x]$, its multiplication matrix is the matrix whose columns are $a, ax, \ldots, ax^{n-1}$.

```c
void fq_nmod_embed_mono_to_dual_matrix(nmod_mat_t res, const fq_nmod_ctx_t ctx)
```
Compute the change of basis matrix from the monomial basis of ctx to its dual basis.

```c
void fq_nmod_embed_dual_to_mono_matrix(nmod_mat_t res, const fq_nmod_ctx_t ctx)
```
Compute the change of basis matrix from the dual basis of ctx to its monomial basis.

```c
void fq_nmod_modulus_pow_series_inv(nmod_poly_t res, const fq_nmod_ctx_t ctx, slong trunc)
```
Compute the power series inverse of the reverse of the modulus of ctx up to $O(x^{\text{trunc}})$.

```c
void fq_nmod_modulus_derivative_inv(fq_nmod_t m_prime, fq_nmod_t m_prime_inv, const fq_nmod_ctx_t ctx)
```
Compute the derivative $m'_\text{prime}$ of the modulus of ctx as an element of ctx, and its inverse $m'_\text{prime}_\text{inv}$.

### 11.17 fq_nmod_mpoly.h – multivariate polynomials over finite fields of word-sized characteristic

The exponents follow the mpoly interface. No references to the coefficients are available.

#### 11.17.1 Types, macros and constants

```c
type fq_nmod_mpoly_struct
```
A structure holding a multivariate polynomial over a finite field of word-sized characteristic.

```c
type fq_nmod_mpoly_t
```
An array of length 1 of fq_nmod_mpoly_struct.

```c
type fq_nmod_mpoly_ctx_struct
```
Context structure representing the parent ring of an fq_nmod_mpoly.

```c
type fq_nmod_mpoly_ctx_t
```
An array of length 1 of fq_nmod_mpoly_ctx_struct.
11.17.2 Context object

```c
void fq_nmod_mpoly_ctx_init(fq_nmod_mpoly_ctx_t ctx, slong nvars, const ordering_t ord, const fq_nmod_mpoly_ctx_t fqctx)
```
Initialise a context object for a polynomial ring with the given number of variables and the given ordering. It will have coefficients in the finite field `fqctx`. The possibilities for the ordering are `ORD_LEX`, `ORD_DEGLEX` and `ORD_DEGREVLEX`.

```c
slong fq_nmod_mpoly_ctx_nvars(const fq_nmod_mpoly_ctx_t ctx)
```
Return the number of variables used to initialize the context.

```c
ordering_t fq_nmod_mpoly_ctx_ord(const fq_nmod_mpoly_ctx_t ctx)
```
Return the ordering used to initialize the context.

```c
void fq_nmod_mpoly_ctx_clear(fq_nmod_mpoly_ctx_t ctx)
```
Release any space allocated by an `ctx`.

11.17.3 Memory management

```c
void fq_nmod_mpoly_init(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```
Initialise `A` for use with the given an initialised context object. Its value is set to zero.

```c
void fq_nmod_mpoly_init2(fq_nmod_mpoly_t A, slong alloc, const fq_nmod_mpoly_ctx_t ctx)
```
Initialise `A` for use with the given an initialised context object. Its value is set to zero. It is allocated with space for `alloc` terms and at least `MPOLY_MIN_BITS` bits for the exponents.

```c
void fq_nmod_mpoly_init3(fq_nmod_mpoly_t A, slong alloc, flint_bitcnt_t bits, const fq_nmod_mpoly_ctx_t ctx)
```
Initialise `A` for use with the given an initialised context object. Its value is set to zero. It is allocated with space for `alloc` terms and `bits` bits for the exponents.

```c
void fq_nmod_mpoly_fit_length(fq_nmod_mpoly_t A, slong len, const fq_nmod_mpoly_ctx_t ctx)
```
Ensure that `A` has space for at least `len` terms.

```c
void fq_nmod_mpoly_realloc(fq_nmod_mpoly_t A, slong alloc, const fq_nmod_mpoly_ctx_t ctx)
```
Reallocate `A` to have space for `alloc` terms. Assumes the current length of the polynomial is not greater than `alloc`.

```c
void fq_nmod_mpoly_clear(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```
Release any space allocated for `A`.

11.17.4 Input/Output

The variable strings in `x` start with the variable of most significance at index 0. If `x` is `NULL`, the variables are named `x1, x2`, etc.

```c
char *fq_nmod_mpoly_get_str_pretty(const fq_nmod_mpoly_t A, const char **x, const fq_nmod_mpoly_ctx_t ctx)
```
Return a string, which the user is responsible for cleaning up, representing `A`, given an array of variable strings `x`.

```c
int fq_nmod_mpoly_fprint_pretty(FILE *file, const fq_nmod_mpoly_t A, const char **x, const fq_nmod_mpoly_ctx_t ctx)
```
Print a string representing `A` to `file`.

```c
int fq_nmod_mpoly_print_pretty(const fq_nmod_mpoly_t A, const char **x, const fq_nmod_mpoly_ctx_t ctx)
```
Print a string representing `A` to `stdout`.
int fq_nmod_mpoly_set_str_pretty(fq_nmod_mpoly_t A, const char *str, const char **x, const fq_nmod_mpoly_ctx_t ctx)

Set A to the polynomial in the null-terminates string str given an array x of variable strings. If parsing str fails, A is set to zero, and −1 is returned. Otherwise, 0 is returned. The operations +, - , * , and / are permitted along with integers and the variables in x. The character ^ must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

11.17.5 Basic manipulation

void fq_nmod_mpoly_gen(fq_nmod_mpoly_t A, slong var, const fq_nmod_mpoly_ctx_t ctx)

Set A to the variable of index var, where var = 0 corresponds to the variable with the most significance with respect to the ordering.

int fq_nmod_mpoly_is_gen(const fq_nmod_mpoly_t A, slong var, const fq_nmod_mpoly_ctx_t ctx)

If var ≥ 0, return 1 if A is equal to the var-th generator, otherwise return 0. If var < 0, return 1 if the polynomial is equal to any generator, otherwise return 0.

void fq_nmod_mpoly_set(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)

Set A to B.

int fq_nmod_mpoly_equal(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)

Return 1 if A is equal to B, else return 0.

void fq_nmod_mpoly_swap(fq_nmod_mpoly_t A, fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)

Efficiently swap A and B.

11.17.6 Constants

int fq_nmod_mpoly_is_fq_nmod(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Return 1 if A is a constant, else return 0.

void fq_nmod_mpoly_get_fq_nmod(fq_nmod_t c, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Assuming that A is a constant, set c to this constant. This function throws if A is not a constant.

void fq_nmod_mpoly_set_fq_nmod(fq_nmod_mpoly_t A, const fq_nmod_t c, const fq_nmod_mpoly_ctx_t ctx)

void fq_nmod_mpoly_set_ui(fq_nmod_mpoly_t A, ulong c, const fq_nmod_mpoly_ctx_t ctx)

Set A to the constant c.

void fq_nmod_mpoly_set_fq_nmod_gen(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Set A to the constant given by fq_nmod_gen().

void fq_nmod_mpoly_zero(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Set A to the constant 0.

void fq_nmod_mpoly_one(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Set A to the constant 1.

int fq_nmod_mpoly_equal_fq_nmod(const fq_nmod_mpoly_t A, const fq_nmod_t c, const fq_nmod_mpoly_ctx_t ctx)

Return 1 if A is equal to the constant c, else return 0.

11.17. fq_nmod_mpoly.h – multivariate polynomials over finite fields of word-sized characteri
int fq_nmod_mpoly_is_zero(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
  Return 1 if A is the constant 0, else return 0.

int fq_nmod_mpoly_is_one(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
  Return 1 if A is the constant 1, else return 0.

11.17.7 Degrees

int fq_nmod_mpoly_degrees_fit_si(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
  Return 1 if the degrees of A with respect to each variable fit into an slong, otherwise return 0.

void fq_nmod_mpoly_degrees_fmpz(fmpz **degs, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
void fq_nmod_mpoly_degrees_si(slong *degs, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
  Set degs to the degrees of A with respect to each variable. If A is zero, all degrees are set to −1.

void fq_nmod_mpoly_degree_fmpz(fmpz_t deg, const fq_nmod_mpoly_t A, slong var, const fq_nmod_mpoly_ctx_t ctx)
slong fq_nmod_mpoly_degree_si(const fq_nmod_mpoly_t A, slong var, const fq_nmod_mpoly_ctx_t ctx)
  Either return or set deg to the degree of A with respect to the variable of index var. If A is zero, the degree is defined to be −1.

int fq_nmod_mpoly_total_degree_fits_si(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
  Return 1 if the total degree of A fits into an slong, otherwise return 0.

void fq_nmod_mpoly_total_degree_fmpz(fmpz_t tdeg, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
slong fq_nmod_mpoly_total_degree_si(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
  Either return or set tdeg to the total degree of A. If A is zero, the total degree is defined to be −1.

void fq_nmod_mpoly_used_vars(int *used, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
  For each variable index i, set used[i] to nonzero if the variable of index i appears in A and to zero otherwise.

11.17.8 Coefficients

void fq_nmod_mpoly_get_coeff_fq_nmod_monomial(fq_nmod_t c, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
  Assuming that M is a monomial, set c to the coefficient of the corresponding monomial in A. This function throws if M is not a monomial.

void fq_nmod_mpoly_set_coeff_fq_nmod_monomial(fq_nmod_mpoly_t A, const fq_nmod_t c, const fq_nmod_mpoly_t M, const fq_nmod_mpoly_ctx_t ctx)
  Assuming that M is a monomial, set the coefficient of the corresponding monomial in A to c. This function throws if M is not a monomial.

void fq_nmod_mpoly_get_coeff_fq_nmod_fmpz(fq_nmod_t c, const fq_nmod_mpoly_t A, fmpz *const *exp, const fq_nmod_mpoly_ctx_t ctx)
void fq_nmod_mpoly_get_coeff_fq_nmod_ui(fq_nmod_t c, const fq_nmod_mpoly_t A, const ulong *exp, const fq_nmod_mpoly_ctx_t ctx)

Set c to the coefficient of the monomial with exponent vector exp.

void fq_nmod_mpoly_set_coeff_fq_nmod_fmpz(fq_nmod_mpoly_t A, const fq_nmod_t c, fmpz *const *exp, const fq_nmod_mpoly_ctx_t ctx)

void fq_nmod_mpoly_set_coeff_fq_nmod_ui(fq_nmod_mpoly_t A, const fq_nmod_t c, const ulong *exp, const fq_nmod_mpoly_ctx_t ctx)

Set the coefficient of the monomial with exponent exp to c.

void fq_nmod_mpoly_get_coeff_vars_ui(fq_nmod_mpoly_t C, const fq_nmod_mpoly_t A, const slong *vars, const ulong *exps, slong length, const fq_nmod_mpoly_ctx_t ctx)

Set C to the coefficient of A with respect to the variables in vars with powers in the corresponding array exps. Both vars and exps point to array of length length. It is assumed that 0 < length ≤ nvars(A) and that the variables in vars are distinct.

11.17.9 Comparison

int fq_nmod_mpoly_cmp(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)

Return 1 (resp. −1, or 0) if A is after (resp. before, same as) B in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when A and B are both monomials.

11.17.10 Container operations

These functions deal with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw.

int fq_nmod_mpoly_is_canonical(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Return 1 if A is in canonical form. Otherwise, return 0. To be in canonical form, all of the terms must have nonzero coefficients, and the terms must be sorted from greatest to least.

slong fq_nmod_mpoly_length(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Return the number of terms in A. If the polynomial is in canonical form, this will be the number of nonzero coefficients.

void fq_nmod_mpoly_resize(fq_nmod_mpoly_t A, slong new_length, const fq_nmod_mpoly_ctx_t ctx)

Set the length of A to new_length. Terms are either deleted from the end, or new zero terms are appended.

void fq_nmod_mpoly_get_term_coeff_fq_nmod(fq_nmod_t c, const fq_nmod_mpoly_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

Set c to the coefficient of the term of index i.

void fq_nmod_mpoly_set_term_coeff_ui(fq_nmod_mpoly_t A, slong i, ulong c, const fq_nmod_mpoly_ctx_t ctx)

Set the coefficient of the term of index i to c.

int fq_nmod_mpoly_term_exp_fits_si(const fq_nmod_mpoly_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

int fq_nmod_mpoly_term_exp_fits_ui(const fq_nmod_mpoly_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

Return 1 if all entries of the exponent vector of the term of index i fit into an slong (resp. a ulong). Otherwise, return 0.

11.17. fq_nmod_mpoly.h – multivariate polynomials over finite fields of word-sized characteristic
void fq_nmod_mpoly_get_term_exp_fmpz(fmpz **exp, const fq_nmod_mpoly_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

void fq_nmod_mpoly_get_term_exp_ui(ulong *exp, const fq_nmod_mpoly_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

void fq_nmod_mpoly_get_term_exp_si(slong *exp, const fq_nmod_mpoly_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

Set exp to the exponent vector of the term of index i. The _ui (resp. _si) version throws if any entry does not fit into a ulong (resp. slong).

ulong fq_nmod_mpoly_get_term_var_exp_ui(const fq_nmod_mpoly_t A, slong i, slong var, const fq_nmod_mpoly_ctx_t ctx)

slong fq_nmod_mpoly_get_term_var_exp_si(const fq_nmod_mpoly_t A, slong i, slong var, const fq_nmod_mpoly_ctx_t ctx)

Return the exponent of the variable var of the term of index i. This function throws if the exponent does not fit into a ulong (resp. slong).

void fq_nmod_mpoly_set_term_exp_fmpz(fq_nmod_mpoly_t A, slong i, fmpz *const *exp, const fq_nmod_mpoly_ctx_t ctx)

void fq_nmod_mpoly_set_term_exp_ui(fq_nmod_mpoly_t A, slong i, const ulong *exp, const fq_nmod_mpoly_ctx_t ctx)

Set the exponent of the term of index i to exp.

void fq_nmod_mpoly_get_term(fq_nmod_mpoly_t M, const fq_nmod_mpoly_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

Set M to the term of index i in A.

void fq_nmod_mpoly_get_term_monomial(fq_nmod_mpoly_t M, const fq_nmod_mpoly_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

Set M to the monomial of the term of index i in A. The coefficient of M will be one.

void fq_nmod_mpoly_push_term_fq_nmod_fmpz(fq_nmod_mpoly_t A, const fq_nmod_t c, fmpz *const *exp, const fq_nmod_mpoly_ctx_t ctx)

void fq_nmod_mpoly_push_term_fq_nmod_ffmpz(fq_nmod_mpoly_t A, const fq_nmod_t c, const fmpz *exp, const fq_nmod_mpoly_ctx_t ctx)

void fq_nmod_mpoly_push_term_fq_nmod_ui(fq_nmod_mpoly_t A, const fq_nmod_t c, const ulong *exp, const fq_nmod_mpoly_ctx_t ctx)

Append a term to A with coefficient c and exponent vector exp. This function runs in constant average time.

void fq_nmod_mpoly_sort_terms(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Sort the terms of A into the canonical ordering dictated by the ordering in ctx. This function simply reorders the terms: It does not combine like terms, nor does it delete terms with coefficient zero. This function runs in linear time in the bit size of A.

void fq_nmod_mpoly_combine_like_terms(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Combine adjacent like terms in A and delete terms with coefficient zero. If the terms of A were sorted to begin with, the result will be in canonical form. This function runs in linear time in the bit size of A.

void fq_nmod_mpoly_reverse(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)

Set A to the reversal of B.
11.17.11 Random generation

```c
void fq_nmod_mpoly_randtest_bound(fq_nmod_mpoly_t A, flint_rand_t state, slong length, ulong exp_bound, const fq_nmod_mpoly_ctx_t ctx)
```
Generate a random polynomial with length up to `length` and exponents in the range \([0, \text{exp\_bound} - 1]\). The exponents of each variable are generated by calls to `n_randint(state, exp\_bound)`.

```c
void fq_nmod_mpoly_randtest_bounds(fq_nmod_mpoly_t A, flint_rand_t state, slong length, ulong *exp_bounds, const fq_nmod_mpoly_ctx_t ctx)
```
Generate a random polynomial with length up to `length` and exponents in the range \([0, \text{exp\_bounds}[i] - 1]\). The exponents of the variable of index `i` are generated by calls to `n_randint(state, \text{exp\_bounds}[i])`.

```c
void fq_nmod_mpoly_randtest_bits(fq_nmod_mpoly_t A, flint_rand_t state, slong length, mp_limb_t exp_bits, const fq_nmod_mpoly_ctx_t ctx)
```
Generate a random polynomial with length up to `length` and exponents whose packed form does not exceed the given bit count.

11.17.12 Addition/Subtraction

```c
void fq_nmod_mpoly_add_fq_nmod(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_t C, const fq_nmod_mpoly_ctx_t ctx)
```
Set `A` to `B + C`.

```c
void fq_nmod_mpoly_sub_fq_nmod(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_t C, const fq_nmod_mpoly_ctx_t ctx)
```
Set `A` to `B - C`.

```c
void fq_nmod_mpoly_add(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_t C, const fq_nmod_mpoly_ctx_t ctx)
```
Set `A` to `B + C`.

```c
void fq_nmod_mpoly_sub(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_t C, const fq_nmod_mpoly_ctx_t ctx)
```
Set `A` to `B - C`.

11.17.13 Scalar operations

```c
void fq_nmod_mpoly_neg(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```
Set `A` to \(-B\).

```c
void fq_nmod_mpoly_scalar_mul_fq_nmod(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_t c, const fq_nmod_mpoly_ctx_t ctx)
```
Set `A` to `B \times c`.

```c
void fq_nmod_mpoly_make_monic(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```
Set `A` to `B` divided by the leading coefficient of `B`. This throws if `B` is zero.
11.17.14 Differentiation

```c
void fq_nmod_mpoly_derivative(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, slong var, const fq_nmod_mpoly_ctx_t ctx)
```

Set $A$ to the derivative of $B$ with respect to the variable of index $\text{var}$.

11.17.15 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

```c
void fq_nmod_mpoly_evaluate_all_fq_nmod(fq_nmod_t ev, const fq_nmod_mpoly_t A, fq_nmod_struct *const *vals, const fq_nmod_mpoly_ctx_t ctx)
```

Set $ev$ the evaluation of $A$ where the variables are replaced by the corresponding elements of the array $vals$.

```c
void fq_nmod_mpoly_evaluate_one_fq_nmod(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, slong var, const fq_nmod_t val, const fq_nmod_mpoly_ctx_t ctx)
```

Set $A$ to the evaluation of $B$ where the variable of index $\text{var}$ is replaced by $\text{val}$.

```c
int fq_nmod_mpoly_compose_fq_nmod(fq_nmod_poly_t A, const fq_nmod_mpoly_t B, fq_nmod_poly_struct *const *C, const fq_nmod_mpoly_ctx_t ctx)
```

Set $A$ to the evaluation of $B$ where the variables are replaced by the corresponding elements of the array $C$. The context object of $B$ is $\text{ctxB}$. Return 1 for success and 0 for failure.

```c
int fq_nmod_mpoly_compose_fq_nmod_mpoly(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, fq_nmod_mpoly_struct *const *C, const fq_nmod_mpoly_ctx_t ctxB, const fq_nmod_mpoly_ctx_t ctxAC)
```

Set $A$ to the evaluation of $B$ where the variables are replaced by the corresponding elements of the array $C$. Both $A$ and the elements of $C$ have context object $\text{ctxAC}$, while $B$ has context object $\text{ctxB}$. Neither $A$ nor $B$ is allowed to alias any other polynomial. Return 1 for success and 0 for failure.

```c
void fq_nmod_mpoly_compose_fq_nmod_mpoly_gen(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const slong *c, const fq_nmod_mpoly_ctx_t ctxB, const fq_nmod_mpoly_ctx_t ctxAC)
```

Set $A$ to the evaluation of $B$ where the variable of index $i$ in $\text{ctxB}$ is replaced by the variable of index $\text{c[i]}$ in $\text{ctxAC}$. The length of the array $C$ is the number of variables in $\text{ctxB}$. If any $\text{c[i]}$ is negative, the corresponding variable of $B$ is replaced by zero. Otherwise, it is expected that $\text{c[i]}$ is less than the number of variables in $\text{ctxAC}$.

11.17.16 Multiplication

```c
void fq_nmod_mpoly_mul(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_t C, const fq_nmod_mpoly_ctx_t ctx)
```

Set $A$ to $B$ times $C$. 
11.17.17 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

```c
int fq_nmod_mpoly_pow_fmpz(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fmpz_t k, const fq_nmod_mpoly_ctx_t ctx)
```

Set A to B raised to the \( k \)-th power. Return 1 for success and 0 for failure.

```c
int fq_nmod_mpoly_pow_ui(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, ulong k, const fq_nmod_mpoly_ctx_t ctx)
```

Set A to B raised to the \( k \)-th power. Return 1 for success and 0 for failure.

11.17.18 Division

```c
int fq_nmod_mpoly_divides(fq_nmod_mpoly_t Q, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

If A is divisible by B, set Q to the exact quotient and return 1. Otherwise, set Q to zero and return 0.

```c
void fq_nmod_mpoly_div(fq_nmod_mpoly_t Q, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

Set Q to the quotient of A by B, discarding the remainder.

```c
void fq_nmod_mpoly_divrem(fq_nmod_mpoly_t Q, constfq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, constfq_nmod_mpoly_ctx_t ctx)
```

Set Q and R to the quotient and remainder of A divided by B.

```c
void fq_nmod_mpoly_divrem_ideal(fq_nmod_mpoly_struct **Q, fq_nmod_mpoly_t R, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

This function is as per `fq_nmod_mpoly_divrem()` except that it takes an array of divisor polynomials B and it returns an array of quotient polynomials Q. The number of divisor (and hence quotient) polynomials, is given by `len`.

11.17.19 Greatest Common Divisor

```c
void fq_nmod_mpoly_term_content(fq_nmod_mpoly_t M, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Set M to the GCD of the terms of A. If A is zero, M will be zero. Otherwise, M will be a monomial with coefficient one.

```c
int fq_nmod_mpoly_content_vars(fq_nmod_mpoly_t g, const fq_nmod_mpoly_t A, slong *vars, slong vars_length, const fq_nmod_mpoly_ctx_t ctx)
```

Set g to the GCD of the coefficients of A when viewed as a polynomial in the variables `vars`. Return 1 for success and 0 for failure. Upon success, g will be independent of the variables `vars`.

```c
int fq_nmod_mpoly_gcd(fq_nmod_mpoly_t G, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

Try to set G to the monic GCD of A and B. The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and G is left untouched.

```c
int fq_nmod_mpoly_gcd_cofactors(fq_nmod_mpoly_t G, fq_nmod_mpoly_t Abar, fq_nmod_mpoly_t Bbar, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

Do the operation of `fq_nmod_mpoly_gcd()` and also compute \( A\bar = A/G \) and \( B\bar = B/G \) if successful.
int fq_nmod_mpoly_gcd_brown(fq_nmod_mpoly_t G, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)

int fq_nmod_mpoly_gcd_hensel(fq_nmod_mpoly_t G, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)

int fq_nmod_mpoly_gcd_zippel(fq_nmod_mpoly_t G, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)

Try to set $G$ to the GCD of $A$ and $B$ using various algorithms.

int fq_nmod_mpoly_resultant(fq_nmod_mpoly_t R, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, slong var, const fq_nmod_mpoly_ctx_t ctx)

Try to set $R$ to the resultant of $A$ and $B$ with respect to the variable of index $\var{}$.

int fq_nmod_mpoly_discriminant(fq_nmod_mpoly_t D, const fq_nmod_mpoly_t A, slong var, const fq_nmod_mpoly_ctx_t ctx)

Try to set $D$ to the discriminant of $A$ with respect to the variable of index $\var{}$.

11.17.20 Square Root

int fq_nmod_mpoly_sqrt(fq_nmod_mpoly_t Q, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

If $Q^2 = A$ has a solution, set $Q$ to a solution and return 1, otherwise return 0 and set $Q$ to zero.

int fq_nmod_mpoly_is_square(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Return 1 if $A$ is a perfect square, otherwise return 0.

int fq_nmod_mpoly_quadratic_root(fq_nmod_mpoly_t Q, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)

If $Q^2 + AQ = B$ has a solution, set $Q$ to a solution and return 1, otherwise return 0.

11.17.21 Univariate Functions

An fq_nmod_mpoly_univar_t holds a univariate polynomial in some main variable with fq_nmod_mpoly_t coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of $F_q[x_1, \ldots, x_m]$ as an element of $(F_q[x_1, \ldots, x_{v-1}, x_{v+1}, \ldots, x_m])\langle x_v \rangle$ and vice versa.

void fq_nmod_mpoly_univar_init(fq_nmod_mpoly_univar_t A, const fq_nmod_mpoly_ctx_t ctx)

Initialize $A$.

void fq_nmod_mpoly_univar_clear(fq_nmod_mpoly_univar_t A, const fq_nmod_mpoly_ctx_t ctx)

Clear $A$.

voidfq_nmod_mpoly_univar_swap(fq_nmod_mpoly_univar_t A, fq_nmod_mpoly_univar_t B, const fq_nmod_mpoly_ctx_t ctx)

Swap $A$ and $B$.

void fq_nmod_mpoly_to_univar(fq_nmod_mpoly_univar_t A, const fq_nmod_mpoly_t B, slong var, const fq_nmod_mpoly_ctx_t ctx)

Set $A$ to a univariate form of $B$ by pulling out the variable of index $\var{}$. The coefficients of $A$ will still belong to the content $ctx$ but will not depend on the variable of index $\var{}$.

voidfq_nmod_mpoly_from_univar(fq_nmod_mpoly_t A, const fq_nmod_mpoly_univar_t B, slong var, const fq_nmod_mpoly_ctx_t ctx)

Set $A$ to the normal form of $B$ by putting in the variable of index $\var{}$. This function is undefined if the coefficients of $B$ depend on the variable of index $\var{}$.  

942 Chapter 11. Finite fields
int fq_nmod_mpoly_univar_degree_fits_si(const fq_nmod_mpoly_univar_t A, const fq_nmod_mpoly_ctx_t ctx)

Return 1 if the degree of A with respect to the main variable fits an slong. Otherwise, return 0.

slong fq_nmod_mpoly_univar_length(const fq_nmod_mpoly_univar_t A, const fq_nmod_mpoly_ctx_t ctx)

Return the number of terms in A with respect to the main variable.

slong fq_nmod_mpoly_univar_get_term_exp_si(fq_nmod_mpoly_univar_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

Return the exponent of the term of index i of A.

void fq_nmod_mpoly_univar_get_term_coeff(fq_nmod_mpoly_t c, const fq_nmod_mpoly_univar_t A, slong i, const fq_nmod_mpoly_ctx_t ctx)

Set (resp. swap) c to (resp. with) the coefficient of the term of index i of A.

11.18 fq_nmod_mpoly_factor.h – factorisation of multivariate polynomials over finite fields of word-sized characteristic

11.18.1 Types, macros and constants

type fq_nmod_mpoly_factor_struct

A struct for holding a factored polynomial. There is a single constant and a product of bases to corresponding exponents.

type fq_nmod_mpoly_factor_t

An array of length 1 of fq_nmod_mpoly_factor_struct.

11.18.2 Memory management

void fq_nmod_mpoly_factor_init(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_ctx_t ctx)

Initialise f.

void fq_nmod_mpoly_factor_clear(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_ctx_t ctx)

Clear f.

11.18.3 Basic manipulation

void fq_nmod_mpoly_factor_swap(fq_nmod_mpoly_factor_t f, fq_nmod_mpoly_factor_t g, const fq_nmod_mpoly_ctx_t ctx)

Efficiently swap f and g.

slong fq_nmod_mpoly_factor_length(const fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_ctx_t ctx)

Return the length of the product in f.

void fq_nmod_mpoly_factor_get_constant_fq_nmod(fq_nmod_t c, const fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_ctx_t ctx)

Set c to the constant of f.
void fq_nmod_mpoly_factor_get_base(fq_nmod_mpoly_t p, const fq_nmod_mpoly_factor_t f, slong i, const fq_nmod_mpoly_ctx_t ctx)

void fq_nmod_mpoly_factor_swap_base(fq_nmod_mpoly_t p, const fq_nmod_mpoly_factor_t f, slong i, const fq_nmod_mpoly_ctx_t ctx)

Set (resp. swap) B to (resp. with) the base of the term of index i in A.

slong fq_nmod_mpoly_factor_get_exp_si(fq_nmod_mpoly_factor_t f, slong i, const fq_nmod_mpoly_ctx_t ctx)

Return the exponent of the term of index i in A. It is assumed to fit an slong.

void fq_nmod_mpoly_factor_sort(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_ctx_t ctx)

Sort the product of f first by exponent and then by base.

11.18.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and f is undefined. None of these functions multiply f by A: f is simply set to a factorisation of A, and thus these functions should not depend on the initial value of the output f.

int fq_nmod_mpoly_factor_squarefree(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Set f to a factorization of A where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call fq_nmod_mpoly_factor_sort() and then multiply the bases with the desired exponent.

int fq_nmod_mpoly_factor(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)

Set f to a factorization of A where the bases are irreducible.

11.19 fq_zech.h – finite fields (Zech logarithm representation)

We represent an element of the finite field as a power of a generator for the multiplicative group of the finite field. In particular, we use a root of \( f(x) \), where \( f(X) \in \mathbb{F}_p[X] \) is a monic, irreducible polynomial of degree \( n \), as a polynomial in \( \mathbb{F}_p[X] \) of degree less than \( n \). The underlying data structure is just an mp_limb_t.

The default choice for \( f(X) \) is the Conway polynomial for the pair \( (p, n) \), enabled by Frank Lübeck’s data base of Conway polynomials using the _nmod_poly_conway() function. If a Conway polynomial is not available, then a random irreducible polynomial will be chosen for \( f(X) \). Additionally, the user is able to supply their own \( f(X) \).

We required that the order of the field fits inside of an mp_limb_t; however, it is recommended that \( p^n < 2^{20} \) due to the time and memory needed to compute the Zech logarithm table.

11.19.1 Types, macros and constants

type fq_zech_ctx_struct

type fq_zech_ctx_t

type fq_zech_struct

type fq_zech_t
11.19.2 Context Management

void fq_zech_ctx_init_ui(fq_zech_ctx_t ctx, ulong p, slong d, const char *var)
Initialises the context for prime \( p \) and extension degree \( d \), with name \( \text{var} \) for the generator. By default, it will try use a Conway polynomial; if one is not available, a random primitive polynomial will be used.

Assumes that \( p \) is a prime and \( p^d < 2^{\text{FLINT\_BITS}} \).

Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

int _fq_zech_ctx_init_conway_ui(fq_zech_ctx_t ctx, ulong p, slong d, const char *var)
Attempts to initialise the context for prime \( p \) and extension degree \( d \), with name \( \text{var} \) for the generator using a Conway polynomial for the modulus.

Returns 1 if the Conway polynomial is in the database for the given size and the initialization is successful; otherwise, returns 0.

Assumes that \( p \) is a prime and \( p^d < 2^{\text{FLINT\_BITS}} \).

Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

void fq_zech_ctx_init_conway_ui(fq_zech_ctx_t ctx, ulong p, slong d, const char *var)
Initialises the context for prime \( p \) and extension degree \( d \), with name \( \text{var} \) for the generator using a Conway polynomial for the modulus.

Assumes that \( p \) is a prime and \( p^d < 2^{\text{FLINT\_BITS}} \).

Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

void fq_zech_ctx_init_random_ui(fq_zech_ctx_t ctx, ulong p, slong d, const char *var)
Initialises the context for prime \( p \) and extension degree \( d \), with name \( \text{var} \) for the generator using a random primitive polynomial.

Assumes that \( p \) is a prime and \( p^d < 2^{\text{FLINT\_BITS}} \).

Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

void fq_zech_ctx_init_modulus(fq_zech_ctx_t ctx, const nmod_poly_t modulus, const char *var)
Initialises the context for given \( \text{modulus} \) with name \( \text{var} \) for the generator.

Assumes that \( \text{modulus} \) is an primitive polynomial over \( \mathbb{F}_p \). An exception is raised if a non-primitive modulus is detected.

Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.

int fq_zech_ctx_init_modulus_check(fq_zech_ctx_t ctx, const nmod_poly_t modulus, const char *var)
As per the previous function, but returns 0 if the modulus was not primitive and 1 if the context was successfully initialised with the given modulus. No exception is raised.

void fq_zech_ctx_init_fq_nmod_ctx(fq_zech_ctx_t ctx, fq_nmod_ctx_t ctxn)
Initializes the context \( \text{ctx} \) to be the Zech representation for the finite field given by \( \text{ctxn} \).

int fq_zech_ctx_init_fq_nmod_ctx_check(fq_zech_ctx_t ctx, fq_nmod_ctx_t ctxn)
As per the previous function but returns 0 if a non-primitive modulus is detected. Returns 0 if the Zech representation was successfully initialised.

void fq_zech_ctx_init_randtest(fq_zech_ctx_t ctx, flint_rand_t state, int type)
Initialises \( \text{ctx} \) to a random finite field, where the prime and degree is set according to \( \text{type} \). If \( \text{type} \) is 0 the prime and degree may be large, else if \( \text{type} \) is 1 the degree is small but the prime may be large, else if \( \text{type} \) is 2 the prime is small but the degree may be large, else if \( \text{type} \) is 3 both prime and degree are small.
void \texttt{fq\_zech\_ctx\_init\_randtest\_reducible}(fq\_zech\_ctx\_t ctx, flint\_rand\_t state, int type)

Since the Zech logarithm representation does not work with a non-irreducible modulus, this function does the same as \texttt{fq\_zech\_ctx\_init\_randtest()}. 

void \texttt{fq\_zech\_ctx\_clear}(fq\_zech\_ctx\_t ctx)

Clears all memory that has been allocated as part of the context.

\begin{verbatim}
const nmod\_poly\_struct *fq\_zech\_ctx\_modulus(const \texttt{fq\_zech\_ctx\_t ctx})

Returns a pointer to the modulus in the context.
\end{verbatim}

\begin{verbatim}
slong fq\_zech\_ctx\_degree(const \texttt{fq\_zech\_ctx\_t ctx})

Returns the degree of the field extension \([\mathbb{F}_q : \mathbb{F}_p]\), which is equal to \(\log_p q\).
\end{verbatim}

\begin{verbatim}
ulong fq\_zech\_ctx\_prime(const \texttt{fq\_zech\_ctx\_t ctx})

Returns the prime \(p\) of the context.
\end{verbatim}

\begin{verbatim}
void \texttt{fq\_zech\_ctx\_order}(fmpz_t f, const \texttt{fq\_zech\_ctx\_t ctx})

Sets \(f\) to be the size of the finite field.
\end{verbatim}

\begin{verbatim}
mp\_limb\_t fq\_zech\_ctx\_order\_ui(const \texttt{fq\_zech\_ctx\_t ctx})

Returns the size of the finite field.
\end{verbatim}

\begin{verbatim}
int \texttt{fq\_zech\_ctx\_fprint}(FILE \*file, const \texttt{fq\_zech\_ctx\_t ctx})

Prints the context information to \{tt\{file\}}. Returns 1 for a success and a negative number for an error.
\end{verbatim}

\begin{verbatim}
void \texttt{fq\_zech\_ctx\_print}(const \texttt{fq\_zech\_ctx\_t ctx})

Prints the context information to \{tt\{stdout\}}.
\end{verbatim}

11.19.3 Memory management

\begin{verbatim}
void \texttt{fq\_zech\_init}(\texttt{fq\_zech\_t rop, const \texttt{fq\_zech\_ctx\_t ctx}})

Initialises the element \(rop\), setting its value to 0.
\end{verbatim}

\begin{verbatim}
void \texttt{fq\_zech\_init2}(\texttt{fq\_zech\_t rop, const \texttt{fq\_zech\_ctx\_t ctx}})

Initialises \(poly\) with at least enough space for it to be an element of \(ctx\) and sets it to 0.
\end{verbatim}

\begin{verbatim}
void \texttt{fq\_zech\_clear}(\texttt{fq\_zech\_t rop, const \texttt{fq\_zech\_ctx\_t ctx}})

Clears the element \(rop\).
\end{verbatim}

\begin{verbatim}
void \_fq\_zech\_sparse\_reduce(mp\_ptr R, slong lenR, const \texttt{fq\_zech\_ctx\_t ctx})

Reduces \((R, \text{len}R)\) modulo the polynomial \(f\) given by the modulus of \(ctx\).
\end{verbatim}

\begin{verbatim}
void \_fq\_zech\_dense\_reduce(mp\_ptr R, slong lenR, const \texttt{fq\_zech\_ctx\_t ctx})

Reduces \((R, \text{len}R)\) modulo the polynomial \(f\) given by the modulus of \(ctx\) using Newton division.
\end{verbatim}

\begin{verbatim}
void \_fq\_zech\_reduce(mp\_ptr r, slong lenR, const \texttt{fq\_zech\_ctx\_t ctx})

Reduces \((R, \text{len}R)\) modulo the polynomial \(f\) given by the modulus of \(ctx\). Does either sparse or dense reduction based on \(ctx->\text{sparse\_modulus}\).
\end{verbatim}

\begin{verbatim}
void \texttt{fq\_zech\_reduce}(\texttt{fq\_zech\_t rop, const \texttt{fq\_zech\_ctx\_t ctx}})

Reduces the polynomial \(rop\) as an element of \(\mathbb{F}_p[X]/(f(X))\).
11.19.4 Basic arithmetic

```c
void fq_zech_add(fq_zech_t rop, const fq_zech_t op1, const fq_zech_t op2, const fq_zech_ctx_t ctx)
    Sets rop to the sum of op1 and op2.

void fq_zech_sub(fq_zech_t rop, const fq_zech_t op1, const fq_zech_t op2, const fq_zech_ctx_t ctx)
    Sets rop to the difference of op1 and op2.

void fq_zech_sub_one(fq_zech_t rop, const fq_zech_t op1, const fq_zech_ctx_t ctx)
    Sets rop to the difference of op1 and 1.

void fq_zech_neg(fq_zech_t rop, const fq_zech_t op, const fq_zech_ctx_t ctx)
    Sets rop to the negative of op.

void fq_zech_mul(fq_zech_t rop, const fq_zech_t op1, const fq_zech_t op2, const fq_zech_ctx_t ctx)
    Sets rop to the product of op1 and op2, reducing the output in the given context.

void fq_zech_mul_fmpz(fq_zech_t rop, const fq_zech_t op, const fmpz_t x, const fq_zech_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_zech_mul_si(fq_zech_t rop, const fq_zech_t op, slong x, const fq_zech_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_zech_mul_ui(fq_zech_t rop, const fq_zech_t op, ulong x, const fq_zech_ctx_t ctx)
    Sets rop to the product of op and x, reducing the output in the given context.

void fq_zech_sqr(fq_zech_t rop, const fq_zech_t op, const fq_zech_ctx_t ctx)
    Sets rop to the square of op, reducing the output in the given context.

void fq_zech_div(fq_zech_t rop, const fq_zech_t op1, const fq_zech_t op2, const fq_zech_ctx_t ctx)
    Sets rop to the quotient of op1 and op2, reducing the output in the given context.

void _fq_zech_inv(mp_ptr *rop, mp_srcptr *op, slong len, const fq_zech_ctx_t ctx)
    Sets (rop, d) to the inverse of the non-zero element (op, len).

void fq_zech_gcdinv(fq_zech_t f, fq_zech_t inv, const fq_zech_t op, const fq_zech_ctx_t ctx)
    Sets inv to be the inverse of op modulo the modulus of ctx and sets f to one. Since the modulus
    for ctx is always irreducible, op is always invertible.

void _fq_zech_pow(fmpz *rop, const fmpz *op, slong len, const fmpz_t e, const fmpz *a, const slong *j, slong lena, const fmpz_t p)
    Sets (rop, 2*d-1) to (op,len) raised to the power e, reduced modulo f(X), the modulus of ctx.
    Assumes that e ≥ 0 and that len is positive and at most d.
    Although we require that rop provides space for 2d – 1 coefficients, the output will be reduced
    modulo f(X), which is a polynomial of degree d.
    Does not support aliasing.

void fq_zech_pow(fq_zech_t rop, const fq_zech_t op, const fmpz_t e, const fq_zech_ctx_t ctx)
    Sets rop the op raised to the power e.
    Currently assumes that e ≥ 0.
    Note that for any input op, rop is set to 1 whenever e = 0.
```
void fq_zech_pow_ui(fq_zech_t rop, const fq_zech_t op, const ulong e, const fq_zech_ctx_t ctx)
    Sets rop the op raised to the power e.
    Currently assumes that e ≥ 0.
    Note that for any input op, rop is set to 1 whenever e = 0.

11.19.5 Roots

int fq_zech_sqrt(fq_zech_t rop, const fq_zech_t op1, const fq_zech_ctx_t ctx)
    Sets rop to the square root of op1 if it is a square, and return 1, otherwise return 0.

void fq_zech_pth_root(fq_zech_t rop, const fq_zech_t op1, const fq_zech_ctx_t ctx)
    Sets rop to a p\text{th} root root of op1. Currently, this computes the root by raising op1 to p^{d-1} where d is the degree of the extension.

int fq_zech_is_square(const fq_zech_t op, const fq_zech_ctx_t ctx)
    Return 1 if op is a square.

11.19.6 Output

int fq_zech_fprint_pretty(FILE *file, const fq_zech_t op, const fq_zech_ctx_t ctx)
    Prints a pretty representation of op to file.
    In the current implementation, always returns 1. The return code is part of the function’s signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

void fq_zech_print_pretty(const fq_zech_t op, const fq_zech_ctx_t ctx)
    Prints a pretty representation of op to stdout.
    In the current implementation, always returns 1. The return code is part of the function’s signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

int fq_zech_fprint(FILE *file, const fq_zech_t op, const fq_zech_ctx_t ctx)
    Prints a representation of op to file.

void fq_zech_print(const fq_zech_t op, const fq_zech_ctx_t ctx)
    Prints a representation of op to stdout.

char *fq_zech_get_str(const fq_zech_t op, const fq_zech_ctx_t ctx)
    Returns the plain FLINT string representation of the element op.

char *fq_zech_get_str_pretty(const fq_zech_t op, const fq_zech_ctx_t ctx)
    Returns a pretty representation of the element op using the null-terminated string x as the variable name.

11.19.7 Randomisation

void fq_zech_randtest(fq_zech_t rop, flint_rand_t state, const fq_zech_ctx_t ctx)
    Generates a random element of \( F_q \).

void fq_zech_randtest_not_zero(fq_zech_t rop, flint_rand_t state, const fq_zech_ctx_t ctx)
    Generates a random non-zero element of \( F_q \).

void fq_zech_randtest_dense(fq_zech_t rop, flint_rand_t state, const fq_zech_ctx_t ctx)
    Generates a random element of \( F_q \) which has an underlying polynomial with dense coefficients.
void fq_zech_rand(fq_zech_t rop, flint_rand_t state, const fq_zech_ctx_t ctx)
    Generates a high quality random element of \( \mathbb{F}_q \).

void fq_zech_rand_not_zero(fq_zech_t rop, flint_rand_t state, const fq_zech_ctx_t ctx)
    Generates a high quality non-zero random element of \( \mathbb{F}_q \).

### 11.19.8 Assignments and conversions

void fq_zech_set(fq_zech_t rop, const fq_zech_t op, const fq_zech_ctx_t ctx)
    Sets rop to op.

void fq_zech_set_si(fq_zech_t rop, const slong x, const fq_zech_ctx_t ctx)
    Sets rop to \( x \), considered as an element of \( \mathbb{F}_p \).

void fq_zech_set_ui(fq_zech_t rop, const ulong x, const fq_zech_ctx_t ctx)
    Sets rop to \( x \), considered as an element of \( \mathbb{F}_p \).

void fq_zech_set_fmpz(fq_zech_t rop, const fmpz_t x, const fq_zech_ctx_t ctx)
    Sets rop to \( x \), considered as an element of \( \mathbb{F}_p \).

void fq_zech_swap(fq_zech_t op1, fq_zech_t op2, const fq_zech_ctx_t ctx)
    Swaps the two elements op1 and op2.

void fq_zech_zero(fq_zech_t rop, const fq_zech_ctx_t ctx)
    Sets rop to zero.

void fq_zech_one(fq_zech_t rop, const fq_zech_ctx_t ctx)
    Sets rop to one, reduced in the given context.

void fq_zech_gen(fq_zech_t rop, const fq_zech_ctx_t ctx)
    Sets rop to a generator for the finite field. There is no guarantee this is a multiplicative generator of the finite field.

int fq_zech_get_fmpz(fmpz_t rop, const fq_zech_t op, const fq_zech_ctx_t ctx)
    If op has a lift to the integers, return 1 and set rop to the lift in \([0, p)\). Otherwise, return 0 and leave rop undefined.

void fq_zech_get_fq_nmod(fq_nmod_t rop, const fq_zech_t op, const fq_zech_ctx_t ctx)
    Sets rop to the \( \mathbb{F}_q \) element corresponding to op.

void fq_zech_set_fq_nmod(fq_zech_t rop, const fq_nmod_t op, const fq_zech_ctx_t ctx)
    Sets rop to the \( \mathbb{F}_q \) element corresponding to op.

void fq_zech_get_nmod_poly(nmod_poly_t a, const fq_zech_t b, const fq_zech_ctx_t ctx)
    Set a to a representative of b in ctx. The representatives are taken in \((\mathbb{Z}/p\mathbb{Z})[x]/h(x)\) where \( h(x) \) is the defining polynomial in ctx.

void fq_zech_set_nmod_poly(fq_zech_t a, const nmod_poly_t b, const fq_zech_ctx_t ctx)
    Set a to the element in ctx with representative b. The representatives are taken in \((\mathbb{Z}/p\mathbb{Z})[x]/h(x)\) where \( h(x) \) is the defining polynomial in ctx.

void fq_zech_get_nmod_mat(nmod_mat_t col, const fq_zech_t a, const fq_zech_ctx_t ctx)
    Convert a to a column vector of length degree(ctx).

void fq_zech_set_nmod_mat(fq_zech_t a, const nmod_mat_t col, const fq_zech_ctx_t ctx)
    Convert a column vector col of length degree(ctx) to an element of ctx.
11.19.9 Comparison

int fq_zech_is_zero(const fq_zech_t op, const fq_zech_ctx_t ctx)
Returns whether op is equal to zero.

int fq_zech_is_one(const fq_zech_t op, const fq_zech_ctx_t ctx)
Returns whether op is equal to one.

int fq_zech_equal(const fq_zech_t op1, const fq_zech_t op2, const fq_zech_ctx_t ctx)
Returns whether op1 and op2 are equal.

int fq_zech_is_invertible(const fq_zech_t op, const fq_zech_ctx_t ctx)
Returns whether op is an invertible element.

int fq_zech_is_invertible_f(fq_zech_t f, const fq_zech_t op, const fq_zech_ctx_t ctx)
Returns whether op is an invertible element. If it is not, then f is set of a factor of the modulus. Since the modulus for an fq_zech_ctx_t is always irreducible, then any non-zero op will be invertible.

11.19.10 Special functions

void fq_zech_trace(fmpz_t rop, const fq_zech_t op, const fq_zech_ctx_t ctx)
Sets rop to the trace of op.

For an element $a \in \mathbb{F}_q$, multiplication by $a$ defines a $\mathbb{F}_p$-linear map on $\mathbb{F}_q$. We define the trace of $a$ as the trace of this map. Equivalently, if $\Sigma$ generates $\text{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ then the trace of $a$ is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$, where $d = \log_p q$.

void fq_zech_norm(fmpz_t rop, const fq_zech_t op, const fq_zech_ctx_t ctx)
Computes the norm of op.

For an element $a \in \mathbb{F}_q$, multiplication by $a$ defines a $\mathbb{F}_p$-linear map on $\mathbb{F}_q$. We define the norm of $a$ as the determinant of this map. Equivalently, if $\Sigma$ generates $\text{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ then the trace of $a$ is equal to $\prod_{i=0}^{d-1} \Sigma^i(a)$, where $d = \dim_{\mathbb{F}_p}(\mathbb{F}_q)$.

Algorithm selection is automatic depending on the input.

void fq_zech_frobenius(fq_zech_t rop, const fq_zech_t op, slong e, const fq_zech_ctx_t ctx)
Evaluates the homomorphism $\Sigma^e$ at op.

Recall that $\mathbb{F}_q/\mathbb{F}_p$ is Galois with Galois group $\langle \sigma \rangle$, which is also isomorphic to $\mathbb{Z}/d\mathbb{Z}$, where $\sigma \in \text{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ is the Frobenius element $\sigma: x \mapsto x^p$.

int fq_zech_multiplicative_order(fmpz *ord, const fq_zech_t op, const fq_zech_ctx_t ctx)
Computes the order of op as an element of the multiplicative group of ctx.

Returns 0 if op is 0, otherwise it returns 1 if op is a generator of the multiplicative group, and -1 if it is not.

Note that ctx must already correspond to a finite field defined by a primitive polynomial and so this function cannot be used to check primitivity of the generator, but can be used to check that other elements are primitive.

int fq_zech_is_primitive(const fq_zech_t op, const fq_zech_ctx_t ctx)
Returns whether op is primitive, i.e., whether it is a generator of the multiplicative group of ctx.
11.19.11 Bit packing

```c
void fq_zech_bit_pack(fmpz_t f, const fq_zech_t op, flint_bitcnt_t bit_size, const fq_zech_ctx_t ctx)
```

Packs `op` into bitfields of size `bit_size`, writing the result to `f`.

```c
void fq_zech_bit_unpack(fq_zech_t rop, const fmpz_t f, flint_bitcnt_t bit_size, const fq_zech_ctx_t ctx)
```

Unpacks into `rop` the element with coefficients packed into fields of size `bit_size` as represented by the integer `f`.

# 11.20 fq_zech_vec.h – vectors over finite fields (Zech logarithm representation)

## 11.20.1 Memory management

```c
fq_zech_struct *fq_zech_vec_init(slong len, const fq_zech_ctx_t ctx)
```

Returns an initialised vector of `fq_zech`'s of given length.

```c
void fq_zech_vec_clear(fq_zech_struct *vec, slong len, const fq_zech_ctx_t ctx)
```

Clears the entries of `(vec, len)` and frees the space allocated for `vec`.

## 11.20.2 Randomisation

```c
void _fq_zech_vec_randtest(fq_zech_struct *f, flint_rand_t state, slong len, const fq_zech_ctx_t ctx)
```

Sets the entries of a vector of the given length to elements of the finite field.

## 11.20.3 Input and output

```c
int _fq_zech_vec_fprint(FILE *file, const fq_zech_struct *vec, slong len, const fq_zech_ctx_t ctx)
```

Prints the vector of given length to the stream `file`. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```c
int _fq_zech_vec_print(const fq_zech_struct *vec, slong len, const fq_zech_ctx_t ctx)
```

Prints the vector of given length to `stdout`.

For further details, see `_fq_zech_vec_fprint()`.

## 11.20.4 Assignment and basic manipulation

```c
void _fq_zech_vec_set(fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len2, const fq_zech_ctx_t ctx)
```

Makes a copy of `(vec2, len2)` into `vec1`.

```c
void _fq_zech_vec_swap(fq_zech_struct *vec1, fq_zech_struct *vec2, slong len2, const fq_zech_ctx_t ctx)
```

Swaps the elements in `(vec1, len2)` and `(vec2, len2)`.

```c
void _fq_zech_vec_zero(fq_zech_struct *vec, slong len, const fq_zech_ctx_t ctx)
```

Zeros the entries of `(vec, len)`.
void _fq_zech_vec_neg(fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len2, const fq_zech_ctx_t ctx)

Negates (vec2, len2) and places it into vec1.

11.20.5 Comparison

int _fq_zech_vec_equal(const fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len, const fq_zech_ctx_t ctx)

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

int _fq_zech_vec_is_zero(const fq_zech_struct *vec, slong len, const fq_zech_ctx_t ctx)

Returns 1 if (vec, len) is zero, and 0 otherwise.

11.20.6 Addition and subtraction

void _fq_zech_vec_add(fq_zech_struct *res, const fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len2, const fq_zech_ctx_t ctx)

Sets (res, len2) to the sum of (vec1, len2) and (vec2, len2).

void _fq_zech_vec_sub(fq_zech_struct *res, const fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len2, const fq_zech_ctx_t ctx)

Sets (res, len2) to (vec1, len2) minus (vec2, len2).

11.20.7 Scalar multiplication and division

void _fq_zech_vec_scalar_addmul_fq_zech(fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len2, const fq_zech_t c, const fq_zech_ctx_t ctx)

Adds (vec2, len2) times c to (vec1, len2), where c is a fq_zech_t.

void _fq_zech_vec_scalar_submul_fq_zech(fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len2, const fq_zech_t c, const fq_zech_ctx_t ctx)

Subtracts (vec2, len2) times c from (vec1, len2), where c is a fq_zech_t.

11.20.8 Dot products

void _fq_zech_vec_dot(fq_zech_t res, const fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len2, const fq_zech_ctx_t ctx)

Sets res to the dot product of (vec1, len) and (vec2, len).

11.21 fq_zech_mat.h – matrices over finite fields (Zech logarithm representation)

11.21.1 Types, macros and constants

type fq_zech_mat_struct

type fq_zech_mat_t
11.21.2 Memory management

void fq_zech_mat_init(fq_zech_mat_t mat, slong rows, slong cols, const fq_zech_ctx_t ctx)
   Initialises mat to a rows-by-cols matrix with coefficients in $\mathbb{F}_q$ given by ctx. All elements are set to zero.

void fq_zech_mat_init_set(fq_zech_mat_t mat, const fq_zech_mat_t src, const fq_zech_ctx_t ctx)
   Initialises mat and sets its dimensions and elements to those of src.

void fq_zech_mat_clear(fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
   Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an fq_zech_mat_t object.

void fq_zech_mat_set(fq_zech_mat_t mat, const fq_zech_mat_t src, const fq_zech_ctx_t ctx)
   Sets mat to a copy of src. It is assumed that mat and src have identical dimensions.

11.21.3 Basic properties and manipulation

fq_zech_struct *fq_zech_mat_entry(const fq_zech_mat_t mat, slong i, slong j)
   Directly accesses the entry in mat in row i and column j, indexed from zero. No bounds checking is performed.

void fq_zech_mat_entry_set(fq_zech_mat_t mat, slong i, slong j, const fq_zech_t x, const fq_zech_ctx_t ctx)
   Sets the entry in mat in row i and column j to x.

slong fq_zech_mat_nrows(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
   Returns the number of rows in mat.

slong fq_zech_mat_ncols(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
   Returns the number of columns in mat.

void fq_zech_mat_swap(fq_zech_mat_t mat1, fq_zech_mat_t mat2, const fq_zech_ctx_t ctx)
   Swaps two matrices. The dimensions of mat1 and mat2 are allowed to be different.

void fq_zech_mat_swap_entrywise(fq_zech_mat_t mat1, fq_zech_mat_t mat2, const fq_zech_ctx_t ctx)
   Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

void fq_zech_mat_zero(fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
   Sets all entries of mat to 0.

void fq_zech_mat_one(fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
   Sets all diagonal entries of mat to 1 and all other entries to 0.

11.21.4 Conversions

void fq_zech_mat_set_nmod_mat(fq_zech_mat_t mat1, const nmod_mat_t mat2, const fq_zech_ctx_t ctx)
   Sets the matrix mat1 to the matrix mat2.

void fq_zech_mat_set_fmpz_mod_mat(fq_zech_mat_t mat1, const fmpz_mod_mat_t mat2, const fq_zech_ctx_t ctx)
   Sets the matrix mat1 to the matrix mat2.
11.21.5 Concatenate

```c
void fq_zech_mat_concat_vertical(fq_zech_mat_t res, const fq_zech_mat_t mat1, const fq_zech_mat_t mat2, const fq_zech_ctx_t ctx)
```

Sets \( res \) to vertical concatenation of \( (mat1, mat2) \) in that order. Matrix dimensions:
- \( mat1 \): \( m \times n \),
- \( mat2 \): \( k \times n \),
- \( res \): \( (m + k) \times n \).

```c
void fq_zech_mat_concat_horizontal(fq_zech_mat_t res, const fq_zech_mat_t mat1, const fq_zech_mat_t mat2, const fq_zech_ctx_t ctx)
```

Sets \( res \) to horizontal concatenation of \( (mat1, mat2) \) in that order. Matrix dimensions:
- \( mat1 \): \( m \times n \),
- \( mat2 \): \( m \times k \),
- \( res \): \( m \times (n + k) \).

11.21.6 Printing

```c
int fq_zech_mat_print_pretty(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
```

Pretty-prints \( mat \) to stdout. A header is printed followed by the rows enclosed in brackets.

```c
int fq_zech_mat_fprint_pretty(FILE *file, const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
```

Pretty-prints \( mat \) to \( file \). A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```c
int fq_zech_mat_print(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
```

Prints \( mat \) to stdout. A header is printed followed by the rows enclosed in brackets.

```c
int fq_zech_mat_fprint(FILE *file, const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
```

Prints \( mat \) to \( file \). A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

11.21.7 Window

```c
void fq_zech_mat_window_init(fq_zech_mat_t window, const fq_zech_mat_t mat, slong r1, slong c1, slong r2, slong c2, const fq_zech_ctx_t ctx)
```

Initializes the matrix \( window \) to be an \( r2 - r1 \) by \( c2 - c1 \) submatrix of \( mat \) whose \( (0,0) \) entry is the \( (r1, c1) \) entry of \( mat \). The memory for the elements of \( window \) is shared with \( mat \).

```c
void fq_zech_mat_window_clear(fq_zech_mat_t window, const fq_zech_ctx_t ctx)
```

Clears the matrix \( window \) and releases any memory that it uses. Note that the memory to the underlying matrix that \( window \) points to is not freed.

11.21.8 Random matrix generation

```c
void fq_zech_mat_randtest(fq_zech_mat_t mat, flint_rand_t state, const fq_zech_ctx_t ctx)
```

Sets the elements of \( mat \) to random elements of \( \mathbb{F}_q \), given by \( ctx \).

```c
int fq_zech_mat_randpermdiag(fq_zech_mat_t mat, flint_rand_t state, fq_zech_struct *diag, slong n, const fq_zech_ctx_t ctx)
```

Sets \( mat \) to a random permutation of the diagonal matrix with \( n \) leading entries given by the vector \( diag \). It is assumed that the main diagonal of \( mat \) has room for at least \( n \) entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.
void fq_zech_mat_randrank(fq_zech_mat_t mat, flint_rand_t state, slong rank, const fq_zech_ctx_t ctx)

Sets mat to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of $F_q$.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling fq_zech_mat_randops().

void fq_zech_mat_randops(fq_zech_mat_t mat, slong count, flint_rand_t state, const fq_zech_ctx_t ctx)

Randomises mat by performing elementary row or column operations. More precisely, at most count random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

void fq_zech_mat_randtril(fq_zech_mat_t mat, flint_rand_t state, int unit, const fq_zech_ctx_t ctx)

Sets mat to a random lower triangular matrix. If unit is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

void fq_zech_mat_randtriu(fq_zech_mat_t mat, flint_rand_t state, int unit, const fq_zech_ctx_t ctx)

Sets mat to a random upper triangular matrix. If unit is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

11.21.9 Comparison

int fq_zech_mat_equal(const fq_zech_mat_t mat1, const fq_zech_mat_t mat2, const fq_zech_ctx_t ctx)

Returns nonzero if mat1 and mat2 have the same dimensions and elements, and zero otherwise.

int fq_zech_mat_is_zero(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)

Returns a non-zero value if all entries mat are zero, and otherwise returns zero.

int fq_zech_mat_is_one(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)

Returns a non-zero value if all entries mat are zero except the diagonal entries which must be one, otherwise returns zero.

int fq_zech_mat_is_empty(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)

Returns a non-zero value if the number of rows or the number of columns in mat is zero, and otherwise returns zero.

int fq_zech_mat_is_square(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)

Returns a non-zero value if the number of rows is equal to the number of columns in mat, and otherwise returns zero.

11.21.10 Addition and subtraction

void fq_zech_mat_add(fq_zech_mat_t C, const fq_zech_mat_t A, const fq_zech_mat_t B, const fq_zech_ctx_t ctx)

Computes $C = A + B$. Dimensions must be identical.

void fq_zech_mat_sub(fq_zech_mat_t C, const fq_zech_mat_t A, const fq_zech_mat_t B, const fq_zech_ctx_t ctx)

Computes $C = A - B$. Dimensions must be identical.

void fq_zech_mat_neg(fq_zech_mat_t A, const fq_zech_mat_t B, const fq_zech_ctx_t ctx)

Sets $B = -A$. Dimensions must be identical.
11.21.11 Matrix multiplication

```c
void fq_zech_mat_mul(fq_zech_mat_t C, const fq_zech_mat_t A, const fq_zech_mat_t B, const fq_zech_ctx_t ctx)
Sets C = AB. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. This function automatically chooses between classical and KS multiplication.
```

```c
void fq_zech_mat_mul_classical(fq_zech_mat_t C, const fq_zech_mat_t A, const fq_zech_mat_t B, const fq_zech_ctx_t ctx)
Sets C = AB. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses classical matrix multiplication.
```

```c
void fq_zech_mat_mul_KS(fq_zech_mat_t C, const fq_zech_mat_t A, const fq_zech_mat_t B, const fq_zech_ctx_t ctx)
Sets C = AB. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses Kronecker substitution to perform the multiplication over the integers.
```

```c
void fq_zech_mat_submul(fq_zech_mat_t D, const fq_zech_mat_t C, const fq_zech_mat_t A, const fq_zech_mat_t B, const fq_zech_ctx_t ctx)
Sets D = C + AB. C and D may be aliased with each other but not with A or B.
```

```c
void fq_zech_mat_mul_vec(fq_zech_struct *c, const fq_zech_mat_t A, const fq_zech_struct *b, slong blen, const fq_zech_ctx_t ctx)
Compute a matrix-vector product of A and (b, blen) and store the result in c. The vector (b, blen) is either truncated or zero-extended to the number of columns of A. The number entries written to c is always equal to the number of rows of A.
```

```c
void fq_zech_mat_mul_vec_ptr(fq_zech_struct *const *c, const fq_zech_mat_t A, const fq_zech_struct *const *b, slong blen, const fq_zech_ctx_t ctx)
Compute a matrix-vector product of (a, alen) and B and and store the result in c. The vector (a, alen) is either truncated or zero-extended to the number of rows of B. The number entries written to c is always equal to the number of columns of B.
```

11.21.12 LU decomposition

```c
slong fq_zech_mat_lu(slong *P, fq_zech_mat_t A, int rank_check, const fq_zech_ctx_t ctx)
Computes a generalised LU decomposition LU = PA of a given matrix A, returning the rank of A.
```

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r, U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for rank_check is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

This function calls fq_zech_mat_lu_recursive.

```c
slong fq_zech_mat_lu_classical(slong *P, fq_zech_mat_t A, int rank_check, const fq_zech_ctx_t ctx)
Computes a generalised LU decomposition LU = PA of a given matrix A, returning the rank of A. The behavior of this function is identical to that of fq_zech_mat_lu. Uses Gaussian elimination.
```
**11.21.11 Reduced row echelon form**

\[
\text{slong } \text{fqzechmatrref}(\text{fqzechmat t } A, \text{ const fqzechctx t } ctx)
\]

Puts \( A \) in reduced row echelon form and returns the rank of \( A \).

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

\[
\text{slong } \text{fqzechmatreduce}(\text{fqzechmat t } A, \text{ slong } *P, \text{ slong } *L, \text{ slong } n, \text{ const fqzechctx t } ctx)
\]

Reduce row \( n \) of the matrix \( A \), assuming the prior rows are in Gauss form. However those rows may not be in order. The entry \( i \) of the array \( P \) is the row of \( A \) which has a pivot in the \( i \)-th column. If no such row exists, the entry of \( P \) will be \(-1\). The function returns the column in which the \( n \)-th row has a pivot after reduction. This will always be chosen to be the first available column for a pivot from the left. This information is also updated in \( P \). Entry \( i \) of the array \( L \) contains the number of possibly nonzero columns of \( A \) row \( i \). This speeds up reduction in the case that \( A \) is chambered on the right. Otherwise the entries of \( L \) can all be set to the number of columns of \( A \). We require the entries of \( L \) to be monotonic increasing.

**11.21.12 Triangular solving**

\[
\text{void } \text{fqzechmat solve tril}(\text{fqzechmat t } X, \text{ const fqzechmat t } L, \text{ const fqzechmat t } B, \text{ int unit, const fqzechctx t } ctx)
\]

Sets \( X = L^{-1}B \) where \( L \) is a full rank lower triangular square matrix. If \( \text{unit} = 1 \), \( L \) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \( X \) and \( B \) are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

\[
\text{void } \text{fqzechmat solve tril classical}(\text{fqzechmat t } X, \text{ const fqzechmat t } L, \text{ const fqzechmat t } B, \text{ int unit, const fqzechctx t } ctx)
\]

Sets \( X = L^{-1}B \) where \( L \) is a full rank lower triangular square matrix. If \( \text{unit} = 1 \), \( L \) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \( X \) and \( B \) are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

\[
\text{void } \text{fqzechmat solve tril recursive}(\text{fqzechmat t } X, \text{ const fqzechmat t } L, \text{ const fqzechmat t } B, \text{ int unit, const fqzechctx t } ctx)
\]

Sets \( X = L^{-1}B \) where \( L \) is a full rank lower triangular square matrix. If \( \text{unit} = 1 \), \( L \) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \( X \) and \( B \) are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

\[
\begin{pmatrix} A & 0 \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}X \\ D^{-1}(Y - CA^{-1}X) \end{pmatrix}
\]

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

\[
\text{void } \text{fqzechmat solve triu}(\text{fqzechmat t } X, \text{ const fqzechmat t } U, \text{ const fqzechmat t } B, \text{ int unit, const fqzechctx t } ctx)
\]

Sets \( X = U^{-1}B \) where \( U \) is a full rank upper triangular square matrix. If \( \text{unit} = 1 \), \( U \) is assumed to have ones on its main diagonal, and the main diagonal will not be read. \( X \) and \( B \) are allowed to
be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical
and recursive algorithms.

```c
void fq_zech_mat_solve_triu_classical(fq_zech_mat_t X, const fq_zech_mat_t U, const
                                      fq_zech_mat_t B, int unit, const fq_zech_ctx_t ctx)
```

Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If `unit` = 1, $U$ is assumed
to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed
to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```c
void fq_zech_mat_solve_triu_recursive(fq_zech_mat_t X, const fq_zech_mat_t U, const
                                       fq_zech_mat_t B, int unit, const fq_zech_ctx_t ctx)
```

Sets $X = U^{-1}B$ where $U$ is a full rank upper triangular square matrix. If `unit` = 1, $U$ is assumed
to have ones on its main diagonal, and the main diagonal will not be read. $X$ and $B$ are allowed
to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$
\begin{pmatrix}
A & B \\
0 & D
\end{pmatrix}^{-1}
\begin{pmatrix}
X \\
Y
\end{pmatrix} =
\begin{pmatrix}
A^{-1}(X - BD^{-1}Y) \\
D^{-1}Y
\end{pmatrix}
$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

11.21.15 Solving

```c
int fq_zech_mat_solve(fq_zech_mat_t X, const fq_zech_mat_t A, const fq_zech_mat_t B, const
                      fq_zech_ctx_t ctx)
```

Solves the matrix-matrix equation $AX = B$.

Returns 1 if $A$ has full rank; otherwise returns 0 and sets the elements of $X$ to undefined values.

The matrix $A$ must be square.

```c
int fq_zech_mat_can_solve(fq_zech_mat_t X, const fq_zech_mat_t A, const fq_zech_mat_t B,
                          const fq_zech_ctx_t ctx)
```

Solves the matrix-matrix equation $AX = B$ over $F_q$.

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of $X$ to zero. If more than
one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of $A$ and it may be singular.

11.21.16 Transforms

```c
void fq_zech_mat_similarity(fq_zech_mat_t M, slong r, fq_zech_t d, const fq_zech_ctx_t ctx)
```

 Applies a similarity transform to the $n \times n$ matrix $M$ in-place.

If $P$ is the $n \times n$ identity matrix the zero entries of whose row $r$ (0-indexed) have been replaced by
$d$, this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

The value $d$ is required to be reduced modulo the modulus of the entries in the matrix.
11.21.17 Characteristic polynomial

void fq_zech_mat_charpoly_danilevsky(fq_zech_poly_t p, const fq_zech_mat_t M, const fq_zech_ctx_t ctx)

Compute the characteristic polynomial $p$ of the matrix $M$. The matrix is assumed to be square.

void fq_zech_mat_charpoly(fq_zech_poly_t p, const fq_zech_mat_t M, const fq_zech_ctx_t ctx)

Compute the characteristic polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised.

11.21.18 Minimal polynomial

void fq_zech_mat_minpoly(fq_zech_poly_t p, const fq_zech_mat_t M, const fq_zech_ctx_t ctx)

Compute the minimal polynomial $p$ of the matrix $M$. The matrix is required to be square, otherwise an exception is raised.

11.22 fq_zech_poly.h – univariate polynomials over finite fields (Zech logarithm representation)

We represent a polynomial in $F_q[X]$ as a struct which includes an array coeffs with the coefficients, as well as the length length and the number alloc of coefficients for which memory has been allocated.

As a data structure, we call this polynomial normalised if the top coefficient is non-zero.

Unless otherwise stated here, all functions that deal with polynomials assume that the $F_q$ context of said polynomials are compatible, i.e., it assumes that the fields are generated by the same polynomial.

11.22.1 Types, macros and constants

type fq_zech_poly_struct

type fq_zech_poly_t

11.22.2 Memory management

void fq_zech_poly_init(fq_zech_poly_t poly, const fq_zech_ctx_t ctx)

Initialises poly for use, with context ctx, and setting its length to zero. A corresponding call to fq_zech_poly_clear() must be made after finishing with the fq_zech_poly_t to free the memory used by the polynomial.

void fq_zech_poly_init2(fq_zech_poly_t poly, slong alloc, const fq_zech_ctx_t ctx)

Initialises poly with space for at least alloc coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to fq_zech_poly_clear() must be made after finishing with the fq_zech_poly_t to free the memory used by the polynomial.

void fq_zech_poly_realloc(fq_zech_poly_t poly, slong alloc, const fq_zech_ctx_t ctx)

Reallocates the given polynomial to have space for alloc coefficients. If alloc is zero the polynomial is cleared and then reinitialised. If the current length is greater than alloc the polynomial is first truncated to length alloc.
void fq_zech_poly_fit_length(fq_zech_poly_t poly, slong len, const fq_zech_ctx_t ctx)
If len is greater than the number of coefficients currently allocated, then the polynomial is reallo-
cated to have space for at least len coefficients. No data is lost when calling this function.

The function efficiently deals with the case where fit_length is called many times in small in-
crements by at least doubling the number of allocated coefficients when length is larger than the
number of coefficients currently allocated.

void _fq_zech_poly_set_length(fq_zech_poly_t poly, slong newlen, const fq_zech_ctx_t ctx)
Sets the coefficients of poly beyond len to zero and sets the length of poly to len.

void fq_zech_poly_clear(fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be
used again.

void _fq_zech_poly_normalise(fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
Sets the length of poly so that the top coefficient is non-zero. If all coefficients are zero, the length
is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void _fq_zech_poly_normalise2(const fq_zech_struct *poly, slong *length, const fq_zech_ctx_t ctx)
Sets the length length of (poly,length) so that the top coefficient is non-zero. If all coefficients
are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee
normalisation.

void fq_zech_poly_truncate(fq_zech_poly_t poly, slong newlen, const fq_zech_ctx_t ctx)
Truncates the polynomial to length at most n.

void fq_zech_poly_set_trunc(fq_zech_poly_t poly1, fq_zech_poly_t poly2, slong newlen, const
fq_zech_ctx_t ctx)
Sets poly1 to poly2 truncated to length n.

void _fq_zech_poly_reverse(fq_zech_struct *output, const fq_zech_struct *input, slong len, slong
m, const fq_zech_ctx_t ctx)
Sets output to the reverse of input, which is of length len, but thinking of it as a polynomial of
length m, notionally zero-padded if necessary. The length m must be non-negative, but there are no
other restrictions. The polynomial output must have space for m coefficients.

void fq_zech_poly_reverse(fq_zech_poly_t output, const fq_zech_poly_t input, slong m, const
fq_zech_ctx_t ctx)
Sets output to the reverse of input, thinking of it as a polynomial of length m, notionally zero-
padded if necessary). The length m must be non-negative, but there are no other restrictions. The
output polynomial will be set to length m and then normalised.

11.22.3 Polynomial parameters

slong fq_zech_poly_degree(const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
Returns the degree of the polynomial poly.

slong fq_zech_poly_length(const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
Returns the length of the polynomial poly.

fq_zech_struct *fq_zech_poly_lead(const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
Returns a pointer to the leading coefficient of poly, or NULL if poly is the zero polynomial.
11.22.4 Randomisation

```c
void fq_zech_poly_randtest(fq_zech_poly_t f, flint_rand_t state, slong len, const fq_zech_ctx_t ctx)
```
Sets \( f \) to a random polynomial of length at most \( \text{len} \) with entries in the field described by \( \text{ctx} \).

```c
void fq_zech_poly_randtest_not_zero(fq_zech_poly_t f, flint_rand_t state, slong len, const fq_zech_ctx_t ctx)
```
Same as `fq_zech_poly_randtest` but guarantees that the polynomial is not zero.

```c
void fq_zech_poly_randtest_monic(fq_zech_poly_t f, flint_rand_t state, slong len, const fq_zech_ctx_t ctx)
```
Sets \( f \) to a random monic polynomial of length \( \text{len} \) with entries in the field described by \( \text{ctx} \).

```c
void fq_zech_poly_randtest_irreducible(fq_zech_poly_t f, flint_rand_t state, slong len, const fq_zech_ctx_t ctx)
```
Sets \( f \) to a random monic, irreducible polynomial of length \( \text{len} \) with entries in the field described by \( \text{ctx} \).

11.22.5 Assignment and basic manipulation

```c
void _fq_zech_poly_set(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
```
Sets \((\text{rop}, \text{len})\) to \((\text{op}, \text{len})\).

```c
void fq_zech_poly_set(fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const fq_zech_ctx_t ctx)
```
Sets the polynomial \( \text{poly1} \) to the polynomial \( \text{poly2} \).

```c
void fq_zech_poly_set_fq_zech(fq_zech_poly_t poly, const fq_zech_t c, const fq_zech_ctx_t ctx)
```
Sets the polynomial \( \text{poly} \) to \( \text{c} \).

```c
void fq_zech_poly_set_fmpz_mod_poly(fq_zech_poly_t rop, const fmpz_mod_poly_t op, const fq_zech_ctx_t ctx)
```
Sets the polynomial \( \text{rop} \) to the polynomial \( \text{op} \).

```c
void fq_zech_poly_set_nmod_poly(fq_zech_poly_t rop, const nmod_poly_t op, const fq_zech_ctx_t ctx)
```
Sets the polynomial \( \text{rop} \) to the polynomial \( \text{op} \).

```c
void fq_zech_poly_swap(fq_zech_poly_t op1, fq_zech_poly_t op2, const fq_zech_ctx_t ctx)
```
Swaps the two polynomials \( \text{op1} \) and \( \text{op2} \).

```c
void _fq_zech_poly_zero(fq_zech_struct *rop, slong len, const fq_zech_ctx_t ctx)
```
Sets \((\text{rop}, \text{len})\) to the zero polynomial.

```c
void fq_zech_poly_zero(fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
```
Sets \( \text{poly} \) to the zero polynomial.

```c
void fq_zech_poly_one(fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
```
Sets \( \text{poly} \) to the constant polynomial \( 1 \).

```c
void fq_zech_poly_gen(fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
```
Sets \( \text{poly} \) to the polynomial \( x \).

```c
void fq_zech_poly_make_monic(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```
Sets \( \text{rop} \) to \( \text{op} \), normed to have leading coefficient \( 1 \).
void _fq_zech_poly_make_monic(fq_zech_struct *rop, const fq_zech_struct *op, slong length, const fq_zech_ctx_t ctx)

Sets rop to (op,length), normed to have leading coefficient 1. Assumes that rop has enough space for the polynomial, assumes that op is not zero (and thus has an invertible leading coefficient).

11.22.6 Getting and setting coefficients

void fq_zech_poly_get_coeff(fq_zech_t x, const fq_zech_poly_t poly, slong n, const fq_zech_ctx_t ctx)

Sets x to the coefficient of $X^n$ in poly.

void fq_zech_poly_set_coeff(fq_zech_poly_t poly, slong n, const fq_zech_t x, const fq_zech_ctx_t ctx)

Sets the coefficient of $X^n$ in poly to x.

void fq_zech_poly_set_coeff_fmpz(fq_zech_poly_t poly, slong n, const fmpz_t x, const fq_zech_ctx_t ctx)

Sets the coefficient of $X^n$ in the polynomial to x, assuming $n \geq 0$.

11.22.7 Comparison

int fq_zech_poly_equal(const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const fq_zech_ctx_t ctx)

Returns nonzero if the two polynomials poly1 and poly2 are equal, otherwise return zero.

int fq_zech_poly_equal_trunc(const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, slong n, const fq_zech_ctx_t ctx)

Notionally truncate poly1 and poly2 to length n and return nonzero if they are equal, otherwise return zero.

int fq_zech_poly_is_zero(const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)

Returns whether the polynomial poly is the zero polynomial.

int fq_zech_poly_is_one(const fq_zech_poly_t op, const fq_zech_ctx_t ctx)

Returns whether the polynomial poly is equal to the constant polynomial 1.

int fq_zech_poly_is_gen(const fq_zech_poly_t op, const fq_zech_ctx_t ctx)

Returns whether the polynomial poly is equal to the polynomial $x$.

int fq_zech_poly_is_unit(const fq_zech_poly_t op, const fq_zech_ctx_t ctx)

Returns whether the polynomial poly is a unit in the polynomial ring $F_q[X]$, i.e. if it has degree 0 and is non-zero.

int fq_zech_poly_equal_fq_zech(const fq_zech_poly_t poly, const fq_zech_t c, const fq_zech_ctx_t ctx)

Returns whether the polynomial poly is equal the (constant) $F_q$ element c
11.22.8 Addition and subtraction

```c
void _fq_zech_poly_add(fq_zech_struct *res, const fq_zech_struct *poly1, slong len1, const fq_zech_struct *poly2, slong len2, const fq_zech_ctx_t ctx)
```

Sets res to the sum of (poly1,len1) and (poly2,len2).

```c
void fq_zech_poly_add(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const fq_zech_ctx_t ctx)
```

Sets res to the sum of poly1 and poly2.

```c
void fq_zech_poly_add_si(fq_zech_poly_t res, const fq_zech_poly_t poly1, slong c, const fq_zech_ctx_t ctx)
```

Sets res to the sum of poly1 and c.

```c
void fq_zech_poly_add_series(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, slong n, const fq_zech_ctx_t ctx)
```

Notionally truncate poly1 and poly2 to length n and set res to the sum.

```c
void _fq_zech_poly_sub(fq_zech_struct *res, const fq_zech_struct *poly1, slong len1, const fq_zech_struct *poly2, slong len2, const fq_zech_ctx_t ctx)
```

Sets res to the difference of (poly1,len1) and (poly2,len2).

```c
void fq_zech_poly_sub(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const fq_zech_ctx_t ctx)
```

Sets res to the difference of poly1 and poly2.

```c
void fq_zech_poly_sub_series(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, slong n, const fq_zech_ctx_t ctx)
```

Notionally truncate poly1 and poly2 to length n and set res to the difference.

```c
void _fq_zech_poly_neg(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
```

Sets rop to the additive inverse of (op,len).

```c
void fq_zech_poly_neg(fq_zech_poly_t res, const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
```

Sets res to the additive inverse of poly.

11.22.9 Scalar multiplication and division

```c
void _fq_zech_poly_scalar_mul_fq_zech(fq_zech_struct *res, const fq_zech_struct *op, slong len, const fq_zech_t x, const fq_zech_ctx_t ctx)
```

Sets (rop,len) to the product of (op,len) by the scalar x, in the context defined by ctx.

```c
void fq_zech_poly_scalar_mul_fq_zech(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_t x, const fq_zech_ctx_t ctx)
```

Sets rop to the product of op by the scalar x, in the context defined by ctx.

```c
void _fq_zech_poly_scalar_addmul_fq_zech(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const fq_zech_t x, const fq_zech_ctx_t ctx)
```

Adds to (rop,len) the product of (op,len) by the scalar x, in the context defined by ctx. In particular, assumes the same length for op and rop.

```c
void fq_zech_poly_scalar_addmul_fq_zech(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_t x, const fq_zech_ctx_t ctx)
```

Adds to rop the product of op by the scalar x, in the context defined by ctx.
void _fq_zech_poly_scalar_submul_fq_zech(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const fq_zech_t x, const fq_zech_ctx_t ctx)

Subtracts from (rop,len) the product of (op,len) by the scalar x, in the context defined by ctx. In particular, assumes the same length for op and rop.

void fq_zech_poly_scalar_submul_fq_zech(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_t x, const fq_zech_ctx_t ctx)

Subtracts from rop the product of op by the scalar x, in the context defined by ctx.

void _fq_zech_poly_scalar_div_fq_zech(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const fq_zech_t x, const fq_zech_ctx_t ctx)

Sets (rop,len) to the quotient of (op,len) by the scalar x, in the context defined by ctx. An exception is raised if x is zero.

void fq_zech_poly_scalar_div_fq_zech(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_t x, const fq_zech_ctx_t ctx)

Sets rop to the quotient of op by the scalar x, in the context defined by ctx. An exception is raised if x is zero.

11.22.10 Multiplication

void _fq_zech_poly_mul_classical(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1, const fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), assuming that len1 is at least len2 and neither is zero.

Permits zero padding. Does not support aliasing of rop with either op1 or op2.

void fq_zech_poly_mul_classical(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t op2, const fq_zech_ctx_t ctx)

Sets rop to the product of op1 and op2 using classical polynomial multiplication.

void _fq_zech_poly_mul reorder(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1, const fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), assuming that len1 and len2 are non-zero.

Permits zero padding. Supports aliasing.

void fq_zech_poly_mul reorder(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t op2, const fq_zech_ctx_t ctx)

Sets rop to the product of op1 and op2, reordering the two indeterminates X and Y when viewing the polynomials as elements of F_p[X,Y].

Suppose F_q = F_p[X]/(f(X)) and recall that elements of F_q are internally represented by elements of type fmpz_poly. For small degree extensions but polynomials in F_q[Y] of large degree n, we change the representation to

\[ g(Y) = \sum_{i=0}^{n} a_i(X)Y^i \]

\[ = \sum_{j=0}^{d} \sum_{i=0}^{n} \text{Coeff}(a_i(X),j)Y^i. \]

This allows us to use a poor algorithm (such as classical multiplication) in the X-direction and leverage the existing fast integer multiplication routines in the Y-direction where the polynomial degree n is large.
void \_fq\_zech\_poly\_mul\_KS(fq\_zech\_struct \*rop, const fq\_zech\_struct \*op1, slong len1, const fq\_zech\_struct \*op2, slong len2, const fq\_zech\_ctx\_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2).

Permits zero padding and places no assumptions on the lengths len1 and len2. Supports aliasing.

void \_fq\_zech\_poly\_mul\_KS(fq\_zech\_poly\_t rop, const fq\_zech\_poly\_t op1, const fq\_zech\_poly\_t op2, const fq\_zech\_ctx\_t ctx)

Sets rop to the product of op1 and op2 using Kronecker substitution, that is, by encoding each coefficient in \( \mathbb{F}_q \) as an integer and reducing this problem to multiplying two polynomials over the integers.

void \_fq\_zech\_poly\_mul(fq\_zech\_struct \*rop, const fq\_zech\_struct \*op1, slong len1, const fq\_zech\_struct \*op2, slong len2, const fq\_zech\_ctx\_t ctx)

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

void \_fq\_zech\_poly\_mul(fq\_zech\_poly\_t rop, const fq\_zech\_poly\_t op1, const fq\_zech\_poly\_t op2, const fq\_zech\_ctx\_t ctx)

Sets rop to the product of op1 and op2, choosing an appropriate algorithm.

void \_fq\_zech\_poly\_mullow\_classical(fq\_zech\_struct \*rop, const fq\_zech\_struct \*op1, slong len1, const fq\_zech\_struct \*op2, slong len2, slong n, const fq\_zech\_ctx\_t ctx)

Sets (rop, n) to the first n coefficients of (op1, len1) multiplied by (op2, len2).

Assumes 0 < n <= len1 + len2 - 1. Assumes neither len1 nor len2 is zero.

void \_fq\_zech\_poly\_mullow\_classical(fq\_zech\_poly\_t rop, const fq\_zech\_poly\_t op1, const fq\_zech\_poly\_t op2, slong n, const fq\_zech\_ctx\_t ctx)

Sets rop to the product of op1 and op2, computed using the classical or schoolbook method.

void \_fq\_zech\_poly\_mullow\_KS(fq\_zech\_struct \*rop, const fq\_zech\_struct \*op1, slong len1, const fq\_zech\_struct \*op2, slong len2, slong n, const fq\_zech\_ctx\_t ctx)

Sets (rop, n) to the lowest n coefficients of the product of (op1, len1) and (op2, len2).

Assumes that len1 and len2 are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between rop, op1 and op2.

void \_fq\_zech\_poly\_mullow\_KS(fq\_zech\_poly\_t rop, const fq\_zech\_poly\_t op1, const fq\_zech\_poly\_t op2, slong n, const fq\_zech\_ctx\_t ctx)

Sets rop to the product of op1 and op2.

void \_fq\_zech\_poly\_mul\_classical(fq\_zech\_struct \*res, const fq\_zech\_struct \*poly1, slong len1, const fq\_zech\_struct \*poly2, slong len2, slong start, const fq\_zech\_ctx\_t ctx)

Computes the product of (poly1, len1) and (poly2, len2) and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but
reduced. Assumes that $\text{len1} \geq \text{len2} > 0$. Aliasing of inputs and output is not permitted. Algorithm is classical multiplication.

```c
void fq_zech_poly_mulhigh_classical(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, slong start, const fq_zech_ctx_t ctx)
```

Computes the product of `poly1` and `poly2` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced. Algorithm is classical multiplication.

```c
void _fq_zech_poly_mulhigh(fq_zech_struct *res, const fq_zech_struct *poly1, slong len1, const fq_zech_struct *poly2, slong len2, slong start, const fq_zech_ctx_t ctx)
```

Computes the product of `(poly1, len1)` and `(poly2, len2)` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced. Assumes that $\text{len1} \geq \text{len2} > 0$. Aliasing of inputs and output is not permitted.

```c
void fq_zech_poly_mulhigh(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, slong start, const fq_zech_ctx_t ctx)
```

Computes the product of `poly1` and `poly2` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced.

```c
void _fq_zech_poly_mulmod(fq_zech_struct *res, const fq_zech_struct *poly1, slong len1, const fq_zech_struct *poly2, slong len2, const fq_zech_struct *f, slong lenf, const fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that $\text{len1} + \text{len2} - \text{lenf} > 0$, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fq_zech_poly_mul` instead.

Aliasing of `f` and `res` is not permitted.

```c
void fq_zech_poly_mulmod(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```c
void _fq_zech_poly_mulmod_preinv(fq_zech_struct *res, const fq_zech_struct *poly1, slong len1, const fq_zech_struct *poly2, slong len2, const fq_zech_struct *f, slong lenf, const fq_zech_struct *finv, slong lenfinv, const fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f mod x^{lenf}`.

Aliasing of `res` with any of the inputs is not permitted.

```c
void fq_zech_poly_mulmod_preinv(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const fq_zech_poly_t finv, const fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`.

### 11.22.11 Squaring

```c
void _fq_zech_poly_sqr_classical(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `(op, len)` is not zero and using classical polynomial multiplication.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`. 
void fq_zech_poly_sqr_classical(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_ctx_t ctx)

Sets rop to the square of op using classical polynomial multiplication.

void _fq_zech_poly_sqr_KS(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)

Sets (rop, 2*len - 1) to the square of (op, len).
Permits zero padding and places no assumptions on the lengths len1 and len2. Supports aliasing.

void fq_zech_poly_sqr_KS(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_ctx_t ctx)

Sets rop to the square of op using Kronecker substitution, that is, by encoding each coefficient in \( F_\mathbb{Q} \) as an integer and reducing this problem to multiplying two polynomials over the integers.

void _fq_zech_poly_sqr(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)

Sets (rop, 2 * len - 1) to the square of (op, len), choosing an appropriate algorithm.
Permits zero padding. Does not support aliasing.

void fq_zech_poly_sqr(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_ctx_t ctx)

Sets rop to the square of op, choosing an appropriate algorithm.

11.22.12 Powering

void _fq_zech_poly_pow(fq_zech_struct *rop, const fq_zech_struct *op, slong len, ulong e, const fq_zech_ctx_t ctx)

Sets rop = op^e, assuming that e, len > 0 and that res has space for e*(len - 1) + 1 coefficients. Does not support aliasing.

void fq_zech_poly_pow(fq_zech_poly_t rop, const fq_zech_poly_t op, ulong e, const fq_zech_ctx_t ctx)

Computes rop = op^e. If e is zero, returns one, so that in particular 0^0 = 1.

void _fq_zech_poly_powmod_ui_binexp(fq_zech_struct *res, const fq_zech_struct *poly, ulong e, const fq_zech_struct *f, slong lenf, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0.
We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_zech_poly_powmod_ui_binexp(fq_zech_poly_t res, const fq_zech_poly_t poly, ulong e, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0.

void _fq_zech_poly_powmod_ui_binexp_preinv(fq_zech_struct *res, const fq_zech_struct *poly, ulong e, const fq_zech_struct *f, slong lenf, const fq_zech_struct *finv, slong lenfinv, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0.
We require finv to be the inverse of the reverse of f.
We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.
void fq_zech_poly_powmod_ui_binexp_preinv(fq_zech_poly_t res, const fq_zech_poly_t poly, ulong e, const fq_zech_poly_t f, const fq_zech_poly_t finv, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require finv to be the inverse of the reverse of f.

void _fq_zech_poly_powmod_fmpz_binexp(fq_zech_struct *res, const fq_zech_struct *poly, const fmpz_t e, const fq_zech_struct *f, slong lenf, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_zech_poly_powmod_fmpz_binexp(fq_zech_poly_t res, const fq_zech_poly_t poly, const fmpz_t e, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void _fq_zech_poly_powmod_fmpz_binexp_preinv(fq_zech_struct *res, const fq_zech_struct *poly, const fmpz_t e, const fq_zech_struct *f, slong lenf, const fq_zech_struct *finv, slong lenfinv, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require finv to be the inverse of the reverse of f. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_zech_poly_powmod_fmpz_binexp_preinv(fq_zech_poly_t res, const fq_zech_poly_t poly, const fmpz_t e, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void _fq_zech_poly_powmod_fmpz_sliding_preinv(fq_zech_struct *res, const fq_zech_struct *poly, const fmpz_t e, ulong k, const fq_zech_struct *f, slong lenf, const fq_zech_struct *finv, slong lenfinv, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using sliding-window exponentiation with window size k. We require e > 0. We require finv to be the inverse of the reverse of f. If k is set to zero, then an “optimum” size will be selected automatically base on e. We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

void fq_zech_poly_powmod_fmpz_sliding_preinv(fq_zech_poly_t res, const fq_zech_poly_t poly, const fmpz_t e, ulong k, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Sets res to poly raised to the power e modulo f, using sliding-window exponentiation with window size k. We require e >= 0. We require finv to be the inverse of the reverse of f. If k is set to zero, then an “optimum” size will be selected automatically base on e.

void _fq_zech_poly_powmod_x_fmpz_preinv(fq_zech_struct *res, const fmpz_t e, const fq_zech_struct *f, slong lenf, const fq_zech_struct *finv, slong lenfinv, const fq_zech_ctx_t ctx)
Sets \( \texttt{res} \) to \( x \) raised to the power \( e \) modulo \( f \), using sliding window exponentiation. We require \( e > 0 \). We require \( \texttt{finv} \) to be the inverse of the reverse of \( f \).

We require \( \texttt{lenf} > 2 \). The output \( \texttt{res} \) must have room for \( \texttt{lenf} - 1 \) coefficients.

```c
void \_fq\_zech\_poly\_pow\_trunc\_binexp(fq\_zech\_struct *\texttt{res}, const fq\_zech\_struct *\texttt{poly}, ulong \texttt{e}, slong \texttt{trunc}, const fq\_zech\_ctx_t \texttt{ctx})
```

Sets \( \texttt{res} \) to the low \( \texttt{trunc} \) coefficients of \( \texttt{poly} \) (assumed to be zero padded if necessary to length \( \texttt{trunc} \)) to the power \( e \). This is equivalent to doing a powering followed by a truncation. We require that \( \texttt{res} \) has enough space for \( \texttt{trunc} \) coefficients, that \( \texttt{trunc} > 0 \) and that \( e > 1 \). Aliasing is not permitted. Uses the binary exponentiation method.

```c
void \texttt{fq\_zech\_poly\_pow\_trunc}\_binexp(fq\_zech\_poly_t \texttt{res}, const fq\_zech\_poly_t \texttt{poly}, ulong \texttt{e}, slong \texttt{trunc}, const fq\_zech\_ctx_t \texttt{ctx})
```

Sets \( \texttt{res} \) to the low \( \texttt{trunc} \) coefficients of \( \texttt{poly} \) to the power \( e \). This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.

```c
void \_fq\_zech\_poly\_pow\_trunc(fq\_zech\_struct *\texttt{res}, const fq\_zech\_struct *\texttt{poly}, ulong \texttt{e}, slong \texttt{trunc}, const fq\_zech\_ctx_t \texttt{ctx})
```

Sets \( \texttt{res} \) to the low \( \texttt{trunc} \) coefficients of \( \texttt{poly} \) (assumed to be zero padded if necessary to length \( \texttt{trunc} \)) to the power \( e \). This is equivalent to doing a powering followed by a truncation. We require that \( \texttt{res} \) has enough space for \( \texttt{trunc} \) coefficients, that \( \texttt{trunc} > 0 \) and that \( e > 1 \). Aliasing is not permitted.

```c
void \texttt{fq\_zech\_poly\_pow\_trunc}(fq\_zech\_poly_t \texttt{res}, const fq\_zech\_poly_t \texttt{poly}, ulong \texttt{e}, slong \texttt{trunc}, const fq\_zech\_ctx_t \texttt{ctx})
```

Sets \( \texttt{res} \) to the low \( \texttt{trunc} \) coefficients of \( \texttt{poly} \) to the power \( e \). This is equivalent to doing a powering followed by a truncation.

11.22.13 Shifting

```c
void \_fq\_zech\_poly\_shift\_left(fq\_zech\_struct *\texttt{rop}, const fq\_zech\_struct *\texttt{op}, slong \texttt{len}, slong \texttt{n}, const fq\_zech\_ctx_t \texttt{ctx})
```

Sets \( \texttt{rop} \), \( \texttt{len} + \texttt{n} \) to \( \texttt{op} \), \( \texttt{len} \) shifted left by \( n \) coefficients.

Inserts zero coefficients at the lower end. Assumes that \( \texttt{len} \) and \( n \) are positive, and that \( \texttt{rop} \) fits \( \texttt{len} + \texttt{n} \) elements. Supports aliasing between \( \texttt{rop} \) and \( \texttt{op} \).

```c
void \texttt{fq\_zech\_poly\_shift\_left}(fq\_zech\_poly_t \texttt{rop}, const fq\_zech\_poly_t \texttt{op}, slong \texttt{n}, const fq\_zech\_ctx_t \texttt{ctx})
```

Sets \( \texttt{rop} \) to \( \texttt{op} \) shifted left by \( n \) coeffs. Zero coefficients are inserted.

```c
void \_fq\_zech\_poly\_shift\_right(fq\_zech\_struct *\texttt{rop}, const fq\_zech\_struct *\texttt{op}, slong \texttt{len}, slong \texttt{n}, const fq\_zech\_ctx_t \texttt{ctx})
```

Sets \( \texttt{rop} \), \( \texttt{len} - \texttt{n} \) to \( \texttt{op} \), \( \texttt{len} \) shifted right by \( n \) coefficients.

Assumes that \( \texttt{len} \) and \( n \) are positive, that \( \texttt{len} > \texttt{n} \), and that \( \texttt{rop} \) fits \( \texttt{len} - \texttt{n} \) elements. Supports aliasing between \( \texttt{rop} \) and \( \texttt{op} \), although in this case the top coefficients of \( \texttt{op} \) are not set to zero.

```c
void \texttt{fq\_zech\_poly\_shift\_right}(fq\_zech\_poly_t \texttt{rop}, const fq\_zech\_poly_t \texttt{op}, slong \texttt{n}, const fq\_zech\_ctx_t \texttt{ctx})
```

Sets \( \texttt{rop} \) to \( \texttt{op} \) shifted right by \( n \) coefficients. If \( n \) is equal to or greater than the current length of \( \texttt{op} \), \( \texttt{rop} \) is set to the zero polynomial.

11.22. \texttt{fq\_zech\_poly.h} – univariate polynomials over finite fields (Zech logarithm representation)
11.22.14 Norms

```c
slong _fq_zech_poly_hamming_weight(const fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
Returns the number of non-zero entries in (op, len).
```

```c
slong fq_zech_poly_hamming_weight(const fq_zech_poly_t op, const fq_zech_ctx_t ctx)
Returns the number of non-zero entries in the polynomial op.
```

11.22.15 Euclidean division

```c
void _fq_zech_poly_divrem(fq_zech_struct *Q, fq_zech_struct *R, const fq_zech_struct *A, slong lenA, const fq_zech_struct *B, slong lenB, const fq_zech_t invB, const fq_zech_ctx_t ctx)
Computes (Q, lenA - lenB + 1), (R, lenA) such that A = BQ + R with 0 ≤ len(R) < len(B).
Assumes that the leading coefficient of B is invertible and that invB is its inverse.
Assumes that len(A), len(B) > 0. Allows zero-padding in (A, lenA). R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.
```

```c
void fq_zech_poly_divrem(fq_zech_poly_t Q, fq_zech_poly_t R, const fq_zech_poly_t A, const fq_zech_poly_t B, const fq_zech_ctx_t ctx)
Computes Q, R such that A = BQ + R with 0 ≤ len(R) < len(B).
Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and f(X) is irreducible.
```

```c
void fq_zech_poly_divrem_f(fq_zech_t f, fq_zech_poly_t Q, fq_zech_poly_t R, const fq_zech_poly_t A, const fq_zech_poly_t B, const fq_zech_ctx_t ctx)
Either finds a non-trivial factor f of the modulus of ctx, or computes Q, R such that A = BQ + R and 0 ≤ len(R) < len(B).
If the leading coefficient of B is invertible, the division with remainder operation is carried out, Q and R are computed correctly, and f is set to 1. Otherwise, f is set to a non-trivial factor of the modulus and Q and R are not touched.
Assumes that B is non-zero.
```

```c
void _fq_zech_poly_rem(fq_zech_struct *R, const fq_zech_struct *A, slong lenA, const fq_zech_struct *B, slong lenB, const fq_zech_t invB, const fq_zech_ctx_t ctx)
Sets R to the remainder of the division of (A, lenA) by (B, lenB). Assumes that the leading coefficient of (B, lenB) is invertible and that invB is its inverse.
```

```c
void fq_zech_poly_rem(fq_zech_poly_t R, const fq_zech_poly_t A, const fq_zech_poly_t B, const fq_zech_ctx_t ctx)
Sets R to the remainder of the division of A by B in the context described by ctx.
```

```c
void _fq_zech_poly_div(fq_zech_struct *Q, const fq_zech_struct *A, slong lenA, const fq_zech_struct *B, slong lenB, const fq_zech_t invB, const fq_zech_ctx_t ctx)
Notationally, computes Q, R such that A = BQ + R with 0 ≤ len(R) < len(B) but only sets (Q, lenA - lenB + 1).
Allows zero-padding in A but not in B. Assumes that the leading coefficient of B is a unit.
```

```c
void fq_zech_poly_div(fq_zech_poly_t Q, const fq_zech_poly_t A, const fq_zech_poly_t B, const fq_zech_ctx_t ctx)
Notionally finds polynomials Q and R such that A = BQ + R with len(R) < len(B), but returns only Q. If len(B) = 0 an exception is raised.
```
void fq_zech_poly_inv_series_newton(fq_zech_struct *Q, const fq_zech_struct *A, slong lenA, const fq_zech_struct *B, slong lenB, const fq_zech_struct *Binv, slong lenBinv, const fq_zech_ctx_t ctx)

Notionally computes polynomials \( Q \) and \( R \) such that \( A = BQ + R \) with \( \text{len}(R) \) less than \( \text{len}(B) \), where \( A \) is of length \( \text{len}(A) \) and \( B \) is of length \( \text{len}(B) \), but return only \( Q \).

We require that \( Q \) have space for \( \text{len}(A) - \text{len}(B) + 1 \) coefficients and assume that the leading coefficient of \( B \) is a unit. Furthermore, we assume that \( \text{Binv} \) is the inverse of the reverse of \( B \) mod \( x^{\text{len}(B)} \).

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

void fq_zech_poly_div_newton_n_preinv(fq_zech_poly_t Q, const fq_zech_poly_t A, const fq_zech_poly_t B, const fq_zech_poly_t Binv, const fq_zech_ctx_t ctx)

Computes \( Q \) and \( R \) such that \( A = BQ + R \) with \( \text{len}(R) < \text{len}(B) \), but returns only \( Q \).

We assume that the leading coefficient of \( B \) is a unit and that \( \text{Binv} \) is the inverse of the reverse of \( B \) mod \( x^{\text{len}(B)} \).

It is required that the length of \( A \) is less than or equal to \( 2 \times \text{length of } B - 2 \).

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

void _fq_zech_poly_divrem_newton_n_preinv(fq_zech_struct *Q, fq_zech_struct *R, const fq_zech_struct *A, slong lenA, const fq_zech_struct *B, slong lenB, const fq_zech_struct *Binv, slong lenBinv, const fq_zech_ctx_t ctx)

Computes \( Q \) and \( R \) such that \( A = BQ + R \) with \( \text{len}(R) \) less than \( \text{len}(B) \), where \( A \) is of length \( \text{len}(A) \) and \( B \) is of length \( \text{len}(B) \). We require that \( Q \) have space for \( \text{len}(A) - \text{len}(B) + 1 \) coefficients.

Furthermore, we assume that \( \text{Binv} \) is the inverse of the reverse of \( B \) mod \( x^{\text{len}(B)} \). The algorithm used is to call \text{div\_newton\_preinv}() and then multiply out and compute the remainder.

void fq_zech_poly_divrem_newton_n_preinv(fq_zech_poly_t Q, const fq_zech_poly_t A, const fq_zech_poly_t B, const fq_zech_poly_t Binv, const fq_zech_ctx_t ctx)

Computes \( Q \) and \( R \) such that \( A = BQ + R \) with \( \text{len}(R) < \text{len}(B) \). We assume \( \text{Binv} \) is the inverse of the reverse of \( B \) mod \( x^{\text{len}(B)} \).

It is required that the length of \( A \) is less than or equal to \( 2 \times \text{length of } B - 2 \).

The algorithm used is to call \text{div\_newton}() and then multiply out and compute the remainder.

void _fq_zech_poly_inv_series_newton(fq_zech_struct *Qinv, const fq_zech_struct *Q, slong n, const fq_zech_t cinv, const fq_zech_ctx_t ctx)

Given \( Q \) of length \( n \) whose constant coefficient is invertible modulo the given modulus, find a polynomial \( \text{Qinv} \) of length \( n \) such that \( Q \times \text{Qinv} \) is 1 modulo \( x^n \). Requires \( n > 0 \). This function can be viewed as inverting a power series via Newton iteration.

void fq_zech_poly_inv_series_newton(fq_zech_poly_t Qinv, const fq_zech_poly_t Q, slong n, const fq_zech_ctx_t ctx)

Given \( Q \) find \( \text{Qinv} \) such that \( Q \times \text{Qinv} \) is 1 modulo \( x^n \). The constant coefficient of \( Q \) must be invertible modulo the modulus of \( Q \). An exception is raised if this is not the case or if \( n = 0 \). This function can be viewed as inverting a power series via Newton iteration.

void _fq_zech_poly_inv_series(fq_zech_struct *Qinv, const fq_zech_struct *Q, slong n, const fq_zech_t cinv, const fq_zech_ctx_t ctx)

Given \( Q \) of length \( n \) whose constant coefficient is invertible modulo the given modulus, find a polynomial \( \text{Qinv} \) of length \( n \) such that \( Q \times \text{Qinv} \) is 1 modulo \( x^n \). Requires \( n > 0 \).
11.22.16 Greatest common divisor

void fq_zech_poly_gcd(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t op2, const fq_zech_ctx_t ctx)

Sets rop to the greatest common divisor of op1 and op2, using the either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

slong _fq_zech_poly_gcd(fq_zech_struct *G, const fq_zech_struct *A, slong lenA, const fq_zech_struct *B, slong lenB, const fq_zech_ctx_t ctx)

Computes the GCD of A of length lenA and B of length lenB, where lenA >= lenB > 0 and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for lenB coefficients.

slong _fq_zech_poly_gcd_euclidean_f(fq_zech_t f, fq_zech_struct *G, const fq_zech_struct *A, slong lenA, const fq_zech_struct *B, slong lenB, const fq_zech_ctx_t ctx)

Either sets f = 1 and G to the greatest common divisor of (A,len(A)) and (B,len(B)) and returns its length, or sets f to a non-trivial factor of the modulus of ctx and leaves the contents of the vector (G,lenB) undefined.

Assumes that len(A) >= len(B) > 0 and that the vector G has space for sufficiently many coefficients.

void fq_zech_poly_gcd_euclidean_f(fq_zech_t f, fq_zech_poly_t G, const fq_zech_poly_t A, const fq_zech_poly_t B, const fq_zech_ctx_t ctx)

Either sets f = 1 and G to the greatest common divisor of A and B or sets f to a factor of the modulus of ctx.

slong _fq_zech_poly_xgcd(fq_zech_struct *G, fq_zech_struct *S, fq_zech_struct *T, const fq_zech_struct *A, slong lenA, const fq_zech_struct *B, slong lenB, const fq_zech_ctx_t ctx)

Computes the GCD of A and B together with cofactors S and T such that SA+TB = G. Returns the length of G.

Assumes that len(A) >= len(B) >= 1 and (len(A),len(B)) ≠ (1,1).

No attempt is made to make the GCD monic.

Requires that G have space for len(B) coefficients. Writes len(B) – 1 and len(A) – 1 coefficients to S and T, respectively. Note that, in fact, len(S) ≤ max(len(B) – len(G),1) and len(T) ≤ max(len(A) – len(G),1).

No aliasing of input and output operands is permitted.
void fq_zech_poly_xgcd(fq_zech_poly_t G, fq_zech_poly_t S, fq_zech_poly_t T, const
fq_zech_poly_t A, const fq_zech_poly_t B, const fq_zech_ctx_t ctx)

Computes the GCD of $A$ and $B$. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial $P$ is defined to be $P$. Except in the case where the GCD is zero, the GCD $G$ is made monic.

Polynomials $S$ and $T$ are computed such that $S*A + T*B = G$. The length of $S$ will be at most $\text{lenB}$ and the length of $T$ will be at most $\text{lenA}$.

int _fq_zech_poly_divides(fq_zech_struct *Q, const fq_zech_struct *A, slong lenA, const
fq_zech_struct *B, slong lenB, const fq_zech_ctx_t ctx)

Returns 1 if $(B, \text{lenB})$ divides $(A, \text{lenA})$ exactly and sets $Q$ to the quotient, otherwise returns 0.

It is assumed that $\text{len}(A) \geq \text{len}(B) > 0$ and that $Q$ has space for $\text{len}(A) - \text{len}(B) + 1$ coefficients.

This function is currently unoptimised and provided for convenience only.

int fq_zech_poly_divides(fq_zech_poly_t Q, const fq_zech_poly_t A, const fq_zech_poly_t B,
const fq_zech_ctx_t ctx)

Returns 1 if $B$ divides $A$ exactly and sets $Q$ to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.
11.22.18 Derivative

void _fq_zech_poly_derivative(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)

Sets \((\text{rop}, \text{len} - 1)\) to the derivative of \((\text{op}, \text{len})\). Also handles the cases where \(\text{len}\) is 0 or 1 correctly. Supports aliasing of \text{rop} and \text{op}.

void fq_zech_poly_derivative(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_ctx_t ctx)

Sets \text{rop} to the derivative of \text{op}.

11.22.19 Square root

void _fq_zech_poly_invsqrt_series(fq_zech_struct *g, const fq_zech_struct *h, slong n, const fq_zech_ctx_t ctx)

Set the first \(n\) terms of \(g\) to the series expansion of \(1/\sqrt{h}\). It is assumed that \(n > 0\), that \(h\) has constant term 1 and that \(h\) is zero-padded as necessary to length \(n\). Aliasing is not permitted.

void fq_zech_poly_invsqrt_series(fq_zech_poly_t g, const fq_zech_poly_t h, slong n, const fq_zech_ctx_t ctx)

Set \text{g} to the series expansion of \(1/\sqrt{h}\) to order \(O(x^n)\). It is assumed that \(h\) has constant term 1.

void _fq_zech_poly_sqrt_series(fq_zech_struct *g, const fq_zech_struct *h, slong n, const fq_zech_ctx_t ctx)

Set the first \(n\) terms of \(g\) to the series expansion of \(\sqrt{h}\). It is assumed that \(n > 0\), that \(h\) has constant term 1 and that \(h\) is zero-padded as necessary to length \(n\). Aliasing is not permitted.

void fq_zech_poly_sqrt_series(fq_zech_poly_t g, const fq_zech_poly_t h, slong n, const fq_zech_ctx_t ctx)

Set \text{g} to the series expansion of \(\sqrt{h}\) to order \(O(x^n)\). It is assumed that \(h\) has constant term 1.

int _fq_zech_poly_sqrt(fq_zech_struct *s, const fq_zech_struct *p, slong n, const fq_zech_ctx_t ctx)

If \((p, n)\) is a perfect square, sets \((s, n / 2 + 1)\) to a square root of \(p\) and returns 1. Otherwise returns 0.

int fq_zech_poly_sqrt(fq_zech_poly_t s, const fq_zech_poly_t p, const fq_zech_ctx_t ctx)

If \(p\) is a perfect square, sets \text{s} to a square root of \(p\) and returns 1. Otherwise returns 0.

11.22.20 Evaluation

void _fq_zech_poly_evaluate_fq_zech(fq_zech_t rop, const fq_zech_struct *op, slong len, const fq_zech_t a, const fq_zech_ctx_t ctx)

Sets \text{rop} to \((\text{op}, \text{len})\) evaluated at \(a\).

void _fq_zech_poly_evaluate_fq_zech(fq_zech_t rop, const fq_zech_poly_t op, const fq_zech_ctx_t ctx)

Sets \text{rop} to the value of \(f(a)\).

As the coefficient ring \(F_q\) is finite, Horner’s method is sufficient.
11.22.21 Composition

```c
void _fq_zech_poly_compose(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1, const fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets rop to the composition of (op1, len1) and (op2, len2).

Assumes that rop has space for (len1-1)*(len2-1) + 1 coefficients. Assumes that op1 and op2 are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```c
void fq_zech_poly_compose(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t op2, const fq_zech_ctx_t ctx)
```

Sets rop to the composition of op1 and op2. To be precise about the order of composition, denoting rop, op1, and op2 by $f$, $g$, and $h$, respectively, sets $f(t) = g(h(t))$.

```c
void _fq_zech_poly_compose_mod_horner(fq_zech_struct *res, const fq_zech_struct *f, slong lenf, const fq_zech_struct *g, const fq_zech_struct *h, slong lenh, const fq_zech_ctx_t ctx)
```

Sets res to the composition $f(g)$ modulo $h$. We require that $h$ is nonzero and that the length of $g$ is one less than the length of $h$ (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner’s rule.

```c
void fq_zech_poly_compose_mod_horner(fq_zech_poly_t res, const fq_zech_poly_t f, const fq_zech_poly_t g, const fq_zech_poly_t h, const fq_zech_ctx_t ctx)
```

Sets res to the composition $f(g)$ modulo $h$. We require that $h$ is nonzero. The algorithm used is Horner’s rule.

```c
void _fq_zech_poly_compose_mod_horner_preinv(fq_zech_struct *res, const fq_zech_struct *f, slong lenf, const fq_zech_struct *g, const fq_zech_struct *h, slong lenh, const fq_zech_ctx_t ctx)
```

Sets res to the composition $f(g)$ modulo $h$. We require that $h$ is nonzero and that the length of $g$ is one less than the length of $h$ (possibly with zero padding). We also require that the length of $f$ is less than the length of $h$. Furthermore, we require $hinv$ to be the inverse of the reverse of $h$. The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner’s rule.

```c
void fq_zech_poly_compose_mod_horner_preinv(fq_zech_poly_t res, const fq_zech_poly_t f, const fq_zech_poly_t g, const fq_zech_poly_t hinv, const fq_zech_ctx_t ctx)
```

Sets res to the composition $f(g)$ modulo $h$. We require that $h$ is nonzero and that $f$ has smaller degree than $h$. Furthermore, we require $hinv$ to be the inverse of the reverse of $h$. The algorithm used is Horner’s rule.

```c
void _fq_zech_poly_compose_mod_brent_kung(fq_zech_struct *res, const fq_zech_struct *f, slong lenf, const fq_zech_struct *g, const fq_zech_struct *h, slong lenh, const fq_zech_ctx_t ctx)
```

Sets res to the composition $f(g)$ modulo $h$. We require that $h$ is nonzero and that the length of $g$ is one less than the length of $h$ (possibly with zero padding). We also require that the length of $f$ is less than the length of $h$. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```c
void fq_zech_poly_compose_mod_brent_kung(fq_zech_poly_t res, const fq_zech_poly_t f, const fq_zech_poly_t g, const fq_zech_poly_t h, const fq_zech_ctx_t ctx)
```

11.22. fq_zech_poly.h – univariate polynomials over finite fields (Zech logarithm representation)
Sets \texttt{res} to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). The algorithm used is the Brent-Kung matrix algorithm.

\begin{verbatim}
void _fq_zech_poly_compose_mod_brent_kung_preinv(fq_zech_struct *res, const fq_zech_struct *f, slong lenf, const fq_zech_struct *g, const fq_zech_struct *h, slong lenh, const fq_zech_ctx_t ctx)
\end{verbatim}

Sets \texttt{res} to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). We also require that the length of \( f \) is smaller than the length of \( h \). Furthermore, we require \texttt{hinv} to be the inverse of the reverse of \( h \). The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

\begin{verbatim}
void fq_zech_poly_compose_mod_brent_kung_preinv(fq_zech_poly_t res, const fq_zech_poly_t f, const fq_zech_poly_t g, const fq_zech_poly_t h, const fq_zech_poly_t hinv, const fq_zech_ctx_t ctx)
\end{verbatim}

Sets \texttt{res} to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). Furthermore, we require \texttt{hinv} to be the inverse of the reverse of \( h \). The algorithm used is the Brent-Kung matrix algorithm.

\begin{verbatim}
void _fq_zech_poly_compose_mod(fq_zech_struct *res, const fq_zech_struct *f, slong lenf, const fq_zech_struct *g, const fq_zech_struct *h, slong lenh, const fq_zech_ctx_t ctx)
\end{verbatim}

Sets \texttt{res} to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

\begin{verbatim}
void fq_zech_poly_compose_mod(fq_zech_poly_t res, const fq_zech_poly_t f, const fq_zech_poly_t g, const fq_zech_poly_t h, const fq_zech_ctx_t ctx)
\end{verbatim}

Sets \texttt{res} to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero.

\begin{verbatim}
void _fq_zech_poly_compose_mod_preinv(fq_zech_struct *res, const fq_zech_struct *f, slong lenf, const fq_zech_struct *g, const fq_zech_struct *h, slong lenh, const fq_zech_ctx_t ctx)
\end{verbatim}

Sets \texttt{res} to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that the length of \( g \) is one less than the length of \( h \) (possibly with zero padding). We also require that the length of \( f \) is smaller than the length of \( h \). Furthermore, we require \texttt{hinv} to be the inverse of the reverse of \( h \). The output is not allowed to be aliased with any of the inputs.

\begin{verbatim}
void fq_zech_poly_compose_mod_preinv(fq_zech_poly_t res, const fq_zech_poly_t f, const fq_zech_poly_t g, const fq_zech_poly_t h, const fq_zech_ctx_t ctx)
\end{verbatim}

Sets \texttt{res} to the composition \( f(g) \) modulo \( h \). We require that \( h \) is nonzero and that \( f \) has smaller degree than \( h \). Furthermore, we require \texttt{hinv} to be the inverse of the reverse of \( h \).

\begin{verbatim}
void _fq_zech_poly_reduce_matrix_mod_poly(fq_zech_mat_t A, const fq_zech_mat_t B, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
\end{verbatim}

Sets the \texttt{ith} row of \texttt{A} to the reduction of the \texttt{ith} row of \texttt{B} modulo \texttt{f} for \( i = 1, \ldots, \sqrt{\deg(f)} \). We require \texttt{B} to be at least a \( \sqrt{\deg(f) \times \deg(f)} \) matrix and \texttt{f} to be nonzero.

\begin{verbatim}
void _fq_zech_poly_precompute_matrix(fq_zech_mat_t A, const fq_zech_struct *f, const fq_zech_struct *g, slong leng, const fq_zech_struct *ginv, slong lenginv, const fq_zech_ctx_t ctx)
\end{verbatim}

Sets the \texttt{ith} row of \texttt{A} to \( f' \) modulo \( g \) for \( i = 1, \ldots, \sqrt{\deg(g)} \). We require \texttt{A} to be a \( \sqrt{\deg(g) \times \deg(g)} \) matrix. We require \texttt{ginv} to be the inverse of the reverse of \texttt{g} and \texttt{g} to be nonzero.
void fq_zech_poly_precompute_matrix(fq_zech_mat_t A, const fq_zech_poly_t f, const fq_zech_poly_t g, const fq_zech_poly_t ginv, const fq_zech_ctx_t ctx)

Sets the ith row of A to $f_i$ modulo $g$ for $i = 1, \ldots, \sqrt{\deg(g)}$. We require $A$ to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require $ginv$ to be the inverse of the reverse of $g$.

void _fq_zech_poly_compose_mod_brent_kung_precomp_preinv(fq_zech_struct *res, const fq_zech_struct *f, slong lenf, const fq_zech_mat_t A, const fq_zech_struct *h, slong lenh, const fq_zech_struct *hinv, slong lenhinv, const fq_zech_ctx_t ctx)

Sets $res$ to the composition $f(g)$ modulo $h$. We require that $h$ is nonzero. We require that the ith row of $A$ contains $g_i$ for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. $A$ is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of $f$ is less than the length of $h$. Furthermore, we require $hinv$ to be the inverse of the reverse of $h$. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

void fq_zech_poly_compose_mod_brent_kung_precomp_preinv(fq_zech_poly_t res, const fq_zech_poly_t f, const fq_zech_mat_t A, const fq_zech_poly_t h, const fq_zech_poly_t hinv, const fq_zech_ctx_t ctx)

Sets $res$ to the composition $f(g)$ modulo $h$. We require that the ith row of $A$ contains $g_i$ for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. $A$ is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that $h$ is nonzero and that $f$ has smaller degree than $h$. Furthermore, we require $hinv$ to be the inverse of the reverse of $h$. This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo $h$ for fixed $g$ and $h$.

11.22.22 Output

int _fq_zech_poly_fprint_pretty(FILE *file, const fq_zech_struct *poly, slong len, const char *x, const fq_zech_ctx_t ctx)

Prints the pretty representation of $(poly, len)$ to the stream $file$, using the string $x$ to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_zech_poly_fprint_pretty(FILE *file, const fq_zech_poly_t poly, const char *x, const fq_zech_ctx_t ctx)

Prints the pretty representation of $poly$ to the stream $file$, using the string $x$ to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fq_zech_poly_print_pretty(const fq_zech_struct *poly, slong len, const char *x, const fq_zech_ctx_t ctx)

Prints the pretty representation of $(poly, len)$ to stdout, using the string $x$ to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_zech_poly_print_pretty(const fq_zech_poly_t poly, const char *x, const fq_zech_ctx_t ctx)

Prints the pretty representation of $poly$ to stdout, using the string $x$ to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

11.22. fq_zech_poly.h – univariate polynomials over finite fields (Zech logarithm representation)
int _fq_zech_fprint(FILE *file, const fq_zech_struct *poly, slong len, const fq_zech_ctx_t ctx)

Prints the pretty representation of \((poly, \text{len})\) to the stream \(file\).

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_zech_poly_fprint(FILE *file, const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)

Prints the pretty representation of \(poly\) to the stream \(file\).

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fq_zech_poly_print(const fq_zech_struct *poly, slong len, const fq_zech_ctx_t ctx)

Prints the pretty representation of \((poly, \text{len})\) to stdout.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int fq_zech_poly_print(const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)

Prints the representation of \(poly\) to stdout.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

char * _fq_zech_poly_get_str(const fq_zech_struct *poly, slong len, const fq_zech_ctx_t ctx)

Returns the plain FLINT string representation of the polynomial \((poly, \text{len})\).

char *fq_zech_poly_get_str(const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)

Returns the plain FLINT string representation of the polynomial \(poly\).

char * _fq_zech_poly_get_str_pretty(const fq_zech_struct *poly, slong len, const char *x, const fq_zech_ctx_t ctx)

Returns a pretty representation of the polynomial \((poly, \text{len})\) using the null-terminated string \(x\) as the variable name.

char *fq_zech_poly_get_str_pretty(const fq_zech_poly_t poly, const char *x, const fq_zech_ctx_t ctx)

Returns a pretty representation of the polynomial \(poly\) using the null-terminated string \(x\) as the variable name.

11.22.23 Inflation and deflation

void fq_zech_poly_inflate(fq_zech_poly_t result, const fq_zech_poly_t input, ulong inflation, const fq_zech_ctx_t ctx)

Sets \(result\) to the inflated polynomial \(p(x^n)\) where \(p\) is given by \(input\) and \(n\) is given by \(inflation\).

void fq_zech_poly_deflate(fq_zech_poly_t result, const fq_zech_poly_t input, ulong deflation, const fq_zech_ctx_t ctx)

Sets \(result\) to the deflated polynomial \(p(x^{1/n})\) where \(p\) is given by \(input\) and \(n\) is given by \(deflation\). Requires \(n > 0\).

ulong fq_zech_poly_deflation(const fq_zech_poly_t input, const fq_zech_ctx_t ctx)

Returns the largest integer by which \(input\) can be deflated. As special cases, returns 0 if \(input\) is the zero polynomial and 1 of \(input\) is a constant polynomial.
11.23 fq_zech_poly_factor.h – factorisation of univariate polynomials over finite fields (Zech logarithm representation)

11.23.1 Types, macros and constants

type fq_zech_poly_factor_struct

type fq_zech_poly_factor_t

11.23.2 Memory Management

void fq_zech_poly_factor_init(fq_zech_poly_factor_t fac, const fq_zech_ctx_t ctx)
                
Initialises fac for use. An fq_zech_poly_factor_t represents a polynomial in factorised form as a product of polynomials with associated exponents.

void fq_zech_poly_factor_clear(fq_zech_poly_factor_t fac, const fq_zech_ctx_t ctx)
                
Frees all memory associated with fac.

void fq_zech_poly_factor_realloc(fq_zech_poly_factor_t fac, slong alloc, const fq_zech_ctx_t ctx)
                
Reallocates the factor structure to provide space for precisely alloc factors.

void fq_zech_poly_factor_fit_length(fq_zech_poly_factor_t fac, slong len, const fq_zech_ctx_t ctx)
                
Ensures that the factor structure has space for at least len factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

11.23.3 Basic Operations

void fq_zech_poly_factor_set(fq_zech_poly_factor_t res, const fq_zech_poly_factor_t fac, const fq_zech_ctx_t ctx)
                
Sets res to the same factorisation as fac.

void fq_zech_poly_factor_print_pretty(const fq_zech_poly_factor_t fac, const char *var, const fq_zech_ctx_t ctx)
                
Pretty-prints the entries of fac to standard output.

void fq_zech_poly_factor_print(const fq_zech_poly_factor_t fac, const fq_zech_ctx_t ctx)
                
Prints the entries of fac to standard output.

void fq_zech_poly_factor_insert(fq_zech_poly_factor_t fac, const fq_zech_poly_t poly, slong exp, const fq_zech_ctx_t ctx)
                
Inserts the factor poly with multiplicity exp into the factorisation fac.

    If fac already contains poly, then exp simply gets added to the exponent of the existing entry.

void fq_zech_poly_factor_concat(fq_zech_poly_factor_t res, const fq_zech_poly_factor_t fac, const fq_zech_ctx_t ctx)
                
Concatenates two factorisations.

    This is equivalent to calling fq_zech_poly_factor_insert() repeatedly with the individual factors of fac.

    Does not support aliasing between res and fac.

void fq_zech_poly_factor_pow(fq_zech_poly_factor_t fac, slong exp, const fq_zech_ctx_t ctx)
                
Raises fac to the power exp.
ulong fq_zech_poly_remove(fq_zech_poly_t f, const fq_zech_poly_t p, const fq_zech_ctx_t ctx)
    
    Removes the highest possible power of p from f and returns the exponent.

11.23.4 Irreducibility Testing

int fq_zech_poly_is_irreducible(const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
    
    Returns 1 if the polynomial f is irreducible, otherwise returns 0.

int fq_zech_poly_is_irreducible_ddf(const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
    
    Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

int fq_zech_poly_is_irreducible_ben_or(const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
    
    Returns 1 if the polynomial f is irreducible, otherwise returns 0. Uses Ben-Or’s irreducibility test.

int _fq_zech_poly_is_squarefree(const fq_zech_struct *f, slong len, const fq_zech_ctx_t ctx)
    
    Returns 1 if (f, len) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

int fq_zech_poly_is_squarefree(const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
    
    Returns 1 if f is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

11.23.5 Factorisation

int fq_zech_poly_factor_equal_deg_prob(fq_zech_poly_t factor, flint_rand_t state, const fq_zech_poly_t pol, slong d, const fq_zech_ctx_t ctx)
    
    Probabilistic equal degree factorisation of pol into irreducible factors of degree d. If it passes, a factor is placed in factor and 1 is returned, otherwise 0 is returned and the value of factor is undetermined.
    
    Requires that pol be monic, non-constant and squarefree.

void fq_zech_poly_factor_equal_deg(fq_zech_poly_factor_t factors, const fq_zech_poly_t pol, slong d, const fq_zech_ctx_t ctx)
    
    Assuming pol is a product of irreducible factors all of degree d, finds all those factors and places them in factors. Requires that pol be monic, non-constant and squarefree.

void fq_zech_poly_factor_split_single(fq_zech_poly_t linfactor, const fq_zech_poly_t input, const fq_zech_ctx_t ctx)
    
    Assuming input is a product of factors all of degree 1, finds a single linear factor of input and places it in linfactor. Requires that input be monic and non-constant.

void fq_zech_poly_factor_distinct_deg(fq_zech_poly_factor_t res, const fq_zech_poly_t poly, slong *const *degs, const fq_zech_ctx_t ctx)
    
    Factorises a monic non-constant squarefree polynomial poly of degree n into factors f[d] such that for 1 ≤ d ≤ n f[d] is the product of the monic irreducible factors of poly of degree d. Factors are stored in res, associated powers of irreducible polynomials are stored in degs in the same order as factors.
    
    Requires that degs have enough space for irreducible polynomials’ powers (maximum space required is n * sizeof(slong)).

void fq_zech_poly_factor_squarefree(fq_zech_poly_factor_t res, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
    
    Sets res to a squarefree factorization of f.
void fq_zech_poly_factor(fq_zech_poly_factor_t res, fq_zech_t lead, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Factorises a non-constant polynomial \( f \) into monic irreducible factors choosing the best algorithm for given modulo and degree. The output \( \text{lead} \) is set to the leading coefficient of \( f \) upon return. Choice of algorithm is based on heuristic measurements.

void fq_zech_poly_factor_cantor_zassenhaus(fq_zech_poly_factor_t res, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Factorises a non-constant polynomial \( f \) into monic irreducible factors using the Cantor-Zassenhaus algorithm.

void fq_zech_poly_factor_kaltofen_shoup(fq_zech_poly_factor_t res, const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)

Factorises a non-constant polynomial \( f \) into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a “baby step/giant step” strategy for the distinct-degree factorization step.

void fq_zech_poly_factor_berlekamp(fq_zech_poly_factor_t factors, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Factorises a non-constant polynomial \( f \) into monic irreducible factors using the Berlekamp algorithm.

void fq_zech_poly_factor_with_berlekamp(fq_zech_poly_factor_t res, fq_zech_t leading_coeff, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Factorises a general polynomial \( f \) into monic irreducible factors and sets \( \text{leading\_coeff} \) to the leading coefficient of \( f \), or 0 if \( f \) is the zero polynomial.

This function first checks for small special cases, deflates \( f \) if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs Berlekamp on all the individual square-free factors.

void fq_zech_poly_factor_with_cantor_zassenhaus(fq_zech_poly_factor_t res, fq_zech_t leading_coeff, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Factorises a general polynomial \( f \) into monic irreducible factors and sets \( \text{leading\_coeff} \) to the leading coefficient of \( f \), or 0 if \( f \) is the zero polynomial.

This function first checks for small special cases, deflates \( f \) if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

void fq_zech_poly_factor_with_kaltofen_shoup(fq_zech_poly_factor_t res, fq_zech_t leading_coeff, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)

Factorises a general polynomial \( f \) into monic irreducible factors and sets \( \text{leading\_coeff} \) to the leading coefficient of \( f \), or 0 if \( f \) is the zero polynomial.

This function first checks for small special cases, deflates \( f \) if it is of the form \( p(x^m) \) for some \( m > 1 \), then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

void fq_zech_poly_iterated_frobenius_preinv(fq_zech_poly_t *rop, slong n, const fq_zech_poly_t v, const fq_zech_poly_t vinv, const fq_zech_ctx_t ctx)

Sets \( \text{rop}[i] \) to be \( x^i \mod v \) for \( 0 \leq i < n \).

It is required that \( \text{vinv} \) is the inverse of the reverse of \( v \mod x^{\text{len}v} \).
11.23.6 Root Finding

void fq_zech_poly_roots(fq_zech_poly_factor_t r, const fq_zech_poly_t f, int with_multiplicity, const fq_zech_ctx_t ctx)

Fill r with factors of the form $x - r_i$ where the $r_i$ are the distinct roots of a nonzero $f$ in $F_q$. If `with_multiplicity` is zero, the exponent $e_i$ of the factor $x - r_i$ is 1. Otherwise, it is the largest $e_i$ such that $(x - r_i)^e$ divides $f$. This function throws if $f$ is zero, but is otherwise always successful.

11.24 fq_zech_embed.h – Computing isomorphisms and embeddings of finite fields

void fq_zech_embed_gens(fq_zech_t gen_sub, fq_zech_t gen_sup, nmod_poly_t minpoly, const fq_zech_ctx_t sub_ctx, const fq_zech_ctx_t sup_ctx)

Given two contexts `sub_ctx` and `sup_ctx`, such that `degree(sub_ctx)` divides `degree(sup_ctx)`, compute:

- an element `gen_sub` in `sub_ctx` such that `gen_sub` generates the finite field defined by `sub_ctx`,
- its minimal polynomial `minpoly`,
- a root `gen_sup` of `minpoly` inside the field defined by `sup_ctx`.

These data uniquely define an embedding of `sub_ctx` into `sup_ctx`.

void _fq_zech_embed_gens_naive(fq_zech_t gen_sub, fq_zech_t gen_sup, nmod_poly_t minpoly, const fq_zech_ctx_t sub_ctx, const fq_zech_ctx_t sup_ctx)

Given two contexts `sub_ctx` and `sup_ctx`, such that `degree(sub_ctx)` divides `degree(sup_ctx)`, compute an embedding of `sub_ctx` into `sup_ctx` defined as follows:

- `gen_sub` is the canonical generator of `sup_ctx` (i.e., the class of $X$),
- `minpoly` is the defining polynomial of `sub_ctx`,
- `gen_sup` is a root of `minpoly` inside the field defined by `sup_ctx`.

void fq_zech_embed_matrices(nmod_mat_t embed, nmod_mat_t project, const fq_zech_t gen_sub, const fq_zech_ctx_t sub_ctx, const fq_zech_ctx_t sup_ctx, const nmod_poly_t gen_minpoly)

Given:

- two contexts `sub_ctx` and `sup_ctx`, of respective degrees $m$ and $n$, such that $m$ divides $n$;
- a generator `gen_sub` of `sub_ctx`, its minimal polynomial `gen_minpoly`, and a root `gen_sup` of `gen_minpoly` in `sup_ctx`, as returned by `fq_zech_embed_gens`;

Compute:

- the $n \times m$ matrix `embed` mapping `gen_sub` to `gen_sup`, and all their powers accordingly;
- an $m \times n$ matrix `project` such that `project` $\times$ `embed` is the $m \times m$ identity matrix.

void fq_zech_embed_trace_matrix(nmod_mat_t res, const nmod_mat_t basis, const fq_zech_ctx_t sub_ctx, const fq_zech_ctx_t sup_ctx)

Given:

- two contexts `sub_ctx` and `sup_ctx`, of degrees $m$ and $n$, such that $m$ divides $n$;
- an $n \times m$ matrix `basis` that maps `sub_ctx` to an isomorphic subfield in `sup_ctx`;

Compute the $m \times n$ matrix of the trace from `sup_ctx` to `sub_ctx`.

This matrix is computed as
Note: if $m = n$, basis represents a Frobenius, and the result is its inverse matrix.

```c
void fq_zech_embed_composition_matrix(nmod_mat_t matrix, const fq_zech_t gen, const fq_zech_ctx_t ctx)
```

Compute the composition matrix of gen.

For an element $a \in \mathbb{F}_{p^n}$, its composition matrix is the matrix whose columns are $a^0, a^1, \ldots, a^{n-1}$.

```c
void fq_zech_embed_composition_matrix_sub(nmod_mat_t matrix, const fq_zech_t gen, const fq_zech_ctx_t ctx, slong trunc)
```

Compute the composition matrix of gen, truncated to trunc columns.

```c
void fq_zech_embed_mul_matrix(nmod_mat_t matrix, const fq_zech_t gen, const fq_zech_ctx_t ctx)
```

Compute the multiplication matrix of gen.

For an element $a \in \mathbb{F}_{p^n} = \mathbb{F}_p[x]$, its multiplication matrix is the matrix whose columns are $a, ax, \ldots, ax^{n-1}$.

```c
void fq_zech_embed_mono_to_dual_matrix(nmod_mat_t res, const fq_zech_ctx_t ctx)
```

Compute the change of basis matrix from the monomial basis of ctx to its dual basis.

```c
void fq_zech_embed_dual_to_mono_matrix(nmod_mat_t res, const fq_zech_ctx_t ctx)
```

Compute the change of basis matrix from the dual basis of ctx to its monomial basis.

```c
void fq_zech_modulus_pow_series_inv(nmod_poly_t res, const fq_zech_ctx_t ctx, slong trunc)
```

Compute the power series inverse of the reverse of the modulus of ctx up to $O(x^{\text{trunc}})$.

```c
void fq_zech_modulus_derivative_inv(fq_zech_t m_prime, fq_zech_t m_prime_inv, const fq_zech_ctx_t ctx)
```

Compute the derivative $m_{\text{prime}}$ of the modulus of ctx as an element of ctx, and its inverse $m_{\text{prime\_inv}}$.  

11.24. fq_zech_embed.h – Computing isomorphisms and embeddings of finite fields
12.1 padic.h – p-adic numbers

12.1.1 Introduction

The padic_t data type represents elements of \( \mathbb{Q}_p \) to precision \( N \), stored in the form \( x = p^v u \) with \( u, v \in \mathbb{Z} \). Arithmetic operations can be carried out with respect to a context containing the prime number \( p \) and various pieces of pre-computed data.

Independent of the context, we consider a \( p \)-adic number \( x = p^v u \) to be in canonical form whenever either \( p \nmid u \) or \( u = v = 0 \), and we say it is reduced if, in addition, for non-zero \( u \), \( u \in (0, p^{N-v}) \).

We briefly describe the interface:

The functions in this module expect arguments of type padic_t, and each variable carries its own precision. The functions have an interface that is similar to the MPFR functions. In particular, they have the same semantics, specified as follows: Compute the requested operation exactly and then reduce the result to the precision of the output variable.

12.1.2 Data structures

A \( p \)-adic number of type padic_t comprises a unit \( u \), a valuation \( v \), and a precision \( N \). We provide the following macros to access these fields, so that code can be developed somewhat independently from the underlying data layout.

\[ \text{fmpz } \ast \text{padic_unit(const padic_t op)} \]

Returns the unit part of the \( p \)-adic number as a FLINT integer, which can be used as an operand for the \text{fmpz} functions.

\[ \text{slong padic_val(const padic_t op)} \]

Returns the valuation part of the \( p \)-adic number.

Note that this function is implemented as a macro and that the expression padic_val(op) can be used as both an \textit{lvalue} and an \textit{rvalue}.

\[ \text{slong padic_get_val(const padic_t op)} \]

Returns the valuation part of the \( p \)-adic number.

\[ \text{slong padic_prec(const padic_t op)} \]

Returns the precision of the \( p \)-adic number.

Note that this function is implemented as a macro and that the expression padic_prec(op) can be used as both an \textit{lvalue} and an \textit{rvalue}.

\[ \text{slong padic_get_prec(const padic_t op)} \]

Returns the precision of the \( p \)-adic number.
12.1.3 Context

A context object for $p$-adic arithmetic contains data pertinent to $p$-adic computations, but which we choose not to store with each element individually. Currently, this includes the prime number $p$, its double inverse in case of word-sized primes, precomputed powers of $p$ in the range given by $\min$ and $\max$, and the printing mode.

```c
void padic_ctx_init(padic_ctx_t ctx, const fmpz_t p, slong min, slong max, enum padic_print_mode mode)
```

Initialises the context $ctx$ with the given data.

Assumes that $p$ is a prime. This is not verified but the subsequent behaviour is undefined if $p$ is a composite number.

Assumes that $\min$ and $\max$ are non-negative and that $\min$ is at most $\max$, raising an abort signal otherwise.

Assumes that the printing mode is one of $\text{PADIC_TERSE}$, $\text{PADIC_SERIES}$, or $\text{PADIC_VAL_UNIT}$. Using the example $x = 7^{-1}12$ in $\mathbb{Q}_7$, these behave as follows:

In $\text{PADIC_TERSE}$ mode, a $p$-adic number is printed in the same way as a rational number, e.g. $12/7$.

In $\text{PADIC_SERIES}$ mode, a $p$-adic number is printed digit by digit, e.g. $5*7^{-1} + 1$.

In $\text{PADIC_VAL_UNIT}$ mode, a $p$-adic number is printed showing the valuation and unit parts separately, e.g. $12*7^{-1}$.

```c
void padic_ctx_clear(padic_ctx_t ctx)
```

Clears all memory that has been allocated as part of the context.

```c
int _padic_ctx_pow_ui(fmpz_t rop, ulong e, const padic_ctx_t ctx)
```

Sets $rop$ to $p^e$ as efficiently as possible, where $rop$ is expected to be an uninitialised $fmpz_t$.

If the return value is non-zero, it is the responsibility of the caller to clear the returned integer.

12.1.4 Memory management

```c
void padic_init(padic_t rop)
```

Initialises the $p$-adic number with the precision set to $\text{PADIC_DEFAULT_PREC}$, which is defined as 20.

```c
void padic_init2(padic_t rop, slong N)
```

Initialises the $p$-adic number $rop$ with precision $N$.

```c
void padic_clear(padic_t rop)
```

Clears all memory used by the $p$-adic number $rop$.

```c
void _padic_canonicalise(padic_t rop, const padic_ctx_t ctx)
```

Brings the $p$-adic number $rop$ into canonical form.

That is to say, ensures that either $u = v = 0$ or $p \nmid u$. There is no reduction modulo a power of $p$.

```c
void _padic_reduce(padic_t rop, const padic_ctx_t ctx)
```

Given a $p$-adic number $rop$ in canonical form, reduces it modulo $p^N$.

```c
void padic_reduce(padic_t rop, const padic_ctx_t ctx)
```

Ensures that the $p$-adic number $rop$ is reduced.
12.1.5 Randomisation

void \texttt{padic\_randtest}(padic\_t rop, flint\_rand\_t state, const padic\_ctx\_t ctx)
Sets \texttt{rop} to a random $p$-adic number modulo $p^N$ with valuation in the range $[-\lceil N/10 \rceil, N)$, $[N - \lceil -N/10 \rceil, N)$, or $[-10, 0)$ as $N$ is positive, negative or zero, whenever \texttt{rop} is non-zero.

void \texttt{padic\_randtest\_not\_zero}(padic\_t rop, flint\_rand\_t state, const padic\_ctx\_t ctx)
Sets \texttt{rop} to a random non-zero $p$-adic number modulo $p^N$, where the range of the valuation is as for the function \texttt{padic\_randtest()}

void \texttt{padic\_randtest\_int}(padic\_t rop, flint\_rand\_t state, const padic\_ctx\_t ctx)
Sets \texttt{rop} to a random $p$-adic integer modulo $p^N$. Note that whenever $N \leq 0$, \texttt{rop} is set to zero.

12.1.6 Assignments and conversions

All assignment functions set the value of \texttt{rop} from \texttt{op}, reduced to the precision of \texttt{rop}.

void \texttt{padic\_set}(padic\_t rop, const padic\_t op, const padic\_ctx\_t ctx)
Sets \texttt{rop} to the $p$-adic number \texttt{op}.

void \texttt{padic\_set\_si}(padic\_t rop, slong op, const padic\_ctx\_t ctx)
Sets the $p$-adic number \texttt{rop} to the \texttt{slong} integer \texttt{op}.

void \texttt{padic\_set\_ui}(padic\_t rop, ulong op, const padic\_ctx\_t ctx)
Sets the $p$-adic number \texttt{rop} to the \texttt{ulong} integer \texttt{op}.

void \texttt{padic\_set\_fmpz}(padic\_t rop, const fmpz\_t op, const padic\_ctx\_t ctx)
Sets the $p$-adic number \texttt{rop} to the \texttt{fmpz} integer \texttt{op}.

void \texttt{padic\_set\_fmpq}(padic\_t rop, const fmpq\_t op, const padic\_ctx\_t ctx)
Sets \texttt{rop} to the rational \texttt{op}.

void \texttt{padic\_set\_mpz}(padic\_t rop, const mpz\_t op, const padic\_ctx\_t ctx)
Sets the $p$-adic number \texttt{rop} to the GMP integer \texttt{op}.

void \texttt{padic\_set\_mpq}(padic\_t rop, const mpq\_t op, const padic\_ctx\_t ctx)
Sets \texttt{rop} to the GMP rational \texttt{op}.

void \texttt{padic\_get\_fmpz}(fmpz\_t rop, const padic\_t op, const padic\_ctx\_t ctx)
Sets the integer \texttt{rop} to the exact $p$-adic integer \texttt{op}.
If \texttt{op} is not a $p$-adic integer, raises an \texttt{abort} signal.

void \texttt{padic\_get\_fmpq}(fmpq\_t rop, const padic\_t op, const padic\_ctx\_t ctx)
Sets the rational \texttt{rop} to the $p$-adic number \texttt{op}.

void \texttt{padic\_get\_mpz}(mpz\_t rop, const padic\_t op, const padic\_ctx\_t ctx)
Sets the GMP integer \texttt{rop} to the $p$-adic integer \texttt{op}.
If \texttt{op} is not a $p$-adic integer, raises an \texttt{abort} signal.

void \texttt{padic\_get\_mpq}(mpq\_t rop, const padic\_t op, const padic\_ctx\_t ctx)
Sets the GMP rational \texttt{rop} to the value of \texttt{op}.

void \texttt{padic\_swap}(padic\_t op1, padic\_t op2)
Swaps the two $p$-adic numbers \texttt{op1} and \texttt{op2}.
Note that this includes swapping the precisions. In particular, this operation is not equivalent to swapping \texttt{op1} and \texttt{op2} using \texttt{padic\_set()} and an auxiliary variable whenever the precisions of the two elements are different.
void \texttt{padic\_zero}(padic\_t \texttt{rop})
Sets the \(p\)-adic number \(\texttt{rop}\) to zero.

void \texttt{padic\_one}(padic\_t \texttt{rop})
Sets the \(p\)-adic number \(\texttt{rop}\) to one, reduced modulo the precision of \(\texttt{rop}\).

12.1.7 Comparison

int \texttt{padic\_is\_zero}(const padic\_t \texttt{op})
Returns whether \(\texttt{op}\) is equal to zero.

int \texttt{padic\_is\_one}(const padic\_t \texttt{op})
Returns whether \(\texttt{op}\) is equal to one, that is, whether \(u = 1\) and \(v = 0\).

int \texttt{padic\_equal}(const padic\_t \texttt{op1}, const padic\_t \texttt{op2})
Returns whether \(\texttt{op1}\) and \(\texttt{op2}\) are equal, that is, whether \(u_1 = u_2\) and \(v_1 = v_2\).

12.1.8 Arithmetic operations

\begin{verbatim}
slong * \texttt{\_padic\_lifts\_exps}(slong *n, slong N)
    Given a positive integer \(N\) define the sequence \(a_0 = N, a_1 = [a_0/2], \ldots, a_{n-1} = [a_{n-2}/2] = 1\).
    Then \(n = \lceil \log_2 N \rceil + 1\).
    This function sets \(n\) and allocates and returns the array \(a\).

void \texttt{\_padic\_lifts\_pows}(fmpz *pow, const slong *a, slong n, const fmpz_t p)
    Given an array \(a\) as computed above, this function computes the corresponding powers of \(p\), that is, \(\texttt{pow}[i] = p^{a_i}\).

void \texttt{padic\_add}(padic\_t \texttt{rop}, const padic\_t \texttt{op1}, const padic\_t \texttt{op2}, const padic\_ctx\_t \texttt{ctx})
    Sets \(\texttt{rop}\) to the sum of \(\texttt{op1}\) and \(\texttt{op2}\).

void \texttt{padic\_sub}(padic\_t \texttt{rop}, const padic\_t \texttt{op1}, const padic\_t \texttt{op2}, const padic\_ctx\_t \texttt{ctx})
    Sets \(\texttt{rop}\) to the difference of \(\texttt{op1}\) and \(\texttt{op2}\).

void \texttt{padic\_neg}(padic\_t \texttt{rop}, const padic\_t \texttt{op}, const padic\_ctx\_t \texttt{ctx})
    Sets \(\texttt{rop}\) to the additive inverse of \(\texttt{op}\).

void \texttt{padic\_mul}(padic\_t \texttt{rop}, const padic\_t \texttt{op1}, const padic\_t \texttt{op2}, const padic\_ctx\_t \texttt{ctx})
    Sets \(\texttt{rop}\) to the product of \(\texttt{op1}\) and \(\texttt{op2}\).

void \texttt{padic\_shift}(padic\_t \texttt{rop}, const padic\_t \texttt{op}, slong \texttt{v}, const padic\_ctx\_t \texttt{ctx})
    Sets \(\texttt{rop}\) to the product of \(\texttt{op}\) and \(p^v\).

void \texttt{padic\_div}(padic\_t \texttt{rop}, const padic\_t \texttt{op1}, const padic\_t \texttt{op2}, const padic\_ctx\_t \texttt{ctx})
    Sets \(\texttt{rop}\) to the quotient of \(\texttt{op1}\) and \(\texttt{op2}\).

void \texttt{\_padic\_inv\_precompute}(padic\_inv\_t \texttt{S}, const fmpz_t \texttt{p}, slong \texttt{N})
    Pre-computes some data and allocates temporary space for \(p\)-adic inversion using Hensel lifting.

void \texttt{\_padic\_inv\_clear}(padic\_inv\_t \texttt{S})
    Frees the memory used by \(\texttt{S}\).

void \texttt{\_padic\_inv\_precomp}(fmpz_t \texttt{rop}, const fmpz_t \texttt{op}, const padic\_inv\_t \texttt{S})
    Sets \(\texttt{rop}\) to the inverse of \(\texttt{op}\) modulo \(p^N\), assuming that \(\texttt{op}\) is a unit and \(N \geq 1\).
    In the current implementation, allows aliasing, but this might change in future versions.
    Uses some data \(S\) precomputed by calling the function \texttt{\_padic\_inv\_precompute()}.
    Note that this object is not declared \texttt{const} and in fact it carries a field providing temporary work space.
\end{verbatim}
allows repeated calls of this function to avoid repeated memory allocations, as used e.g. by the function \texttt{padic_log}.

\begin{verbatim}
void _padic_inv(fmpz_t rop, const fmpz_t op, const fmpz_t p, slong N)
Sets \texttt{rop} to the inverse of \texttt{op} modulo \(p^N\), assuming that \texttt{op} is a unit and \(N \geq 1\).

In the current implementation, allows aliasing, but this might change in future versions.
\end{verbatim}

\begin{verbatim}
void _padic_exp_balanced(fmpz_t rop, const fmpz_t u, slong v, const fmpz_t p, slong N)
void _padic_exp_rectangular(fmpz_t rop, const fmpz_t u, slong v, const fmpz_t p, slong N)
void _padic_exp(fmpz_t rop, const fmpz_t u, slong v, const fmpz_t p, slong N)
Sets \texttt{rop} to the \(p\)-exponential function evaluated at \(x = p^u\), reduced modulo \(p^N\).

Assumes that \(x \neq 0\), that \(\text{ord}_p(x) < N\) and that \(\exp(x)\) converges, that is, that \(\text{ord}_p(x)\) is at least 2 or 1 depending on whether the prime \(p\) is 2 or odd.

Supports aliasing between \texttt{rop} and \texttt{u}.
\end{verbatim}

\begin{verbatim}
int padic_exp(padic_t y, const padic_t x, const padic_ctx_t ctx)
Returns whether the \(p\)-adic exponential function converges at the \(p\)-adic number \(x\), and if so sets \texttt{y} to its value.

The \(p\)-adic exponential function is defined by the usual series
\[
\exp_p(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}
\]
but this only converges only when \(\text{ord}_p(x) > 1/(p - 1)\). For elements \(x \in \mathbb{Q}_p\), this means that \(\text{ord}_p(x) \geq 1\) when \(p \geq 3\) and \(\text{ord}_2(x) \geq 2\) when \(p = 2\).
\end{verbatim}
int `padic_exp_rectangular`(`padic_t y, const padic_t x, const padic_ctx_t ctx`)
    Returns whether the \( p \)-adic exponential function converges at the \( p \)-adic number \( x \), and if so sets \( y \) to its value.
    Uses a rectangular splitting algorithm to evaluate the series expression of \( \exp(x) \mod p^N \).

int `padic_exp_balanced`(`padic_t y, const padic_t x, const padic_ctx_t ctx`)
    Returns whether the \( p \)-adic exponential function converges at the \( p \)-adic number \( x \), and if so sets \( y \) to its value.
    Uses a balanced approach, balancing the size of chunks of \( x \) with the valuation and hence the rate of convergence, which results in a quasi-linear algorithm in \( N \), for fixed \( p \).

12.1.10 Logarithm

`slong _padic_log_bound`( `slong v, slong N, const fmpz_t p`)
    Returns \( b \) such that for all \( i \geq b \) we have
    \[ iv - \text{ord}_p(i) \geq N \]
    where \( v \geq 1 \).
    Assumes that \( 1 \leq v < N \) or \( 2 \leq v < N \) when \( p \) is odd or \( p = 2 \), respectively, and also that \( N < 2^{f-2} \) where \( f \) is FLINT_BITS.

void `_padic_log`( `fmpz_t z, const fmpz_t y, slong v, const fmpz_t p, slong N`)
void `_padic_log_rectangular`( `fmpz_t z, const fmpz_t y, slong v, const fmpz_t p, slong N`)
void `_padic_log_satoh`( `fmpz_t z, const fmpz_t y, slong v, const fmpz_t p, slong N`)
void `_padic_log_balanced`( `fmpz_t z, const fmpz_t y, slong v, const fmpz_t p, slong N`)
    Computes
    \[ z = -\sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N}, \]
    reduced modulo \( p^N \).
    Note that this can be used to compute the \( p \)-adic logarithm via the equation
    \[ \log(x) = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i} \]
    \[ = -\sum_{i=1}^{\infty} \frac{(1-x)^i}{i}. \]
    Assumes that \( y = 1 - x \) is non-zero and that \( v = \text{ord}_p(y) \) is at least \( 1 \) when \( p \) is odd and at least \( 2 \) when \( p = 2 \) so that the series converges.
    Assumes that \( v < N \), and hence in particular \( N \geq 2 \).
    Does not support aliasing between \( y \) and \( z \).

int `padic_log`( `padic_t rop, const padic_t op, const padic_ctx_t ctx`)
    Returns whether the \( p \)-adic logarithm function converges at the \( p \)-adic number \( op \), and if so sets \( rop \) to its value.
    The \( p \)-adic logarithm function is defined by the usual series
    \[ \log_p(x) = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i} \]
    but this only converges when \( \text{ord}_p(x-1) \) is at least \( 2 \) or \( 1 \) when \( p = 2 \) or \( p > 2 \), respectively.
int **padic_log_rectangular** (padic_t rop, const padic_t op, const padic_ctx_t ctx)

Returns whether the p-adic logarithm function converges at the p-adic number op, and if so sets rop to its value.

Uses a rectangular splitting algorithm to evaluate the series expression of \(\log(x) \mod p^N\).

int **padic_log_satoh** (padic_t rop, const padic_t op, const padic_ctx_t ctx)

Returns whether the p-adic logarithm function converges at the p-adic number op, and if so sets rop to its value.

Uses an algorithm based on a result of Satoh, Skjernaa and Taguchi that \(\ord_p(a^{p^k} - 1) > k\), which implies that

\[
\log(a) \equiv p^{-k} \left( \log(a^{p^k}) \pmod{p^{N+k}} \right) \pmod{p^N}.
\]

int **padic_log_balanced** (padic_t rop, const padic_t op, const padic_ctx_t ctx)

Returns whether the p-adic logarithm function converges at the p-adic number op, and if so sets rop to its value.

### 12.1.11 Special functions

void **_padic_teichmuller** (fmpz_t rop, const fmpz_t op, const fmpz_t p, slong N)

Computes the Teichmüller lift of the p-adic unit op, assuming that \(N \geq 1\).

Supports aliasing between rop and op.

void **padic_teichmuller** (padic_t rop, const padic_t op, const padic_ctx_t ctx)

Computes the Teichmüller lift of the p-adic unit op.

If op is a p-adic integer divisible by p, sets rop to zero, which satisfies \(t^p - t = 0\), although it is clearly not a \((p - 1)\)-st root of unity.

If op has negative valuation, raises an abort signal.

ulong **padic_val_fac_ui_2** (ulong n)

Computes the 2-adic valuation of n!.

Note that since n fits into an ulong, so does \(\ord_2(n!)\) since \(\ord_2(n!) \leq (n - 1)/(p - 1) = n - 1\).

ulong **padic_val_fac_ui** (ulong n, const fmpz_t p)

Computes the p-adic valuation of n!.

Note that since n fits into an ulong, so does \(\ord_p(n!)\) since \(\ord_p(n!) \leq (n - 1)/(p - 1)\).

void **padic_val_fac** (fmpz_t rop, const fmpz_t op, const fmpz_t p)

Sets rop to the p-adic valuation of the factorial of op, assuming that op is non-negative.

### 12.1.12 Input and output

char * **padic_get_str** (char *str, const padic_t op, const padic_ctx_t ctx)

Returns the string representation of the p-adic number op according to the printing mode set in the context.

If str is NULL then a new block of memory is allocated and a pointer to this is returned. Otherwise, it is assumed that the string str is large enough to hold the representation and it is also the return value.

int **_padic_fprint** (FILE *file, const fmpz_t u, slong v, const padic_ctx_t ctx)
int padic_fprint(FILE *file, const padic_t op, const padic_ctx_t ctx)
   Prints the string representation of the \( p \)-adic number \( \text{op} \) to the stream \( \text{file} \).
   In the current implementation, always returns 1.

int _padic_print(const fmpz_t u, slong v, const padic_ctx_t ctx)
int padic_print(const padic_t op, const padic_ctx_t ctx)
   Prints the string representation of the \( p \)-adic number \( \text{op} \) to the stream \text{stdout}.
   In the current implementation, always returns 1.

void padic_debug(const padic_t op)
   Prints debug information about \( \text{op} \) to the stream \text{stdout}, in the format "(u v N)".

### 12.2 padic_poly.h – polynomials over \( p \)-adic numbers

#### 12.2.1 Module documentation

We represent a polynomial in \( \mathbb{Q}_p[\times] \) as a product \( p^v f(\times) \), where \( p \) is a prime number, \( v \in \mathbb{Z} \) and \( f(\times) \in \mathbb{Z}[\times] \). As a data structure, we call this polynomial \textit{normalised} if the polynomial \( f(\times) \) is \textit{normalised}, that is, if the top coefficient is non-zero. We say this polynomial is in \textit{canonical form} if one of the coefficients of \( f(\times) \) is a \( p \)-adic unit. If \( f(\times) \) is the zero polynomial, we require that \( v = 0 \). We say this polynomial is \textit{reduced} modulo \( p^N \) if it is canonical form and if all coefficients lie in the range \([0, p^N)\).

#### 12.2.2 Memory management

void padic_poly_init(padic_poly_t poly)
   Initialises \( \text{poly} \) for use, setting its length to zero. The precision of the polynomial is set to \text{PADIC_DEFAULT_PREC}. A corresponding call to \text{padic_poly_clear()} must be made after finishing with the \text{padic_poly_t} to free the memory used by the polynomial.

void padic_poly_init2(padic_poly_t poly, slong alloc, slong prec)
   Initialises \( \text{poly} \) with space for at least \( \text{alloc} \) coefficients and sets the length to zero. The allocated coefficients are all set to zero. The precision is set to \( \text{prec} \).

void padic_poly_realloc(padic_poly_t poly, slong alloc, const fmpz_t p)
   Reallocates the given polynomial to have space for \( \text{alloc} \) coefficients. If \( \text{alloc} \) is zero the polynomial is cleared and then reinitialised. If the current length is greater than \( \text{alloc} \) the polynomial is first truncated to length \( \text{alloc} \).

void padic_poly_fit_length(padic_poly_t poly, slong len)
   If \( \text{len} \) is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least \( \text{len} \) coefficients. No data is lost when calling this function.
   The function efficiently deals with the case where \text{fit_length} is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void _padic_poly_set_length(padic_poly_t poly, slong len)
   Denotes the coefficients of \( \text{poly} \) beyond \( \text{len} \) and sets the length of \( \text{poly} \) to \( \text{len} \).
   Note that if the current length is greater than \( \text{len} \) the polynomial may no longer be in canonical form.

void padic_poly_clear(padic_poly_t poly)
   Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.
void _padic_poly_normalise(padic_poly_t poly)
    Sets the length of poly so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void _padic_poly_canonicalise(fmpz *poly, slong *v, slong len, const fmpz_t p)
void padic_poly_canonicalise(padic_poly_t poly, const fmpz_t p)
    Brings the polynomial poly into canonical form, assuming that it is normalised already. Does not carry out any reduction.

void padic_poly_reduce(padic_poly_t poly, const padic_ctx_t ctx)
    Reduces the polynomial poly modulo \( p^N \), assuming that it is in canonical form already.

void padic_poly_truncate(padic_poly_t poly, slong n, const fmpz_t p)
    Truncates the polynomial to length at most \( n \).

12.2.3 Polynomial parameters

slong padic_poly_degree(const padic_poly_t poly)
    Returns the degree of the polynomial poly.

slong padic_poly_length(const padic_poly_t poly)
    Returns the length of the polynomial poly.

slong padic_poly_val(const padic_poly_t poly)
    Returns the valuation of the polynomial poly, which is defined to be the minimum valuation of all its coefficients.
    The valuation of the zero polynomial is \( \sim 0 \).
    Note that this is implemented as a macro and can be used as either a lvalue or a rvalue.

slong padic_poly_prec(padic_poly_t poly)
    Returns the precision of the polynomial poly.
    Note that this is implemented as a macro and can be used as either a lvalue or a rvalue.
    Note that increasing the precision might require a call to padic_poly_reduce().

12.2.4 Randomisation

void padic_poly_randtest(padic_poly_t f, flint_rand_t state, slong len, const padic_ctx_t ctx)
    Sets \( f \) to a random polynomial of length at most \( len \) with entries reduced modulo \( p^N \).

void padic_poly_randtest_not_zero(padic_poly_t f, flint_rand_t state, slong len, const padic_ctx_t ctx)
    Sets \( f \) to a non-zero random polynomial of length at most \( len \) with entries reduced modulo \( p^N \).

void padic_poly_randtest_val(padic_poly_t f, flint_rand_t state, slong val, slong len, const padic_ctx_t ctx)
    Sets \( f \) to a random polynomial of length at most \( len \) with at most the prescribed valuation \( val \) and entries reduced modulo \( p^N \).
    Specifically, we aim to set the valuation to be exactly equal to \( val \), but do not check for additional cancellation when creating the coefficients.
12.2.5 Assignment and basic manipulation

```c
void padic_poly_set_padic(padic_poly_t poly, const padic_t x, const padic_ctx_t ctx)
    Sets the polynomial poly to the \( p \)-adic number \( x \), reduced to the precision of the polynomial.

void padic_poly_set(padic_poly_t poly1, const padic_poly_t poly2, const padic_ctx_t ctx)
    Sets the polynomial poly1 to the polynomial poly2, reduced to the precision of poly1.

void padic_poly_set_si(padic_poly_t poly, slong x, const padic_ctx_t ctx)
    Sets the polynomial poly to the signed slong integer \( x \) reduced to the precision of the polynomial.

void padic_poly_set_ui(padic_poly_t poly, ulong x, const padic_ctx_t ctx)
    Sets the polynomial poly to the unsigned slong integer \( x \) reduced to the precision of the polynomial.

void padic_poly_set_fmpz(padic_poly_t poly, const fmpz_t x, const padic_ctx_t ctx)
    Sets the polynomial poly to the integer \( x \) reduced to the precision of the polynomial.

void padic_poly_set_fmpq(padic_poly_t poly, const fmpq_t x, const padic_ctx_t ctx)
    Sets the polynomial poly to the value of the rational \( x \), reduced to the precision of the polynomial.

void padic_poly_set_fmpz_poly(padic_poly_t rop, const fmpz_poly_t op, const padic_ctx_t ctx)
    Sets the polynomial rop to the integer polynomial op reduced to the precision of the polynomial.

void padic_poly_set_fmpq_poly(padic_poly_t rop, const fmpq_poly_t op, const padic_ctx_t ctx)
    Sets the polynomial rop to the value of the rational polynomial op, reduced to the precision of the polynomial.

int padic_poly_get_fmpz_poly(fmpz_poly_t rop, const padic_poly_t op, const padic_ctx_t ctx)
    Sets the integer polynomial rop to the value of the \( p \)-adic polynomial op and returns 1 if the polynomial is \( p \)-adically integral. Otherwise, returns 0.

void padic_poly_get_fmpq_poly(fmpq_poly_t rop, const padic_poly_t op, const padic_ctx_t ctx)
    Sets rop to the rational polynomial corresponding to the \( p \)-adic polynomial op.

void padic_poly_zero(padic_poly_t poly)
    Sets poly to the zero polynomial.

void padic_poly_one(padic_poly_t poly)
    Sets poly to the constant polynomial 1, reduced to the precision of the polynomial.

void padic_poly_swap(padic_poly_t poly1, padic_poly_t poly2)
    Swaps the two polynomials poly1 and poly2, including their precisions.
    This is done efficiently by swapping pointers.
```

12.2.6 Getting and setting coefficients

```c
void padic_poly_get_coeff_padic(padic_t c, const padic_poly_t poly, slong n, const padic_ctx_t ctx)
    Sets \( c \) to the coefficient of \( x^n \) in the polynomial, reduced modulo the precision of \( c \).

void padic_poly_set_coeff_padic(padic_poly_t f, slong n, const padic_t c, const padic_ctx_t ctx)
    Sets the coefficient of \( x^n \) in the polynomial \( f \) to \( c \), reduced to the precision of the polynomial \( f \).
    Note that this operation can take linear time in the length of the polynomial.
```
12.2.7 Comparison

int padic_poly_equal(const padic_poly_t poly1, const padic_poly_t poly2)
    Returns whether the two polynomials poly1 and poly2 are equal.

int padic_poly_is_zero(const padic_poly_t poly)
    Returns whether the polynomial poly is the zero polynomial.

int padic_poly_is_one(const padic_poly_t poly)
    Returns whether the polynomial poly is equal to the constant polynomial `1`, taking the precision of the polynomial into account.

12.2.8 Addition and subtraction

void _padic_poly_add(fmpz *rop, slong *rval, slong N, const fmpz *op1, slong vall, slong len1, slong N1, const fmpz *op2, slong val2, slong len2, slong N2, const padic_ctx_t ctx)
    Sets (rop, *rval, FLINT_MAX(len1, len2)) to the sum of (op1, vall1, len1) and (op2, val2, len2).
    Assumes that the input is reduced and guarantees that this is also the case for the output.
    Assumes that \( \min\{v_1, v_2\} < N \).
    Supports aliasing between the output and input arguments.

void padic_poly_add(padic_poly_t f, const padic_poly_t g, const padic_poly_t h, const padic_ctx_t ctx)
    Sets \( f \) to the sum \( g + h \).

void _padic_poly_sub(fmpz *rop, slong *rval, slong N, const fmpz *op1, slong vall, slong len1, slong N1, const fmpz *op2, slong val2, slong len2, slong N2, const padic_ctx_t ctx)
    Sets (rop, *rval, FLINT_MAX(len1, len2)) to the difference of (op1, vall1, len1) and (op2, val2, len2).
    Assumes that the input is reduced and guarantees that this is also the case for the output.
    Assumes that \( \min\{v_1, v_2\} < N \).
    Support aliasing between the output and input arguments.

void padic_poly_sub(padic_poly_t f, const padic_poly_t g, const padic_poly_t h, const padic_ctx_t ctx)
    Sets \( f \) to the difference \( g - h \).

void padic_poly_neg(padic_poly_t f, const padic_poly_t g, const padic_ctx_t ctx)
    Sets \( f \) to \(-g\).

12.2.9 Scalar multiplication

void _padic_poly_scalar_mul_padic(fmpz *rop, slong *rval, slong N, const fmpz *op, slong val, slong len, const padic_t c, const padic_ctx_t ctx)
    Sets (rop, *rval, len) to (op, val, len) multiplied by the scalar \( c \).
    The result will only be correctly reduced if the polynomial is non-zero. Otherwise, the array (rop, len) will be set to zero but the valuation *rval might be wrong.

void padic_poly_scalar_mul_padic(padic_poly_t rop, const padic_poly_t op, const padic_t c, const padic_ctx_t ctx)
    Sets the polynomial rop to the product of the polynomial op and the \( p \)-adic number c, reducing the result modulo \( p^N \).
12.2.10 Multiplication

```c
void _padic_poly_mul(fmpz *rop, slong *rval, slong N, const fmpz *op1, slong val1, slong len1, const fmpz *op2, slong val2, slong len2, const padic_ctx_t ctx)
```

Sets \((rop, *rval, \text{len1} + \text{len2} - 1)\) to the product of \((\text{op1}, \text{val1}, \text{len1})\) and \((\text{op2}, \text{val2}, \text{len2})\).

Assumes that the resulting valuation \(*rval\), which is the sum of the valuations \(\text{val1}\) and \(\text{val2}\), is less than the precision \(\sim N\) of the context.

Assumes that \(\text{len1} \geq \text{len2} > 0\).

```c
void padic_poly_mul(padic_poly_t res, const padic_poly_t poly1, const padic_poly_t poly2, const padic_ctx_t ctx)
```

Sets the polynomial \(\text{res}\) to the product of the two polynomials \(\text{poly1}\) and \(\text{poly2}\), reduced modulo \(p^N\).

12.2.11 Powering

```c
void _padic_poly_pow(fmpz *rop, slong *rval, slong N, const fmpz *op, slong val, slong len, ulong e, const padic_ctx_t ctx)
```

Sets the polynomial \((\text{rop}, *rval, e (\text{len} - 1) + 1)\) to the polynomial \((\text{op}, \text{val}, \text{len})\) raised to the power \(\sim e^\cdot\).

Assumes that \(e > 1\) and \(\text{len} > 0\).

Does not support aliasing between the input and output arguments.

```c
void padic_poly_pow(padic_poly_t rop, const padic_poly_t op, ulong e, const padic_ctx_t ctx)
```

Sets the polynomial \(\text{rop}\) to the polynomial \(\text{op}\) raised to the power \(\sim e\), reduced to the precision in \(\text{rop}\).

In the special case \(e = 0\), sets \(\text{rop}\) to the constant polynomial one reduced to the precision of \(\text{rop}\).

Also note that when \(e = 1\), this operation sets \(\text{rop}\) to \(\text{op}\) and then reduces \(\text{rop}\).

When the valuation of the input polynomial is negative, this results in a loss of \(p\)-adic precision.

Suppose that the input polynomial is given to precision \(\sim N\) and has valuation \(\sim v < 0\). The result then has valuation \(ev < 0\) but is only correct to precision \(N + (e - 1)v\).

12.2.12 Series inversion

```c
void padic_poly_inv_series(padic_poly_t g, const padic_poly_t f, slong n, const padic_ctx_t ctx)
```

Computes the power series inverse \(g\) of \(f\) modulo \(X^n\), where \(n \geq 1\).

Given the polynomial \(f \in \mathbb{Q}[X] \subset \mathbb{Q}_p[X]\), there exists a unique polynomial \(f^{-1} \in \mathbb{Q}[X]\) such that \(ff^{-1} = 1\) modulo \(X^n\). This function sets \(g\) to \(f^{-1}\) reduced modulo \(p^N\).

Assumes that the constant coefficient of \(f\) is non-zero.

Moreover, assumes that the valuation of the constant coefficient of \(f\) is minimal among the coefficients of \(f\).

Note that the result \(g\) is zero if and only if \(- \text{ord}_p(f) \geq N\).
12.2.13 Derivative

```c
void _padic_poly_derivative(const fmpz_t rop, fmpz_t rval, slong N, const fmpz_t op, slong val, slong len, const padic_ctx_t ctx)
```

Sets \((rop, rval)\) to the derivative of \((op, val)\) reduced modulo \(p^N\).

Supports aliasing of the input and the output parameters.

```c
void padic_poly_derivative(const padic_poly_t rop, const padic_poly_t op, const padic_ctx_t ctx)
```

Sets \(rop\) to the derivative of \(op\), reducing the result modulo the precision of \(rop\).

12.2.14 Shifting

```c
void _padic_poly_shift_left(const padic_poly_t rop, const padic_poly_t op, slong n, const padic_ctx_t ctx)
```

Notationally, sets the polynomial \(rop\) to the polynomial \(op\) multiplied by \(x^n\), where \(n \geq 0\), and reduces the result.

```c
void _padic_poly_shift_right(const padic_poly_t rop, const padic_poly_t op, slong n, const padic_ctx_t ctx)
```

Notationally, sets the polynomial \(rop\) to the polynomial \(op\) after floor division by \(x^n\), where \(n \geq 0\), ensuring the result is reduced.

12.2.15 Evaluation

```c
void _padic_poly_evaluate_padic(const fmpz_t u, slong *v, slong N, const fmpz_t poly, slong val, slong len, const fmpz_t a, slong b, const padic_ctx_t ctx)
```

```c
void padic_poly_evaluate_padic(padic_t y, const padic_poly_t poly, const padic_t a, const padic_ctx_t ctx)
```

Sets the \(p\)-adic number \(y\) to \(poly\) evaluated at \(a\), reduced in the given context.

Suppose that the polynomial can be written as \(F(X) = p^w f(X)\) with \(\text{ord}_p(f) = 1\), that \(\text{ord}_p(a) = b\) and that both are defined to precision \(\sim N\). Then \(f\) is defined to precision \(N - w\) and so \(f(a)\) is defined to precision \(N - w\) when \(a\) is integral and \(N - w + (n-1)b\) when \(b < 0\), where \(n = \text{deg}(f)\). Thus, \(y = F(a)\) is defined to precision \(N\) when \(a\) is integral and \(N + (n-1)b\) when \(b < 0\).

12.2.16 Composition

```c
void _padic_poly-compose(const fmpz_t rop, slong *rval, slong N, const fmpz_t op1, slong val1, slong len1, const fmpz_t op2, slong val2, slong len2, const padic_ctx_t ctx)
```

Sets \((rop, *rval, (len1-1)*(len2-1)+1)\) to the composition of the two input polynomials, reducing the result modulo \(p^N\).

Assumes that \(len1\) is non-zero.

Does not support aliasing.

```c
void padic_poly-compose(const padic_poly_t rop, const padic_poly_t op1, const padic_poly_t op2, const padic_ctx_t ctx)
```

Sets \(rop\) to the composition of \(op1\) and \(op2\), reducing the result in the given context.

To be clear about the order of composition, let \(f(X)\) and \(g(X)\) denote the polynomials \(op1\) and \(op2\), respectively. Then \(rop\) is set to \(f(g(X))\).
void _padic_poly_compose_pow(fmpz *rop, slong *rval, slong N, const fmpz *op, slong val, slong len, slong k, const padic_ctx_t ctx)

Sets \((rop, \text{val}, (\text{len} - 1) \cdot k + 1)\) to the composition of \((op, \text{val}, \text{len})\) and the monomial \(x^k\), where \(k \geq 1\).

Assumes that \(\text{len}\) is positive.

Supports aliasing between the input and output polynomials.

void padic_poly_compose_pow(padic_poly_t rop, const padic_poly_t op, slong k, const padic_ctx_t ctx)

Sets \(rop\) to the composition of \(op\) and the monomial \(x^k\), where \(k \geq 1\).

Note that no reduction takes place.

### 12.2.17 Input and output

int padic_poly_debug(const padic_poly_t poly)

Prints the data defining the \(p\)-adic polynomial \(poly\) in a simple format useful for debugging purposes.

In the current implementation, always returns 1.

int _padic_poly_fprint(FILE *file, const fmpz *poly, slong val, slong len, const padic_ctx_t ctx)

int padic_poly_fprint(FILE *file, const padic_poly_t poly, const padic_ctx_t ctx)

Prints a simple representation of the polynomial \(poly\) to the stream \(file\).

A non-zero polynomial is represented by the number of coefficients, two spaces, followed by a list of the coefficients, which are printed in a way depending on the print mode,

In the PADIC_TERSE mode, the coefficients are printed as rational numbers.

The PADIC_SERIES mode is currently not supported and will raise an abort signal.

In the PADIC_VAL_UNIT mode, the coefficients are printed in the form \(p^u\).

The zero polynomial is represented by "0".

In the current implementation, always returns 1.

int _padic_poly_print(const fmpz *poly, slong val, slong len, const padic_ctx_t ctx)

int padic_poly_print(const padic_poly_t poly, const padic_ctx_t ctx)

Prints a simple representation of the polynomial \(poly\) to stdout.

In the current implementation, always returns 1.

int _padic_poly_fprint_pretty(FILE *file, const fmpz *poly, slong val, slong len, const char *var, const padic_ctx_t ctx)

int padic_poly_fprint_pretty(FILE *file, const padic_poly_t poly, const char *var, const padic_ctx_t ctx)

int _padic_poly_print_pretty(const fmpz *poly, slong val, slong len, const char *var, const padic_ctx_t ctx)

int padic_poly_print_pretty(const padic_poly_t poly, const char *var, const padic_ctx_t ctx)
12.2.18 Testing

```c
int _padic_poly_isCanonical(const fmpz *op, slong val, slong len, const padic_ctx_t ctx)
int _padic_poly_isCanonical(const padic_poly_t op, const padic_ctx_t ctx)
int _padic_poly_isReduced(const fmpz *op, slong val, slong len, slong N, const padic_ctx_t ctx)
int padic_poly_isReduced(const padic_poly_t op, const padic_ctx_t ctx)
```

12.3 padic_mat.h – matrices over p-adic numbers

12.3.1 Module documentation

We represent a matrix over \( \mathbb{Q}_p \) as a product \( p^v M \), where \( p \) is a prime number, \( v \in \mathbb{Z} \) and \( M \) a matrix over \( \mathbb{Z} \). We say this matrix is in canonical form if either \( M \) is zero, in which case we choose \( v = 0 \), too, or if \( M \) contains at least one \( p \)-adic unit. We say this matrix is reduced modulo \( p^N \) if it is canonical form and if all coefficients of \( M \) lie in the range \( [0, p^{N-v}) \).

12.3.2 Macros

```c
fmpz_mat_struct *padic_mat(const padic_mat_t A)
    Returns a pointer to the unit part of the matrix, which is a matrix over \( \mathbb{Z} \).
    The return value can be used as an argument to the functions in the fmpz_mat module.

fmpz *padic_mat_entry(const padic_mat_t A, slong i, slong j)
    Returns a pointer to unit part of the entry in position \( (i, j) \). Note that this is not necessarily a unit.
    The return value can be used as an argument to the functions in the fmpz module.

slong padic_mat_val(const padic_mat_t A)
    Allow access (as L-value or R-value) to \( \text{val} \) field of \( A \). This function is implemented as a macro.

slong padic_mat_prec(const padic_mat_t A)
    Allow access (as L-value or R-value) to \( \text{prec} \) field of \( A \). This function is implemented as a macro.

slong padic_mat_get_val(const padic_mat_t A)
    Returns the valuation of the matrix.

slong padic_mat_get_prec(const padic_mat_t A)
    Returns the \( p \)-adic precision of the matrix.

slong padic_mat_nrows(const padic_mat_t A)
    Returns the number of rows of the matrix \( A \).

slong padic_mat_ncols(const padic_mat_t A)
    Returns the number of columns of the matrix \( A \).
```
12.3.3 Memory management

```c
void padic_mat_init(padic_mat_t A, slong r, slong c)
    Initialises the matrix A as a zero matrix with the specified numbers of rows and columns and precision PADIC_DEFAULT_PREC.

void padic_mat_init2(padic_mat_t A, slong r, slong c, slong prec)
    Initialises the matrix A as a zero matrix with the specified numbers of rows and columns and the given precision.

void padic_mat_clear(padic_mat_t A)
    Clears the matrix A.

void _padic_mat_canonicalise(padic_mat_t A, const padic_ctx_t ctx)
    Ensures that the matrix A is in canonical form.

void _padic_mat_reduce(padic_mat_t A, const padic_ctx_t ctx)
    Ensures that the matrix A is reduced modulo \(p^N\), assuming that it is in canonical form already.

void padic_mat_reduce(padic_mat_t A, const padic_ctx_t ctx)
    Ensures that the matrix A is reduced modulo \(p^N\), without assuming that it is necessarily in canonical form.

int padic_mat_is_empty(const padic_mat_t A)
    Returns whether the matrix A is empty, that is, whether it has zero rows or zero columns.

int padic_mat_is_square(const padic_mat_t A)
    Returns whether the matrix A is square.

int padic_mat_is_canonical(const padic_mat_t A, const padic_ctx_t p)
    Returns whether the matrix A is in canonical form.
```

12.3.4 Basic assignment

```c
void padic_mat_set(padic_mat_t B, const padic_mat_t A, const padic_ctx_t p)
    Sets B to a copy of A, respecting the precision of B.

void padic_mat_swap(padic_mat_t A, padic_mat_t B)
    Swaps the two matrices A and B. This is done efficiently by swapping pointers.

void padic_mat_swap_entrywise(padic_mat_t mat1, padic_mat_t mat2)
    Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

void padic_mat_zero(padic_mat_t A)
    Sets the matrix A to zero.

void padic_mat_one(padic_mat_t A)
    Sets the matrix A to the identity matrix. If the precision is negative then the matrix will be the zero matrix.
```
12.3.5 Conversions

void **padic_mat_set_fmpq_mat**(padic_mat_t B, const fmpq_mat_t A, const padic_ctx_t ctx)

Sets the p-adic matrix $B$ to the rational matrix $A$, reduced according to the given context.

void **padic_mat_get_fmpq_mat**(fmpq_mat_t B, const padic_mat_t A, const padic_ctx_t ctx)

Sets the rational matrix $B$ to the p-adic matrices $A$; no reduction takes place.

12.3.6 Entries

Because of the choice of the data structure, representing the matrix as $p^v M$, setting an entry of the matrix might lead to changes in all entries in the matrix $M$. Also, a specific entry is not readily available as a p-adic number. Thus, there are separate functions available for getting and setting entries.

void **padic_mat_get_entry_padic**(padic_t rop, const padic_mat_t op, slong i, slong j, const padic_ctx_t ctx)

Sets $rop$ to the entry in position $(i,j)$ in the matrix $op$.

void **padic_mat_set_entry_padic**(padic_mat_t rop, slong i, slong j, const padic_t op, const padic_ctx_t ctx)

Sets the entry in position $(i,j)$ in the matrix to $rop$.

12.3.7 Comparison

int **padic_mat_equal**(const padic_mat_t A, const padic_mat_t B)

Returns whether the two matrices $A$ and $B$ are equal.

int **padic_mat_is_zero**(const padic_mat_t A)

Returns whether the matrix $A$ is zero.

12.3.8 Input and output

int **padic_mat_fprint**(FILE *file, const padic_mat_t A, const padic_ctx_t ctx)

Prints a simple representation of the matrix $A$ to the output stream $file$. The format is the number of rows, a space, the number of columns, two spaces, followed by a list of all the entries, one row after the other.

In the current implementation, always returns 1.

int **padic_mat_fprint_pretty**(FILE *file, const padic_mat_t A, const padic_ctx_t ctx)

Prints a pretty representation of the matrix $A$ to the output stream $file$.

In the current implementation, always returns 1.

int **padic_mat_print**(const padic_mat_t A, const padic_ctx_t ctx)

int **padic_mat_print_pretty**(const padic_mat_t A, const padic_ctx_t ctx)
12.3.9 Random matrix generation

void \texttt{padic\_mat\_randtest}(\texttt{padic\_mat\_t A}, \texttt{flint\_rand\_t state}, const \texttt{padic\_ctx\_t ctx})

Sets $A$ to a random matrix.

The valuation will be in the range $[-\lceil N/10 \rceil, N)$, $[N - \lceil -N/10 \rceil, N)$, or $[-10,0)$ as $N$ is positive, negative or zero.

12.3.10 Transpose

void \texttt{padic\_mat\_transpose}(\texttt{padic\_mat\_t B}, const \texttt{padic\_mat\_t A})

Sets $B$ to $A^t$.

12.3.11 Addition and subtraction

void \texttt{\_padic\_mat\_add}(\texttt{padic\_mat\_t C}, const \texttt{padic\_mat\_t A}, const \texttt{padic\_mat\_t B}, const \texttt{padic\_ctx\_t ctx})

Sets $C$ to the exact sum $A + B$, ensuring that the result is in canonical form.

void \texttt{padic\_mat\_add}(\texttt{padic\_mat\_t C}, const \texttt{padic\_mat\_t A}, const \texttt{padic\_mat\_t B}, const \texttt{padic\_ctx\_t ctx})

Sets $C$ to the sum $A + B$ modulo $p^N$.

void \texttt{\_padic\_mat\_sub}(\texttt{padic\_mat\_t C}, const \texttt{padic\_mat\_t A}, const \texttt{padic\_mat\_t B}, const \texttt{padic\_ctx\_t ctx})

Sets $C$ to the exact difference $A - B$, ensuring that the result is in canonical form.

void \texttt{padic\_mat\_sub}(\texttt{padic\_mat\_t C}, const \texttt{padic\_mat\_t A}, const \texttt{padic\_mat\_t B}, const \texttt{padic\_ctx\_tctx})

Sets $C$ to $A - B$, ensuring that the result is reduced.

void \texttt{padic\_mat\_neg}(\texttt{padic\_mat\_t B}, const \texttt{padic\_mat\_t A})

Sets $B$ to $-A$ in canonical form.

void \texttt{padic\_mat\_neg}(\texttt{padic\_mat\_t B}, const \texttt{padic\_mat\_t A}, const \texttt{padic\_ctx\_t ctx})

Sets $B$ to $-A$, ensuring the result is reduced.

12.3.12 Scalar operations

void \texttt{\_padic\_mat\_scalar\_mul\_padic}(\texttt{padic\_mat\_t B}, const \texttt{padic\_mat\_t A}, const \texttt{padic\_t c}, const \texttt{padic\_ctx\_t ctx})

Sets $B$ to $cA$, ensuring that the result is in canonical form.

void \texttt{padic\_mat\_scalar\_mul\_padic}(\texttt{padic\_mat\_t B}, const \texttt{padic\_mat\_t A}, const \texttt{padic\_t c}, const \texttt{padic\_ctx\_t ctx})

Sets $B$ to $cA$, ensuring that the result is reduced.

void \texttt{\_padic\_mat\_scalar\_mul\_fmpz}(\texttt{padic\_mat\_t B}, const \texttt{padic\_mat\_t A}, const \texttt{fmpz\_t c}, const \texttt{padic\_ctx\_t ctx})

Sets $B$ to $cA$, ensuring that the result is in canonical form.

void \texttt{padic\_mat\_scalar\_mul\_fmpz}(\texttt{padic\_mat\_t B}, const \texttt{padic\_mat\_t A}, const \texttt{fmpz\_t c}, const \texttt{padic\_ctx\_t ctx})

Sets $B$ to $cA$, ensuring that the result is reduced.
void 

```c
    padic_mat_scalar_div_fmpz(padic_mat_t B, const padic_mat_t A, const fmpz_t c, const padic_ctx_t ctx)
```
Sets $B$ to $c^{-1}A$, assuming that $c \neq 0$. Ensures that the result $B$ is reduced.

### 12.3.13 Multiplication

void 

```c
    _padic_mat_mul(padic_mat_t C, const padic_mat_t A, const padic_mat_t B, const padic_ctx_t ctx)
```
Sets $C$ to the product $AB$ of the two matrices $A$ and $B$, ensuring that $C$ is in canonical form.

void 

```c
    padic_mat_mul(padic_mat_t C, const padic_mat_t A, const padic_mat_t B, const padic_ctx_t ctx)
```
Sets $C$ to the product $AB$ of the two matrices $A$ and $B$, ensuring that $C$ is reduced.

### 12.4 qadic.h – unramified extensions over $p$-adic numbers

#### 12.4.1 Data structures

We represent an element of the extension $\mathbb{Q}_q \cong \mathbb{Q}_p[X]/(f(X))$ as a polynomial in $\mathbb{Q}_p[X]$ of degree less than $\deg(f)$. As such, `qadic_struct` and `qadic_t` are typedef'ed as `padic_poly_struct` and `padic_poly_t`.

#### 12.4.2 Context

We represent an unramified extension of $\mathbb{Q}_p$ via $\mathbb{Q}_q \cong \mathbb{Q}_p[X]/(f(X))$, where $f \in \mathbb{Q}_p[X]$ is a monic, irreducible polynomial which we assume to actually be in $\mathbb{Z}[X]$. The first field in the context structure is a $p$-adic context struct `pctx`, which contains data about the prime $p$, precomputed powers, the printing mode etc. The polynomial $f$ is represented as a sparse polynomial using two arrays $j$ and $a$ of length `len`, where $f(X) = \sum_i a_i X^{j_i}$. We also assume that the array $j$ is sorted in ascending order. We choose this data structure to improve reduction modulo $f(X)$ in $\mathbb{Q}_p[X]$, assuming a sparse polynomial $f(X)$ is chosen. The field `var` contains the name of a generator of the extension, which is used when printing the elements.

void 

```c
    qadic_ctx_init(qadic_ctx_t ctx, const fmpz_t p, slong d, slong min, slong max, const char *var, enum padic_print_mode mode)
```
Initialises the context `ctx` with prime $p$, extension degree $d$, variable name `var` and printing mode `mode`. The defining polynomial is chosen as a Conway polynomial if possible and otherwise as a random sparse polynomial.

Stores powers of $p$ with exponents between `min` (inclusive) and `max` exclusive. Assumes that `min` is at most `max`.

Assumes that $p$ is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

Assumes that the printing mode is one of `PADIC_TERSE`, `PADIC_SERIES`, or `PADIC_VAL_UNIT`.

This function also carries out some relevant precomputation for arithmetic in $\mathbb{Q}_p/(p^N)$ such as powers of $p$ close to $p^N$.

void 

```c
    qadic_ctx_init_conway(qadic_ctx_t ctx, const fmpz_t p, slong d, slong min, slong max, const char *var, enum padic_print_mode mode)
```
Initialises the context `ctx` with prime $p$, extension degree $d$, variable name `var` and printing mode `mode`. The defining polynomial is chosen as a Conway polynomial, hence has restrictions on the prime and the degree.
Stores powers of \( p \) with exponents between \( \min \) (inclusive) and \( \max \) exclusive. Assumes that \( \min \) is at most \( \max \).

Assumes that \( p \) is a prime.
Assumes that the string \( \text{var} \) is a null-terminated string of length at least one.
Assumes that the printing mode is one of \texttt{PADIC_TERSE}, \texttt{PADIC_SERIES}, or \texttt{PADIC_VAL_UNIT}.
This function also carries out some relevant precomputation for arithmetic in \( \mathbb{Q}_p/(p^N) \) such as powers of \( p \) close to \( p^N \).

```c
void qadic_ctx_clear(qadic_ctx_t ctx)
```
Clears all memory that has been allocated as part of the context.

```c
slong qadic_ctx_degree(const qadic_ctx_t ctx)
```
Returns the extension degree.

```c
void qadic_ctx_print(const qadic_ctx_t ctx)
```
Prints the data from the given context.

### 12.4.3 Memory management

```c
void qadic_init(qadic_t rop)
```
Initialises the element \( \text{rop} \), setting its value to \( 0 \).

```c
void qadic_init2(qadic_t rop, slong prec)
```
Initialises the element \( \text{rop} \) with the given output precision, setting the value to \( 0 \).

```c
void qadic_clear(qadic_t rop)
```
Clears the element \( \text{rop} \).

```c
void _fmpz_poly_reduce(fmpz *R, slong lenR, const fmpz *a, const slong *j, slong len)
```
Reduces a polynomial \( (R, \text{lenR}) \) modulo a sparse monic polynomial \( f(X) = \sum_i a_i X^{j_i} \) of degree at least 2.
Assumes that the array \( j \) of positive length \( \text{len} \) is sorted in ascending order.
Allows zero-padding in \( (R, \text{lenR}) \).

```c
void _fmpz_mod_poly_reduce(fmpz *R, slong lenR, const fmpz *a, const slong *j, slong len, const fmpz_t p)
```
Reduces a polynomial \( (R, \text{lenR}) \) modulo a sparse monic polynomial \( f(X) = \sum_i a_i X^{j_i} \) of degree at least 2 in \( \mathbb{Z}/(p) \), where \( p \) is typically a prime power.
Assumes that the array \( j \) of positive length \( \text{len} \) is sorted in ascending order.
Allows zero-padding in \( (R, \text{lenR}) \).

```c
void qadic_reduce(qadic_t rop, const qadic_ctx_t ctx)
```
Reduces \( \text{rop} \) modulo \( f(X) \) and \( p^N \).
12.4.4 Properties

\texttt{slong qadic\_val(const qadic\_t op)}

Returns the valuation of \texttt{op}.

\texttt{slong qadic\_prec(const qadic\_t op)}

Returns the precision of \texttt{op}.

12.4.5 Randomisation

\texttt{void qadic\_randtest(qadic\_t rop, flint\_rand\_t state, const qadic\_ctx\_t ctx)}

Generates a random element of \( \mathbb{Q}_q \).

\texttt{void qadic\_randtest\_not\_zero(qadic\_t rop, flint\_rand\_t state, const qadic\_ctx\_t ctx)}

Generates a random non-zero element of \( \mathbb{Q}_q \).

\texttt{void qadic\_randtest\_val(qadic\_t rop, flint\_rand\_t state, slong v, const qadic\_ctx\_t ctx)}

Generates a random element of \( \mathbb{Q}_q \) with prescribed valuation \( v \).

Note that if \( v \geq N \) then the element is necessarily zero.

\texttt{void qadic\_randtest\_int(qadic\_t rop, flint\_rand\_t state, const qadic\_ctx\_t ctx)}

Generates a random element of \( \mathbb{Q}_q \) with non-negative valuation.

12.4.6 Assignments and conversions

\texttt{void qadic\_set(qadic\_t rop, const qadic\_t op, const qadic\_ctx\_t ctx)}

Sets \texttt{rop} to \texttt{op}.

\texttt{void qadic\_zero(qadic\_t rop)}

Sets \texttt{rop} to zero.

\texttt{void qadic\_one(qadic\_t rop)}

Sets \texttt{rop} to one, reduced in the given context.

Note that if the precision \( N \) is non-positive then \texttt{rop} is actually set to zero.

\texttt{void qadic\_gen(qadic\_t rop, const qadic\_ctx\_t ctx)}

Sets \texttt{rop} to the generator \( X \) for the extension when \( N > 0 \), and zero otherwise. If the extension degree is one, raises an abort signal.

\texttt{void qadic\_set\_ui(qadic\_t rop, ulong op, const qadic\_ctx\_t ctx)}

Sets \texttt{rop} to the integer \texttt{op}, reduced in the context.

\texttt{int qadic\_get\_padic(qadic\_t rop, const qadic\_t op, const qadic\_ctx\_t ctx)}

If the element \texttt{op} lies in \( \mathbb{Q}_p \), sets \texttt{rop} to its value and returns 1; otherwise, returns 0.

12.4.7 Comparison

\texttt{int qadic\_is\_zero(const qadic\_t op)}

Returns whether \texttt{op} is equal to zero.

\texttt{int qadic\_is\_one(const qadic\_t op)}

Returns whether \texttt{op} is equal to one in the given context.

\texttt{int qadic\_equal(const qadic\_t op1, const qadic\_t op2)}

Returns whether \texttt{op1} and \texttt{op2} are equal.
12.4.8 Basic arithmetic

void qadic_add(qadic_t rop, const qadic_t op1, const qadic_t op2, const qadic_ctx_t ctx)
        Sets rop to the sum of op1 and op2.
        Assumes that both op1 and op2 are reduced in the given context and ensures that rop is, too.

void qadic_sub(qadic_t rop, const qadic_t op1, const qadic_t op2, const qadic_ctx_t ctx)
        Sets rop to the difference of op1 and op2.
        Assumes that both op1 and op2 are reduced in the given context and ensures that rop is, too.

void qadic_neg(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
        Sets rop to the negative of op.
        Assumes that op is reduced in the given context and ensures that rop is, too.

void qadic_mul(qadic_t rop, const qadic_t op1, const qadic_t op2, const qadic_ctx_t ctx)
        Sets rop to the product of op1 and op2, reducing the output in the given context.

void _qadic_inv(fmpz *rop, const fmpz *op, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N)
        Sets (rop, d) to the inverse of (op, len) modulo f(X) given by (a,j,lena) and p^N.
        Assumes that (op,len) has valuation 0, that is, that it represents a p-adic unit.
        Assumes that len is at most d.
        Does not support aliasing.

void qadic_inv(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
        Sets rop to the inverse of op, reduced in the given context.

void _qadic_pow(fmpz *rop, const fmpz *op, slong len, const fmpz_t e, const fmpz *a, const slong *j, slong lena, const fmpz_t p)
        Sets (rop, 2*d-1) to (op,len) raised to the power e, reduced modulo f(X) given by (a, j, lena) and p, which is expected to be a prime power.
        Assumes that e ≥ 0 and that len is positive and at most d.
        Although we require that rop provides space for 2d – 1 coefficients, the output will be reduced modulo f(X), which is a polynomial of degree d.
        Does not support aliasing.

void qadic_pow(qadic_t rop, const qadic_t op, const fmpz_t e, const qadic_ctx_t ctx)
        Sets rop the op raised to the power e.
        Currently assumes that e ≥ 0.
        Note that for any input op, rop is set to one in the given context whenever e = 0.

12.4.9 Square root

int qadic_sqrt(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
        Return 1 if the input is a square (to input precision). If so, set rop to a square root (truncated to output precision).
12.4.10 Special functions

```c
void _qadic_exp_rectangular(fmpz *rop, const fmpz *op, slong v, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N, const fmpz_t pN)
```

Sets \((rop, 2*d - 1)\) to the exponential of \((op, v, len)\) reduced modulo \(p^N\), assuming that the series converges.

Assumes that \((op, v, len)\) is non-zero.

Does not support aliasing.

```c
int qadic_exp_rectangular(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
```

Returns whether the exponential series converges at \(op\) and sets \(rop\) to its value reduced modulo the given context.

```c
void _qadic_exp_balanced(fmpz *rop, const fmpz *x, slong v, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N, const fmpz_t pN)
```

Sets \((rop, d)\) to the exponential of \((op, v, len)\) reduced modulo \(p^N\), assuming that the series converges.

Assumes that \(len\) is in \([1, d]\) but supports zero padding, including the special case when \((op, len)\) is zero.

Supports aliasing between \(rop\) and \(op\).

```c
int qadic_exp_balanced(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
```

Returns whether the exponential series converges at \(op\) and sets \(rop\) to its value reduced modulo the given context.

```c
void _qadic_exp(fmpz *rop, const fmpz *op, slong v, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N, const fmpz_t pN)
```

Sets \((rop, 2*d - 1)\) to the exponential of \((op, v, len)\) reduced modulo \(p^N\), assuming that the series converges.

Assumes that \((op, v, len)\) is non-zero.

Does not support aliasing.

```c
int qadic_exp(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
```

Returns whether the exponential series converges at \(op\) and sets \(rop\) to its value reduced modulo the given context.

The exponential series converges if the valuation of \(op\) is at least 2 or 1 when \(p\) is even or odd, respectively.

```c
void _qadic_log_rectangular(fmpz *z, const fmpz *y, slong v, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N, const fmpz_t pN)
```

Computes

\[
z = - \sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N}.
\]

Note that this can be used to compute the \(p\)-adic logarithm via the equation

\[
\log(x) = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x - 1)^i}{i}
\]

\[
= - \sum_{i=1}^{\infty} \frac{(1 - x)^i}{i}.
\]

Assumes that \(y = 1 - x\) is non-zero and that \(v = \text{ord}_p(y)\) is at least 1 when \(p\) is odd and at least 2 when \(p = 2\) so that the series converges.
Assumes that $y$ is reduced modulo $p^N$.
Assumes that $v < N$, and in particular $N \geq 2$.
Supports aliasing between $y$ and $z$.

```c
int qadic_log_rectangular(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
Returns whether the $p$-adic logarithm function converges at op, and if so sets rop to its value.
```

```c
void _qadic_log_balanced(fmpz *z, const fmpz *y, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N, const fmpz_t pN)
Computes $(z, d)$ as
$$z = -\sum_{i=1}^{\infty} \frac{y_i}{i} \pmod{p^N}.$$ 
Assumes that $v = \text{ord}_p(y)$ is at least 1 when $p$ is odd and at least 2 when $p = 2$ so that the series converges.
Supports aliasing between $z$ and $y$.
```

```c
int qadic_log_balanced(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
Returns whether the $p$-adic logarithm function converges at op, and if so sets rop to its value.
```

```c
void _qadic_log(fmpz *z, const fmpz *y, slong v, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N, const fmpz_t pN)
Computes $(z, d)$ as
$$z = -\sum_{i=1}^{\infty} \frac{y_i}{i} \pmod{p^N}.$$ 
Note that this can be used to compute the $p$-adic logarithm via the equation
$$\log(x) = \sum_{i=1}^{\infty} \frac{(-1)^{i-1} (x - 1)^i}{i} = -\sum_{i=1}^{\infty} \frac{(1 - x)^i}{i}.$$ 
Assumes that $y = 1 - x$ is non-zero and that $v = \text{ord}_p(y)$ is at least 1 when $p$ is odd and at least 2 when $p = 2$ so that the series converges.
Assumes that $(y, d)$ is reduced modulo $p^N$.
Assumes that $v < N$, and hence in particular $N \geq 2$.
Supports aliasing between $z$ and $y$.
```

```c
int qadic_log(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
Returns whether the $p$-adic logarithm function converges at op, and if so sets rop to its value.
```

The $p$-adic logarithm function is defined by the usual series
$$\log_p(x) = \sum_{i=1}^{\infty} \frac{(-1)^{i-1} (x - 1)^i}{i}$$
but this only converges when $\text{ord}_p(x)$ is at least 2 or 1 when $p = 2$ or $p > 2$, respectively.

```c
void _qadic_frobenius_a(fmpz *rop, slong e, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N)
Computes $\sigma^e(X) \mod p^N$ where $X$ is such that $Q_q \cong Q_p[X]/(f(X))$.
Assumes that the precision $N$ is at least 2 and that the extension is non-trivial, i.e. $d \geq 2$.
Assumes that $0 < e < d$.
Sets $(rop, 2d-1)$, although the actual length of the output will be at most $d$. 
```
void _qadic_frobenius(fmpz *rop, const fmpz *op, slong len, slong e, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N)
Sets (rop, 2*d-1) to Σ evaluated at (op, len).
Assumes that len is positive but at most d.
Assumes that 0 < e < d.
Does not support aliasing.

void qadic_frobenius(qadic_t rop, const qadic_t op, slong e, const qadic_ctx_t ctx)
Evaluates the homomorphism Σe at op.
Recall that \( \mathbb{Q}_q/\mathbb{Q}_p \) is Galois with Galois group \( \langle \Sigma \rangle \cong \langle \sigma \rangle \), which is also isomorphic to \( \mathbb{Z}/d\mathbb{Z} \), where \( \sigma \in \text{Gal}(\mathbb{F}_q/\mathbb{F}_p) \) is the Frobenius element \( \sigma : x \mapsto x^p \) and Σ is its lift to \( \text{Gal}(\mathbb{Q}_q/\mathbb{Q}_p) \).
This functionality is implemented as GaloisImage() in Magma.

void _qadic_teichmuller(fmpz *rop, const fmpz *op, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N)
Sets (rop, d) to the Teichmüller lift of (op, len) modulo \( p^N \).
Does not support aliasing.

void qadic_teichmuller(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
Sets rop to the Teichmüller lift of op to the precision given in the context.
For a unit op, this is the unique \((q - 1)\)th root of unity which is congruent to op modulo p.
Sets rop to zero if op is zero in the given context.
Raises an exception if the valuation of op is negative.

void _qadic_trace(fmpz_t rop, const fmpz *op, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t pN)
void qadic_trace(padic_t rop, const qadic_t op, const qadic_ctx_t ctx)
Sets rop to the trace of op.
For an element \( a \in \mathbb{Q}_q \), multiplication by a defines a \( \mathbb{Q}_p \)-linear map on \( \mathbb{Q}_q \). We define the trace of a as the trace of this map. Equivalently, if Σ generates \( \text{Gal}(\mathbb{Q}_q/\mathbb{Q}_p) \) then the trace of a is equal to \( \sum_{d-1}^{d-1} \Sigma^i(a) \).

void _qadic_norm(fmpz_t rop, const fmpz *op, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N)
void qadic_norm(padic_t rop, const qadic_t op, const qadic_ctx_t ctx)
Sets rop to the norm of the element (op, len) in \( \mathbb{Z}_q \) to precision N, where len is at least one.
The result will be reduced modulo \( p^N \).
Note that whenever (op, len) is a unit, so is its norm. Thus, the output rop of this function will typically not have to be canonicalised or reduced by the caller.

void qadic_norm(padic_t rop, const qadic_t op, const qadic_ctx_t ctx)
Computes the norm of op to the given precision.
Algorithm selection is automatic depending on the input.

void qadic_norm_analytic(padic_t rop, const qadic_t op, const qadic_ctx_t ctx)
Whenever op has valuation greater than \((p - 1)^{-1}\), this routine computes its norm rop via
\[
\text{Norm}(x) = \exp\left(\left(\text{Trace} \log(x)\right)\right).
\]
In the special case that op lies in \( \mathbb{Q}_p \), returns its norm as \( \text{Norm}(x) = x^d \), where \( d \) is the extension degree.
Otherwise, raises an abort signal.
The complexity of this implementation is quasi-linear in \( d \) and N, and polynomial in \( \log p \).
void qadic_norm_resultant(padic_t rop, const qadic_t op, const qadic_ctx_t ctx)
    Sets rop to the norm of op, using the formula

    \[ \text{Norm}(x) = \ell(f)^{-\deg(a)} \text{Res}(f(X), a(X)), \]

    where \( Q_p \cong \mathbb{Q}_p[X]/(f(X)) \), \( \ell(f) \) is the leading coefficient of \( f(X) \), and \( a(X) \in \mathbb{Q}_p[X] \) denotes the same polynomial as \( x \).

    The complexity of the current implementation is given by \( O(d^4 M(N \log p)) \), where \( M(n) \) denotes the complexity of multiplying to \( n \)-bit integers.

### 12.4.11 Output

int qadic_fprint_pretty(FILE *file, const qadic_t op, const qadic_ctx_t ctx)
    Prints a pretty representation of op to file.

    In the current implementation, always returns 1. The return code is part of the function’s signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

int qadic_print_pretty(const qadic_t op, const qadic_ctx_t ctx)
    Prints a pretty representation of op to stdout.

    In the current implementation, always returns 1. The return code is part of the function’s signature to allow for a later implementation to return the number of characters printed or a non-positive error code.
13.1 double_extras.h – support functions for double arithmetic

13.1.1 Random functions

double \texttt{d\_randtest}(\texttt{flint\_rand\_t} state)

Returns a random number in the interval \([0.5, 1)\).

double \texttt{d\_randtest\_signed}(\texttt{flint\_rand\_t} state, \texttt{slong} minexp, \texttt{slong} maxexp)

Returns a random signed number with exponent between \texttt{minexp} and \texttt{maxexp} or zero.

double \texttt{d\_randtest\_special}(\texttt{flint\_rand\_t} state, \texttt{slong} minexp, \texttt{slong} maxexp)

Returns a random signed number with exponent between \texttt{minexp} and \texttt{maxexp}, zero, \texttt{D\_NAN} or \(\pm\texttt{D\_INF}\).

13.1.2 Arithmetic

double \texttt{d\_polyval}(\texttt{const double *}poly, \texttt{int} len, \texttt{double} x)

Uses Horner's rule to evaluate the polynomial defined by the given \texttt{len} coefficients. Requires that \texttt{len} is nonzero.

13.1.3 Special functions

double \texttt{d\_lambertw}(\texttt{double} x)

Computes the principal branch of the Lambert W function, solving the equation \(x = W(x) \exp(W(x))\). If \(x < -1/e\), the solution is complex, and NaN is returned.

Depending on the magnitude of \(x\), we start from a piecewise rational approximation or a zeroth-order truncation of the asymptotic expansion at infinity, and perform 0, 1 or 2 iterations with Halley's method to obtain full accuracy.

A test of \(10^7\) random inputs showed a maximum relative error smaller than 0.95 times \texttt{DBL\_EPSILON} \((2^{-52})\) for positive \(x\). Accuracy for negative \(x\) is slightly worse, and can grow to about 10 times \texttt{DBL\_EPSILON} close to \(-1/e\). However, accuracy may be worse depending on compiler flags and the accuracy of the system \texttt{libm} functions.

int \texttt{d\_is\_nan}(\texttt{double} x)

Returns a nonzero integral value if \(x\) is \texttt{D\_NAN}, and otherwise returns 0.

double \texttt{d\_log2}(\texttt{double} x)

Returns the base 2 logarithm of \(x\) provided \(x\) is positive. If a domain or pole error occurs, the appropriate error value is returned.
13.2 d_vec.h – double precision vectors

13.2.1 Memory management

double * _d_vec_init (slong len)
Returns an initialised vector of doubles of given length. The entries are not zeroed.

void _d_vec_clear (double *vec)
Frees the space allocated for vec.

13.2.2 Randomisation

void _d_vec_randtest (double *f, flint_rand_t state, slong len, slong minexp, slong maxexp)
Sets the entries of a vector of the given length to random signed numbers with exponents between minexp and maxexp or zero.

13.2.3 Assignment and basic manipulation

void _d_vec_set (double *vec1, const double *vec2, slong len2)
Makes a copy of (vec2, len2) into vec1.

void _d_vec_zero (double *vec, slong len)
Zeroes the entries of (vec, len).

13.2.4 Comparison

int _d_vec_equal (const double *vec1, const double *vec2, slong len)
Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

int _d_vec_is_zero (const double *vec, slong len)
Returns 1 if (vec, len) is zero, and 0 otherwise.

int _d_vec_is_approx_zero (const double *vec, slong len, double eps)
Returns 1 if the entries of (vec, len) are zero to within eps, and 0 otherwise.

int _d_vec_approx_equal (const double *vec1, const double *vec2, slong len, double eps)
Compares two vectors of the given length and returns 1 if their entries are within eps of each other, otherwise returns 0.

13.2.5 Addition and subtraction

void _d_vec_add (double *res, const double *vec1, const double *vec2, slong len2)
Sets (res, len2) to the sum of (vec1, len2) and (vec2, len2).

void _d_vec_sub (double *res, const double *vec1, const double *vec2, slong len2)
Sets (res, len2) to (vec1, len2) minus (vec2, len2).
13.2.6 Dot product and norm

double _d_vec_dot(const double *vec1, const double *vec2, slong len2)

Returns the dot product of (vec1, len2) and (vec2, len2).

double _d_vec_norm(const double *vec, slong len)

Returns the square of the Euclidean norm of (vec, len).

double _d_vec_dot_heuristic(const double *vec1, const double *vec2, slong len2, double *err)

Returns the dot product of (vec1, len2) and (vec2, len2) by adding up the positive and negative products, and doing a single subtraction of the two sums at the end. err is a pointer to a double in which an error bound for the operation will be stored.

double _d_vec_dot_thrice(const double *vec1, const double *vec2, slong len2, double *err)

Returns the dot product of (vec1, len2) and (vec2, len2) using error-free floating point sums and products to compute the dot product with three times (thrice) the working precision. err is a pointer to a double in which an error bound for the operation will be stored.

This implements the algorithm of Ogita-Rump-Oishi. See http://www.ti3.tuhh.de/paper/rump/OgRuOi05.pdf.

13.3 d_mat.h – double precision matrices

13.3.1 Memory management

void d_mat_init(d_mat_t mat, slong rows, slong cols)

Initialises a matrix with the given number of rows and columns for use.

void d_mat_clear(d_mat_t mat)

Clears the given matrix.

13.3.2 Basic assignment and manipulation

void d_mat_set(d_mat_t mat1, const d_mat_t mat2)

Sets mat1 to a copy of mat2. The dimensions of mat1 and mat2 must be the same.

void d_mat_swap_entrywise(d_mat_t mat1, d_mat_t mat2)

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

double d_mat_entry(d_mat_t mat, slong i, slong j)

Returns the entry of mat at row i and column j. Both i and j must not exceed the dimensions of the matrix. This function is implemented as a macro.

double d_mat_get_entry(const d_mat_t mat, slong i, slong j)

Returns the entry of mat at row i and column j. Both i and j must not exceed the dimensions of the matrix.

double *d_mat_entry_ptr(const d_mat_t mat, slong i, slong j)

Returns a pointer to the entry of mat at row i and column j. Both i and j must not exceed the dimensions of the matrix.

void d_mat_zero(d_mat_t mat)

Sets all entries of mat to 0.
13.3.3 Random matrix generation

void d_mat_randtest(d_mat_t mat, flint_rand_t state, slong minexp, slong maxexp)

Sets the entries of mat to random signed numbers with exponents between minexp and maxexp or zero.

13.3.4 Input and output

void d_mat_print(const d_mat_t mat)

Prints the given matrix to the stream stdout.

13.3.5 Comparison

int d_mat_equal(const d_mat_t mat1, const d_mat_t mat2)

Returns a non-zero value if mat1 and mat2 have the same dimensions and entries, and zero otherwise.

int d_mat_approx_equal(const d_mat_t mat1, const d_mat_t mat2, double eps)

Returns a non-zero value if mat1 and mat2 have the same dimensions and entries within eps of each other, and zero otherwise.

int d_mat_is_square(const d_mat_t mat)

Returns a non-zero value if the number of rows is equal to the number of columns in mat, and otherwise returns zero.

13.3.6 Transpose

void d_mat_transpose(d_mat_t B, const d_mat_t A)

Sets B to \( A^T \), the transpose of A. Dimensions must be compatible. A and B are allowed to be the same object if A is a square matrix.

13.3.7 Matrix multiplication

void d_mat_mul_classical(d_mat_t C, const d_mat_t A, const d_mat_t B)

Sets C to the matrix product \( C = AB \). The matrices must have compatible dimensions for matrix multiplication (an exception is raised otherwise). Aliasing is allowed.

13.4 mpfr_vec.h – vectors of MPFR floating-point numbers

13.4.1 Memory management

mpfr_ptr _mpfr_vec_init(slong len, flint_bitcnt_t prec)

Returns a vector of the given length of initialised mpfr’s with the given exact precision.

void _mpfr_vec_clear(mpfr_ptr vec, slong len)

Clears the given vector.
13.4.2 Arithmetic

void _mpfr_vec_zero(mpfr_ptr vec, slong len)
    Zeros the vector (vec, len).

void _mpfr_vec_set(mpfr_ptr vec1, mpfr_srcptr vec2, slong len)
    Copies the vector vec2 of the given length into vec1. No check is made to ensure vec1 and vec2
    are different.

void _mpfr_vec_add(mpfr_ptr res, mpfr_srcptr vec1, mpfr_srcptr vec2, slong len)
    Adds the given vectors of the given length together and stores the result in res.

void _mpfr_vec_scalar_mul_mpfr(mpfr_ptr res, mpfr_srcptr vec, slong len, mpfr_t c)
    Multiplies the vector with given length by the scalar c and sets res to the result.

void _mpfr_vec_scalar_mul_2exp(mpfr_ptr res, mpfr_srcptr vec, slong len, flint_bitcnt_t exp)
    Multiplies the given vector of the given length by 2^exp.

void _mpfr_vec_scalar_product(mpfr_t res, mpfr_srcptr vec1, mpfr_srcptr vec2, slong len)
    Sets res to the scalar product of (vec1, len) with (vec2, len). Assumes len > 0.

13.5 mpfr_mat.h – matrices of MPFR floating-point numbers

13.5.1 Memory management

void mpfr_mat_init(mpfr_mat_t mat, slong rows, slong cols, mpfr_prec_t prec)
    Initialises a matrix with the given number of rows and columns and the given precision for use.
    The precision is the exact precision of the entries.

void mpfr_mat_clear(mpfr_mat_t mat)
    Clears the given matrix.

13.5.2 Basic manipulation

__mpfr_struct *mpfr_mat_entry(const mpfr_mat_t mat, slong i, slong j)
    Return a reference to the entry at row i and column j of the given matrix. The values i and j
    must be within the bounds for the matrix. The reference can be used to either return or set the
    given entry.

void mpfr_mat_swap(mpfr_mat_t mat1, mpfr_mat_t mat2)
    Efficiently swap matrices mat1 and mat2.

void mpfr_mat_swap_entrywise(mpfr_mat_t mat1, mpfr_mat_t mat2)
    Swaps two matrices by swapping the individual entries rather than swapping the contents of the
    structs.

void mpfr_mat_set(mpfr_mat_t mat1, const mpfr_mat_t mat2)
    Set mat1 to the value of mat2.

void mpfr_mat_zero(mpfr_mat_t mat)
    Set mat to the zero matrix.
13.5.3 Comparison

```c
int mpfr_mat_equal(const mpfr_mat_t mat1, const mpfr_mat_t mat2)
    Return 1 if the two given matrices are equal, otherwise return 0.
```

13.5.4 Randomisation

```c
void mpfr_mat_randtest(mpfr_mat_t mat, flint_rand_t state)
    Generate a random matrix with random number of rows and columns and random entries for use in test code.
```

13.5.5 Basic arithmetic

```c
void mpfr_mat_mul_classical(mpfr_mat_t C, const mpfr_mat_t A, const mpfr_mat_t B,
    mpfr_rnd_t rnd)
    Set C to the product of A and B with the given rounding mode, using the classical algorithm.
```


14.1 flint_ctypes - Python interface

There is a Python wrapper (flint_ctypes) included with FLINT available in the src/python directory. This wrapper is not currently officially supported and should not be used in production, but it can be useful for experimenting with FLINT.

14.1.1 Introduction

Examples:

```python
>>> from flint_ctypes import *
>>> QQ.bernoulli(50)
495057205241079648212477525/66
>>> sign, primes, exponents = _.factor()
>>> sign
1
>>> primes
[5, 417202699, 47464429777438199, 2, 3, 11]
>>> exponents
[2, 1, 1, -1, -1, -1]
>>> sign * (primes ** exponents).product()
495057205241079648212477525/66
```

Types, parents and coercions

```python
>>> ZZ(5)
5
>>> _.parent()
Integer ring (fmpz)
>>> QQ(5)
5
>>> _.parent()
Rational field (fmpq)
>>> ZZ(10) / ZZ(6)
Traceback (most recent call last):
...
FlintDomainError: x / y is not an element of {Integer ring (fmpz)} for {x = 10}, {y = 6}
>>> x = QQ(1) / 2; x ** x
Traceback (most recent call last):
...
```

(continues on next page)
Flint DomainError: \( x \times y \) is not an element of \( \mathbb{R}(\text{fmpq}) \) for \( x = 1/2 \), \( y = 1/2 \)

\[
\begin{align*}
\text{>>> } & ZZ(10) / QQ(6) \\
& 5/3 \\
\text{>>> } & x = QQbar(1) / 2; x ** x \\
& \text{Root } a = 0.707107 \text{ of } 2*a^2 - 1
\end{align*}
\]

Real and complex numbers

\[
\begin{align*}
\text{>>> } & RR.zeta(2) \\
& [1.644934066848226 +/- 4.57e-16] \\
\text{>>> } & RR.prec = 128 \\
\text{>>> } & RR.zeta(2) \\
& [1.644934066848226436472415166664602518922 +/- 2.88e-39] \\
\text{>>> } & RR.prec = 53 \quad \# \text{restore default}
\end{align*}
\]

14.1.2 API documentation
15.1 References

(In the PDF edition, this section is empty. See the bibliography listing at the end of the document.)

CHAPTER
SIXTEEN

VERSION HISTORY

16.1 History and changes

16.1.1 FLINT version history

2024 – FLINT 3.1.0-dev

Main contributors: Fredrik Johansson (FJ), Albin Ahlbäck (AA), Jean Kieffer (JK).

- Features
  - New module `acb_theta` for computing complex Riemann theta functions with characteristics in any dimension (JK).
  - `flint_printf` and related functions now supports printing common FLINT types, e.g. using the format string `%fmpz` for `fmpz_t` (AA).
  - Generic expression parsing: `gr_set_str` supports parsing expressions like `a*x^2+1/3` or `(0.1 + 0.2*i) +/- (0.001 + 0.001*i)` in any ring (FJ).
  - Added `fmpz_mpoly_q_set_str_pretty` and `fmpz_mpoly_q_get_str_pretty`.
  - Added `flint_mpn_mulhigh` (AA).
  - Primality testing for Gaussian integers (`fmpzi_is_prime`, `fmpzi_is_probabprime`) (Mathieu Gouttenoire).
  - Modular splitting evaluation of polynomials (`_gr_poly_evaluate_modular`) (David Berghaus).
  - Reversion of generic power series (`gr_poly_revert_series` and variants) (FJ).
  - Support inversion for `gr` vectors (FJ).
  - Split generic division into `gr_div` and `gr_div_nonunique` to make the semantics of division more precise (FJ).
  - Added `gr_ctx_is_zero_ring` (FJ).
  - Added `nmod_divides` (FJ).
  - Reciprocal Fibonacci constant (`arb_const_reciprocal_fibonacci`) (FJ).
  - Added functions for working with symmetric positive-definite matrices (`fmpz_mat_is_spd`, `arb_mat_spd_get_fmpz_mat`, `arb_mat_spd_is_lil_reduced`, `arb_mat_spd_lil_reduce`, `arb_mat_randtest_cho`, `arb_mat_randtest_spd`) (JK).
– Added **acb_urandom** and **arb_randtest_positive** (JK).
– Added **acb_mul_i_pow_si** (JK).
– Handle modulus 1 in **fmpz_CRT** functions (Fabian Gundlach).
– Added **_nmod_poly_conway** and **_nmod_poly_conway_rand** (AA).
– Added **fft_small_mulmod_satisfies_bounds**. The function **sd_fft_ctx_init** now verifies that the modulus satisfies the assumptions for correct modular arithmetic. (Daniel Schultz).
– Allow setting generator names for **gr_mpoly**, **gr_series**, **fmpz_mpoly** and **fmpz_mpoly_q** generic rings (FJ).
– Added **gr_gens_recursive** (FJ).
– Allow overriding **flint_aligned_alloc** (AA).
– Added **_fmpz_vec_dot_general** (FJ).

• **Bugs**
– Fixed threading problem in **gr_method_tab_init**: FLINT would occasionally crash when calling generics-based internal code when using a large number of threads (FJ, after debugging by Alexander Smirnov).
– Fixed comparison of **gr** vectors with **fmpq** elements (FJ).
– Fixed allocation bug in **gr_mpoly_mul_monomial** (FJ).
– Fixed aliasing in **fmpzdivrem_approx** (FJ).
– Avoid division by zero in **acb_poly_refine_roots_durand_kerner**: in rare instances, computing roots of an integer polynomial could hang (FJ).
– Allow large arguments in **arb_atan_frac_bsplit** (FJ).
– Fixed printing large coefficients in **nmod_mpoly** (Alexander Smirnov, AA).
– Fixed **fq_*_poly_powmod** (AA).
– Fixed initialization of **fq_default_ctx** (Claus Fieker, Tommy Hofmann).
– Fixed memory leak in **gr_poly_write** (FJ).
– Fixed printing **nmod32** elements on 32-bit systems (FJ).
– Fixed **ldconfig** for BSD systems (AA).
– Fixed **FLINT_WANT_ASSERT** for CMake (AA).

• **Performance**
– FLINT is now built with `-O3 -march=native` by default (AA).
– FLINT is no longer built with `-funroll-loops` by default except for select modules. This reduces the library size by more than 20%. (AA).
– Assembly routines are now used as intended on ARM64 when compiling with GCC (AA).
– New basecase code for **flint_mpn_mul**, **flint_mpn_mul_n** and **flint_mpn_sqr** (generic C versions, assembly for x86-64). This can yield up to a 2x speedup over GMP for short integer multiplications when calling **mpn** functions directly, though few applications currently benefit significantly due to wrapper overheads (some Arb benchmarks run ~5% faster with this change). (AA, FJ).
– Faster **_fmpz_vec_dot** (FJ).
– Faster basecase **fmpz_poly** and **fmpz_mat** basecase algorithms based on dot products.
– Optimized **fmpz_mat_mul_classical** (FJ).
– Added **fmpz_mat_mul_waksman**, speeding up **fmpz_mat** multiplication for balanced matrices with huge entries (Éric Schost, Vincent Neiger, FJ).
- Strip trailing zeros in `fmpz_poly_gcd`: this gives a 50x speedup computing \( \text{gcd}(x^{1000}, x^{1001}) \).
- Faster CLD bound computation, speeding up `fmpz_poly_factor` for some polynomials.
- Use new formulas from Jorge Zuniga to compute Catalan’s constant and \( \zeta(3) \) faster (FJ).
- Compressed the database of Conway polynomials to 10% of the original size (AA).
- Optimized context initializers for `fq`, `fq_zech`, `fq_nmod` and `qadic` (AA).
- Improved `fmpz_is_probabprime` for word-size input (FJ).
- Optimized header files (AA).
- Changed several internal helper functions to forced inlines (AA).
- Faster computation of Swinnerton-Dyer polynomials (FJ).
- Added benchmark script (`dev/bench.py`) (FJ).

**Test code**
- Unified test programs per module: compiling FLINT’s test suite is now an order of magnitude faster (AA).
- Added pretty-printing and timing output for unit tests (AA).
- Improve use of test multiplier in some long-running unit tests (AA, FJ).
- Improved test coverage (AA, FJ).
- Allow `gr_ctx_init_random` to generate composite rings (FJ).

**Maintenance**
- Require GMP >= 6.2.1 and MPFR >= 4.1.0 (AA).
- Drop support for Itanium (AA).
- Drop support for MPIR (AA).
- Cleaned up `longlong.h` (AA).
- Removed `ARB_VERSION`, `ANTIC_VERSION`, `CALCIUM_VERSION` and associated constants (AA).
- Removed `_long_vec_print` and `_perm_print` (use `flint_printf` instead) (AA).
- Removed unused functions in the `d_mat` and `aprcl` modules (AA).
- Removed `fmpq_get_mpz_frac` (AA).
- Removed `arb_fmpz_poly_cos_minpoly` (AA).
- Removed `fmpz_mat_mul_classical_inline` (FJ).
- Introduce `FLINT_SWAP` macro to replace several older macros (FJ).
- Replaced `invert_limb` by `n_preinvert_limb_prenorm` (AA).
- Renamed `_perm_set_one` to `_perm_one` (AA).
- Only define some multithreaded “divides” function when the CPU is strongly ordered (AA).
- Enable `fft_small` for MSVC builds (AA).
- Use binary files in `qsieve` (AA).
- Use C11 atomics in the `fmpz` memory manager (AA).
- Merged some repeated code in the `mpoly` modules (FJ).
- Replaced more functions by generics-based versions (FJ).
- Do not include `pthread.h` when opted out (AA).
– Test ARM NEON in CI (AA).
– Test examples in CI (AA).
– Detect MPFR and GMP internals in configure (AA).
– Add `-lflint` to PKG-CONFIG (Josh Rickmar).
– Silenced some GCC compiler warnings (AA).
– Unified exception handling (AA).
– Corrected some function signatures in the documentation (Vincent Delecroix, Joel-Dahne, Edgar Costa).
– Rename the default Git-branch from `trunk` to `main` (FJ).
– Document some macros defined in `flint.h` (AA).
– Other code cleanup and modernisation (AA).

2023-11-10 – FLINT 3.0.1

• Build issues
  – Fix LIBS2 order for static linking (Tomás Oliveira e Silva).
  – Fix substitution of version number for older autotools (Albin Ahlbäck).
  – Fix use of AC_SEARCH_LIBS to find cblas_dgemm (Gonzalo Tornaría).
  – Add FlexiBLAS as a blas option (Mahrud Sayrafi).
  – Don’t use deprecated PythonInterp in CMake build (Mahrud Sayrafi).
  – Fix setting version numbers and strings in CMake build (Mahrud Sayrafi).
  – Only link with NTL for the tests on CMake (Mahrud Sayrafi).

• Bugs
  – Fix bug in nmod32 on 32-bit systems.
  – Fix missing modulus assignment in nmod_poly_mat_window_init (Vincent Neiger).
  – Fix tmp allocation size in _fmpz_set_str_basecase.
  – Fix rare arithmetic bug and memory leak in n_factor_ecm_select_curve.

• Other
  – Some corrections to the documentation.

2023-10-20 – FLINT 3.0.0

Merged libraries and reorganisation

• The following libraries have been merged into FLINT:
  – Arb 2.23 (arbitrary-precision ball arithmetic)
  – Calcium 0.4 (exact real and complex arithmetic)
  – Antic 0.2.5 (number fields, binary quadratic forms)

• Arb, Calcium and Antic will no longer be maintained as separate libraries. Users upgrading to FLINT 3.0 should ensure that they no longer link to the old Arb, Calcium or Antic library files or include any header files from those libraries which may be incompatible.
• The FLINT 3.0 API is largely backwards-compatible with FLINT 2.9, Arb 2.23, Calcium 0.4 and Antic 0.2.5, except for changes to rarely-used and internal functions documented below. However, the following changes to the handling of header files are likely to require (trivial) patches in many downstream codebases:
  – Header files belonging to Arb, Calcium and Antic now appear in the flint/ subdirectory. For example, instead of `#include "arb.h"`, it is necessary to `#include "flint/arb.h"` unless `<INCLUDE_DIR>/flint` has been added to the include path.
  – Most header files no longer include their implicit dependencies. For example, `fmpz_poly.h` no longer includes `fmpz.h`. Code that used functions from the `fmpz` module but only included `fmpz_poly.h` may thus now need to include `fmpz.h` explicitly. Likewise, many inclusions of system libraries like `stdlib.h` have been removed.
• The following people helped with the merge: Fredrik Johansson, Isuru Fernando, Albin Ahlbäck.
• FLINT 3.0 has a new build system based on Autotools, contributed by Albin Ahlbäck. Among other improvements, parallel builds are much faster and it is possible to build individual targets. Additional build system and CI improvements have been made by Marc Mezzarobba, Max Horn, Edgar Costa, Alex Best, Andreas Enge, and others.
• It is now necessary to run `bootstrap.sh` to generate the `configure` script in order to build FLINT from the git repository.
• Some `configure` options have changed: for example, `--reentrant` is now `--enable-reentrant`.
• The root directory has been cleaned up by moving all source code into the `src` directory. This should not affect any users.
• The NTL interface has been moved to a single header file. The `--with-ntl` build flag is now only needed to build the test code for this interface.
• The C++ interface (flintxx) has been removed. This interface is now maintained in the separate repository `https://github.com/flintlib/flintxx` (Edgar Costa).

Generic rings

• The new `gr` module supports generic programming. It provides wrappers for most built-in FLINT types and allows constructing generic structures (polynomials, matrices, etc.) over arbitrary base rings. The following modules are available:
  – `gr_generic` (various generic algorithms)
  – `gr_mat` (matrices with generic elements)
  – `gr_mpoly` (multivariate polynomials with generic elements)
  – `gr_poly` (univariate polynomials with generic elements)
  – `gr_special` (special functions for generic elements)
  – `gr_vec` (vectors with generic elements)
This feature is experimental: it is highly likely that some interfaces will change in a future FLINT release.
• There is also a Python wrapper (flint_ctypes) included with FLINT available in the `src/python` directory. Unlike other third-party FLINT wrappers available currently, this wrapper uses the `gr` interface to wrap (nearly) all FLINT types at once. This wrapper is not officially supported and will likely be deprecated in the future, but it can be useful for experimenting with FLINT.
• The generics system supports certain representations that do not have dedicated FLINT modules, for example 8-bit and 32-bit nmods.
Small-prime FFT

- The new `fft_small` module implements FFTs modulo word-size primes and multiplication based on such FFTs. This module requires AVX2 or NEON vector instructions and will not be built on targets that do not support them. The small-prime FFT speeds up the following functions for huge input, sometimes by a factor 2x to 10x:
  - `flint_mpm_mul` and variants, and indirectly any function based on FLINT’s integer multiplication for large inputs. For example, `fmpz_mul` and `arb_mul` are faster, but `fmpz_gcd` is currently unaffected since it calls GMP.
  - `nmod_poly_mul` and variants, and indirectly any function based on `nmod_poly` multiplication.
  - `fmpz_poly_mul` and variants, and indirectly any function based on `fmpz_poly` multiplication.
  - Division functions for `fmpz` and `arb`, which now use Newton iteration instead of calling GMP for huge input.
  - `fmpz_mod` arithmetic.
  - Radix conversion functions like `fmpz_get_str`, `fmpz_set_str` and `arb_get_str`.

- The FFT was contributed by Daniel Schultz, with final integration work and adaptations for other FLINT functions (Newton iteration implementations, etc.) done by Fredrik Johansson.

Other changes

- Changed the order of the `alloc` and `length` fields in `arb_poly_t`, `acb_poly_t` and `ca_poly_t` to match the FLINT types.
- Added `fmpzi` division, norm and GCD functions (`gcd_shortest` by Daniel Schultz).
- Added an `acf` type for complex floating-point numbers.
- Added error handling to `dirichlet_group_init`.
- Increased the prime factor limit in `dirichlet_group_init` from 1e12 to 1e16.
- Added `arb_nonnegative_abs` (Erik Postma).
- Fixed `arb_pow` for x just barely containing 0, y > 0 (Erik Postma).
- Improved precision handling in `arb_gamma` for huge input.
- Faster `arb_contains_arf`, `arb_overlaps`, `arb_gt`, `arb_lt`.
- Changed the argument order of `_fmpz_mod_poly_mullow` and `_fmpz_mod_poly_div_series`.
- Changed the call signature of many `_fmpz_mod_poly` methods to take a context object as input instead of the raw modulus.
- Support test coverage reports (`--enable-coverage`).
- Added `fmpz_poly_randtest_irreducible`.
- Improved tuning for various `nmod_poly` functions.
- Most Newton polynomial division and square root functions now use the Karp-Markstein algorithm.
- Replaced `count_leading_zeros` and `count_trailing_zeros` macros with `flint_clz` and `flint_ctz`.
- Fixed `nmod_polyCompose` which was not using an asymptotically fast algorithm.
- Various functions in the `nmod`, `fmpz_mod`, `fq` modules and elsewhere have been rewritten to use algorithms in the `generics` module. In many cases the corresponding type-specific algorithm implementation has been removed entirely (for example, `nmod_poly_divrem_newton` no longer exists).
- Fixed `fmpz_mod_poly_factor_squarefree`, `nmod_poly_factor_squarefree` and `fq_*_poly_factor_squarefree` sometimes returning non-monic factors. Among other consequences, this could lead to functions like `fq_poly_roots` returning incorrect roots.
- Fixed several bugs in the `fq_default` modules (Tommy Hofmann).
- Fixed stack overflow in `mpoly_divrem_ideal` functions.
- Handle the zero polynomial correctly in `nmod_poly_shift_left` (Vincent Neiger).
- Fixed handling of permutations in `invert_cols` matrix methods (Vincent Neiger).
- Added `nmod_mat_permute_rows` (Vincent Neiger).
- Fixed bug in `mpoly_monomial_halves` (Daniel Schultz).
- Fixed overflow bug in `fmpz_mod_mpoly_divrem_ideal` (Daniel Schultz).
- Optimized `fmpz_addmul`, `fmpz_addmul_ui`, `fmpz_submul`, `fmpz_submul_ui` for small arguments.
- Fixed demotion bug in `fmpz_addmul_si` and `fmpz_submul_si`.
- Optimized `fmpq_cmp`, `fmpq_cmp_ui`, `fmpq_cmp_si`, `fmpq_cmp_fmpz` for small arguments.
- Optimized `fmpz_poly_resultant_modular` by using a tighter bound.
- Allow `lll` to work with rank deficient Z basis (Daniel Schultz).
- Added `fmpq_mat_can_solve_dixon` (William Hart).
- Inlined `n_gcd` (Albin Ahlbäck).
- Fixed fallback code for `sub_ddmmss` when given signed arguments.
- Many documentation fixes (Håvard Damm-Johnsen, Joel Dahne, Albin Ahlbäck, David Einstein, Alex Best, and others).
- Code simplifications (Vincent Neiger).
- Fixed several type signatures (Ricardo Buring).
- Fixed several memory leaks (Ricardo Buring).
- Fixed `fmpz_poly_factor_squarefree` crashing when given the zero polynomial.
- Added `arb_minmax` (Joel Dahne).
- Added `_push_term_ffmpz` functions to `mpoly` types (David Einstein).
- Added functions for printing `nmod` vectors (Vincent Neiger).
- Added `nmod_poly_is_monic` (Vincent Neiger).
- Fixed threaded Arb functions to use the thread pool (Albin Ahlbäck).
- Removed `nmod_poly_mpz` functions (Ricardo Buring).
- Fixed file handling in `qsieve` (Michiel de Wilde, Oscar Benjamin).
- Free memory in case of failure in `fq_zech_ctx_init` (Claus Fieker).
- Fixed corrupted output in `fmpz_or`.
- Added several `nmod_poly_mat` utility functions (Vincent Neiger).
List of additions

- FLINT 3.0 includes all functions in FLINT 2.9, Arb 2.23, Calcium 0.4 and Antic 0.2.5 except those listed under “list of removals”. On top of this, the following functions have been added. This list is incomplete; many internal functions and functions starting with an underscore have been omitted.

- mpn_mul_default_mpn_ctx,
- _nmod_poly_mul_mid_default_mpn_ctx,
- _fmpz_poly_mul_mid_default_mpn_ctx and many internal functions in the new fft_small module

- acb_poly_nth_derivative, arb_div_arf_newton, arb_div_newton, arb_fmpz_divapprox,
- arb_nint, arb_poly_nth_derivative, arb_sqrt_arf, arb_sqrt_arf_newton,
- arb_sqrt_newton, arb_trunc, arb_minmax

- ca_set_fmpzi
- flint_aligned_alloc, flint_aligned_free
- flint_get_num_available_threads
- flint_mpn_add_inplace_c, flint_mpn_cmp_ui_2exp, flint_mpn_mul_large,
- flint_mpn_nbits
- fmpz_get_str_bsplit_threaded
- fmpz_mat_equal_col, fmpz_mat_equal_row, fmpz_neg_ui_array
- fmpz_poly_randtest_irreducible
- fmpz_poly_q_evaluate_fmpq, fmpz_poly_q_scalar_div_fmpq, fmpz_poly_q_scalar_div_fmpz,
- fmpz_poly_q_scalar_mul_fmpq, fmpz_poly_q_scalar_mul_fmpz
- fmpz_ui_pow_ui
- fmpzi_set_qqbar
- get_default_mpn_ctx

- gr_abs, gr_acos, gr_acos_pi, gr_acosh, gr_acot, gr_acot_pi, gr_acoth, gr_acsc,
- gr_acsc_pi, gr_acsch, gr_add, gr_add_fmpq, gr_add_fmpz, gr_add_other,
- gr_add_si, gr_add_ui, gr_addmul, gr_addmul_fmpq, gr_addmul_fmpz,
- gr_addmul_other, gr_addmul_ui, gr_agm, gr_agm1, gr_airy,
- gr_airy_ai, gr_airy_ai_prime, gr_airy_ai_prime_zero, gr_airy_ai_zero,
- gr_airy_bi, gr_airy_bi_prime, gr_airy_bi_prime_zero, gr_airy_bi_zero,
- gr_asem, gr_asem_pi, gr_asemch, gr_asin, gr_asin_pi, gr_asinh, gr_atan,
- gr_atan2, gr_atan_pi, gr_atanh, gr_barnes_g, gr_bellnum_fmpq, gr_bellnum_ui,
- gr_bellnum_vec, gr_bernoulli_fmpq, gr_bernoulli_ui, gr_bernoulli_vec,
- gr_bernopoly_ui, gr_bessel_i, gr_bessel_i_scaled, gr_bessel_j, gr_bessel_j_y,
- gr_bessel_k, gr_bessel_k_scaled, gr_bessel_y, gr_betal, gr_beta_lower,
- gr_bin, gr_bin_ui, gr_bin_ui_vec, gr_binui, gr_binvec, gr_carlson_rc,
- gr_carlson_rd, gr_carlson_rf, gr_carlson_rg, gr_carlson_rj, gr_catalan, gr_ceil,
- gr_chebyshev_t, gr_chebyshev_t_fmpq, gr_chebyshevs_fmpq, gr_chebyshev_fmpq,
- gr_clear, gr_cmp, gr_cmp_other, gr_cmpabs, gr_cmpabs_other, gr_conj,
- gr_cos, gr_cos_integral, gr_cos_pi, gr_cosh, gr_cosh_integral, gr_cot,
- gr_cot_pi, gr_coth, gr_coulomb, gr_coulomb_f, gr_coulomb_g, gr_coulomb_hneg,
- gr_coulomb_hpos, gr_csc, gr_csc_pi, gr_csch, gr_csgn, gr_ctx_ca_get_option,
- gr_ctx_ca_set_option, gr_ctx_clear, gr_ctx_cmp_coercion, gr_ctx_data_as_ptr,
- gr_ctx_data_ptr, gr_ctx_fmpz_mod_set_primality, gr_ctx_fq_degree,
- gr_ctx_fq_order, gr_ctx_fq_prime, gr_ctx_get_real_prec, gr_ctx_get_str,
- gr_ctx_has_real_prec, gr_ctx_init_complex_acb, gr_ctx_init_complex_algebraic_ca,
gr_generic_log, gr_generic_log10, gr_generic_loglp, gr_generic_log2,
gr_generic_mul_2exp_fmpz, gr_generic_mul_2exp_si, gr_generic_mul_fmpq,
gr_generic_mul_fmpz, gr_generic_mul_other, gr_generic_mul_si, gr_generic_mul_two,
gr_generic_mul_ui, gr_generic_mul_ui_via_ZZ, gr_generic_neg_one,
gr_generic_other_add, gr_generic_other_add_vec, gr_generic_other_div,
gr_generic_other_div_vec, gr_generic_other_divexact_vec, gr_generic_other_mul,
gr_generic_other_mul_vec, gr_generic_other_pow, gr_generic_other_pow_vec,
gr_generic_other_sub, gr_generic_other_sub_vec, gr_generic_partitions_fmpz,
gr_generic_partitions_ui, gr_generic_partitions_vec, gr_generic_pow_fmpq,
gr_generic_pow_fmpz, gr_generic_pow_fmpz_binexp, gr_generic_pow_other,
gr_generic_pow_si, gr_generic_pow_ui, gr_generic_pow_ui_binexp,
gr_generic_randtest_not_zero, gr_generic_rfac, gr_generic_rfac_fmpz,
gr_generic_rfac_ui, gr_generic_rfac_vec, gr_generic_rising, gr_generic_rising_ui,
gr_generic_rqqrt, gr_generic_scalar_add_vec, gr_generic_scalar_div_vec,
gr_generic_scalar_divexact_vec, gr_generic_scalar_mul_vec,
gr_generic_scalar_other_add_vec, gr_generic_scalar_other_div_vec,
gr_generic_scalar_other_divexact_vec, gr_generic_scalar_other_mul_vec,
gr_generic_scalar_other_mul_vec, gr_generic_scalar_other_sub_vec,
gr_generic_set_fmpq, gr_generic_set_fmpz_2exp_fmpz, gr_generic_set_other,
gr_generic_set_vec, gr_generic_set_shallow,
gr_generic_sin, gr_generic_sin_cos, gr_generic_sqr, gr_generic_sqrt,
gr_generic_stirling_s1_ui_vec, gr_generic_stirling_s1_uiui,
gr_generic_stirling_s1u_ui_vec, gr_generic_stirling_s1u_uiui,
gr_generic_stirling_s2_ui_vec, gr_generic_stirling_s2_uiui, gr_generic_sub_fmpq,
gr_generic_sub_fmpz, gr_generic_sub_other, gr_generic_sub_si, gr_generic_sub_ui,
gr_generic_submul, gr_generic_submul_fmpq, gr_generic_submul_fmpz,
gr_generic_submul_other, gr_generic_submul_si, gr_generic_submul_ui,
gr_generic_tan, gr_generic_vec_add, gr_generic_vec_add_other,
gr_generic_vec_add_scalar, gr_generic_vec_add_scalar_fmpq,
gr_generic_vec_add_scalar_fmpz, gr_generic_vec_add_scalar_other,
gr_generic_vec_add_scalar_si, gr_generic_vec_add_scalar_ui,
gr_generic_vec_clear, gr_generic_vec_div, gr_generic_vec_div_other,
gr_generic_vec_div_scalar, gr_generic_vec_div_scalar_fmpq,
gr_generic_vec_div_scalar_fmpz, gr_generic_vec_div_scalar_other,
gr_generic_vec_div_scalar_si, gr_generic_vec_div_scalar_ui,
gr_generic_vec_divexact, gr_generic_vec_divexact_other, gr_generic_vec_divexact_scalar,
gr_generic_vec_divexact_scalar_fmpq, gr_generic_vec_divexact_scalar_fmpz,
gr_generic_vec_divexact_scalar_other, gr_generic_vec_divexact_scalar_si,
gr_generic_vec_dot, gr_generic_vec_dot_fmpz, gr_generic_vec_dot_fmpq,
gr_generic_vec_dot_vec, gr_generic_vec_dot_vec_fmpz, gr_generic_vec_dot_vec_fmpq,
gr_generic_vec_equal, gr_generic_vec_init, gr_generic_vec_is_zero,
gr_generic_vec_mul, gr_generic_vec_mul_other, gr_generic_vec_mul_scalar,
gr_generic_vec_mul_scalar_2exp_si, gr_generic_vec_mul_scalar_fmpq,
gr_generic_vec_mul_scalar_fmpz, gr_generic_vec_mul_scalar_other,
gr_generic_vec_mul_scalar_si, gr_generic_vec_mul_scalar_ui,
gr_generic_vec_neg, gr_generic_vec_normalise, gr_generic_vec_normalise_weak,
gr_generic_vec_pow, gr_generic_vec_pow_other, gr_generic_vec_pow_scalar,
gr_generic_vec_pow_scalar_fmpq, gr_generic_vec_pow_scalar_fmpz,
gr_generic_vec_pow_scalar_other, gr_generic_vec_pow_scalar_si,
gr_generic_vec_pow_scalar ui, gr_generic_vec_reciprocals,
gr_generic_vec_pow_scalar_addmul, gr_generic_vec_pow_scalar_addmul si,
gr_generic_vec_pow_scalar_submul, gr_generic_vec_pow_scalar_submul si,
gr_generic_vec_set, gr_generic_vec_set_powers, gr_generic_vec_sub,
gr_generic_vec_sub_other, gr_generic_vec_sub_scalar, gr_generic_vec_sub_scalar_fmpq,
gr_generic_vec_sub_scalar_fmpz, gr_generic_vec_sub_scalar_other,
gr_generic_vec_sub_scalar_si, gr_generic_vec_sub_scalar_ui, gr_generic_vec_swap,
gr_generic_vec_zero, gr_generic_write n, gr_get_d, gr_get_fmpq,
gr_get_fmpz, gr_get_fmpq_2exp_fmpz, gr_get_si, gr_get_str, gr_get_str n,
gr_get_ui, gr_glaisher, gr_harmonic, gr_harmonic_ui, gr_heap_clear,
gr_heap_clear_vec, gr_heap_init, gr_heap_init_vec, gr_hermite_h,
gr_hilbert_class_poly, gr_hurwitz_zeta, gr_hypgeom_0f1, gr_hypgeom_1f1,
gr_hypgeom_2f1, gr_hypgeom_pfq, gr_hypgeom_u, gr_i, gr_im, gr_init, gr_inv,
gr_is_invertible, gr_is_neg_one, gr_is_one, gr_is_square, gr_is_zero,
gr_jacobi_p, gr_jacobi_theta, gr_jacobi_theta_1, gr_jacobi_theta_2,
gr_jacobi_theta_3, gr_jacobi_theta_4, gr_khinchin, gr_laguerre_1,
gr_lambertw, gr_lambertw_fmpz, gr_lcm, gr_legendre_p_root_ui,
gr_legendre_q, gr_lerch_phi, gr_log, gr_log10, gr_log1p, gr_log2,
gr_log_barnes_g, gr_log_integral, gr_log_pi_i, gr_mat_add, gr_mat_add_scalar,
gr_mat_addmul_scalar, gr_mat_adjugate, gr_mat_adjugate_cofactor,
gr_mat_adjugate_cofactor, gr_mat_apply_row_similarity, gr_mat_charpoly,
gr_mat_charpoly_berkowitz, gr_mat_charpoly_danielevsky, gr_mat_charpoly_faddeev,
gr_mat_charpoly_faddeev_bags, gr_mat_charpoly_from_hessenberg,
gr_mat_charpoly_gauss, gr_mat_charpoly_householder, gr_mat_clear,
gr_mat_concat_horizontal, gr_mat_concat_vertical, gr_mat_det,
gr_mat_det_berkowitz, gr_mat_det_cofactor, gr_mat_det_fflu, gr_mat_det_generic,
gr_mat_det_generic_field, gr_mat_det_generic_integral_domain, gr_mat_det_lu,
gr_mat_diag_mul, gr_mat_diagonalization, gr_mat_diagonalization_generic,
gr_mat_diagonalization_pcomp, gr_mat_div_scalar, gr_mat_eigenvalues,
gr_mat_eigenvalues_other, gr_mat_entry_ptr, gr_mat_entry_srcptr, gr_mat_equal,
gr_mat_exp, gr_mat_exp_jordan, gr_mat_fflu, gr_mat_find_nonzero_pivot,
gr_mat_find_nonzero_pivot_generic, gr_mat_find_nonzero_pivot_large_abs,
gr_mat_gr_poly_evaluate, gr_mat_hadamard, gr_mat_hessenberg,
gr_mat_hessenberg_gauss, gr_mat_hessenberg_householder, gr_mat_hilbert,
gr_mat_init, gr_mat_init_set, gr_mat_inv, gr_mat_invert_cols,
gr_mat_invert_rows, gr_mat_is_diagonal, gr_mat_is_empty, gr_mat_is_hessenberg,
gr_mat_is_lower_triangular, gr_mat_is_neg_one, gr_mat_is_one, gr_mat_is_scalar,
gr_mat_is_square, gr_mat_is_upper_triangular, gr_mat_is_zero,
gr_mat_jordan_blocks, gr_mat_jordan_form, gr_mat_jordan_transformation,
gr_mat_log, gr_mat_log_jordan, gr_mat_lu, gr_mat_lu_classical,
gr_mat_lu_recursive, gr_mat_mnumpy_field, gr_mat_mul, gr_mat_mul_classical,
gr_mat_mul_diag, gr_mat_mul_generic, gr_mat_mul_scalar, gr_mat_mul_strassen,
gr_mat_neg, gr_mat_nonsingular_solve, gr_mat_nonsingular_solve_den,
gr_mat_nonsingular_solve_den_fflu, gr_mat_nonsingular_solve_fflu,
gr_mat_nonsingular_solve_fflu_pcomp, gr_mat_nonsingular_solve_fflu_pcomp, gr_mat_nonsingular_solve_tril,
gr_mat_nonsingular_solve_tril_pcomp, gr_mat_nonsingular_solve_tril_recursive,
gr_mat_nonsingular_solve_triu, gr_mat_nonsingular_solve_triu_classical,
gr_mat_nonsingular_solve_triu_recursive, gr_mat_nullspace, gr_mat_one,
gr_mat_ones, gr_mat_pascal, gr_mat_print, gr_mat_randops, gr_mat_randpermdiag,
gr_mat_randrank, gr_mat_randtest, gr_mat_rank, gr_mat_rank_fflu, gr_mat_rank_lu,
gr_mat_reduce_row, gr_mat_ref, gr_mat_ref_den, gr_mat_ref_den_fflu,
gr_mat_ref_fflu, gr_mat_ref_lu, gr_mat_set, gr_mat_set_ffmpz, gr_mat_set_ffmpq, gr_mat_set_ffmpz_mat,
gr_mat_set_ffmpq_mat, gr_mat_set_jordan_blocks, gr_mat_set_scalar, gr_mat_set_si,
gr_mat_set_ui, gr_mat_solve_field, gr_mat_sqrt, gr_mat_stirling,
gr_mat_sub, gr_mat_sub_scalar, gr_mat_submul_scalar, gr_mat_swap, gr_mat_swap_cols,
gr_mat_swap_entrywise, gr_mat_swap_rows, gr_mat_trace, gr_mat_trace_prod2,
gr_mat_transpose, gr_mat_transpose_resize, gr_mat_window_clear, gr_mat_window_init,
gr_mat_write, gr_mat_zero, gr_method_tab_init, gr_modular_delta, gr_modular_j,
gr_modular_lambda, gr_mpolo_add, gr_mpolo_assert_canonical, gr_mpolo_clear,
gr_mpolo_combine_like_terms, gr_mpolo_equal, gr_mpolo_fit_bits, gr_mpolo_fit_length,
gr_mpolo_fit_length_fit_bits, gr_mpolo_fit_length_reset_bits, gr_mpolo_gen,
gr_mpolo_get_coeff_ffmpz, gr_mpolo_get_coeff_ffmpq, gr_mpolo_init2, gr_mpolo_init3, gr_mpolo_is_canonical, gr_mpolo_is_gen,
gr_mpolo_is_one, gr_mpolo_is_zero, gr_mpolo_mul, gr_mpolo_mpolo_ffmpz,
gr_mpolo_mpolo_ffmpq, gr_mpolo_mpolo_johnson, gr_mpolo_mpolo_monomial, gr_mpolo_mpolo_scalar,
gr_mpolo_mpolo_scalar, gr_mpolo_mpolo_ui, gr_mpolo_neg, gr_mpolo_one, gr_mpolo_print_pretty,
The following functions that were present in FLINT 2.9, Arb 2.23 or Calcium 0.4 have been removed, deprecated, or replaced. Most are algorithms obsoleted by new gr implementations, functions dealing with removed types (fmpr) or GMP types (mpz, etc.), and internal functions that are no longer needed.

- `__fmpz_clear`, `__fmpz_eq`, `__fmpz_gt`, `__fmpz_gte`, `__fmpz_init`, `__fmpz_init_set`, `__fmpz_init_set_ui`, `__fmpz_lt`, `__fmpz_lte`, `__fmpz_neg`, `__fmpz_neq`, `__fmpz_set_si`, `__fmpz_set_ui`
- `__fmpz_mod_poly_div_divconquer`, `__fmpz_mod_poly_divrem_divconquer`, `__fq_nmod_poly_divrem_divconquer`, `__fq_poly_divrem_divconquer`, `__fq_zech_poly_divrem_divconquer`
- `__nmmod_poly_div_divconquer`, `__nmmod_poly_divrem_divconquer`, `__nmmod_poly_invsqrt_series_prealloc`
- `acb_rising_ui_bs`, `acb_rising_ui_rs`, `acb_rising_ui_rec`
- `__arb_poly_compose_axnc`, `__arb_poly_compose_divconquer`, `__arb_poly_compose_horner`, `__arb_poly_compose_series_brent_kung`, `__arb_poly_compose_series_horner`, `__arb_poly_sin_cos_series_basecase`, `__arb_poly_sin_cos_series_tangent`, `__arb_poly_taylor_shift_convol`, `__arb_poly_taylor_shift_divconquer`, `__arb_poly_taylor_shift_horner`
- `arb_rising_ui_bs`, `arb_rising_ui_rs`, `arb_rising_ui_rec`, `arb_rising2_ui_bs`, `arb_rising2_ui_rs`, `arb_rising2_ui`
- `arith_bernoulli_number_vec_zeta`, `arith_bernoulli_number_zeta`, `arith_cos_minpoly`, `arith_euler_number_zeta`, `arith_number_of_partitions_mpfr`
- `ca_poly_atan_series`, `ca_poly_compose_divconquer`, `ca_poly_compose_horner`
- `fmpq_poly_set_array_mpq`
- `fmpq_add_1x1`, `fmpq_add_eps`, `fmpq_add_mpn`, `fmpq_mul_1x1`, `fmpq_mul_mpn`, `fmpq_normalise_naive`, `fmpq_set_round`, `fmpq_set_round_mpn`
- `fmpz_ui_pow_ui`, `fmpz_vec_get_mpf_vec`
- `fq_nmod_poly_compose_divconquer`, `fq_nmod_poly_compose_horner`, `fq_nmod_poly_div_basecase`, `fq_nmod_poly_divrem_basecase`, `fq_nmod_poly_divrem_divconquer`, `fq_nmod_poly_divrem_divconquer_recursive`
_fq_nmod_poly_gcd_euclidean, _fq_nmod_poly_gcd_hgcd, _fq_nmod_poly_hgcd,
_fq_nmod_poly_hgcd_recursive, _fq_nmod_poly_hgcd_recursive_iter,
_fq_nmod_poly_xgcd_euclidean

• _fq_poly_compose_divconquer, _fq_poly_compose_horner, _fq_poly_div_basecase,
_fq_poly_divrem_basecase, _fq_poly_divrem_divconquer, _fq_poly_divrem_divconquer_recursive,
_fq_poly_gcd_euclidean, _fq_poly_gcd_hgcd, _fq_poly_hgcd,
_fq_poly_hgcd_recursive, _fq_poly_hgcd_recursive_iter, _fq_poly_xgcd_euclidean

• _fq_zech_poly_compose_divconquer, _fq_zech_poly_compose_horner,
_fq_zech_poly_div_basecase, _fq_zech_poly_divrem_basecase,
_fq_zech_poly_divrem_divconquer, _fq_zech_poly_divrem_divconquer_recursive,
_fq_zech_poly_gcd_euclidean, _fq_zech_poly_gcd_hgcd, _fq_zech_poly_hgcd,
_fq_zech_poly_hgcd_recursive, _fq_zech_poly_hgcd_recursive_iter,
_fq_zech_poly_xgcd_euclidean

• _nmod_mat_set_mod

• _nmod_poly_compose_divconquer, _nmod_poly_compose_series_brent_kung,
_nmod_poly_compose_series_divconquer, _nmod_poly_compose_series_horner,
_nmod_poly_div_basecase, _nmod_poly_div_basecase_1, _nmod_poly_div_basecase_2,
_nmod_poly_div_basecase_3, _nmod_poly_div_divconquer, _nmod_poly_div_divconquer_recursive,
_nmod_poly_div_newton, _nmod_poly_divrem_basecase_1, _nmod_poly_divrem_basecase_2,
_nmod_poly_divrem_basecase_3, _nmod_poly_divrem_divconquer, _nmod_poly_divrem_divconquer_recursive,
_nmod_poly_divrem_q0, _nmod_poly_divrem_q1, _nmod_poly_exp_series_basecase,
_nmod_poly_exp_series_monomial_ui, _nmod_poly_exp_series_newton,
_nmod_poly_hgcd_recursive, _nmod_poly_hgcd_recursive_iter, _nmod_poly_hgcd_res,
_nmod_poly_integral_offset, _nmod_poly_log_series_monomial_ui,
_nmod_poly_rem_basecase, _nmod_poly_rem_basecase_1, _nmod_poly_rem_basecase_2,
_nmod_poly_rem_basecase_3

• acb_poly_compose_divconquer, acb_poly_compose_horner, acb_poly_compose_series_brent_kung,
acb_poly_compose_series_horner, acb_poly_sin_cos_series_basecase,
acb_poly_sin_cos_series_tangent, acb_poly_taylor_shift_convolution,
acb_poly_taylor_shift_divconquer, acb_poly_taylor_shift_horner

• arb_flint_get_num_available_threads

• arb_poly_compose_divconquer, arb_poly_compose_horner, arb_poly_compose_series_brent_kung,
arb_poly_compose_series_horner, arb_poly_sin_cos_series_basecase,
arb_poly_sin_cos_series_tangent, arb_poly_taylor_shift_convolution,
arb_poly_taylor_shift_divconquer, arb_poly_taylor_shift_horner

• arb_test_multiplier

• arb_thread_pool_num_available

• arf_get_fmpr, arf_set_fmpr

• arith_cos_minpoly, arith_number_of_partitions_mpfr

• ca_mat_transpose_resize, ca_poly_atan_series, ca_poly_compose_divconquer,
ca_poly_compose_horner, calcium_test_multiplier

• cos_minpoly, cos_pi_pq

• fmpq_poly_evaluate_mpq, fmpq_poly_evaluate_mpz, fmpq_poly_get_coeff_mpq,
fmpq_poly_scalar_div_mpq, fmpq_poly_scalar_div_mpz, fmpq_poly_scalar_mul_mpq,
fmpq_poly_scalar_mul_mpz, fmpq_poly_set_array_mpq, fmpq_poly_set_coeff_mpq,
fmpq_poly_set_coeff_mpz, fmpq_poly_set_mpq, fmpq_poly_set_mpz

• fmpq_add, fmpq_add_fmpz, fmpq_add_num, fmpq_add_si, fmpq_add_ui, fmpq_addmul,
fmpq_addmul_fmpq, fmpq_addmul_si, fmpq_addmul_ui, fmpq_check_ulp, fmpq_cmp,
fmpq_cmp_2exp_si, fmpq_cmpabs, fmpq_cmpabs_2exp_si, fmpq_cmpabs_ui,
fmpq_div, fmpq_div_fmpz, fmpq_div_mpq, fmpq_div_mpz, fmpq_exp, fmpq_exp1,
fmpr_fmpz_div, fmpr_fmpz_div_fmpz, fmpr_get_d, fmpr_get_fmpq, fmpr_get_fmpz,
fmpr_get_fmpz_2exp, fmpr_get_fmpz_fixed_fmpz, fmpr_get_fmpz_fixed_si,
fmpr_log, fmpr_log1p, fmpr_mul, fmpr_mul_fmpz,
fmpr_mul_naive, fmpr_mul_si, fmpr_mul_ui, fmpr_pow_sloppy_fmpz,
fmpr_pow_sloppy_si, fmpr_pow_sloppy_ui, fmpr_print, fmpr_print_d, fmpr_randtest,
fmpr_randtest_not_zero, fmpr_randtest_special, fmpr_root, fmpr_rsqrt, fmpr_set_d,
fmpr_set_fmpz, fmpr_set_fmpz_2exp, fmpr_set_fmpq, fmpr_set_round_ui_2exp_fmpz,
fmpr_set_round_uiui_2exp_fmpz, fmpr_sub, fmpr_submul, fmpr_submul_fmpz, fmpr_submul_si,
fmpr_submul_ui, fmpr_ui_div, fmpr_ulp

• fmpr_deprecated_multi_crt, fmpr_deprecated_multi_crt_clear,
fmpr_deprecated_multi_crt_init, fmpr_deprecated_multi_crt_precomp,
fmpr_deprecated_multi_crt_precomp_p, fmpr_deprecated_multi_crt_precompute,
fmpr_deprecated_multi_crt_precompute_p

• fmpr_mat_col_equal, fmpr_mat_get_mpf_mat, fmpr_mat_row_equal

• fmpr_mod_ctx_get_modulus_mpz_read_only

• fmpr_mod_poly_compose_divconquer, fmpr_mod_poly_compose_horner,
fmpr_mod_poly_div_basecase, fmpr_mod_poly_div_divconquer,
fmpr_mod_poly_div_newton, fmpr_mod_poly_divrem_divconquer,
fmpr_mod_poly_gcd_euclidean, fmpr_mod_poly_gcd_hgcd, fmpr_mod_poly_get_coeff_mpz,
fmpr_mod_poly_set_coeff_mpz, fmpr_mod_poly_xgcd_euclidean,
fmpr_mod_poly_xgcd_hgcd

• fmpr_poly_evaluate_mpf, fmpr_poly_evaluate_mpq, fmpr_poly_get_coeff_mpz,
mpr_poly_q_evaluate, fmpr_poly_q_scalar_div_mpq, fmpr_poly_q_scalar_div_mpz,
mpr_poly_q_scalar_mul_mpq, fmpr_poly_q_scalar_mul_mpz, fmpr_poly_scalar_divexact_mpz,
mpr_poly_scalar_div_mpz, fmpr_poly_set_coeff_mpz, fmpr_poly_set_mpz

• fq_mmod_poly_compose_divconquer, fq_mmod_poly_compose_horner,
fq_mmod_poly_divrem_basecase, fq_mmod_poly_divrem_divconquer,
fq_mmod_poly_gcd_euclidean, fq_mmod_poly_gcd_hgcd, fq_mmod_poly_xgcd_euclidean,
fq_poly_compose_divconquer, fq_poly_compose_horner, fq_poly_divrem_basecase,
fq_poly_divrem_divconquer, fq_poly_gcd_euclidean, fq_poly_gcd_hgcd,
fq_poly_xgcd_euclidean, fq_zech_poly_compose_divconquer, fq_zech_poly_compose_horner,
fq_zech_poly_divrem_basecase, fq_zech_poly_divrem_divconquer,
fq_zech_poly_gcd_euclidean, fq_zech_poly_gcd_hgcd, fq_zech_poly_xgcd_euclidean

• mag_get_fmpr, mag_set_fmpr

• mpfr_cos_pi_pq, mpfr_zeta_inv_euler_product

• nmod_poly_compose_divconquer, nmod_poly_compose_series_brent_kung,
nmod_poly_compose_series_divconquer, nmod_poly_compose_series_horner,
nmod_poly_div_basecase, nmod_poly_div_divconquer, nmod_poly_div_newton,
nmod_poly_divrem_divconquer, nmod_poly_divrem_newton, nmod_poly_exp_series_basecase,
nmod_poly_log_series_monomial_ui, nmod_poly_factor_get_nmod_poly,
nmod_poly_log_series_monomial_ui, nmod_poly_rem_basecase,
nmod_poly_set_fmpz_poly, sinh_cosh_divk_precomp

• _nmod_poly_powmod_mpz_binexp, nmod_poly_powmod_mpz_binexp,
_nmod_poly_powmod_mpz_binexp, nmod_poly_powmod_mpz_binexp_preinv,
_nmod_poly_powmod_mpz_binexp, nmod_poly_powmod_mpz_binexp_preinv,
_nmod_poly_powmod_mpz_binexp, nmod_poly_powmod_mpz_binexp_preinv
2022-06-24 – FLINT 2.9.0

- Add fmpz_mod_poly_div function
- Add _flint_get_memory function
- Add Eulerian polynomials
- Support “multivariate” polynomials with zero variables
- Improve Stirling numbers of both kinds
- Speed up numerous fmpz functions for small inputs
- Improve Bell numbers
- Speedups to nmod arithmetic
- Improve nmod_mat LU decomposition
- Fully separate nmod module from nmod_vec
- Speed up Hermite polynomials
- Add n-th derivative for Z[x] and Q[x]
- Improve fq_default module (nmod is now used where optimal)
- Add sqrt functions for numerous polynomial/series modules and finite fields
- Add FFT matrix multiplication
- Improve CI
- Improve LLL for general use
- Add matrix-vector products over Q
- Add can_solve function for fmpq_mat, handling non-square/singular matrices
- Document fmpz_mod_vec module
- Fix and document qadic_sqrt function
- Add parallel programming helpers

2022-04-25 – FLINT 2.8.5

- Fix a serious bug in LLL

2021-11-17 – FLINT 2.8.4

- Fix a serious bug in fmpz_mod_poly_xgcd for polynomials of large length
- Fix an assertion failure in fmpz_mat_solve_fllu (only relevant if asserts enabled)
- Fix some bugs on 32 bit machines
- Work around a compiler bug on Apple M1
- Fix bug in nmod_mpoly_factor (returned 0 for some factorisations)
- Fix some documentation build errors and some doc formatting issues
2021-11-03 – FLINT 2.8.3

- Fix a serious bug in nmod_poly_xgcd_hgcd, nmod_poly_xgcd, fmpz_poly_xgcd_modular, fmpz_poly_xgcd, fmpq_poly_xgcd for polynomials of length >= 340.
- Fix some copyright assignments
- Fix some documentation errors

2021-10-15 – FLINT 2.8.2

- Fix an issue with –disable-dependency-tracking on some distributions

2021-10-01 – FLINT 2.8.1

- Numerous bug fixes
- Adjust soname on android
- Allow disabling of dependency tracking

2021-07-23 – FLINT 2.8.0

- New fq_default module which combines existing finite fields
- Speedups for linear algebra when using BLAS and/or threading
- New series expansions with coefficients in QQ
- Faster CRT
- New fmpz_mod_mpoly module
- Polynomial factoring improvements over ZZ
- Fixed bugs in gmpcompat on Windows
- Add fmpz_mat_can_solve_fllu and fmpz_mat_can_solve
- Cleanup of the nmod_poly and nmod_poly_factor code
- Implement nmod_mat_det_howell
- Add fmpz_mod_poly_divides, fmpz_divides, n_divides, nmod_poly_divides
- Interface for multiplying matrices by vectors and arrays
- Nearest Euclidean division
- Subresultant GCD
- XGCD over ZZ with canonical Bezout coefficients
- Add fmpz_mpoly resultant and discriminant
- Add deprecations list
- Add FLINT_SGN macro
- Speedups for series computations
- Switch to GitHub Actions for CI
- Improve Taylor shift
- Numerous bug fixes and speedups
2021-01-18 – FLINT 2.7.1

- Fix build bug due to missing test files
- Fix bug in fmpzmod_poly_factor when there are more than five factors
- Fix issue when using MPIR 3.0.0 on Win64 with command line build
- Fix bug in fmpzmod_poly_div_series
- Fix some broken asserts
- Support standard GNU installation directories in CMake build
- Fix stack overflow with ICC

2020-12-18 – FLINT 2.7.0

- Multivariate factorisation
- Square root and square testing for finite fields
- Square root and square testing for multivariates
- Zassenhaus factoring speedups (incl. degree pruning)
- Fast factorisation of cubic univariate polynomials
- Add context objects to fmpzmod_poly functions
- Use BLAS for matrix multiplication over Z/nZ (small n)
- Linear solving for non-square/singular matrices (can_solve)
- Speed up factorisation over Z/nZ (for multiprecision n)

2020-08-12 – FLINT 2.6.3

- Fix a bug in generator of finite field in characteristic 2
- Allow Flint to work with GMP 6.1.2 and 6.2.0 interchangeably
- Fix some old license headers

2020-07-31 – FLINT 2.6.2

- Fix for choice of generator in an fq finite field of degree one
- Fix an incorrectly written test

2020-07-23 – FLINT 2.6.1

- Fix issues on Debian major architectures
- Fix numerous reported bugs (mpoly, fq_poly, mpn_mul_1, mod 2 code, etc.)
2020-06-05 – FLINT 2.6.0

- multivariate polynomials over most standard rings (sparse distributed)
- APR-CL primality proving
- elliptic curve integer factoring
- minpoly and charpoly
- improved quadratic sieve for integer factoring
- embeddings of finite fields
- pollard rho integer factoring
- p+1 integer factoring
- best of breed smooth integer factoring routine
- best of breed general integer factoring routine
- howell and strong echelon form
- large speedups for solve and hence inverse over Z and Q
- randprime and nextprime functions
- pernet-stein HNF improvements
- moller-granlund precomputed inverses
- resultant_modular_div
- fibonacci polynomials
- exception mechanism/flint_abort
- sqrt of series and polynomials
- division of series over Z
- power sums
- improved base cases of various power series functions
- ability to switch memory allocators
- fast recurrence for Hermite polys
- shifted Legendre polynomials
- Laguerre polynomials
- Gegenbauer polys
- sphinx documentation
- van houten with gradual feeding implementation of polynomial factoring over Z
- perfect power detection
- divisibility testing for polynomials
- fast block based memory manager for bundling fmpz allocations
- uniform random generation
- CMake build system
- linear algebra speedups when everything can be kept in longs
- nmod module for integers mod (small) n
- fmpz_mod_mat module for matrices over integers mod multiprecision n
- kronecker product (tensor product)
• random primitive polys (for finite fields)
• thread pool implementation
• threading of FFT for integer and polynomial multiplication over Z
• threading of quadratic sieve for integer factoring
• improved threading of factoring of polynomials mod p
• threading for multivariate polynomial multiplication, division and GCD
• threaded multiplication of matrices mod p
• Berlekamp-Massey (nmod)
• fmpz_mod module for integers mod multiprecision n
• Pohlig-Hellman (discrete log)
• farey_neighbours
• remove openMP option
• additional integer division variants
• speed up mpn_mulmod_preinv
• fft precaching
• cyclotomic polynomial detection
• polynomial root finding over finite fields
• GMP 6.2 support
• MPIR 3.0.0 support
• many small speedups and additional convenience functions added

2015-08-13 – FLINT 2.5.2

• Fix further issues with soname versioning and ldconfig
• Fix a bug when using GMP instead of MPIR.

2015-08-12 – FLINT 2.5.1

• Fix some build bugs related to soname versioning and ldconfig
• Fix issue with Windows MSVC build

2015-08-07 – FLINT 2.5.0

• LLL (rational, Nguyen-Stehle, from Gram matrix, with_removal, Storjohann/ULLL)
• Hermite normal form (naive, xgcd, Domich-Kannan-Trotter, Kannan-Bachem, Pernet-Stein)
• Smith normal form (diagonal, Kannan-Bachem, Iliopoulos)
• Paterson-Stockmeyer algorithm
• modular resultant
• hgcd resultant
• polynomial discriminant
• multithreaded multimodular Taylor shift
- multithreaded Brent-Kung composition
- multithreaded Kaltofen-Shoup distinct degree factorisation
- multiplication based reduced row echelon form
- place inline functions in library for foreign function interfaces
- Primality tests for large integers (Pocklington, Morrison)
- Probable prime tests for large integers (Lucas, Baillie-PSW, strong-psp, Brillhart-Lehmer-Selfridge)
- CRT for large integers
- Dixon algorithm for nullspace
- Brent-Kung composition in irreducibility and distinct degree factorisation
- floating point QR decomposition
- Schwarz-Rutishauser Gram-Schmidt algorithm
- Ogita-Rump-Oishi dot product
- matrix window functions
- MSVC support (Brian Gladman)
- fast cube/nth-root (Newton, Kahan, magic number, Chebyshev)
- Bodrato matrix squaring
- matrix concatenation functions
- matrix content
- faster n_gcd
- faster n_sqrtmod and fmpz_sqrtmod
- additional functions for returning factor of modulus in polys over \( \mathbb{Z}/n\mathbb{Z} \)
- Hadamard matrix construction
- series addition/subtraction
- faster prime_pi bounds
- speedup creation of sparse polynomials
- speedup n_isprime n_nextprime
- speedup n_isprime_pocklington
- speedups to fmpq_poly and fmpz_poly arithmetic
- speedup polynomial irreducibility testing over \( \mathbb{Z}/p\mathbb{Z} \)
- speedup of rank computation over \( \mathbb{Z}/\mathbb{Z} \)
- made CPimport compile time dependency only
- teach flint_printf/sprintf about explicit width format specifiers
- support relative paths in configure
- library soname versioning
- ARM64 patches
- Support MSYS2
- Progress towards supporting MIPS64
- Fix a serious bug in fmpz_poly_signature
• fixed a severe bug in flint’s fmpz_poly_gcd_heuristic, reported by Anton Mellit.

• fixed a severe bug in flint’s primality code (n_is_prime() affecting n_factor())

• Fix a linker issue on Mac OSX.

• Fix bug in ARM assembly

• C++ expressions template wrapper
• Fast factorisation of polynomials over $\mathbb{Z}/n\mathbb{Z}$
• improved p-adics
• polynomials/matrices over p-adics
• qadics
• Finite fields (small and large $F_q$), polynomials/matrices over $F_q$
• Finite fields with Zech logarithm representation
• Fast factorisation of polynomials over $F_q$
• Faster Brent-Kung modular composition
• New prime sieving code
• Lambert-W function
• Precomputed inverses for polynomials and large integers
• Williams’ P+1 integer factoring algorithm
• Harvey’s KS2/KS4 polynomial multiplication
• Faster primality testing up to 64 bits
• Support for Cygwin64 and MinGW64
• Support for Clang
• Support for GMP
• Support for Boehm-Demers-Weiser GC
• Support for flint extension modules
2012-07-01 – FLINT 2.3

- **general**
  - many changes to the build system
  - added NTL interface
  - switched to custom memory allocation functions flint_malloc etc
  - in addition to the entries below, fixed a large number of memory leaks, problems with the test code, and bugs in corner cases of various functions
  - added _fmpz_cleanup_mpz_content as an alternative to _fmpz_cleanup
  - support MinGW32
  - support Cygwin
  - bugfix on ia64
  - support sparc32/sparc64
  - support OSX
  - support Solaris, NetBSD, OpenBSD, etc (if bash, GNU Make present)
- **ulong_extras**
  - implemented the improved Lehman algorithm
  - added n_jacobi_unsigned to allow n > WORD_MAX
  - fixed n_sqrtmod for n > WORD_MAX
  - fixed bug causing n_sqrtmod to hang
  - added sublinear algorithm for computing factorials mod p
  - added n_sqrtmod_primepow, n_sqrtmodn and associated functions for computing square roots modulo composite integers
  - fixed bugs in n_is_prime_pocklington
  - fixed UWORD_MAX case in powmod and powmod2
  - fixed problems with the random number generators
  - fixed rare bug in n_mod_precomp
  - fixed rare bug in n_is_prime_pseudosquare
- **long_extras**
  - added z_sizeinbase
- **qsieve**
  - new module implementing a quadratic sieve for numbers up to two limbs
- **fft**
  - new module providing an efficient Schoenhage-Strassen FFT
- **longlong**
  - added assembly code for ia64 and ARM
  - fixed bug in fallback version of add_sssaaaaaaaa
- **fmpz**
  - added fmpz_fib_ui
  - added double precision natural logarithm
- added fmpz_val2 for 2-valuation
- added mul_2exp, div_2exp, cdiv_q_2exp, tdiv_q_2exp, fdiv_r, fdiv_r_2exp, tdiv_ui, mul_tdiv_q_2exp
- added get_d/set_d
- added fmpz_divisible, divisible_si
- optimised fmpz_powm and fmpz_powm_ui
- added clog, clogui, flog, flogui for computing logarithms
- added abs_lbound_ui_2exp, ubound_ui_2exp
- added fmpz_rfac_ui and fmpz_rfac_uiui for rising factorials
- added functions to obtain read-only fmpz_t’s from mpz_t’s
- added fmpz_init_set, init_set_ui
- added fmpz_gcdinv
- added fmpz_is_square
- added fmpz_tstbit, setbit, clrbit, complement, combit, and, or, xor, popcnt
- added a sign flag for CRT instead of using separate methods
- fixed bugs in fmpz_sqrtmod
- fixed a bug in fmpz_bit_unpack that could cause corruption of the global fmpz array when compiled in single mode
- fixed a bug in fmpz_sub_ui that could cause memory corruption

• fmpz_vec
  - added functions for obtaining the largest absolute value coefficient
  - added functions for computing the sum and product of an integer vector
  - made max_bits much faster
  - added _fmpz_vec_mod_fmpz
  - made randtest produce sparse output

• fmpz_poly
  - added fmpz_poly_sqr, fmpz_poly_sqrlow for squaring a polynomial
  - added fmpz_poly_lcm
  - made multipoint interpolation faster by using the Newton basis
  - added a function for fast division by a linear polynomial
  - added power series composition (classical and Brent-Kung)
  - added power series reversion (classical, Newton, fast Lagrange)
  - added a function for obtaining the largest absolute value coefficient
  - fixed quadratic memory usage and stack overflow when performing unbalanced division or pseudo division using the divconquer algorithm
  - fixed a bug in poly_zero_coeffs
  - fixed a bug in xgcd_modular
  - allowing +/-1 in the constant term of power series inversion
  - fixed aliasing bug in divrem
  - added restartable Hensel lifting and associated utility functions
- fixed rem, which used to only call the basecase algorithm
- fixed pseudo_divrem, which used to only call the basecase algorithm
- implemented Schoenhage-Strassen multiplication (mul_SS, mullow_SS) and enabled this by default
- fixed a bug in the heuristic GCD algorithm
- added functions for Newton basis conversion
- added functions for fast Taylor shift
- added fmpz_poly_sqrt implementing a basecase algorithm
- added scalar mul_2exp, fddiv_2exp, tddiv_2exp
- made randtest produce sparse output
- added fmpz_poly_equal_fmpz
- improved performance by always using basecase multiplication when one polynomial is short
- improved algorithm selection for fmpz_poly_gcd
- fixed several bugs in gcd_modular
- improved performance of gcd_modular

- **fmpz_poly_factor**
  - new module for factorisation of fmpz_polys
  - added a naive implementation of the Zassenhaus algorithm

- **fmpz_mod_poly**
  - new module for polynomials modulo over \( \mathbb{Z}/n\mathbb{Z} \) for arbitrary-precision \( n \)
  - multiplication, powering
  - classical and divconquer division
  - series inversion
  - Euclidean GCD and XGCD
  - invmod
  - radix conversion
  - divconquer composition
  - GCD and division functions that test invertibility of the leading coefficient

- **fmpz_mat**
  - added det_divisor for computing a random divisor of the determinant
  - faster determinant computation using divisor trick
  - faster determinant computation by using multimodular updates
  - fixed \( n \times 0 \times m \) product not zeroing the result
  - various interface improvements
  - faster implementation of Cramer’s rule for multiple right hand sides
  - added fmpz_mat_fread and read
  - added multi CRT/mod functions
  - added trace

- **fmpz_poly_mat**
- fixed n x 0 x m product not zeroing the result
- added inverse
- added rank computation
- added reduced row echelon form and nullspace computation
- added more utility functions
- added squaring and exponentiation
- added balanced product of a sequence of matrices
- added truncate, mullow, sqrlow, pow_trunc
- added trace

- fmpz_factor
  - new module providing interface for integer factorisation
  - fast expansion of a factored integer

- fmpq
  - cleaned up and improved performance of rational reconstruction code
  - allow separate numerator and denominator bounds for rational reconstruction
  - added continued fraction expansion
  - added functions for summation using binary splitting
  - added fmpq_swap
  - added fmpq_print, fmpq_get_str
  - added fmpq_pow_si
  - added functions to obtain read-only fmpq_t's from mpq_t's
  - added fmpq_cmp

- fmpq_mat
  - fixed n x 0 x m product not zeroing the result
  - added fmpq_mat_transpose
  - added trace

- fmpq_poly
  - improved speed of multipoint interpolation using _fmpz_poly_div_root
  - fmpq_poly: added power series composition (classical and Brent-Kung)
  - fmpq_poly: added power series reversion (classical, Newton, fast Lagrange)
  - fixed bug wherein set_array_mpq modified the input
  - added gcd, xgcd, lcm, resultant
  - added fmpq_poly_set_fmpq
  - added fmpq_poly_get_slice, fmpq_poly_reverse
  - fixed aliasing bug in divrem
  - changed some functions to use FLINT scalar types instead of MPIR data types
  - added fmpq_poly_get_numerator

- nmod_poly
  - implemented the half gcd algorithm for subquadratic gcd and xgcd
– added multipoint evaluation and interpolation
– added asymptotically fast multipoint evaluation and interpolation
– added a function for forming the product of linear factors
– added a function for fast division by a linear polynomial
– added power series composition (classical and Brent-Kung)
– added power series reversion (classical, Newton, fast Lagrange)
– added nmod_poly_mulmod, powmod and related functions (ported from flint1)
– added squarefree, irreducibility tests (ported from flint1)
– added Berlekamp and Cantor-Zassenhaus factoring (ported from flint1)
– fixed quadratic memory usage and stack overflow when performing unbalanced division using the divconquer algorithm
– added compose_series_divconquer
– added resultant
– fixed aliasing bug in divrem
– added rem functions
– added divrem_q0, q1 for special cases of division
– added functions for fast Taylor shift
– added nmod_poly_sqrt
– made fread read the modulus from the file
– made randtest produce sparse output
– fixed bug in xgcd_euclidean with scalar inputs

• nmod_vec
  – added functions and macros for computing dot products
  – made randtest produce sparse output

• nmod_mat
  – added addmul/submul functions
  – asymptotically fast solving of triangular systems
  – asymptotically fast LUP decomposition
  – asymptotically fast determinant and rank computation
  – asymptotically fast reduced row echelon form and nullspace
  – asymptotically fast nonsingular solving
  – asymptotically fast inverse
  – tidied some interfaces
  – fixed n x 0 x m product not zeroing the result
  – added trace
  – made multiplication faster for tiny moduli by means of bit packing

• nmod_poly_mat
  – new module for matrices over Z/nZ[x], with similar functionality as the fmpz_poly_mat module
  – determinant, rank, solving, reduced echelon form, nullspace
– fraction-free Gaussian elimination
– multiplication using bit packing
– multiplication using evaluation-interpolation
– determinant using evaluation-interpolation

• padic
– restructured and improved much of the code
– added padic_log
– improved log and exp using rectangular splitting
– added asymptotically fast log and exp based on binary splitting

• perm
– added the perm module for permutation matrices
– computing the parity of a permutation
– inverting a permutation

• arith
– added generation of cyclotomic polynomials
– added functions for evaluating Dedekind sums
– fast computation of the partition function
– added a function for factoring a Hardy-Ramanujan-Rademacher type exponential sum
– added Chebyshev polynomials T and U
– added computation of the minimal polynomial of $\cos(2\pi/n)$
– added asymptotically fast high-precision approximation of $\zeta(n)$
– added asymptotically fast computation of Euler’s constant
– added new algorithms and functions for computing Bell numbers
– fast computation of $\pi$ (adapting code written by Hanhong Xue)
– added functions for computing the number of sum of squares representations of an integer
– renamed functions to have an arith prefix

2011-06-04 – FLINT 2.2

• fmpq (multiprecision rational numbers)
  – Basic arithmetic functions
  – Utility functions
  – Rational reconstruction
  – Functions for enumerating the rationals

• fmpq_mat (matrices over $\mathbb{Q}$)
  – Basic arithmetic functions
  – Utility functions
  – Fast multiplication
  – Classical and fraction-free reduced row echelon form
  – Determinants
Fast non-singular solving

- fmpz_poly_mat (matrices over \( \mathbb{Z}[x] \))
  - Basic arithmetic functions
  - Utility functions
  - Fraction-free row reduction and determinants
  - Fast determinants (experimental)

- fmpz_mat
  - Added more utility functions (scalar multiplication, etc)
  - Added Dixon’s p-adic algorithm (used by fast nonsingular rational system solving)
  - Added reduced row echelon form
  - Added conversions between fmpz_mat and nmod_mat
  - Added CRT functions for fmpz_mats
  - Faster matrix multiplication for small to medium dimensions

- longlong.h
  - Added x86 assembly macros for accumulating sums of two limb operands

- nmod_mat
  - Sped up arithmetic for moduli close to FLINT_BITS

- arith
  - Changed interface of various functions to use new fmpq type

- fmpz
  - Added fmpz_set_ui_mod
  - Inlined fmpz_neg, fmpz_set_si, fmpz_set_ui for better performance
  - Added fmpz_lcm
  - Small performance improvement to fmpz_CRT_ui

- fmpz_vec
  - Added _fmpz_vec_lcm

- fmpz_poly_q (rational functions over \( \mathbb{Q} \), modeled as quotients of fmpz_polys)
  - Basic arithmetic functions
  - Conversion and IO functions
  - Evaluation

- padic (p-adic numbers – experimental)
  - Basic arithmetic
  - Data conversion and IO
  - Inverse and square root using Newton iteration
  - Teichmuller lifts (not optimised)
  - p-adic exponential function (not optimised)

- fmpz_poly
  - Added fmpz_poly_gcd_modular (and fmpz_poly_gcd wrapper)
  - Added fmpz_poly_xgcd_modular (and fmpz_poly_xgcd wrapper)
– Added conversions between fmpz_poly and nmod_poly
– Added CRT functions
– Added multipoint evaluation and interpolation

• nmod_poly
  – Added nmod_poly_xgcd_euclidean (and nmod_poly_xgcd wrapper)
  – nmod_poly_gcd wrapper

• mpn_extras
  – Added MPN_NORM and MPN_SWAP macros.
  – Added mpn_gcd_full to remove some of the restrictions from the usual mpn_gcd

• build fixes
  – fixed make install to create nonexistent dirs (reported by Serge Torres)
  – -L use /usr instead of /usr/local by default (reported by Serge Torres)
  – guards for system headers because of flint’s use of ulong

2011-03-09 – FLINT 2.1

• fmpz
  – Simplified interface for fast multimodular reduction and CRT reconstruction
  – Fixed segmentation fault in fmpz_multi_mod_ui when the input exceeds the product of the moduli
  – Added simple incremental CRT functions (fmpz_CRT_ui, fmpz_CRT_ui_unsigned) to complement the existing fast ones
  – Added example programs for CRT
  – Added random number generators designed for testing modular code (fmpz_randtest_mod, fmpz_randtest_mod_signed)
  – Added fmpz_fdiv_ui for remainder on division by an ulong
  – Added fmpz_bin_uiui for computing binomial coefficients
  – Added fmpz_mul2_uiui and fmpz_divexact2_uiui for multiplying or dividing an fmpz by a pair of ulongs (efficiently when their product fits in a single limb)

• fmpz_mat
  – Added utility functions for basic arithmetic and creating unit matrices
  – Added multimodular determinant computation (certified or heuristic)
  – Added support for computing right nullspaces (fmpz_mat_kernel). Fast only for small matrices.
  – Some internal code cleanup and various small fixes

• nmod_mat
  – Faster Gaussian elimination for small moduli
  – Faster determinants
  – Faster matrix inversion and nonsingular solving

• nmod_poly
  – Added nmod_poly_integral for computing integrals
Added fast square root and inverse square root of power series
Added fast transcendental functions of power series (log, exp, sin, cos, tan, sinh, cosh, tanh, asin, atan, asinh, atanh)
Made nmod_poly_inv_series_newton more memory efficient

- **fmpq_poly**
  - Added fmpq_poly_integral for computing integrals
  - Added fast transcendental functions of power series (log, exp, sin, cos, tan, sinh, cosh, tanh, asin, atan, asinh, atanh)

- **arith**
  - Made computation of vectors of Bernoulli numbers faster
  - Added fast computation of single Bernoulli numbers
  - Added a separate function for computing denominators of Bernoulli numbers
  - Added fast computation of Bell numbers (vector and single)
  - Added fast computation of Euler numbers (vector and single)
  - Added fast computation of Euler polynomials
  - Added fast computation of Swinnerton-Dyer polynomials
  - Added fast computation of Legendre polynomials
  - Added fast vector computation of the partition function
  - Added fast vector computation of Landau’s function

- **ulong_extras**
  - Added a function for computing factorials mod n

- **build system**
  - Added support for building static and shared libraries
  - All object files and test/profile/example binaries now build in separate build directory

- **documentation**
  - Large number of corrections

2011-01-16 — FLINT 2.0

N.B: FLINT 2 is a complete rewrite of flint from scratch. It includes the following modules:

- **ulong_extras**: (word sized integers and modular arithmetic)
  - random numbers (randint, randbits, randprime, randint)
  - powering
  - reverse binary
  - mod, divrem, mulmod all with precomputed inverses
  - gcd, invgcd, xgcd
  - jacobi symbols
  - addmod, submod, invmod, powmod
  - prime sieve, nextprime, prime-pi, nth-prime
  - primality testing (small, binary search, Pocklington-Lehmer, Pseudosquare)
- probably prime tests (strong base-a, Fermat, Fibonacci, BPSW, Lucas)
- sqrt, sqrtrem, is-square, perfect-power (2,3,5)
- remove, is-squarefree
- factorisation (trial-range, trial, power (2,3,5), one-line, SQUFOF)
- Moebius mu, Euler phi

- fmpz: (memory managed multiple precision integers)
  - memory management (init, clear)
  - random numbers (randbits, randm)
  - conversion to and from long, ulong, doubles, mpz’s, strings
  - read/write to file, stdin, stdout
  - sizeinbase, bits, size, sgn, swap, set, zero
  - cmp, cmp-ui, cmpabs, equal, is-zero, is-one
  - neg, abs, add, add-ui, sub, sub-ui, mul, mul-si, mul-ui, mul-2exp
  - addmul, addmul-ui, submul, submul-ui
  - cdiv-q, cdiv-q-si, cdiv-q-ui
  - fdiv-q, fdiv-q-si, fdiv-q-ui, fdiv-qr, fdiv-q-2exp
  - tdiv-q, tdiv-q-si
  - divexact, divexact-si, divexact-ui
  - mod, mod-ui
  - powering
  - sqrt, sqrt-rem
  - factorial
  - ged, invmod
  - bit-pack, bit-unpack
  - multimodular reduction, CRT

- fmpz_vec: (vectors over fmpz’s)
  - memory management (init, clear)
  - random vectors
  - max-bits, max-limbs
  - read/write to file/stdin/stdout
  - set, swap, zero, neg
  - equal, is-zero
  - sort
  - add, sub
  - scalar multiplication by fmpz, ulong, long, 2exp
  - exact division by fmpz, long, ulong
  - fdiv-q by fmpz, long, ulong, 2exp
  - tdiv-q by fmpq, long, ulong
  - addmul by fmpz, long, long by 2exp
• submul by fmpz, long, long by 2exp
  • Gaussian content

• fmpz_poly: (polys over fmpz's)
  • memory management (init, realloc, fit-length, clear)
  • random polys
  • normalise, set-length, truncate
  • length, degree, max-limbs, max-bits
  • set, set-si, set-ui, set-fmpz, set-str
  • get-str, get-str-pretty
  • zero, one, zero-coeffs
  • swap, reverse
  • get/set coeffs from fmpz, long, ulong
  • get-coeff-ptr, lead
  • equal, is-zero
  • add, sub
  • scalar multiplication by fmpz, long, ulong
  • scalar addmul/submul by fmpz
  • scalar fdiv by fmpz, long, ulong
  • scalar tdiv by fmpz, long, ulong
  • scalar divexact by fmpz, long, ulong
  • bit pack, bit unpack
  • multiplication (classical, karatsuba, KS)
  • mullow (classical, karatsuba, KS)
  • mullhigh (classical, karatsuba)
  • middle product (classical)
  • powering (small, binary exponent, binomial, multinomial, addition chains)
  • truncated powering (binary exponent)
  • shift left/right
  • euclidean norm
  • gcd (subresultant)
  • resultant
  • content, primitive part
  • divrem (basecase, divide-and-conquer)
  • div (basecase, divide-and-conquer)
  • rem (basecase)
  • invert series (basecase, Newton)
  • div series
  • pseudo divrem (basecase, divide-and-conquer, Cohen)
  • rem (Cohen)
- div
- evaluate (Horner) at fmpz, mpq, a mod n
- composition (Horner, divide-and-conquer)
- signature
- read/write to file/stdin/stdout

- fmpq_poly: (polynomials over Q stored as poly over fmpz with fmpz denominator)
  - memory management (init, realloc, fit-length, clear)
  - random polys
  - set-length, canonicalise, normalise, truncate
  - is-canonical, length, degree
  - reference to numerator, denominator
  - set, set-si, set-ui, set-fmpz, set-mpq
  - set-array-mpq, set-str
  - get-str, get-str-pretty
  - zero, neg, swap
  - invert
  - set coefficient to mpq, long, ulong, fmpz, mpz
  - get coefficient as mpq
  - equal, cmp, is-one, is-zero
  - add, sub
  - scalar multiplication by long, ulong, fmpz, mpq
  - scalar division by fmpz, long, ulong, mpq
  - multiplication, mullow
  - powering
  - shift left/right
  - divrem, div, rem
  - invert series (Newton iteration)
  - divide series
  - derivative
  - evaluate at fmpz, mpq
  - composition, scale by constant
  - content, primitive part
  - make-monic, is-monic
  - is-squarefree
  - read/write to file/stdin/stdout

- nmod_vec: (vectors over Z/nZ for n fitting in a machine word)
  - memory management (init/clear)
  - macros for efficient reduction of 1, 2 and 3 limb integers mod n
  - macro for addmul mod n
- add/sub/neg individual coefficients mod n
- random vectors
- set, zero, swap
- reduce, max-bits
- equal
- add, sub, neg
- scalar multiplication by a value reduced mod n
- scalar addmul by a value reduced mod n
- nmod_poly: (polynomials over Z/nZ for n fitting in a machine word)
  - memory management (init, realloc, fit-length, clear)
  - random polys
  - normalise, truncate
  - length, degree, modulus, max-bits
  - set, swap, zero, reverse
  - get/set coefficients as ulongs, strings
  - read/write to file, stdin, stdout
  - equal, is-zero
  - shift left/right
  - add, sub, neg
  - scalar multiplication by a value reduced mod n
  - make-monic
  - bit pack, bit unpack
  - multiplication (classical, KS)
  - mullow (classical, KS)
  - mulhigh (classical)
  - powering (binary exponent)
  - pow-trunc (binary exponent)
  - divrem (basecase, divide-and-conquer, Newton iteration)
  - div (basecase, divide-and-conquer, Newton iteration)
  - invert series (basecase, Newton iteration)
  - divide series (Newton iteration)
  - derivative
  - evaluation at a value taken mod n
  - composition (Horner, divide-and-conquer)
  - gcd (euclidean)
- fmpz_mat: (matrices over fmpz’s)
  - memory management (init, clear)
  - random matrices (bits, integer relations, simultaneous diophantine equations NTRU-like, aj-tai, permutation of rows and cols of a diagonal matrix, random of given rank, random of given determinant, random elementary operations)
– set, init-set, swap, entry pointer
– write to file or stdout
– equal
– transpose
– multiplication (classical, multimodular)
– inverse
– determinant
– row reduce (Gaussian and Gauss-Jordan fraction-free elimination)
– rank
– solve $Ax = b$, solve $AX = B$
– fraction free LU decomposition

• nmod_mat: (matrices over $\mathbb{Z}/n\mathbb{Z}$ for $n$ fitting in a machine word)
  – memory management (init, clear)
  – random matrices (uniform, full, permutations of diagonal matrix, random of given rank, random elementary operations)
  – set, equal
  – print to stdout
  – add
  – transpose
  – multiplication (classical, Strassen, $A*B^T$)
  – row reduction (Gaussian and Gauss-Jordan)
  – determinant
  – rank
  – solve ($Ax = b$, $AX = B$, solve with precomputed LU)
  – invert

• arith: (arithmetic functions)
  – Bernoulli numbers
  – Bernoulli polynomials
  – primorials (product of primes up to $n$)
  – harmonic numbers
  – Stirling numbers
  – Euler phi function
  – Moebius mu function
  – Sigma (sum of powers of divisors)
  – Ramanujan tau function

• examples: (example programs)
  – compute coefficients of q-series of Delta function
• mpfr_vec: (vectors over mpfr reals)
  – memory management (init, clear)
- add
- set, zero
- scalar multiplication by mpfr, 2exp
- scalar product

- mpfr_mat: (matrices over mpfr reals)
  - memory management (init, clear)

2010-12-24 – FLINT 1.6.0

- Bugs:
  - Fixed a memory leak in mpz_poly_to_string_pretty
  - Fixed a bug inherited from an old version of fpLLL
  - Makefile to respect CC and CXX
  - Fixed bug in F_mpz_set_si
  - Fixed bug in F_mpz_equal
  - Most for loops to C90 standard (for easier MSVC porting)
  - Better Cygwin support
  - Fixed a bug in zmod_poly_resultant
  - Fixed bug in F_mpz_mul_KS/2
  - Fixed bug in tinyQS
  - Worked around some known bugs in older GMP/MPIR's

- Major new functionality
  - F_mpz_poly_factor_zassenhaus
  - F_mpz_poly_factor (incl. fmpz_poly_factor wrapper) using new vH-N approach (see the paper of van Hoeij and Novocin and the paper of van Hoeij, Novocin and Hart)
  - Implementation of new CLD bounds function for polynomial factors (see the paper of van Hoeij, Novocin and Hart)
  - Restartable Hensel lifting
  - Heuristic LLL implementations using doubles and mpfr
  - LLL implementations optimised for knapsack lattices
  - New (probably subquadratic) LLL implementation (ULLL)
  - zmod_poly_factor_cantor_zassenhaus
  - New F_mpz_mod_poly module for polynomials over Z/pZ for multiprec. p

- Some of the other new functions added
  - F_mpz
  - F_mpz_gcd
  - F_mpz_smod
  - F_mpz_mod_preinv
  - F_mpz_fdiv_qr
  - F_mpz_get/set/mpfr/2exp
- F_mpz_sscanf
- F_mpz_set_d
- F_mpz_poly:
  - read F_mpz_poly to_string/from_string/fprint/print/fread/pretty
- F_mpz_poly_to/from_zmod_poly
- F_mpz_poly_scalar_div_exact
- F_mpz_poly_sm
- F_mpz_poly_derivative, F_mpz_poly_content, F_mpz_poly_eval_horner_d/2exp
- F_mpz_poly_scalar_abs
- F_mpz_poly_set_d_2exp
- F_mpz_poly_div/divrem
- F_mpz_poly_gcd
- F_mpz_poly_is_squarefree
- F_mpz_poly_factor_squarefree
- F_mpz_poly_mul_trunc_left
- F_mpz_poly_pseudo_div
- F_mpz_poly_set_coeff
- F_mpz_poly_pow_ui
- Inflation/deflation trick for factorisation
- zmod_poly:
  - Inflation/deflation trick for factorisation
- mpz_mat:
  - mpz_mat_from_string/to_string/fprint/fread/pretty
- mpz_mat:
  - mpz_mat_init/clear
  - Gramm-schmidt Orthogonalisation
- F_mpz_mat:
  - F_mpz_mat_print/fprint/fread/pretty
  - F_mpz_mat_mul_classical
  - F_mpz_mat_max_bits/2
  - F_mpz_mat_scalar_mul/div_2exp
  - F_mpz_mat_col_equal
  - F_mpz_mat_sm
  - F_mpz_vec_scalar_product/norm
  - F_mpz_vec_add/submul_ui/si/F_mpz/2exp
- zmod_mat:
  - classical multiplication
  - strassen multiplication
  - scalar multiplication
• zmod_mat_equal
• zmod_mat_add/sub
• zmod_mat_addmul_classical
• d_mat:
  • d_vec_norm, d_vec_scalar_product
• mpfr_mat:
  • mpfr_vec_scalar_product/norm

2009-09-22 – FLINT 1.5.0

• Added multimodular reduction and CRT to F_mpz module
• Fixed some bugs in F_mpz module and numerous bugs in test code
• Added zmod_poly_compose
• Added zmod_poly_evaluate
• Added functions for reduced evaluation and composition to mpfr_poly module (contributed by Burcin Erocal)
• Fixed bugs in the primality tests in long_extras
• Removed all polynomial multimodular multiplication functions
• Added new thompson code used in the 1 trillion triangles computation
• Fixed a severe bug in the mpfr_poly_pseudo_div function reported by Sebastian Pancratz
• Added mpz_comb_temp_init/clear functions
• Fixed a normalisation buglet in mpz_poly_pack_bytes
• Added F_mpz_pow_ui function (contributed by Andy Novocin)
• Fixed a severe bug in the FFT reported by William Stein and Mariah Lenox (fix contributed by David Harvey)
• Removed some memory leaks from F_mpz test code
• Fixed bug in zmod_poly_evaluate test code

2009-07-06 – FLINT 1.4.0

• Sped up zmod_poly division in case where operands are the same length
• Sped up zmod_poly division in case where operands have lengths differing by 1
• Fixed a bug in zmod_poly_gcd for polynomials of zero length
• Sped up zmod_poly_gcd considerably (both euclidean and half gcd)
• Sped up zmod_poly_gcd_invert and zmod_poly_xgcd considerably
• Made zmod_poly_gcd_invert and zmod_poly_xgcd asymptotically fast
• Made zmod_poly_resultant asymptotically fast
• Added optimised zmod_poly_rem function
• Fixed a divide by zero bug in zmod_poly_factor_berlekamp
• Added test code for z_factor_tinyQS and z_factor_HOLF
• Fixed many bugs in the z_factor code, tinyQS and mpQS
• Corrected numerous typos in the documentation and added missing descriptions
• Added F_mpz_cmp function
• Added documentation to the manual for the new F_mpz module

2009-06-09 – FLINT 1.3.0

• Added new code for checking 2nd, 3rd and 5th roots
• Fixed a bug in z_factor
• Connected quadratic sieve for factoring large ulongs
• Added one line factor algorithm
• constructed best of breed factor algorithm
• Fixed termination conditions for z_intcuberooot and z_intfifthroot which were broken
• Added some code for special cases which cause infinite loops in cuberooot and fifthroot
• Went back to ceil(pow(n, 0.33333333)) and ceil(pow(n, 0.2)) for initial guesses in cube and fifth root functions as these were about 50% faster than sqrt(n) and sqrt(sqrt(n)) respectively.
• Added test code for z_intfifthroot
• Added test code for z_factor_235power
• Fixed some uninitialised data found by valgrind in intcuberooot and intfifthroot
• Fixed multiply defined PRIME_COUNT in long_extras-test
• Got rid of gotos in some functions in long_extras
• Knocked optimisation level back to -O2 because it miscompiles on sage.math
• Changed tables to use uint64_t’s instead of ulongs which are not 64 bits on a 32 bit machine
• Only checked MAX_HOLF on 64 bit machine
• Changed MAX_SQUFOF to WORD(-1)
• Check constant 0x3FFFFFFFFFUL only on a 64 bit machine
• Fixed a bug in z_oddprime_lt_4096 only on a 32 bit machine
• Fixed some TLS issues with Cygwin
• Fixed some typos in makefile
• Fixed a wrong path in fmpz.c

2009-04-18 – FLINT 1.2.5

• Upgraded to zn_poly-0.9 to avoid a serious error in squaring of large polynomials over Z/nZ
2009-04-04 – FLINT 1.2.4

- Defined THREAD to be blank on Apple CC and __thread for thread local storage on other gcc’s (where it’s defined)
- #undef ulong in profiler.h where time.h and other system time headers are included (both reported by M. Abshoff)

2009-03-31 – FLINT 1.2.3

- Fixed bugs in all fmpz_poly evaluation functions, identified by Burcin Eracal.

2009-03-20 – FLINT 1.2.2

- Fixed a memory leak in zmod_poly_factor
- Fixed zmod_poly-profile build

2009-03-14 – FLINT 1.2.1

- Removed some FLINT 2.0 code which was interfering with the build of the NTL-interface
- Removed an omp.h from fmpz_poly.c.

2009-03-10 – FLINT 1.2.0

- made memory manager, fmpz and fmpz_poly threadsafe
- Code for running tests in parallel (not activated)
- Sped up fmpz_poly_scalar_div_ui/si when scalar is 1/-1
- Parallelise _fmpz_poly_mul_modular
- fmpz_mul_modular_packed to pack coefficients to the byte before running _fmpz_poly_mul_modular
- fmpz_poly_pseudo_rem_cohen (not documented)
- special case for leading coeff 1/-1 in fmpz_poly_pseudo_divrem_basecase
- removed a memory allocation bug which caused a massive slowdown in fmpz_poly_pseudo_divrem_basecase
- fmpz_poly_pseudo_rem_basecase (not documented)
- fmpz_poly_pseudo_rem (not asymptotically fast)
- fmpz_poly_signature (not asymptotically fast)
- basic fmpz_poly_is_squarefree function
- included zn_poly in trunk and made FLINT build zn_poly as part of its build process
- switched to using zn_poly for polynomial multiplication, newton inversion, scalar multiplication in zmod_poly
- Integer cube root of word sized integers
- Fibonacci pseudoprime test
- BSPW probable prime test
- n - 1 primality test
- Complete implementation of z_issquarefree
• Significantly improved the thetaproduct example program.
• Fixed bug in `fmpz_poly_byte_pack` which is triggered when trying to pack into fields a multiple of 8 bytes (could cause a segfault)
• Fixed a bug in `fmpz_poly_pseudo_divrem` (relied on an uninitialised poly to have length 0)
• Fixed bug in `fmpz_multiCRT_ui` (could segfault)
• Fixed bug in `fmpz_multi_mod_ui` (could segfault)
• Fixed memory leak in `zmod_poly_factor_squarefree`
• Fixed memory leak in `zmod_poly_from_string`

2009-03-01 – FLINT 1.1.3

• Inserted some missing return values in `zmod_poly` test code.

2009-03-01 – FLINT 1.1.2

• Fixed some memory allocation slowdowns and bugs in `fmpz_poly` division and pseudo division functions (reported by William Stein).

2009-02-11 – FLINT 1.1.1

• Fixed bugs in `__fmpz_poly_scalar_mul_fmpz`, `fmpz_poly_gcd_heuristic` and `fmpz_poly_gcd_subresultant` and fixed bugs in `test__fmpz_poly_scalar_div_fmpz`, `test_fmpz_poly_scalar_div_fmpz` and `test_fmpz_poly_scalar_div_mpz`.

2008-12-21 – FLINT 1.1.0

Some of the following features were previewed in FLINT 1.0.11.

• integer gcd (this just wraps the GMP gcd code)
• polynomial content
• primitive part
• convert to and from FLINT and NTL integers and polynomials
• get a coefficient of a polynomial efficiently as a read only `mpz_t`
• print polynomials in a prettified format with a specified variable
• Sped up integer multiplication
• Convert to and from `zmod_polys` from `fmpz_polys`
• Chinese remainder for `fmpz_polys`
• Leading coeff macro
• Euclidean norm of polynomials
• Exact division testing of polynomials
• Polynomial GCD (subresultant, heuristic, modular)
• Modular inversion of polynomials
• Resultant
• XGCD (Pohst-Zassenhaus)
• Multimodular polynomial multiplication
• Rewritten karatsuba_trunc function
• Rewritten division functions
• Polynomial derivative
• Polynomial evaluation
• Polynomial composition
• Addition and subtraction of zmod_polys
• Sped up multiplication of zmod_polys
• Extended multiplication of zmod_polys to allow up to 63 bit moduli
• zmod_poly subpolynomials
• zmod_poly reverse
• Truncated multiplication for zmod_polys (left, right, classical and KS)
• Scalar multiplication
• Division for zmod_polys (divrem and div, classical, divide and conquer and newton)
• Series inversion for zmod_polys
• Series division for zmod_polys
• Resultant for zmod_polys
• GCD for zmod_polys including half-gcd
• Inversion modulo a polynomial for zmod_polys
• XGCD for zmod_polys
• Squarefree factorisation for zmod_polys
• Berlekamp factorisation for zmod_polys
• Irreducibility testing for zmod_polys
• Derivative for zmod_polys
• Middle product for zmod_polys (sped up newton division)
• addmod, submod and divmod for ulongs
• Sped up limb sized integer square root
• Partial factoring of ulongs
• z_randbits
• Pocklington-Lehmer primality testing
• BSPW pseudo-primality testing
• Fermat pseudo-primality testing
• Fast Legendre symbol computation
• Chinese remainder for fmpzs
• Square root with remainder for fmpzs
• Left and right shift for fmpzs
• Reduction modulo a ulong for fmpzs
• Montgomery redc, mulmod, divmod and mod for fmpzs
• Multimodular reduction and CRT for fmpzs
• fmpz_mulmod and fmpz_divmod
• fmpz_invert for inversion modulo an fmpz
• Dramatically sped up gcd for small fmpzs
• Computation of 1D, 2D and some 3D theta functions
• Example program for multiplying theta functions
• Test code now times test functions
• Quick and dirty timing function for profiler
• Tiny quadratic sieve for small one and two limb integers
• Completely rewritten self initialising multiple polynomial quadratic sieve
• Build fix for 64 bit OSX dylibs (reported by Michael Abshoff)

2008-12-25 – FLINT 1.0.21
• Fixed the Christmas bug reported by Michael Abshoff which causes a test failure in fmpz_poly_gcd_modular and a hang in fmpz_poly_invmod_modular on 32 bit machines

2008-12-13 – FLINT 1.0.20
• Fixed some bugs in conversion of zmod_poly’s to and from strings

2008-12-12 – FLINT 1.0.19
• Fixed a bug in z_remove_precomp

2008-12-05 – FLINT 1.0.18
• Fixed another bug in the fmpz_poly_set_coeff_* functions which resulted in dirty coefficients

2008-11-30 – FLINT 1.0.17
• Fixed a segfault caused by left shifting of polynomials with zero limbs allocated in division and pseudo division functions.
• Fixed a bound used in fmpz_gcd_modular to use a proven bound
• Fixed a bug in fmpz_poly-profile where the top bit of random coefficients of n bits was always set

2008-10-22 – FLINT 1.0.16
• Fixed a segfault when trying to truncate a polynomial to an longer length than it currently is, with the function fmpz_poly_truncate (reported by Craig Citro)
2008-10-15 – FLINT 1.0.15

- Fixed a bug which causes a segfault when setting a coefficient of the zero polynomial to zero
- Fixed build bug in longlong.h on s390 platform

2008-09-23 – FLINT 1.0.14

- Update long_extras and test code for the sake of new quadratic sieve (new functions in long_extras remain undocumented)
- Removed many bugs from tinyQS and mpQS and joined them into a single program for factoring integers

2008-07-13 – FLINT 1.0.13

- Fixed memory leaks and dirty memory issues in test code for numerous modules.
- Removed further issues with cache prefetching in mpn_extras.c

2008-07-11 – FLINT 1.0.12

- Removed some Opteron tuning flags which cause illegal instruction errors on Pentium4
- Fixed numerous memory leaks in fmpz_poly test code
- Fixed memory leak in fmpz_poly_power_trunc_n
- Fixed major memory leaks in fmpz_poly_xgcd_modular
- Rewrote __fmpz_poly_mul_karaturc_recursive and __fmpz_poly_mul_karatsuba_trunc to “prove code” and got rid of some dirty memory issues
- Fixed some potential illegal memory accesses to do with cache prefetching in fmpz_poly.c

2008-07-09 – FLINT 1.0.11

- Fixed a bug in z_ll_mod_precomp on ia64 (reported by Michael Abshoff and William Stein)

2008-06-16 – FLINT 1.0.10

- integer gcd (this just wraps the GMP gcd code)
- polynomial content
- convert to and from FLINT and NTL integers and polynomials
- get a coefficient of a polynomial efficiently as a read only mpz_t
- print polynomials in a prettified format with a specified variable
2008-03-11 – FLINT 1.0.9

- Fixed a memory allocation bug in fmpz_poly_power

2008-02-15 – FLINT 1.0.8

- Fixed a bug in fmpz_poly_right_shift (reported by Kiran Kedlaya)

2008-01-22 – FLINT 1.0.7

- Made F_mpn_mul binary compatible with the way mpn_mul operates in practice.

2008-01-17 – FLINT 1.0.6

- Fixed an issue with FLINT_BIT_COUNT on certain machines (probably due to arithmetic shift issues)

2008-01-05 – FLINT 1.0.5

- Fixed some inline issues which cause problems because of the C99 inline rules (reported by David Harvey).
- Fixed a makefile issue reported (and solved) by David Harvey when not linking against NTL.

2008-01-04 – FLINT 1.0.4

- Fixed a bug in the bernoulli_zmod example program and associated polynomial zmod code which caused memory corruption.
- Fixed a bug in the fmpz-test code which manifested on 32 bit machines, reported by David Harvey.
- Fixed some bugs in the pari profiling code.

2007-12-16 – FLINT 1.0.3

- Fixed a bug in the polynomial memory management code which caused a segfault
- Fixed a bug in the pseudo division code which caused a block overrun

2007-12-10 – FLINT 1.0.2

- Rewrote tuning code for integer multiplication functions, making it more robust and fixing a bug which showed up on 32 bit machines (reported by Michael Abshoff and Jaap Spies). Factored the tuning code out into a number of macros.
2007-12-07 – FLINT 1.0.1

- Fixed a bug in _fmpz_poly_maxbits1 on 32 bit machines, reported by Michael Abshoff and Carl Witty
- Removed some instances of u_int64_t and replaced them with uint64_t, reported by Michael Abshoff
- Replaced sys/types.h with stdint.h
- Added FLINT macros to documentation
- Corrected numerous typos in documentation

2007-12-02 – FLINT 1.0

- First version of FLINT, includes fmpz_poly, fmpz and mpQS

16.1.2 Antic version history

2021-06-24 – Antic 0.2.5

- TODO: list changes here

2021-04-15 – Antic 0.2.4

- TODO: list changes here

2020-12-11 – Antic 0.2.3

- TODO: list changes here

2020-06-30 – Antic 0.2.2

- TODO: list changes here

2020-06-16 – Antic 0.2.1

- TODO: list changes here

2019-02-12 – Antic 0.2

- Many bug fixes, standalone build system, continuous integration.
2013-05-12 – Antic 0.1

- First version of antic, including a qfb module for (positive definite) binary quadratic forms.

16.1.3 Calcium version history

2021-05-28 – Calcium 0.4

- Algebraic numbers
  - Fixed bug in special-casing of roots of unity in qqbar_root_ui.
  - Fixed qqbar_randtest with bits == 1.
  - Faster qqbar_cmp_re for nearby reals.
  - Faster qqbar polynomial evaluation and powering using linear algebra.
  - Improved qqbar_abs, qqbar_abs2 to produce cleaner enclosures.
  - Use a slightly better method to detect real numbers in qqbar_sgn_im.
  - Added qqbar_hash.
  - Added qqbar_get_fmpq, qqbar_get_fmpz.
  - Added qqbar_pow_fmpq, qqbar_pow_fmpz, qqbar_pow_si.
  - Added qqbar_numerator, qqbar_denominator.

- Basic arithmetic and elementary functions
  - Improved ca_condense_field: automatically demote to a simple number field when the only used extension number is algebraic.
  - Improved multivariate field arithmetic to automatically remove algebraic or redundant monomial factors from denominators.
  - Added ca_pow_si_arithmetic (guaranteed in-field exponentiation).
  - Added polynomial evaluation functions (ca_fmpz_poly_evaluate, ca_fmpq_poly_evaluate, ca_fmpzmpoly_q_evaluate).
  - Added several helper functions (ca_is_special, ca_is_qq_elem, ca_is_qq_elem_zero, ca_is_qq_elem_one, ca_is_qq_elem_integer, ca_is_nf_elem, ca_is_cyclotomic_nf_elem, ca_is_generic_elem).
  - Added ca_rewrite_complex_normal_form.
  - Perform direct complex conjugation in cyclotomic fields.
  - Use ca_get_acb_raw instead of ca_get_acb when printing to avoid expensive recomputations.
  - Added alternative algorithms for various basic functions.
  - Deep complex conjugation.
  - Use complex conjugation in is_real, is_imaginary, is_negative_real.
  - Added functions for unsafe inversion for internal use.
  - Significantly stronger zero testing in mixed algebraic-transcendental fields.
  - Added ca_arg.
  - Added special case for testing equality between number field elements and rationals.
  - Added ca_sin_cos, ca_sin, ca_cos, ca_tan and variants.
  - Added ca_atan, ca_asin, ca_acos and variants.
- Added ca_csgn.
- Improved ca_get_acb and ca_get_acb_accurate_parts to fall back on exact zero tests when
direct numerical evaluation does not give precise enclosures.
- Added ca_get_decimal_str.
- More automatic simplifications of logarithms (simplify logarithms of exponentials, square roots
and powers raised to integer powers).
- More automatic simplifications of square roots (simplify square roots of exponentials, square
roots and powers raised to integer powers).
- Improved order comparisons (ca_check_ge etc.) to handle special values and to fall back on
strong equality tests.
- Fixed a test failure in the ca_mat module.

• Polynomials
- Added ca_poly_inv_series, ca_poly_div_series (power series division).
- Added ca_poly_exp_series (power series exponential).
- Added ca_poly_log_series (power series logarithm).
- Added ca_poly_atan_series (power series arctangent).

• Other
- Added fmpz_mpoly_q_used_vars.
- Remove useless rpath line from configure (reported by Julien Puydt).
- Added missing declaration of fexpr_hash.
- Fixed crashes on OS X in Python interface (contributed by deinst).
- Fixed memory leaks in Python string conversions (contributed by deinst).
- Reserve I, E for symbolic expressions in Python interface.

2021-04-23 – Calcium 0.3

• Symbolic expressions
- Added the fexpr module for flat-packed unevaluated symbolic expressions.
- LaTeX output.
- Basic manipulation (construction, replacement, accessing subexpressions).
- Numerical evaluation with Arb.
- Expanded normal form.
- Conversion methods for other types.
- Enable LaTeX rendering of objects in Jupyter notebooks.

• Algebraic numbers
- Fix a major performance issue (slow root refinement) that made Calcium as a whole far slower
than necessary.
- Added qqbar_cmp_root_order; sort polynomial roots consistently by default.
- Added qqbar_get_quadratic.
- Added qqbar_equal_fmpq_poly_val and use it to speed up checking guessed values.
- Conversion of \texttt{qqbar\_t} to and from symbolic expression (\texttt{qqbar\_set\_fexpr}, \texttt{qqbar\_get\_fexpr\_repr}, \texttt{qqbar\_get\_fexpr\_root\_nearest}, \texttt{qqbar\_get\_fexpr\_root\_indexed}, \texttt{qqbar\_get\_fexpr\_formula}).
- Fixed bugs in \texttt{qqbar\_cmp\_abs\_re}, \texttt{cmp\_abs\_im}.
- Optimized \texttt{qqbar\_cmp\_im} and \texttt{qqbar\_cmp\_abs\_im} for conjugates with mirror symmetry.
- Added \texttt{qqbar\_pow} (taking a \texttt{qqbar} exponent).
- Special-case roots of unity in \texttt{qqbar\_pow\_ui}, \texttt{qqbar\_root\_ui}, \texttt{qqbar\_abs} and \texttt{qqbar\_abs2}.
- Wrapped \texttt{qqbar} in Python.

- **Polynomials**
  - Added several utility functions.
  - Optimized polynomial multiplication with rational entries.
  - Fast polynomial multiplication over number fields.

- **Matrices**
  - Fast matrix multiplication over number fields.
  - Right kernel (\texttt{ca\_mat\_right\_kernel}).
  - Matrix diagonalization (\texttt{ca\_mat\_diagonalization}).
  - Jordan normal form (\texttt{ca\_mat\_jordan\_form}, \texttt{ca\_mat\_jordan\_transformation}, \texttt{ca\_mat\_jordan\_blocks}).
  - Matrix exponential (\texttt{ca\_mat\_exp}).
  - Matrix logarithm (\texttt{ca\_mat\_log}).
  - Polynomial evaluation (\texttt{ca\_mat\_ca\_poly\_evaluate}).
  - Cofactor expansion algorithm for determinant and adjugate (\texttt{ca\_mat\_adjugate\_cofactor}).
  - Added several utility functions.
  - Improved algorithm selection in \texttt{ca\_mat\_inv}.
  - Solving using the adjugate matrix.
  - Danilevsky characteristic polynomial algorithm (\texttt{ca\_mat\_charpoly\_danilevsky}).

- **Field elements**
  - Use factoring in \texttt{ca\_sqrt} to enable more simplifications.
  - Simplify square roots and logarithms of negative real numbers.
  - Optimized \texttt{ca\_sub}.
  - Conversion of \texttt{ca\_t} to and from symbolic expressions (\texttt{ca\_set\_fexpr}, \texttt{ca\_get\_fexpr}).
  - Added function for assigning elements between context objects (\texttt{ca\_transfer}).
  - Fixed a possible memory corruption bug when Vieta’s formulas are used.
  - Optimized constructing square roots of rational numbers.

- **Other**
  - Added demonstration notebook to documentation.
  - Fixed OSX compatibility in Python wrapper (contributed by deinst).
  - Fixed bug in \texttt{calculus\_write\_acb}.
  - Fixed bug in \texttt{fmpz\_mpoly\_vec\_set\_primitive\_unique} (contributed by gbunting).
2020-10-16 – Calcium 0.2

- Basic arithmetic and expression simplification
  - Use Gröbner basis for reduction ideals, making simplification much more robust.
  - Compute all linear relations with LLL simultaneously instead of piecemeal.
  - Make monomial ordering configurable (default is lex as before).
  - Use Vieta’s formulas to simplify expressions involving conjugate algebraic numbers.
  - Denest exponentials of symbolic logarithms.
  - Denest logarithms of symbolic powers and square roots.
  - Denest powers of symbolic powers.
  - Simplify exponentials that evaluate to roots of unity.
  - Simplify logarithms of roots of unity.
  - Improve ideal reduction to avoid some unnecessary GCD computations.

- Python wrapper
  - Calcium now includes a minimal ctypes-based Python wrapper for testing.

- New ca_mat module for matrices
  - Mostly using naive basecase algorithms.
  - Matrix arithmetic, basic manipulation.
  - Construction of special matrices (Hilbert, Pascal, Stirling, DFT).
  - LU factorization.
  - Fraction-free LU decomposition.
  - Nonsingular solving and inverse.
  - Reduced row echelon form.
  - Rank.
  - Trace and determinant.
  - Characteristic polynomial.
  - Computation of eigenvalues with multiplicities.

- New ca_poly module for polynomials
  - Mostly using naive basecase algorithms.
  - Polynomial arithmetic, basic manipulation.
  - Polynomial division.
  - Evaluation and composition.
  - Derivative and integral.
  - GCD (Euclidean algorithm).
  - Squarefree factorization.
  - Computation of roots with multiplicities.
  - Construction from given roots.

- New ca_vec module for vectors.
  - Memory management and basic scalar operations.

- Bug fixes
- Fix bug in powering number field elements.
- Fix bug in qqbar_log_pi_i.
- Fix aliasing bug in ca_pow.

**New basic functions**
- Conversion from double: ca_set_d, ca_set_d_d.
- Special functions: ca_erf, ca_erfi, ca_erfc, with algebraic relations.
- Special functions: ca_gamma (incomplete simplification algorithms).

**New utils_flint module for Flint utilities**
- Vectors of multivariate polynomials.
- Construction of elementary symmetric polynomials.
- GrÃ¶bner basis computation (naive Buchberger algorithm).

**Documentation and presentation**
- Various improvements to the documentation.
- DFT example program.

**2020-09-08 – Calcium 0.1**
- Initial test release.
- ca module (exact real and complex numbers).
- fmpz_mpoly_q module (multivariate rational functions over Q).
- qqbar module (algebraic numbers represented by minimal polynomials).
- Example programs.

**16.1.4 Arb version history**

**2022-06-29 – Arb 2.23.0**
- Performance
  - Multithreaded numerical integration.
  - Multithreaded binary splitting computation of mathematical constants.
  - Multithreaded computation of Bernoulli numbers.
  - Multithreaded computation of Euler numbers.
  - Multithreaded refinement of Riemann zeta zeros.
  - Multithreaded complex_plot example program.
  - Multithreaded elementary functions.
  - Multithreaded computation of Hilbert class polynomials.
  - Improved multithreaded partition function.
  - Use FLINT’s FFT multiplication instead of GMP in appropriate ranges.
  - New, faster algorithm for elementary functions between roughly 10^3 and 10^6 digits.
  - Faster computation of log using Newton-like iteration instead of using MPFR.
  - Faster computation of atan using Newton-like iteration instead of the bit-burst algorithm.
Fix performance bug in atan() leading to quadratic running time with large arguments in high precision.
- Optimized high-precision complex squaring.
- Added internal function arb_flint_get_num_available_threads() to improve tuning for multithreaded algorithms.
- Fixed performance bug making erf() slower at high precision with multiple threads.

- Features
  - Implemented the Lerch transcendent (acb_dirichlet_lerch_phi()).
  - Fpwrap wrapper for Lerch transcendent (contributed by Valentin Boettcher).
  - Added a rudimentary module for Gaussian integers (fmpzi.h).
  - Added zeta_zeros example program (contributed by D.H.J. Polymath).
  - Added functions for simultaneous high-precision computation of logarithms of primes and arctangents for primitive angles.
  - Added bernoulli, class_poly, functions_benchmark example programs for benchmarking use.
  - Multiplying a signed number by an infinity yields an infinity instead of [0 +/- inf] (contributed by Erik Postma).

- Miscellaneous
  - Deprecated doubles version of partition function.
  - Fix crash in erf on some systems including mips64el (reported by Julien Puydt).
  - Fixed MINGW64 build (contributed by Massoud Mazar).
  - Avoid deprecated FLINT function n_gcd_full.
  - Documentation fixes.

2022-01-25 – Arb 2.22.1
- Fixed bugs causing some hypergeometric functions hang or crash for some input on various non-x86 architectures.
- Fixed a minor bug in acb_hypgeom_m (NaN result sometimes only set the real part to NaN).

2022-01-15 – Arb 2.22.0
- Special functions
  - Use numerical integration in some cases to compute the hypergeometric functions 0F1, 1F1, U, 2F1, incomplete gamma and beta, modified Bessel, etc. with real parameters and argument, improving performance and accuracy when the parameters are large.
  - Much faster computation of Bernoulli numbers using hybrid numerical-modular algorithm (modular code adapted from bernmm by David Harvey).
  - Faster computation of Euler numbers using hybrid algorithm; added arb_fmpz_euler_number_ui.
  - Added inverse error function (arb_hypgeom_erfinv, arb_hypgeom_erfcinv).
  - New (faster, more accurate) implementations of real error functions (arb_hypgeom_erp, arb_hypgeom_erkc) and trigonometric integrals (arb_hypgeom_si, arb_hypgeom_ci).
  - Added acb_dirichlet_l_fmpq and acb_dirichlet_l_fmpq_afe: reduced-complexity evaluation of L-functions at rational points.
Added functions for computing primorials (arb_primal_ui, arb_primal_nth_ui).

- New, highly optimized internal code for real hypergeometric series (arb_hypgeom_sum_fmpq_arb, etc.; currently only used in some functions).
- Fix arb_fpwrap_double_hypgeom_2f1 which computed the wrong thing.

Core arithmetic and functions
- Faster implementation of arb_ui_pow_ui.
- Added arb_fma_si, arb_fma_fmpz.
- Added arf_equal_ui, arf_equal_d.
- Added arf_get_str.
- Use arb-based printing code instead of MPFR in arf_printd and mag_printd so that large exponents work.
- Fixed bug in arb_get_str that caused loss of precision when printing more than about $10^6$ digits.
- Allow negative exponents in mag_pow_fmpz.
- Added the double_interval module for fast machine-precision interval arithmetic (experimental, intended for internal use).

2021-10-20 – Arb 2.21.1

- Fixed 32-bit test failures for arb_hypgeom_gamma_fmpq.
- Added pow function to the fpwrap module.
- Added missing header file includes.
- Do not encode the library version in the SONAME on Android (contributed by Andreas Enge).

2021-09-25 – Arb 2.21.0

- Experimental new arb_fpwrap module: accurate floating-point wrappers of Arb mathematical functions (supersedes the external arbcmath.h).
- Fixed memory leak in arf_load_file (reported by Dave Platt).
- New and faster gamma function code.
- Most gamma function internals are now located in the arb_hypgeom and acb_hypgeom modules. The user-facing functions (arb_gamma, etc.) are still available under the old names for compatibility. The internal algorithms for rising factorials (binary splitting, etc.) have been moved without aliases.
- Added arb_fma, arb_fma_arf, arb_fma_ui (like addmul, but take a separate input and output).
- Slightly faster internal Bernoulli number generation for small n.
- Better enclosures for acb_barnes_g at negative reals.
- Added Graeffe transforms (arb_poly_graeffe_transform, acb_poly_graeffe_transform) (contributed by Matthias Gessinger).
- Fixed conflict with musl libc (reported by Gonzalo Tornaría).
- Added acb_add_error_arb (contributed by Albin Ahlbäck).
2021-07-25 – Arb 2.20.0

- Flint 2.8 support.
- Change `arb_get_str` with `ARB_STR_NO_RADIUS`: [+/- 1.20e-15] now prints as 0e-14.
- Uniformly distributed random number functions `arf_urandom`, `arb_urandom` (contributed by Albin Ahlbäck).
- Use quasilinear algorithm in `arb_gamma_fmpq` for all small fractions.
- Added derivative of Weierstrass elliptic function (`acb_elliptic_p_prime`) (contributed by Daniel Schultz).
- Added dot products with integer coefficients: `arb_dot_fmpz`, `arb_dot_siui`, `arb_dot_uiui`, `arb_dot_si`, `arb_dot_ui`, `acb_dot_fmpz`, `acb_dot_siui`, `acb_dot/uiui`, `acb_dot_si`, `acb_dot_ui`.
- Faster `arb_fmpz_poly_evaluate_arb` and `arb_fmpz_poly_evaluate_acb`.
- Explicitly guarantee that roots are isolated in `arb_fmpz_poly_complex_roots` (could previously theoretically fail when using the deflation hack).
- Use GNUInstallDirs in `CMakeLists.txt` to support standard GNU installation directories (contributed by Michael Orlitzky).
- Fixed bug for aliased multiplication of window matrices (contributed by David Berghaus).
- Documentation fixes (contributed by Joel Dahne, Hanno Rein).

2020-12-06 – Arb 2.19.0

- Significant improvements to the implementation of Platt’s algorithm for computing Riemann zeta function zeros at large height (contributed by p15-git-acc).
- Better criterion for selecting asymptotic expansion of incomplete gamma function (contributed by p15-git-acc).
- Multithreaded `acb_dft` for power-of-two lengths (contributed by p15-git-acc).
- Added `acb_csc_pi`, `arb_csc_pi` (contributed by p15-git-acc).
- Fixed segfault in `acb_mat_eig_simple_rump` when called with L non-NULL and R NULL (contributed by p15-git-acc).
- Fixed bug in `acb_real_abs` (contributed by Joel Dahne).
- Changed several functions to more consistently return infinities instead of NaNs where reasonable (contributed by p15-git-acc).
- Added Fransen-Robinson as an integral example (contributed by p15-git-acc).
- Cleaned up makefile (contributed by p15-git-acc).
- Fixed several typos and some omitted functions in the documentation (contributed by Joel-Dahne, p15-git-acc).
### 2020-06-25 – Arb 2.18.1

- Support MinGW64.
- Added version numbers (__ARB_VERSION, __ARB_RELEASE, ARB_VERSION) to arb.h.

### 2020-06-09 – Arb 2.18.0

- **General**
  - Flint 2.6 support.
  - Several build system improvements (contributed by Isuru Fernando).
  - Changed arf_get_mpf to return an MPFR underflow/overflow result (rounding to 0 or infinity with the right sign and MPFR overflow flags) instead of throwing flint_abort() if the exponent is out of bounds for MPFR.
  - Documentation and type corrections (contributed by Joel Dahne).
- **Arithmetic**
  - The number of iterations per precision level in arb_fmpz_poly_complex_roots has been tweaked to avoid extreme slowdown for some polynomials with closely clustered roots.
  - Added arb_contains_interior, acb_contains_interior.
- **Special functions**
  - Fixed unsafe shifts causing Dirichlet characters for certain moduli exceeding 32 bits to crash.
  - Added acb_agm for computing the arithmetic-geometric mean of two complex numbers.
  - acb_elliptic_rj now uses a slow fallback algorithm in cases where Carlson’s algorithm is not known to be valid. This fixes instances where acb_elliptic_pi, acb_elliptic_pi_inc and acb_elliptic_rj previously ended up on the wrong branch. Users should be cautioned that the new version can give worse enclosures and sometimes fails to converge in some cases where the old algorithm did (the pi flag for acb_elliptic_pi_inc is useful as a workaround).
  - Optimized some special cases in acb_hurwitz_zeta.

### 2019-10-16 – Arb 2.17.0

- **General**
  - Removed many obsolete fmpz methods and de-inlined several helper functions to slightly improve compile time and library size.
  - Fixed a namespace clash for an internal function (contributed by Julian Rüth).
  - Added the helper function arb_sgn_nonzero.
  - Added the helper function acb_rel_one_accuracy_bits.
- **Riemann zeta function**
  - Added a function for efficiently computing individual zeros of the Riemann zeta function using Turing’s method (acb_dirichlet_zeta_zero) (contributed by D.H.J. Polymath).
  - Added a function for counting zeros of the Riemann zeta function up to given height using Turing’s method (acb_dirichlet_zeta_nzeros) (contributed by D.H.J. Polymath).
  - Added the Backlund S function (acb_dirichlet_backlund_s).
- Added a function for computing Gram points (acb_dirichlet_gram_point).
- Added acb_dirichlet_zeta_deriv_bound for quickly bounding the derivative of the Riemann zeta function.

Other special functions
- Improved the algorithm in acb_hypgeom_u to estimate precision loss more accurately.
- Implemented Coulomb wave functions (acb_hypgeom_coulomb, acb_hypgeom_coulomb_series and other functions).
- Faster algorithm for Catalan’s constant.
- Added acb_modular_theta_series.
- Improved tuning in acb_hypgeom_pfq_series_sum for higher derivatives at high precision (reported by Mark Watkins).

2018-12-07 – Arb 2.16.0

Linear algebra and arithmetic
- Added acb_mat_approx_eig_qr for approximate computation of eigenvalues and eigenvectors of complex matrices.
- Added acb_mat_eig_enclosure_rump implementing Rump’s algorithm for certification of eigenvalue-eigenvector pairs as well as clusters.
- Added acb_mat_eig_simple_rump for certified diagonalization of matrices with simple eigenvalues.
- Added acb_mat_eig_simple_vdhoeven_mourrain, acb_mat_eig_simple for fast certified diagonalization of matrices with simple eigenvalues.
- Added acb_mat_eig_multiple_rump, acb_mat_eig_multiple for certified computation of eigenvalues with possible overlap.
- Added acb_mat_eig_global_enclosure for fast global inclusion of eigenvalues without isolation.
- Added arb_mat_companion, acb_mat_companion for constructing companion matrices.
- Added several arb_mat and acb_mat helper functions: indeterminate, is_exact, is_zero, is_finite, is_triu, is_tril, is_diag, diag_prod.
- Added arb_mat_approx_inv, acb_mat_approx_inv.
- Optimized arb_mat_mul_block by using arb_dot when the blocks are small.
- Added acb_get_mid.
- Updated hilbert_matrix example program.
2018-10-25 – Arb 2.15.1

- Fixed precision issue leading to spurious NaN results in incomplete elliptic integrals

2018-09-18 – Arb 2.15.0

- Arithmetic
  - Added arb_dot and acb_dot for efficient evaluation of dot products.
  - Added arb_approx_dot and acb_approx_dot for efficient evaluation of dot products without error bounds.
  - Converted loops to arb_dot and acb_dot in the arb_poly and acb_poly methods mul_low_classical, inv_series, div_series, exp_series_basecase, sin_cos_series_basecase, sinh_cosh_series_basecase, evaluate_rectangular, evaluate2_rectangular, revert_series_lagrange_fast. Also changed the algorithm cutoffs for mullow, exp_series, sin_cos_series, sinh_cosh_series.
  - Converted loops to arb_dot and acb_dot in the arb_mat and acb_mat methods mul_classical, mul_threaded, solve_tril, solve_triu, charpoly. Also changed the algorithm cutoffs for mul, solve_tril, solve_triu.
  - Converted loops to arb_approx_dot and acb_approx_dot in the arb_mat and acb_mat methods approx_solve_tril, approx_solve_triu. Also changed the algorithm cutoffs.
  - Added arb_mat_approx_mul and acb_mat_approx_mul for matrix multiplication without error bounds.

- Miscellaneous
  - Added arb_hypgeom_airy_zero for computing zeros of Airy functions.
  - Added arb_hypgeom_dilog wrapper.
  - Optimized arb_const_pi and arb_const_log2 by using a static table at low precision, giving a small speedup and avoiding common recomputation when starting threads.
  - Optimized mag_set_ui_2exp_si.
  - Remove obsolete and unused function _arb_vec_dot.
  - Converted some inline functions to ordinary functions to reduce library size.
  - Fixed arb_dirichlet_stieltjes to use the integration algorithm also when a != 1.
  - Fixed test failure for acb_dirichlet_stieltjes on ARM64 (reported by Gianfranco Costamagna). Special thanks to Julien Puydt for assistance with debugging.
  - Fixed crash in acb_dft_bluestein with zero length (reported by Gianfranco Costamagna).

2018-07-22 – Arb 2.14.0

- Linear algebra
  - Faster and more accurate real matrix multiplication using block decomposition, scaling, and multiplying via FLINT integer matrices in combination with safe use of doubles for radius matrix multiplications.
  - Faster and more accurate complex matrix multiplication by reordering and taking advantage of real matrix multiplication.
  - The new multiplication algorithm methods (arb_mat_mul_block, acb_mat_mul_reorder) are used automatically by the main multiplication methods.
- Faster and more accurate LU factorization by using a block recursive algorithm that takes advantage of matrix multiplication. Added separate algorithm methods \( \text{arb/acb}_\text{mat}_\text{lu} \) (recursive/classical) with an automatic algorithm choice in the default \text{lu} methods.

- Added methods \( \text{arb/acb}_\text{mat}_\text{solve} \) (tril/triu) (and variants) for solving upper or lower triangular systems using a block recursive algorithm taking advantage of matrix multiplication.

- Improved linear solving and inverse for large well-conditioned matrices by using a preconditioning algorithm. Added separate solving algorithm methods \( \text{arb/acb}_\text{mat}_\text{solve} \) (lu/precond) with an automatic algorithm choice in the default solve methods (contributed by anonymous user arbguest).

- Improved determinants using a preconditioning algorithm. Added separate determinant algorithm methods \( \text{arb/acb}_\text{mat}_\text{det} \) (lu/precond) with an automatic algorithm choice in the default \text{det} methods.

- Added automatic detection of triangular matrices in \text{arb}_\text{mat}_\text{det} and \text{acb}_\text{mat}_\text{det}.

- Added \text{arb}_\text{mat}_\text{solve}_\text{preapprox} which allows certifying a precomputed approximate solution (contributed by anonymous user arbguest).

- Added methods for constructing various useful test matrices: \text{arb}_\text{mat}_\text{ones}, \text{arb}_\text{mat}_\text{hilbert}, \text{arb}_\text{mat}_\text{pascal}, \text{arb}_\text{mat}_\text{stirling}, \text{arb}_\text{mat}_\text{det}, \text{acb}_\text{mat}_\text{ones}, \text{acb}_\text{mat}_\text{det}.

- Added support for window matrices \( \text{arb/acb}_\text{mat}_\text{window}_\text{init}/\text{clear} \).

- Changed random test matrix generation \( \text{arb/acb}_\text{mat}_\text{randtest} \) to produce sparse matrices with higher probability.

- Added \text{acb}_\text{mat}_\text{conjugate} and \text{acb}_\text{mat}_\text{conjugate}_\text{transpose}.

- Arithmetic and elementary functions

  - Improved \text{arb}_\text{sin}_\text{cos}, \text{arb}_\text{sin} and \text{arb}_\text{cos} to produce more accurate enclosures for wide input intervals. The working precision is also reduced automatically based on the accuracy of the input to improve efficiency.

  - Improved \text{arb}_\text{sinh}_\text{cosh}, \text{arb}_\text{sinh} and \text{arb}_\text{cosh} to produce more accurate enclosures for wide input intervals. The working precision is also reduced automatically based on the accuracy of the input to improve efficiency.

  - Improved \text{arb}_\text{exp}_\text{invexp} and \text{arb}_\text{expml} to produce more accurate enclosures for wide input intervals. The working precision is also reduced automatically based on the accuracy of the input to improve efficiency.

  - Improved \text{acb}_\text{rsqrt} to produce more accurate enclosures for wide intervals.

  - Made \text{mag}_{\text{add}/\text{ui}}_{\text{lower}} public.

  - Added \text{mag}_\text{sinh}, \text{mag}_\text{cosh}, \text{mag}_\text{sinh}_\text{lower}, \text{mag}_\text{cosh}_\text{lower}.

  - Fixed minor precision loss near -1 in \text{arb}_\text{log}_\text{hypot} and \text{acb}_\text{log}.

  - Return imaginary numbers with exact zero real part when possible in \text{acb}_\text{acos} and \text{acb}_\text{acosh} (contributed by Ralf Stephan).

  - Improved special cases in \text{arb}_\text{set}_\text{interval}_\text{arf} (reported by Marc Mezzarobba).

- Special functions

  - Added a function for computing isolated generalized Stieltjes constants \( \text{acb\_dirichlet\_stieltjes} \).

  - Added scaled versions of Bessel functions \( \text{acb\_hypgeom\_bessel\_i\_scaled}, \text{acb\_hypgeom\_bessel\_k\_scaled} \).

  - The interface for the internal methods computing Bessel functions \( \text{i\_asym}, \text{k\_asym}, \text{etc.} \) has been changed to accommodate computing scaled versions.
- Added Riemann xi function (acb_dirichlet_xi) (contributed by D.H.J Polymath).
- Fixed infinite error bounds in the Riemann zeta function when evaluating at a ball containing zero centered in the left plane (contributed by D.H.J Polymath).
- Fixed precision loss in Airy functions with huge input and high precision.
- Legendre functions of the first kind (legendre_p): handle inexact integer a+b-c in 2F1 better (contributed by Joel Dahne).

- Example programs and documentation
  - Added more color functions to complex_plot.c.
  - Added more example integrals suggested by Nicolas Brisebarre and Bruno Salvy to integrals.c
  - Changed Sphinx style and redesigned the documentation front page.
  - Miscellaneous documentation cleanups.
  - Added documentation page about contributing.

- Other
  - Fixed a crash on some systems when calling acb_dft methods with a length of zero.
  - Fixed issue with setting rpath in configure (contributed by Vincent Delecroix).

2018-02-23 – Arb 2.13.0

- Major bugs
  - Fixed rounding direction in arb_get_abs_lbound_arf() which in some cases could result in an invalid lower bound being returned, and added forgotten test code for this and related functions (reported by deinst). Although this bug could lead to incorrect results, it probably had limited impact in practice (explaining why it was not caught indirectly by other test code) since a single rounding in the wrong direction in this operation generally will be dwarfed by multiple roundings in the correct direction in surrounding operations.

- Important notes about bounds
  - Many functions have been modified to compute tighter enclosures when the input balls are wide. In most cases the bounds should be improved, but there may be some regressions. Bug reports about any significant regressions are welcome.
  - Division by zero in arb_div() has been changed to return [NaN +/- inf] instead of [+/- inf]. This change might be reverted in the future if it proves to be too inconvenient. In either case, users should only assume that division by zero produces something non-finite, and user code that depends on division by zero to produce [0 +/- inf] should be modified to handle zero-containing denominators as a separate case.

- Improvements to arithmetic and elementary functions
  - Faster implementation of acb_get_mag_lower().
  - Optimized arb_get_mag_lower(), arb_get_mag_lower_nonnegative().
  - Added arb_set_interval_mag() and arb_set_interval_neg_pos_mag() for constructing an arb_t from a pair of mag_t endpoints.
  - Added mag_const_pi_lower(), mag_atan(), mag_atan_lower().
  - Added mag_div_lower(), mag_inv(), mag_inv_lower().
  - Added mag_sqrt_lower() and mag_rsqrt_lower().
  - Added mag_log(), mag_log_lower(), mag_neg_log(), mag_neg_log_lower().
  - Added mag_exp_lower(), mag_expinv_lower() and tweaked mag_exp().
- Added `mag_pow_fmpz_lower()`, `mag_get_fmpz()`, `mag_get_fmpz_lower()`.
- Improved `arb_exp()` for wide input.
- Improved `arb_log()` for wide input.
- Improved `arb_sqrt()` for wide input.
- Improved `arb_rsqrt()` for wide input.
- Improved `arb_div()` for wide input.
- Improved `arb_atan()` for wide input and slightly optimized `arb_atan2()` for input spanning multiple signs.
- Improved `acb_rsqrt()` for wide input and improved stability of this function generally in the left half plane.
- Added `arb_log_hypot()` and improved `acb_log()` for wide input.
- Slightly optimized trigonometric functions (`acb_sin()`, `acb_sin_pi()`, `acb_cos()`, `acb_cos_pi()`, `acb_sin_cos()`, `acb_sin_cos_pi()`) for pure real or imaginary input.

**Special functions**

- Slightly improved bounds for gamma function (`arb_gamma()`, `acb_gamma()`, `arb_rgamma()`, `acb_rgamma()`) for wide input.
- Improved bounds for Airy functions for wide input.
- Simplifications to code for computing Gauss period minimal polynomials (contributed by Jean-Pierre Flori).
- Optimized `arb_hypgeom_legendre_p_ui()` further by avoiding divisions in the basecase recurrence and computing the prefactor more quickly in the asymptotic series (contributed by Marc Mezzarobba).
- Small further optimization of `arb_hypgeom_legendre_p_ui_root()` (contributed by Marc Mezzarobba).
- Improved derivative bounds for Legendre polynomials (contributed by Marc Mezzarobba).

**Numerical integration**

- Increased default quadrature `deg_limit` at low precision to improve performance for integration of functions without singularities near the path.
- Added several more integrals to examples/integrals.c
- Added utility functions `acb_real_abs()`, `acb_real_sgn()`, `acb_real_heaviside()`, `acb_real_floor()`, `acb_real ceil()`, `acb_real_min()`, `acb_real_max()`, `acb_real_sqrt()`, useful for numerical integration.
- Added utility functions `acb_sqrt analytic()`, `acb_rsqrt analytic()`, `acb_log analytic()`, `acb_pow analytic()` with branch cut detection, useful for numerical integration.

**Build system and compatibility issues**

- Removed `-Wl` flag from Makefile.subdirs to fix “-r and -pie may not be used together” compilation error on some newer Linux distributions (reported by many users).
- Fixed broken test code for `l_vec_hurwitz` which resulted in spurious failures on 32-bit systems (originally reported by Thierry Monteil on Sage trac).
- Avoid using deprecated MPFR function `mpfr_root()` with MPFR versions >= 4.0.0.
- Remark: the recently released MPFR 4.0.0 has a bug in `mpfr_div()` leading to test failures in Arb (though not affecting correctness of Arb itself). Users should make sure to install the patched version MPFR 4.0.1.
- Added missing C++ include guards in `arb_fmpz_poly.h` and `dlog.h` (reported by Marc Mezzarobba).
– Fixed Travis builds on Mac OS again (contributed by Isuru Fernando).
– Added missing declaration for arb_bell_ui() (reported by numsys).

2017-11-29 – Arb 2.12.0

• Numerical integration
  – Added a new function (acb_calc_integrate) for rigorous numerical integration using adaptive subdivision and Gauss-Legendre quadrature. This largely obsoletes the old integration code using Taylor series.
  – Added new integrals.c example program (old example program moved to integrals_taylor.c).

• Discrete Fourier transforms
  – Added acb_dft module with various FFT algorithm implementations, including top level O(n log n) acb_dft and acb_dft_inverse functions (contributed by Pascal Molin).

• Legendre polynomials
  – Added arb_hypgeom_legendre_p_ui for fast and accurate evaluation of Legendre polynomials. This is also used automatically by the Legendre functions, where it is substantially faster and gives better error bounds than the generic algorithm.
  – Added arb_hypgeom_legendre_p_ui_root for fast computation of Legendre polynomial roots and Gauss-Legendre quadrature nodes (used internally by the new integration code).
  – Added arb_hypgeom_central_bin_ui for fast computation of central binomial coefficients (used internally for Legendre polynomials).

• Dirichlet L-functions and zeta functions
  – Fixed a bug in the Riemann zeta function involving a too small error bound in the implementation of the Riemann-Siegel formula for inexact input. This bug could result in a too small enclosure when evaluating the Riemann zeta function at an argument of large imaginary height without also computing derivatives, if the input interval was very wide.
  – Add acb_dirichlet_zeta_jet; also made computation of the first derivative of Riemann zeta function use the Riemann-Siegel formula where appropriate.
  – Added acb_dirichlet_l_vec_hurwitz for fast simultaneous evaluation of Dirichlet L-functions for multiple characters using Hurwitz zeta function and FFT (contributed by Pascal Molin).
  – Simplified interface for using hurwitz_precomp functions.
  – Added lcentral.c example program (contributed by Pascal Molin).
  – Improved error bounds when evaluating Dirichlet L-functions using Euler product.

• Elementary functions
  – Faster custom implementation of sin, cos at 4600 bits and above instead of using MPFR (30-40% asymptotic improvement, up to a factor two speedup).
  – Faster code for exp between 4600 and 19000 bits.
  – Improved error bounds for acb_atan by using derivative.
  – Improved error bounds for arb_sinh_cosh, arb_sinh and arb_cosh when the input has a small midpoint and large radius.
  – Added reciprocal trigonometric and hyperbolic functions (arb_sec, arb_csc, arb_sech, arb_csch, acb_sec, acb_csc, acb_sech, acb_csch).
  – Changed the interface of _acb_vec_unit_roots to take an extra length parameter (compatibility-breaking change).
- Improved `arb_pow` and `acb_pow` with an inexact base and a negative integer or negative half-integer exponent; the inverse is now computed before performing binary exponentiation in this case to avoid spurious blow-up.

**Elliptic functions**
- Improved Jacobi theta functions to reduce the argument modulo the lattice parameter, greatly improving speed and numerical stability for large input.
- Optimized `arb_agm` by using a final series expansion and using special code for wide intervals.
- Optimized `acb_agm1` by using a final series expansion and using special code for positive real input.
- Optimized derivative of AGM for high precision by using a central difference instead of a forward difference.
- Optimized `acb_elliptic_rf` and `acb_elliptic_rj` for high precision by using a variable length series expansion.

**Other**
- Fixed incorrect handling of subnormals in `arf_set_d`.
- Added `mag_bin_uiui` for bounding binomial coefficients.
- Added `mag_set_d_lower`, `mag_sqrt_lower`, `mag_set_d_2exp_fmpz_lower`.
- Implemented multithreaded complex matrix multiplication.
- Optimized `arb_rel_accuracy_bits` by adding fast path.
- Fixed a spurious floating-point exception (division by zero) in the `t-gauss_period_minpoly` test program triggered by new code optimizations in recent versions of GCC that are unsafe together with FLINT inline assembly functions (a workaround was added to the test code, and a proper fix for the assembly code has been added to FLINT).

2017-07-10 – Arb 2.11.1

- Avoid use of a function that was unavailable in the latest public FLINT release

2017-07-09 – Arb 2.11.0

**Special functions**
- Added the Lambert W function (`arb_lambertw`, `acb_lambertw`, `arb_poly_lambertw_series`, `acb_poly_lambertw_series`). All complex branches and evaluation of derivatives are supported.
- Added the `acb_expm1` method, complementing `arb_expm1`.
- Added `arb_sinc_pi`, `acb_sinc_pi`.
- Optimized handling of more special cases in the Hurwitz zeta function.

**Polynomials**
- Added the `arb_fmpz_poly` module to provide Arb methods for FLINT integer polynomials.
- Added methods for evaluating an `fmpz_poly` at `arb_t` and `acb_t` arguments.
- Added `arb_fmpz_poly_complex_roots` for computing the real and complex roots of an integer polynomial, turning the functionality previously available in the `poly_roots.c` example program into a proper library function.
- Added a method (`arb_fmpz_poly_gauss_period_minpoly`) for constructing minimal polynomials of Gaussian periods.
- Added \texttt{arb\_poly\_product\_roots\_complex} for constructing a real polynomial from complex conjugate roots.

- **Miscellaneous**
  - Fixed test code in the dirichlet module for 32-bit systems (contributed by Pascal Molin).
  - Use \texttt{flint\_abort()} instead of \texttt{abort()} (contributed by Tommy Hofmann).
  - Fixed the static library install path (contributed by François Bissey).
  - Made \texttt{arb\_nonnegative\_part()} a publicly documented method.
  - Arb now requires FLINT version 2.5 or later.

**2017-02-27 – Arb 2.10.0**

- **General**
  - Changed a large number of methods from inline functions to normal functions, substantially reducing the size of the built library.
  - Fixed a few minor memory leaks (missing clear() calls).

- **Basic arithmetic**
  - Added \texttt{arb\_is\_int\_2exp\_si} and \texttt{acb\_is\_int\_2exp\_si}.
  - Added \texttt{arf\_sosq} for computing $x^2+y^2$ of floating-point numbers.
  - Improved error bounds for complex square roots in the left half plane.
  - Improved error bounds for complex reciprocal (\texttt{acb\_inv}) and division.
  - Added the internal helper \texttt{mag\_get\_d\_log2\_approx} as a public method.

- **Elliptic functions and integrals**
  - New module \texttt{acb\_elliptic.h} for elliptic functions and integrals.
  - Added complete elliptic integral of the third kind.
  - Added Legendre incomplete elliptic integrals (first, second, third kinds).
  - Added Carlson symmetric incomplete elliptic integrals (RF, RC, RG, RJ, RD).
  - Added Weierstrass elliptic zeta and sigma functions.
  - Added inverse Weierstrass elliptic p-function.
  - Added utility functions for computing the Weierstrass invariants and lattice roots.
  - Improved computation of derivatives of Jacobi theta functions by using modular transformations, and added a main evaluation function (\texttt{acb\_modular\_theta\_jet}).
  - Improved detection of pure real or pure imaginary parts in various cases of evaluating theta and modular functions.

- **Other special functions**
  - New, far more efficient implementation of the dilogarithm function (\texttt{acb\_polylog} with $s = 2$).
  - Fixed an issue in the Hurwitz zeta function leading to unreasonable slowdown for certain complex input.
  - Added \texttt{acb\_poly\_exp\_pi\_i\_series}.
  - Added \texttt{arb\_poly\_log1p\_series}, \texttt{acb\_poly\_log1p\_series}.
2016-12-02 – Arb 2.9.0

- **License**
  - Changed license from GPL to LGPL.

- **Build system and compatibility**
  - Fixed FLINT includes to use flint/foo.h instead of foo.h, simplifying compilation on many systems.
  - Added another alias for the dynamic library to fix make check on certain systems (contributed by Andreas Enge).
  - Travis CI support (contributed by Isuru Fernando).
  - Added support for ARB_TEST_MULTIPLIER environment variable to control the number of test iterations.
  - Support building with CMake (contributed by Isuru Fernando).
  - Support building with MSVC on Windows (contributed by Isuru Fernando).
  - Fixed unsafe use of FLINT_ABS for slong -> ulong conversion in arf.h, which caused failures on MIPS and ARM systems.

- **Basic arithmetic and methods**
  - Fixed mag_addmul(x,x,x) with x having a mantissa of all ones. This could produce a non-normalized mag_t value, potentially leading to incorrect results in arb and acb level arithmetic. This bug was caught by new test code, and fortunately would have been hard to trigger accidentally.
  - Added fastPath paths for error bound calculations in arb_sqrt and arb_div, speeding up these operations significantly at low precision
  - Added support for round-to-nearest in all arf methods.
  - Added fprint methods (contributed by Alex Griffing).
  - Added acb_printn and acb_fprintn methods to match arb_printn.
  - Added arb_equal_si and acb_equal_si.
  - Added arb_can_round_mpfr.
  - Added arb_get_ubound_arf, arb_get_lbound_arf (contributed by Tommy Hofmann).
  - Added sign function (arb_sgn).
  - Added complex sign functions (acb_sgn, acb_csgn).
  - Rewrote arb_contains_fmpq to make the test exact.
  - Optimized mag_get_fmpq.
  - Optimized arf_get_fmpz and added more robust test code.
  - Rewrote arb_get_unique_fmpz and arb_get_interval_fmpz_2exp, reducing overhead, making them more robust with huge exponents, and documenting their behavior more carefully.
  - Optimized arb_union.
  - Optimized arf_is_int, arf_is_int_2exp_si and changed these from inline to normal functions.
  - Added mag_const_pi, mag_sub, mag_exppinv.
  - Optimized binary-to-decimal conversion for huge exponents by using exponential function instead of binary powering.
  - Added arb_intersection (contributed by Alex Griffing).
  - Added arb_min, arb_max (contributed by Alex Griffing).
Fixed a bug in arb_log and in test code on 64-bit Windows due to unsafe use of MPFR which only uses 32-bit exponents on Win64.

Improved some test functions to reduce the chance of reporting spurious failures.

Added squaring functions (arb_sqr, acb_sqr) (contributed by Ricky Farr).

Added arf_frexp.

Added arf_cmp_si, arf_cmp_ui, arf_cmp_d.

Added methods to count allocated bytes (arb_allocated_bytes, _arb_vec_allocated_bytes, etc.).

Added methods to predict memory usage for large vectors (_arb/_acb_vec_estimate_allocated_bytes).

Changed clear() methods from inline to normal functions, giving 8% faster compilation and 25% smaller libarb.so.

Added acb_unit_root and _acb_vec_unit_roots (contributed by Pascal Molin).

• Polynomials

  Added sinh and cosh functions of power series (arb/acb_poly_sinh/cosh_series and sinh_cosh_series).

  Use basecase series inversion algorithm to improve speed and error bounds in arb/acb_poly_inv_series.

  Added functions for fast polynomial Taylor shift (arb_poly_taylor_shift, acb_poly_taylor_shift and variants).

  Fast handling of special cases in polynomial composition.

  Added acb_poly scalar mul and div convenience methods (contributed by Alex Griffing).

  Added set_trunc, set_trunc_round convenience methods.

  Added add_series, sub_series methods for truncating addition.

  Added polynomial is_zero, is_one, is_x, valuation convenience methods.

  Added hack to arb_poly_mullow and acb_poly_mullow to avoid overhead when doing an in-place multiplication with length at most 2.

  Added binomial and Borel transform methods for acb_poly.

• Matrices

  Added Cholesky decomposition plus solving and inverse for positive definite matrices (arb_mat_cho, arb_mat_spd_solve, arb_mat_spd_inv and related methods) (contributed by Alex Griffing).

  Added LDL decomposition and inverse and solving based on LDL decomposition for real matrices (arb_mat_ldl, arb_mat_solve_ldl_precomp, arb_mat_inv_ldl_precomp) (contributed by Alex Griffing).

  Improved the entrywise error bounds in matrix exponential computation to preserve sparsity and give exact entries where possible in many cases (contributed by Alex Griffing).

  Added public functions for computing the truncated matrix exponential Taylor series (arb_mat_exp_taylor_sum, acb_mat_exp_taylor_sum).

  Added functions related to sparsity structure (arb_mat_entrywise_is_zero, arb_mat_count_is_zero, etc.) (contributed by Alex Griffing).

  Entrywise multiplication (arb_mat_nul_entrywise, acb_mat_nul_entrywise) (contributed by Alex Griffing).

  Added is_empty and is_square convenience methods (contributed by Alex Griffing).
– Added the bool_mat helper module for matrices over the boolean semiring (contributed by Alex Griffing).
– Added Frobenius norm computation (contributed by Alex Griffing).

- Miscellaneous special functions
  – Added evaluation of Bernoulli polynomials (arb_bernoulli_poly_ui, acb_bernoulli_poly_ui).
  – Added convenience function for evaluation of huge Bernoulli numbers (arb_bernoulli_fmpz).
  – Added Euler numbers (arb_euler_number_ui, arb_euler_number_fmpz).
  – Added fast approximate partition function (arb_partitions_fmpz/ui).
  – Optimized partition function for n < 1000 by using recurrence for the low 64 bits.
  – Improved the worst-case error bound in arb_atan.
  – Added arb_log_base_ui.
  – Added complex sinc function (acb_sinc).
  – Special handling of z = 1 when computing polylogarithms.
  – Fixed agm(-1,-1) to output 0 instead of indeterminate.
  – Made working precision in arb_gamma and acb_gamma more sensitive to the input accuracy.

- Hypergeometric functions
  – Compute erf and erfc without cancellation problems for large or complex z.
  – Avoid re-computing the square root of pi in several places.
  – Added generalized hypergeometric function (acb_hypgeom_pfq).
  – Implement binary splitting and rectangular splitting for evaluation of hypergeometric series with a power series parameter, greatly speeding up Y_n, K_n and other functions at high precision, as well as speeding up high-order parameter derivatives.
  – Use binary splitting more aggressively in acb_hypgeom_pfq_sum to reduce error bound inflation.
  – Asymptotic expansions of hypergeometric functions: more accurate parameter selection, and better handling of terminating cases.
  – Tweaked algorithm selection and working precision in acb_hypgeom_m.
  – Avoid dividing by the denominator of the next term in acb_hypgeom_sum, which would lead to a division by zero when evaluating hypergeometric polynomials.
  – Fixed a bug in hypergeometric series evaluation resulting in near-integers not being skipped in some cases, leading to unnecessary loss of precision.
  – Added series expansions of Airy functions (acb_hypgeom_airy_series, acb_hypgeom_airy_jet).
  – Fixed a case where Airy functions accidentally chose the worst algorithm instead of the best one.
  – Added functions for computing erf, erfc, erfi of power series in the acb_hypgeom module.
  – Added series expansion of the logarithmic integral (acb_hypgeom_li_series).
  – Added Fresnel integrals (acb_hypgeom_fresnel, acb_hypgeom_fresnel_series).
  – Added the lower incomplete gamma function (acb_hypgeom_gamma_lower) (contributed by Alex Griffing).
  – Added series expansion of the lower incomplete gamma function (acb_hypgeom_gamma_lower_series) (contributed by Alex Griffing).
- Added support for computing the regularized incomplete gamma functions.
- Use slightly sharper error bound for analytic continuation of 2F1.
- Added support for computing finite limits of 2F1 with inexact parameters differing by integers.
- Added the incomplete beta function \((\text{acb\_hypgeom\_beta\_lower}, \text{acb\_hypgeom\_beta\_lower\_series})\)
- Improved \text{acb\_hypgeom\_u} to use a division-avoiding algorithm for small polynomial cases.
- Added \text{arb\_hypgeom} module, wrapping the complex hypergeometric functions for more convenient use with the \text{arb\_t} type.

- **Dirichlet L-functions and Riemann zeta function**
  - New module \text{dirichlet} for working algebraically with Dirichlet groups and characters (contributed by Pascal Molin).
  - New module \text{acb\_dirichlet} for numerical evaluation of Dirichlet characters and L-functions (contributed by Pascal Molin).
  - Efficient representation and manipulation of Dirichlet characters using the Conrey representation (contributed by Pascal Molin).
  - New module \text{dlog} for word-size discrete logarithm evaluation, used to support algorithms on Dirichlet characters (contributed by Pascal Molin).
  - Methods for properties, evaluation, iteration, pairing, lift, lowering etc. of Dirichlet characters (contributed by Pascal Molin).
  - Added \text{acb\_dirichlet\_roots} methods for fast evaluation of many roots of unity (contributed by Pascal Molin).
  - Added \text{acb\_dirichlet\_hurwitz\_precomp} methods for fast multi-evaluation of the Hurwitz zeta function for many parameter values.
  - Added methods for computing Gauss, Jacobi and theta sums over Dirichlet characters (contributed by Pascal Molin).
  - Added methods (\text{acb\_dirichlet\_l}, \text{acb\_dirichlet\_l\_jet}, \text{acb\_dirichlet\_l\_series}) for evaluation of Dirichlet L-functions and their derivatives.
  - Implemented multiple algorithms for evaluation of Dirichlet L-functions depending on the argument (Hurwitz zeta function decomposition, Euler product, functional equation).
  - Added methods (\text{acb\_dirichlet\_hardy\_z}, \text{acb\_dirichlet\_hardy\_z\_series}, etc.) for computing the Hardy Z-function corresponding to a Dirichlet L-function.
  - Added fast bound for Hurwitz zeta function (\text{mag\_hurwitz\_zeta\_uiui}).
  - Improved parameter selection in Hurwitz zeta function to target relative instead of absolute error for large positive \(s\).
  - Improved parameter selection in Hurwitz zeta function to avoid computing unnecessary Bernoulli numbers for large imaginary \(s\).
  - Added \text{acb\_dirichlet\_eta} function (\text{acb\_dirichlet\_eta}).
  - Implemented the Riemann-Siegel formula for faster evaluation of the Riemann zeta function at large height.
  - Added smooth-index algorithm for the main sum when evaluating the Riemann zeta function, avoiding the high memory usage of the full sieving algorithm when the number of terms gets huge.
  - Improved tuning for using the Euler product when computing the Riemann zeta function.

- **Example programs**
  - Added logistic map example program.
- Added lvalue example program.
- Improved poly_roots in several ways: identify roots that are exactly real, automatically perform squarefree factorization, use power hack, and allow specifying a product of polynomials as input on the command line.

**Housekeeping**

- New section in the documentation giving an introduction to ball arithmetic and using the library.
- Tidied, documented and added test code for the fmpz_extras module.
- Added proper documentation and test code for many helper methods.
- Removed the obsolete fmprb module entirely.
- Documented more algorithms and formulas.
- Clarified integer overflow issues and use of ARF_PREC_EXACT in the documentation.
- Added .gitignore file.
- Miscellaneous improvements to the documentation.

**2015-12-31 – Arb 2.8.1**

- Fixed 32-bit test failure for the Laguerre function.
- Made the Laguerre function indeterminate at negative integer orders, to be consistent with the test code.

**2015-12-29 – Arb 2.8.0**

- Compatibility and build system
  - Windows64 support (contributed by Bill Hart).
  - Fixed a bug that broke basic arithmetic on targets where FLINT uses fallback code instead of assembly code, such as PPC64 (contributed by Jeroen Demeyer).
  - Fixed configure to use EXTRA_SHARED_FLAGS/LDFLAGS, and other build system fixes (contributed by Tommy Hofmann, Bill Hart).
  - Added soname versioning (contributed by Julien Puydt).
  - Fixed test code on MinGW (contributed by Hrvoje Abraham).
  - Miscellaneous fixes to simplify interfacing Arb from Julia.

- Arithmetic and elementary functions
  - Fixed arf_get_d to handle underflow/overflow correctly and to support round-to-nearest.
  - Added more complex inverse hyperbolic functions (acb_asin, acb_acos, acb_asinh, acb_acosh, acb_atanh).
  - Added arb_contains_int and acb_contains_int for testing whether an interval contains any integer.
  - Added acb_quadratic_roots_fmpz.
  - Improved arb_sinh to use a more accurate formula for x < 0.
  - Added sinc function (arb_sinc) (contributed by Alex Griffing).
  - Fixed bug in arb_exp affecting convergence for huge input.
  - Faster implementation of arb_div_2expm1_ui.
- Added `mag_root`, `mag_geom_series`.
- Improved and added test code for `arb_add_error` functions.
- Changed `arb_pow` and `acb_pow` to make `pow(0, positive) = 0` instead of `nan`.
- Improved `acb_sqrt` to return finite output for finite input straddling the branch cut.
- Improved `arb_set_interval_arf` so that `[inf, inf] = inf` instead of an infinite interval.
- Added computation of Bell numbers (`acb_bell_fmpz`).
- Added `arb_power_sum_vec` for computing power sums using Bernoulli numbers.
- Added computation of the Fujiwara root bound for `acb_poly`.
- Added code to identify all the real roots of a real polynomial (`acb_poly_validate_real_roots`).
- Added several convenient assignment functions, including `arb_set_d`, `acb_set_d`, `acb_set_d_d`, `acb_set_fmpz_fmpz` (contributed by Ricky Farr).
- Added many accessor functions (`_arb/acb_vec_entry_ptr`, `arb_get_mid/rad_arb`, `acb_real/imag_ptr`, `arb_mid/rad_ptr`, `acb_get_real/imag`).
- Added missing functions `acb_add_si`, `acb_sub_si`.
- Renamed `arb_root` to `arb_root_ui` (keeping alias) and added `acb_root_ui`.

- **Special functions**
  - Implemented the Gauss hypergeometric function 2F1 and its regularized version.
  - Fixed two bugs in `acb_hypgeom_pfq_series_direct` discovered while implementing 2F1. In rare cases, these could lead to incorrect values for functions depending on parameter derivatives of hypergeometric series.
    - The first bug involved incorrect handling of negative integer parameters. The bug only affected 2F1 and higher functions; it did not affect correctness of any previously implemented functions that relied on `acb_hypgeom_pfq_series_direct` (such as Bessel Y and K functions of integer order).
    - The second bug involved a too small bound being computed for the sum of a geometric series. The geometric series bound is nearly tight for 2F1, and the incorrect version caused immediate test failures for that function. Theoretically, this bug affected correctness of some previously-implemented functions that relied on `acb_hypgeom_pfq_series_direct` (such as Bessel Y and K functions of integer order), but since the geometric bound is not as tight in those cases, those functions were still reliable in practice (no failing test case has been found).
  - Implemented Airy functions and their derivatives (`acb_hypgeom_airy`).
  - Implemented the confluent hypergeometric function 0F1 (`acb_hypgeom_0f1`).
  - Implemented associated Legendre functions P and Q.
  - Implemented Chebyshev, Jacobi, Gegenbauer, Laguerre, Hermite functions.
  - Implemented spherical harmonics.
  - Added function for computing Bessel J and Y functions simultaneously.
  - Added rising factorials for non-integer n (`arb_rising`, `acb_rising`).
  - Made rising factorials use gamma function for large integer n.
  - Faster algorithm for theta constants and Dedekind eta function at very high precision.
  - Fixed erf to give finite values instead of +/-inf for big imaginary input.
  - Improved `acb_zeta` (and `arb_zeta`) to automatically use fast code for integer zeta values.
  - Added double factorial (`arb_doublefac_ui`).

16.1. History and changes
- Added code for generating Hilbert class polynomials (acb_modular_hilbert_class_poly).

- Matrices
  - Added faster matrix squaring (arb/acb_mat_sqr) (contributed by Alex Griffing).
  - Added matrix trace (arb/acb_mat_trace) (contributed by Alex Griffing).
  - Added arb/acb_mat_set_round_fmpz_mat, acb_mat_set(_round)_arb_mat (contributed by Tommy Hofmann).
  - Added arb/acb_mat_transpose (contributed by Tommy Hofmann).
  - Added comparison methods arb/acb_mat_eq/ne (contributed by Tommy Hofmann).

- Other
  - Added complex_plot example program.
  - Added Airy functions to real_roots example program.
  - Other minor patches were contributed by Alexander Kobel, Marc Mezzarobba, Julien Puydt.
  - Removed obsolete file config.h.

---

2015-07-14 – Arb 2.7.0

- Hypergeometric functions
  - Implemented Bessel I and Y functions (acb_hypgeom_bessel_i, acb_hypgeom_bessel_y).
  - Fixed bug in Bessel K function giving the wrong branch for negative real arguments.
  - Added code for evaluating complex hypergeometric series binary splitting.
  - Added code for evaluating complex hypergeometric series using fast multipoint evaluation.

- Gamma related functions
  - Implemented the Barnes G-function and its continuous logarithm (acb_barnes_g, acb_log_barnes_g).
  - Implemented the generalized polygamma function (acb_polygamma).
  - Implemented the reflection formula for the logarithmic gamma function (acb_lgamma, acb_poly_lgamma_series).
  - Implemented the digamma function of power series (arb_poly_digamma_series, acb_poly_digamma_series).
  - Improved acb_poly_zeta_series to produce exact zero imaginary parts in most cases when the result should be real-valued.
  - Made the real logarithmic gamma function (arb_lgamma, arb_poly_lgamma_series) abort more quickly for negative input.

- Elementary functions
  - Added arb_exp_expinv and acb_exp_expinv functions for simultaneously computing exp(x), exp(-x).
  - Improved acb_tan, acb_tan_pi, acb_cot and acb_cot_pi for input with large imaginary parts.
  - Added complex hyperbolic functions (acb_sinh, acb_cosh, acb_sinh_cosh, acb_tanh, acb_coth).
  - Added acb_log_sin_pi for computing the logarithmic sine function without branch cuts away from the real line.
  - Added arb_poly_cot_pi_series, acb_poly_cot_pi_series.
- Added arf_root and improved speed of arb_root.
- Tuned algorithm selection in arb_pow_fmpz.

- Other
  - Added documentation for arb and acb vector functions.

**2015-04-19 – Arb 2.6.0**

- Special functions
  - Added the Bessel K function.
  - Added the confluent hypergeometric functions M and U.
  - Added exponential, trigonometric and logarithmic integrals ei, si, shi, ci, chi, li.
  - Added the complete elliptic integral of the second kind E.
  - Added support for computing hypergeometric functions with power series as parameters.
  - Fixed special cases in Bessel J function returning useless output.
  - Fixed precision of zeta function accidentally being capped at 7000 digits (bug in 2.5).
  - Special-cased real input in the gamma functions for complex types.
  - Fixed exp of huge numbers outputting unnecessarily useless intervals.
  - Fixed broken code in erf that sometimes gave useless output.
  - Made selection of number of terms in hypergeometric series more robust.

- Polynomials and power series.
  - Added sin_pi, cos_pi and sin_cos_pi for real and complex power series.
  - Speeded up series reciprocal and division for length = 2.
  - Added add_si methods for polynomials.
  - Made inv_series and div_series with zero input produce indeterminates instead of aborting.
  - Added arb_poly_majorant, acb_poly_majorant.

- Basic functions
  - Added comparison methods arb_eq, arb_ne, arb_lt, arb_le, arb_gt, arb_ge, acb_eq, acb_ne.
  - Added acb_rel_accuracy_bits and improved the real version.
  - Fixed precision of constants like pi behaving more nondeterministically than necessary.
  - Fixed arf_get_mag_lower(nan) to output 0 instead of inf.

- Other
  - Removed call to fmpz_dedekind_sum which only exists in the git version of flint.
  - Fixed a test code bug that could cause crashes on some systems.
  - Added fix for static build on OS X (thanks Marcello Seri).
  - Miscellaneous corrections to the documentation.
2015-01-28 – Arb 2.5.0

- **String conversion**
  - Added `arb_set_str`.
  - Added `arb_get_str` and `arb_printn` for pretty-printed rigorous decimal output.
  - Added helper functions for binary to decimal conversion.

- **Core arithmetic**
  - Improved speed of division when using GMP instead of MPIR.
  - Improved complex division with a small denominator.
  - Removed a little bit of overhead for complex squaring.

- **Special functions**
  - Faster code for `atan` at very high precision, used instead of `mpfr_atan`.
  - Optimized elementary functions slightly for small input.
  - Added modified error functions `erf` and `erfi`.
  - Added the generalized exponential integral.
  - Added the upper incomplete gamma function.
  - Implemented the complete elliptic integral of the first kind.
  - Implemented the arithmetic-geometric mean of complex numbers.
  - Optimized `arb_digamma` for small integers.
  - Made `mag_log_ui`, `mag_binpow_uiui` and `mag_polylog_tail` proper functions.
  - Added `pow`, `agm`, `erf`, `elliptic_k`, `elliptic_p` as functions of complex power series.
  - Added incomplete gamma function of complex power series.
  - Improved code for bounding complex rising factorials (the old code could potentially have given wrong results in degenerate cases).
  - Added `arb_sqrt1pm1`, `arb_atanh`, `arb_asinh`, `arb_atan`.
  - Added `arb_log1p`, `acb_log1p`, `acb_atan`.
  - Added `arb_hurwitz_zeta`.
  - Improved parameter selection in the Hurwitz zeta function to try to avoid stalling when given enormous input.
  - Optimized `sqrt` and `rsqrt` of power series when given a binomial as input.
  - Made `arb_bernoulli_ui(2^64-2)` not crash.
  - Fixed `rgamma` of negative integers returning indeterminate.

- **Polynomials and matrices**
  - Added characteristic polynomial computation for real and complex matrices.
  - Added polynomial `set_round` methods.
  - Added `is_real` methods for more types.
  - Added `get_unique_fmpz` methods.
  - Added code for generating Swinnerton-Dyer polynomials.
  - Improved error bounding in `det()` and `exp()` of complex matrices to recognize when the result is real-valued.
  - Changed polynomial `divrem` to return success/fail instead of aborting on divide by zero.
• Miscellaneous
  – Added logo to documentation.
  – Made inlined functions build as part of the library.
  – Silenced a clang warning.
  – Made _acb_vec_sort_pretty a library function.

2014-11-15 – Arb 2.4.0

• Arithmetic and core functions
  – Made evaluation of sin, cos and exp at medium precision faster using the sqrt trick.
  – Optimized arb_sinh and arb_sinh_cosh.
  – Optimized complex division with a small denominator.
  – Optimized cubing of complex numbers.
  – Added floor and ceil functions for the arf and arb types.
  – Added acb_poly powering functions.
  – Added acb_exp_pi_i.
  – Added functions for evaluation of Chebyshev polynomials.
  – Fixed arb_div to output nan for input containing nan.

• Added a module acb_hypgeom for hypergeometric functions
  – Evaluation of the generalized hypergeometric function in convergent cases.
  – Evaluation of confluent hypergeometric functions using asymptotic expansions.
  – The Bessel function of the first kind for complex input.
  – The error function for complex input.

• Added a module acb_modular for modular forms and elliptic functions
  – Support for working with modular transformations.
  – Mapping a point to the fundamental domain.
  – Evaluation of Jacobi theta functions and their series expansions.
  – The Dedekind eta function.
  – The j-invariant and the modular lambda and delta function.
  – Eisenstein series.

• Miscellaneous
  – Fixed mag_print printing a too large exponent.
  – Fixed printd methods to use a fallback instead of aborting when printing numbers too large for MPFR.
  – Added version number string (arb_version).
  – Various additions to the documentation.
2014-09-25 – Arb 2.3.0

- Removed most of the legacy (Arb 1.x) modules.
- Updated build scripts, hopefully fixing various issues.
- New implementations of `arb_sin`, `arb_cos`, `arb_sin_cos`, `arb.atan`, `arb_log`, `arb_exp`, `arb.expm1`, much faster up to a few thousand bits.
- Ported the bit-burst code for high-precision exponentials to the `arb` type.
- Speeded up `arb_log.ui.from.prev`.
- Added `mag_exp`, `mag.expm1`, `mag.exp_tail`, `mag.pow.fmpz`.
- Improved various `mag` functions.
- Added `arb.get/set.interval.mpfr`, `arb.get_interval.arf`, and improved `arb.set_interval.arf`.
- Improved `arf.get.fmpz`.
- Prettier printing of complex numbers with negative imaginary part.
- Changed some frequently-used functions from inline to non-inline to reduce code size.

2014-08-01 – Arb 2.2.0

- Added functions for computing polylogarithms and order expansions of polylogarithms, with support for real and complex `s`, `z`.
- Added a missing cast affecting C++ compatibility.
- Generalized `powsum` functions to allow a geometric factor.
- Improved `powsum` functions slightly when the exponent is an integer.
- Faster `arb_log.ui.from.prev`.
- Added `mag.sqrt` and `mag.rsqrt` functions.
- Fixed various minor bugs and added missing tests and documentation entries.

2014-06-20 – Arb 2.1.0

- Ported most of the remaining functions to the new `arb/ach` types, including:
  - Elementary functions (`log`, `atan`, etc.).
  - Hypergeometric series summation.
  - The gamma function.
  - The Riemann zeta function and related functions.
  - Bernoulli numbers.
  - The partition function.
  - The calculus modules (rigorous real root isolation, rigorous numerical integration of complex-valued functions).
  - Example programs.
- Added several missing utility functions to the `arf` and `mag` modules.
2014-05-27 – Arb 2.0.0

- New modules mag, arf, arb, arb_poly, arb_mat, acb, acb_poly, acb_mat for higher-performance ball arithmetic.
- Poly_roots2 and hilbert_matrix2 example programs.
- Vector dot product and norm functions (contributed by Abhinav Baid).

2014-05-03 – Arb 1.1.0

- Faster and more accurate error bounds for polynomial multiplication (error bounds are now always as good as with classical multiplication, and multiplying high-degree polynomials with approximately equal coefficients now has proper quasilinear complexity).
- Faster and much less memory-hungry exponentials at very high precision.
- Improved the partition function to support n bigger than a single word, and enabled the possibility to use two threads for the computation.
- Fixed a bug in floating-point arithmetic that caused a too small bound for the rounding error to be reported when the result of an inexact operation was rounded up to a power of two (this bug did not affect the correctness of ball arithmetic, because operations on ball midpoints always round down).
- Minor optimizations to floating-point arithmetic.
- Improved argument reduction of the digamma function and short series expansions of the rising factorial.
- Removed the holonomic module for now, as it did not really do anything very useful.

2013-12-21 – Arb 1.0.0

- New example programs directory
  - poly_roots example program.
  - real_roots example program.
  - pi_digits example program.
  - hilbert_matrix example program.
  - keiper_li example program.
- New fmprb_calc module for calculus with real functions
  - Bisection-based root isolation.
  - Asymptotically fast Newton root refinement.
- New fmpcb_calc module for calculus with complex functions
  - Numerical integration using Taylor series.
- Scalar functions
  - Simplified fmprb_const_euler using published error bound.
  - Added fmprb_inv.
  - Added fmprb_trim, fmpcb_trim.
  - Added fmpcb_rsqrt (complex reciprocal square root).
  - Fixed bug in fmprb_sqrtpos with nonfinite input.
  - Slightly improved fmprb powering code.
- Added various functions for bounding fmprs by powers of two.
- Added fmpr_is_int.

- Polynomials and power series
  - Implemented scaling to speed up blockwise multiplication.
  - Slightly faster basecase power series exponentials.
  - Improved sin/cos/tan/exp for short power series.
  - Added complex sqrt_series, rsqrt_series.
  - Implemented the Riemann-Siegel Z and theta functions for real power series.
  - Added fmprb_poly_pow_series, fmprb_poly_pow_ui and related methods.
  - Added fmprb/fmpcb_poly_contains_fmpz_poly.
  - Faster composition by monomials.
  - Implemented Borel transform and binomial transform for real power series.

- Matrices
  - Implemented matrix exponentials.
  - Multithreaded fmprb_mat_mul.
  - Added matrix infinity norm functions.
  - Added some more matrix-scalar functions.
  - Added matrix contains and overlaps methods.

- Zeta function evaluation
  - Multithreaded power sum evaluation.
  - Faster parameter selection when computing many derivatives.
  - Implemented binary splitting to speed up computing many derivatives.

- Miscellaneous
  - Corrections for C++ compatibility (contributed by Jonathan Bober).
  - Several minor bugfixes and test code enhancements.

2013-08-07 – Arb 0.7

- Floating-point and ball functions
  - Documented, added test code, and fixed bugs for various operations involving a ball containing
    an infinity or NaN.
  - Added reciprocal square root functions (fmpr_rsqrt, fmprb_rsqrt) based on mpfr_rec_sqrt.
  - Faster high-precision division by not computing an explicit remainder.
  - Slightly faster computation of pi by using new reciprocal square root and division code.
  - Added an fmpr function for approximate division to speed up certain radius operations.
  - Added fmpr_set_d for conversion from double.
  - Allow use of doubles to optionally compute the partition function faster but without an error
    bound.
  - Bypass mpfr overflow when computing the exponential function to extremely high precision
    (approximately 1 billion digits).
- Made fmprb_exp faster for large numbers at extremely high precision by skipping the log(2) removal.
- Made fnmpcb_lgamma faster at high precision by speeding up the argument reduction branch computation.
- Added fmprb_asin, fmprb_acos.
- Added various other utility functions to the fmprb module.
- Added a function for computing the Glaisher constant.
- Optimized evaluation of the Riemann zeta function at high precision.

- Polynomials and power series
  - Made squaring of polynomials faster than generic multiplication.
  - Implemented power series reversion (various algorithms) for the fnmpcb_poly type.
  - Added many fnmpcb_poly utility functions (shifting, truncating, setting/getting coefficients, etc.).
  - Improved power series division when either operand is short
  - Improved power series logarithm when the input is short.
  - Improved power series exponential to use the basecase algorithm for short input regardless of the output size.
  - Added power series square root and reciprocal square root.
  - Added atan, tan, sin, cos, sin_cos, asin, acos fnmpcb_poly power series functions.
  - Added Newton iteration macros to simplify various functions.
  - Added gamma functions of real and complex power series ([fnmpcb/fmpcb_poly_[gamma/rgamma/lgamma]_series]).
  - Added wrappers for computing the Hurwitz zeta function of a power series ([fnmpcb/fmpcb_poly_zeta_series]).
  - Implemented sieving and other optimizations to improve performance for evaluating the zeta function of a short power series.
  - Improved power series composition when the inner series is linear.
  - Added many fnmpcb_poly versions of nearly all fnmprb_poly functions.
  - Improved speed and stability of series composition/reversion by balancing the power table exponents.

- Other
  - Added support for freeing all cached data by calling flint_cleanup().
  - Introduced fnmprb_ptr, fnmprb_srcptr, fnmpcb_ptr, fnmpcb_srcptr typedefs for cleaner function signatures.
  - Various bug fixes and general cleanup.
**2013-05-31 – Arb 0.6**

- Made fast polynomial multiplication over the reals numerically stable by using a blockwise algorithm.
- Disabled default use of the Gauss formula for multiplication of complex polynomials, to improve numerical stability.
- Added division and remainder for complex polynomials.
- Added fast multipoint evaluation and interpolation for complex polynomials.
- Added missing `fmprb_poly_sub` and `fmpcb_poly_sub` functions.
- Faster exponentials (`fmprb_exp` and dependent functions) at low precision, using precomputation.
- Rewrote `fmpr_add` and `fmpr_sub` using mpn level code, improving efficiency at low precision.
- Ported the partition function implementation from flint (using ball arithmetic in all steps of the calculation to guarantee correctness).
- Ported algorithm for computing the cosine minimal polynomial from flint (using ball arithmetic to guarantee correctness).
- Support using GMP instead of MPIR.
- Only use thread-local storage when enabled in flint.
- Slightly faster error bounding for the zeta function.
- Added some other helper functions.

**2013-03-28 – Arb 0.5**

- Arithmetic and elementary functions
  - Added `fmpr_get_fmpz`, `fmpr_get_si`.
  - Fixed accuracy problem with `fmprb_div_2expm1`.
  - Special-cased squaring of complex numbers.
  - Added various `fmpcb` convenience functions (addmul_ui, etc).
  - Optimized `fmpr_cmp_2exp_si` and `fmpr_cmpabs_2exp_si`, and added test code for comparison functions.
  - Added `fmprb_atan2`, also fixing a bug in `fmpcb_arg`.
  - Added `fmprb_sin_pi`, `cos_pi`, `sin_cos_pi`, etc.
  - Added `fmprb_sin_pi_fmpq` (etc.) using algebraic methods for fast evaluation of roots of unity.
  - Faster `fmprb_poly_evaluate` and `evaluate_fmpcb` using rectangular splitting.
  - Added `fmprb_poly_evaluate2`, `evaluate2_fmpcb` for simultaneously evaluating the derivative.
  - Added `fmprb_poly_root` polishing code using near-optimal Newton steps (experimental).
  - Added `fmpr_root`, `fmprb_root` (currently based on MPFR).
  - Added `fmpr_min`, `fmpr_max`.
  - Added `fmpr_set_interval_fmpr`, `fmprb_union`.
  - Added `fmpr_bits`, `fmprb_bits`, `fmpcb_bits` for obtaining the mantissa width.
  - Added `fmprb_hypot`.
  - Added complex square roots.
- Improved fmprb_log to slightly improve speed, and properly support huge arguments.
- Fixed exp, cosh, sinh to work with huge arguments.
- Added fmprb_expm1.
- Fixed sin, cos, atan to work with huge arguments.
- Improved fmprb_pow and fmpcb_pow, including automatic detection of small integer and half-integer exponents.
- Added many more elementary functions: fmprb_tan/cot/tanh/coth, fmpcb_tan/cot, and pi versions.
- Added fmprb_const_e, const_log2, const_log10, const_catalan.
- Fixed ball containment/overlap checking to work operate efficiently and correctly with huge exponents.
- Strengthened test code for many core operations.

- Special functions
  - Reorganized zeta function related code.
  - Faster evaluation of the Riemann zeta function via sieving.
  - Documented and improved efficiency of the zeta constant binary splitting code.
  - Calculate error bound in Borwein’s algorithm with fmprs instead of using doubles.
  - Optimized divisions in zeta evaluation via the Euler product.
  - Use functional equation for Riemann zeta function of a negative argument.
  - Compute single Bernoulli numbers using ball arithmetic instead of relying on the floating-point code in flint.
  - Initial code for evaluating the gamma function using its Taylor series.
  - Much faster rising factorials at high precision, using difference polynomials.
  - Much faster gamma function at high precision.
  - Added complex gamma function, log gamma function, and other versions.
  - Added fmprb_agm (real arithmetic-geometric mean).
  - Added fmprb_gamma_fmpq, supporting rapid computation of gamma(p/q) for q = 1,2,3,4,6.
  - Added real and complex digamma function.
  - Fixed unnecessary recomputation of Bernoulli numbers.
  - Optimized computation of Euler’s constant, and added proper error bounds.
  - Avoid reliance on doubles in the hypergeometric series tail bound.
  - Cleaned up factorials and binomials, computing factorials via gamma.

- Other
  - Added an fmpz_extras module to collect various internal fmpz helper functions.
  - Fixed detection of flint header files.
  - Fixed various other small bugs.
2013-01-26 – Arb 0.4

- Much faster fmpr_mul, fmprb_mul and set_round, resulting in general speed improvements.
- Code for computing the complex Hurwitz zeta function with derivatives.
- Fixed and documented error bounds for hypergeometric series.
- Better algorithm for series evaluation of the gamma function at a rational point.
- Much faster generation of Bernoulli numbers.
- Complex log, exp, pow, trigonometric functions (currently based on MPFR).
- Complex nth roots via Newton iteration.
- Added code for arithmetic on fmpcb_polys.
- Code for computing Khinchin’s constant.
- Code for rising factorials of polynomials or power series
  - Faster sin_cos.
  - Better div_2expm1.
- Many other new helper functions.
- Improved thread safety.
- More test code for core operations.

2012-11-07 – Arb 0.3

- Converted documentation to Sphinx.
- New module fmpcb for ball interval arithmetic over the complex numbers
  - Conversions, utility functions and arithmetic operations.
- New module fmpcb_mat for matrices over the complex numbers
  - Conversions, utility functions and arithmetic operations.
  - Multiplication, LU decomposition, solving, inverse and determinant.
- New module fmpcb_poly for polynomials over the complex numbers
  - Root isolation for complex polynomials.
- New module fmpz_holonomic for functions/sequences defined by linear differential/difference equations with polynomial coefficients
  - Functions for creating various special sequences and functions.
  - Some closure properties for sequences.
  - Taylor series expansion for differential equations.
  - Computing the nth entry of a sequence using binary splitting.
  - Computing the nth entry mod p using fast multipoint evaluation.
- Generic binary splitting code with automatic error bounding is now used for evaluating hypergeometric series.
- Matrix powering.
- Various other helper functions.
2012-09-29 – Arb 0.2

- Code for computing the gamma function (Karatsuba, Stirling's series).
- Rising factorials.
- Fast exp_series using Newton iteration.
- Improved multiplication of small polynomials by using classical multiplication.
- Implemented error propagation for square roots.
- Polynomial division (Newton-based).
- Polynomial evaluation (Horner) and composition (divide-and-conquer).
- Product trees, fast multipoint evaluation and interpolation (various algorithms).
- Power series composition (Horner, Brent-Kung).
- Added the fmprb_mat module for matrices of balls of real numbers.
- Matrix multiplication.
- Interval-aware LU decomposition, solving, inverse and determinant.
- Many helper functions and small bugfixes.

2012-09-14 – Arb 0.1

- 2012-08-05 - Began simplified rewrite.
- 2012-04-05 - Experimental ball and polynomial code (first commit).


Bibliography 1107


Bibliography


Bibliography 1111


INDEX

Symbols

acb_dirichlet_definite_hardy_z (C function), 713
acb_dirichlet_euler_product_real_ui (C function), 710
acb_dirichlet_exact_zeta_nzeros (C function), 713
acb_dirichlet_hardy_theta_series (C function), 712
acb_dirichlet_hardy_z_series (C function), 712
acb_dirichlet_isolate_gram_hardy_z_zero (C function), 713
acb_dirichlet_isolate_rosser_hardy_z_zero (C function), 713
acb_dirichlet_isolate_turing_hardy_z_zero (C function), 713
acb_dirichlet_l_series (C function), 711
acb_dirichlet_platt_local_hardy_z_zeros (C function), 714
acb_dirichlet_refine_hardy_z_zero (C function), 713
acb_elliptic_k_series (C function), 665
acb_elliptic_p_series (C function), 668
acb_hypgeom_airy_series (C function), 647
acb_hypgeom_beta_lower_series (C function), 649
acb_hypgeom_chi_series (C function), 650
acb_hypgeom_ci_series (C function), 650
acb_hypgeom_coulomb_series (C function), 647
acb_hypgeom_ei_series (C function), 649
acb_hypgeom_erf_series (C function), 643
acb_hypgeom_erfc_series (C function), 644
acb_hypgeom_erfi_series (C function), 644
acb_hypgeom_fresnel_series (C function), 644
acb_hypgeom_gamma_lower_series (C function), 648
acb_hypgeom_gamma_upper_series (C function), 648
acb_hypgeom_li_series (C function), 651
acb_hypgeom_shi_series (C function), 650
acb_mat_charpoly (C function), 635
acb_mat_companion (C function), 635
acb_mat_diag_prod (C function), 635
acb_mat_vector_mul_col (C function), 632
acb_mat_vector_mul_row (C function), 632
acb_modular_theta_series (C function), 674
acb_poly_add (C function), 597
acb_poly_agm1_series (C function), 608
acb_poly_atan_series (C function), 604
acb_poly_binomial_transform (C function), 602
acb_poly_binomial_transform_basecase (C function), 602
acb_poly_binomial_transform_convolution (C function), 602
acb_poly_borel_transform (C function), 602
acb_poly_compose (C function), 599
acb_poly_compose_series (C function), 599
acb_poly_cos_pi_series (C function), 605
acb_poly_cos_series (C function), 605
acb_poly_cosh_series (C function), 606
acb_poly_cot_pi_series (C function), 605
acb_poly_derivative (C function), 602
acb_poly_digamma_series (C function), 606
acb_poly_div (C function), 598
acb_poly_div_root (C function), 599
acb_poly_div_series (C function), 598
acb_poly_divrem (C function), 599
acb_poly_elliptic_k_series (C function), 609
acb_poly_elliptic_p_series (C function), 609
acb_poly_erf_series (C function), 608
acb_poly_evaluate (C function), 600
acb_poly_evaluate2 (C function), 600
acb_poly_evaluate2_horner (C function), 600
acb_poly_evaluate2_rectangular (C function), 600
acb_poly_evaluate_horner (C function), 600
acb_poly_evaluate_rectangular (C function), 600
acb_poly_evaluation (C function), 601
acb_poly_evaluation2_rectangular (C function), 601
acb_poly_evaluation_vec_fast (C function), 601
acb_poly_evaluation_vec_fast_precomp (C function), 601
acb_poly_evaluation_vec_iter (C function), 601
acb_poly_exp_pi_i_series (C function), 604
acb_poly_exp_series (C function), 604
acb_poly_exp_series_basecase (C function), 604
acb_poly_find_roots (C function), 609
acb_poly_gamma_series (C function), 606
acb_poly_graeffe_transform (C function), 602

1115
_acb_poly_integral (C function), 602
_acb_poly_interpolate_barycentric (C function), 601
_acb_poly_interpolate_fast (C function), 601
_acb_poly_interpolate_fast_precomp (C function), 601
_acb_poly_interpolate_newton (C function), 601
_acb_poly_interpolation_weights (C function), 601
_acb_poly_inv_borel_transform (C function), 602
_acb_poly_inv_series (C function), 598
_acb_poly_lambertw_series (C function), 606
_acb_poly_igamma_series (C function), 606
_acb_poly_loglp_series (C function), 604
_acb_poly_log_series (C function), 604
_acb_poly_majorant (C function), 597
_acb_poly_mul (C function), 598
_acb_poly_mullow (C function), 597
_acb_poly_mullow_classical (C function), 597
_acb_poly_mullowTranspose (C function), 597
_acb_poly_mullowTranspose_gauss (C function), 597
_acb_poly_normalise (C function), 594
_acb_poly_nth_derivative (C function), 602
_acb_poly_overlaps (C function), 596
_acb_poly_polylog_cpx (C function), 608
_acb_poly_polylog_cpx_small (C function), 608
_acb_poly_polylog_cpx_zeta (C function), 608
_acb_poly_polylog_series (C function), 608
_acb_poly_pow_acb_series (C function), 603
_acb_poly_pow_series (C function), 603
_acb_poly_pow_ui (C function), 603
_acb_poly_pow_ui_trunc_binexp (C function), 603
_acb_poly_powsum_one_series_sieved (C function), 607
_acb_poly_powsum_series_naive (C function), 607
_acb_poly_powsum_series_naive_threaded (C function), 607
_acb_poly_product_roots (C function), 600
_acb_poly_refine_roots_durand_kerner (C function), 609
_acb_poly_rem (C function), 598
_acb_poly_revert_series (C function), 599
_acb_poly_riemann_zeta_series (C function), 606
_acb_poly_root_bound_fujiwara (C function), 609
_acb_poly_root_inclusion (C function), 609
_acb_poly_sqrt_series (C function), 604
_acb_poly_set_length (C function), 594
_acb_poly_shift_left (C function), 595
_acb_poly_shift_right (C function), 595
_acb_poly_sin_cos_pi_series (C function), 605
_acb_poly_sin_cos_series (C function), 604
_acb_poly_sin_pi_series (C function), 605
_acb_poly_sin_series (C function), 605
_acb_poly_sinc_series (C function), 606
_acb_poly_sinh_cosh_series (C function), 605
_acb_poly_sinh_cosh_series_basecase (C function), 605
_acb_poly_sinh_cosh_series_exponential (C function), 605
_acb_poly_sinh_series (C function), 605
_acb_poly_sqr_series (C function), 603
_acb_poly_sub (C function), 597
_acb_poly_tan_series (C function), 605
_acb_poly_taylor_shift (C function), 599
_acb_poly_tree_alloc (C function), 600
_acb_poly_tree_build (C function), 600
_acb_poly_tree_free (C function), 600
_acb_poly_validate_real_roots (C function), 610
_acb_poly_validate_roots (C function), 609
_acb_poly_zeta_cpx_series (C function), 607
_acb_poly_zeta_em_bound (C function), 607
_acb_poly_zeta_em_bound1 (C function), 607
_acb_poly_zeta_em_choose_param (C function), 607
_acb_poly_zeta_em_sum (C function), 607
_acb_poly_zeta_em_tail_bsplit (C function), 607
_acb_poly_zeta_em_tail_naive (C function), 607
_acb_poly_zeta_series (C function), 608
_acb_vec_add (C function), 577
_acb_vec_add_error Arf_vec (C function), 577
_acb_vec_add_error Mag_vec (C function), 577
_acb_vec_allocated_bytes (C function), 562
_acb_vec_bits (C function), 577
_acb_vec_clear (C function), 562
_acb_vec_contains (C function), 576
_acb_vec_equal (C function), 576
_acb_vec_estimate_allocated_bytes (C function), 562
_acb_vec_get_imag (C function), 576
_acb_vec_get_real (C function), 576
_acb_vec_get_unique_fmpz_vec (C function), 562
_acb_vec_get_unique_fmpz_vec (C function), 577
_acb_vec_get_unique_mpz_vec (C function), 577
_acb_vec_get_unique_mpq_vec (C function), 577
_acb_vec_get_unique_mpqz_vec (C function), 577
_acb_vec_get_unique_mpqz_vec (C function), 577
_acb_vec_in_determinate (C function), 577
_acb_vec_init (C function), 562
_acb_vec_is_finite (C function), 576
_acb_vec_is_real (C function), 576
_acb_vec_is_zero (C function), 576
_acb_vec_neg (C function), 577
_acb_vec_overlaps (C function), 576
_acb_vec_print (C function), 578
_acb_vec_printn (C function), 578
_acb_vec_scalar_addmul (C function), 577
_acb_vec_scalar_div (C function), 577
_acb_vec_scalar_div_arb (C function), 577
_acb_vec_scalar_div_fmpz (C function), 577
_acb_vec_scalar_div_ui (C function), 577
Index

_d_vec_norm (C function), 1013
_d_vec_randtest (C function), 1012
_d_vec_set (C function), 1012
_d_vec_sub (C function), 1012
_d_vec_zero (C function), 1012
dirichlet_char_exp (C function), 460
fexpr_vec_clear (C function), 810
fexpr_vec_init (C function), 810
fexpr_vec_sort_fast (C function), 815
fft_mulmod_2exp (C function), 265
fmpq_add (C function), 276
fmpq_add_fmpz (C function), 276
fmpq_add_si (C function), 276
fmpq_add_small (C function), 277
fmpq_add_ui (C function), 276
fmpq_adddmul (C function), 276
fmpq_canonicalise (C function), 272
fmpq_div (C function), 276
fmpq_fprint (C function), 275
fmpq_gcd (C function), 277
fmpq_gcd_fcofactors (C function), 277
fmpq_get_str (C function), 274
fmpq_harmonic_ui (C function), 280
fmpq_is_canonical (C function), 272
fmpq_mat_charpoly (C function), 289
fmpq_mat_mimpoly (C function), 290
fmpq_mod_fmpz (C function), 278
fmpq_mul (C function), 276
fmpq_mul_fmpz (C function), 276
fmpq_mul_small (C function), 277
fmpq_mul_ui (C function), 276
fmpq_next_calkin_wilf (C function), 278
fmpq_next_minimal (C function), 278
fmpq_next_signed_calkin_wilf (C function), 279
fmpq_next_signed_minimal (C function), 278
fmpq_poly_add (C function), 295
fmpq_poly_add_can (C function), 295
fmpq_poly_add_series (C function), 295
fmpq_poly_add_series_can (C function), 296
fmpq_poly_asin_series (C function), 305
fmpq_poly_asinh_series (C function), 305
fmpq_poly_atan_series (C function), 304
fmpq_poly_atanh_series (C function), 305
fmpq_poly_canonicalize (C function), 301
fmpq_poly_cmp (C function), 301
fmpq_poly_compose (C function), 308
fmpq_poly_compose_series (C function), 309
fmpq_poly_compose_series_brent_kung (C function), 309
fmpq_poly_compose_series_horner (C function), 308
fmpq_poly_content (C function), 311
fmpq_poly_cos_series (C function), 305
fmpq_poly_cosh_series (C function), 306
fmpq_poly_derivative (C function), 303
fmpq_poly_div (C function), 299
fmpq_poly_div_series (C function), 301
fmpq_poly_divides (C function), 301
fmpq_poly_divrem (C function), 299
fmpq_poly_equal_trunc (C function), 295
fmpq_poly_eval_fmpz (C function), 307
fmpq_poly_eval_fmpq (C function), 307
fmpq_poly_exp_N_series (C function), 304
fmpq_poly_exp_series (C function), 304
fmpq_poly_fprint (C function), 312
fmpq_poly_fprint_pretty (C function), 312
fmpq_poly_gcd (C function), 302
fmpq_poly_gegenbauer_c (C function), 307
fmpq_poly_integral (C function), 303
fmpq_poly_interpolate_fmpz_vec (C function), 308
fmpq_poly_inv_series (C function), 301
fmpq_poly_inv_series_newton (C function), 301
fmpq_poly_isqrt (C function), 303
fmpq_poly_is_monic (C function), 291
fmpq_poly_laguerre_l (C function), 307
fmpq_poly_lcm (C function), 302
fmpq_poly_legendre_p (C function), 307
fmpq_poly_log_series (C function), 304
fmpq_poly_make_monic (C function), 311
fmpq_poly_mul (C function), 298
fmpq_poly_mullow (C function), 298
fmpq_poly_normalise (C function), 291
fmpq_poly_nth_derivative (C function), 303
fmpq_poly_pow (C function), 299
fmpq_poly_pow_trunc (C function), 299
fmpq_poly_power_sums (C function), 304
fmpq_poly_power_sums_to_poly (C function), 304
fmpq_poly_powers_clear (C function), 300
fmpq_poly_powers_precompute (C function), 300
fmpq_poly_primitive_part (C function), 311
fmpq_poly_print (C function), 311
fmpq_poly_print_pretty (C function), 311
fmpq_poly_revert_series (C function), 300
fmpq_poly_revert_series_precomp (C function), 300
fmpq_poly_rescale (C function), 308
fmpq_poly_resultant (C function), 302
fmpq_poly_revert_series (C function), 310
fmpq_poly_revert_series_lagrange (C function), 310
fmpq_poly_revert_series_lagrange_fast (C function), 310
fmpq_poly_revert_series_newton (C function), 310
fmpq_poly_scalar_div_fmpz (C function), 298
fmpq_poly_scalar_div_fmpq (C function), 297
fmpq_poly_scalar_div_ui (C function), 298
fmpq_poly_scalar_mul_fmpz (C function), 297
fmpq_poly_scalar_mul_fmpq (C function), 297
_{fmpz_mod_poly_gcdinv_euclidean_f} (C function), 427
_{fmpz_mod_poly_gcdinv_hgcd} (C function), 427
_{fmpz_mod_poly_scalar_div_fmpz} (C function), 415
_{fmpz_mod_poly_scalar_mul_fmpz} (C function), 414
_{fmpz_mod_poly_set_length} (C function), 410
_{fmpz_mod_poly_shift_left} (C function), 413
_{fmpz_mod_poly_shift_right} (C function), 413
_{fmpz_mod_poly_sqrt} (C function), 415
_{fmpz_mod_poly_sqrt_basecase} (C function), 430
_{fmpz_mod_poly_sqrt_series} (C function), 430
_{fmpz_mod_poly_gcdinv} (C function), 423
_{fmpz_mod_poly inversion} (C function), 424
_{fmpz_mod_poly_gcd} (C function), 423
_{fmpz_mod_poly_hgcd} (C function), 423
_{fmpz_mod_poly_interval_poly_worker} (C function), 440
_{fmpz_mod_poly_inv_series} (C function), 422
_{fmpz_mod_poly_invmodule} (C function), 425
_{fmpz_mod_poly_invmodule_f} (C function), 425
_{fmpz_mod_poly_invseries} (C function), 430
_{fmpz_mod_poly_is_squarefree} (C function), 438
_{fmpz_mod_poly_is_squarefree_f} (C function), 439
_{fmpz_mod_poly_minpoly} (C function), 426
_{fmpz_mod_poly_minpoly_bm} (C function), 426
_{fmpz_mod_poly_minpoly_hgcd} (C function), 426
_{fmpz_mod_poly_mul} (C function), 415
_{fmpz_mod_poly_mulmod} (C function), 415
_{fmpz_mod_poly_mulmod_preinv} (C function), 416
_{fmpz_mod_poly_mulmod_preinv_naive} (C function), 418
_{fmpz_mod_poly_mulmod_preinv_threaded_pool} (C function), 417
_{fmpz_mod_poly_mulmod_preinv_worker} (C function), 417
_{fmpz_mod_poly_pow} (C function), 416
_{fmpz_mod_poly_pow_trunc} (C function), 417
_{fmpz_mod_poly_pow_trunc_binexp} (C function), 417
_{fmpz_mod_poly_powmod} (C function), 417
_{fmpz_mod_poly_powmod_preinv} (C function), 417
_{fmpz_mod_poly_powmod_preinv_worker} (C function), 417
_{fmpz_mod_poly_powmod_x} (C function), 417
_{fmpz_mod_poly_precompute_matrix} (C function), 431
_{fmpz_mod_poly_precompute_matrix_worker} (C function), 431
_{fmpz_mod_poly_product_roots_fmpz_vec} (C function), 416
_{fmpz_mod_poly_radix} (C function), 435
_{fmpz_mod_poly_radix_init} (C function), 434
_{fmpz_mod_poly_reduce} (C function), 1004
_{fmpz_mod_poly_reduce_matrix_mod_poly} (C function), 431
_{fmpz_mod_poly_rem} (C function), 421
_{fmpz_mod_poly_rem_basecase} (C function), 420
_{fmpz_mod_poly_resultant} (C function), 428
_{fmpz_mod_poly_resultant_euclidean} (C function), 427
_{fmpz_mod_poly_resultant_hgcd} (C function), 427
_{fmpz_mod_poly_scalar_div_fmpz} (C function), 415
_{fmpz_mod_poly_scalar_mul_fmpz} (C function), 414
_{fmpz_mod_poly_set_length} (C function), 410
_{fmpz_mod_poly_shift_left} (C function), 413
_{fmpz_mod_poly_shift_right} (C function), 413
_{fmpz_mod_poly_sqrt} (C function), 415
_{fmpz_mod_poly_sqrt_basecase} (C function), 430
_{fmpz_mod_poly_sqrt_series} (C function), 430
_{fmpz_mod_poly_gcdinv} (C function), 423
_{fmpz_mod_poly_gcd} (C function), 423
_{fmpz_mod_poly_hgcd} (C function), 423
_{fmpz_mod_poly_interval_poly_worker} (C function), 440
_{fmpz_mod_poly_inv_series} (C function), 422
_{fmpz_mod_poly_invmodule} (C function), 425
_{fmpz_mod_poly_invmodule_f} (C function), 425
_{fmpz_mod_poly_invseries} (C function), 430
_{fmpz_mod_poly_is_squarefree} (C function), 438
_{fmpz_mod_poly_is_squarefree_f} (C function), 439
_{fmpz_mod_poly_minpoly} (C function), 426
_{fmpz_mod_poly_minpoly_bm} (C function), 426
_{fmpz_mod_poly_minpoly_hgcd} (C function), 426
_{fmpz_mod_poly_mul} (C function), 415
_{fmpz_mod_poly_mulmod} (C function), 415
_{fmpz_mod_poly_mulmod_preinv} (C function), 416
_{fmpz_mod_poly_mulmod_preinv_naive} (C function), 418
_{fmpz_mod_poly_mulmod_preinv_threaded_pool} (C function), 417
_{fmpz_mod_poly_pow} (C function), 416
_{fmpz_mod_poly_pow_trunc} (C function), 417
_{fmpz_mod_poly_pow_trunc_binexp} (C function), 417
_{fmpz_mod_poly_powmod} (C function), 417
_{fmpz_mod_poly_powmod_preinv} (C function), 417
_{fmpz_mod_poly_powmod_preinv_worker} (C function), 417
_{fmpz_mod_poly_powmod_x} (C function), 417
_{fmpz_mod_poly_precompute_matrix} (C function), 431
_{fmpz_mod_poly_precompute_matrix_worker} (C function), 431
_{fmpz_mod_poly_product_roots_fmpz_vec} (C function), 416
_{fmpz_mod_poly_radix} (C function), 435
_{fmpz_mod_poly_radix_init} (C function), 434
_{fmpz_mod_poly_reduce} (C function), 1004
_{fmpz_mod_poly_reduce_matrix_mod_poly} (C function), 431
_{fmpz_mod_poly_rem} (C function), 421
_{fmpz_mod_poly_rem_basecase} (C function), 420
_{fmpz_mod_poly_resultant} (C function), 428
Index

_fmpz_poly_num_real_roots_sturm (C function), 212
_fmpz_poly_pow (C function), 186
_fmpz_poly_pow_addchains (C function), 186
_fmpz_poly_pow_binexp (C function), 186
_fmpz_poly_pow_binomial (C function), 185
_fmpz_poly_pow_multinomial (C function), 185
_fmpz_poly_pow_small (C function), 186
_fmpz_poly_pow_trunc (C function), 186
_fmpz_poly_power_sums_naive (C function), 206
_fmpz_poly_power_sums_to_poly (C function),

_fmpz_poly_powers_clear (C function), 196
_fmpz_poly_powers_precompute (C function),

_fmpz_poly_preinvert (C function), 195
_fmpz_poly_primitive_part (C function), 191
_fmpz_poly_print (C function), 209
_fmpz_poly_print_pretty (C function), 210
_fmpz_poly_product_roots_fmpq_vec (C function),

_fmpz_poly_product_roots_fmpz_vec (C function), 211
_fmpz_poly_pseudo_div (C function), 199
_fmpz_poly_pseudo_divrem (C function), 199
_fmpz_poly_pseudo_divrem_basecase (C function),

_fmpz_poly_pseudo_divrem_cohen (C function),

_fmpz_poly_pseudo_divrem_conquer (C function), 198
_fmpz_poly_pseudo_divrem_conquer (C function), 198
_fmpz_poly_pseudo_divrem_conquer (C function),

_fmpz_poly_pseudo_rem (C function), 199
_fmpz_poly_pseudo_rem_cohen (C function), 199
_fmpz_poly_reduce (C function), 1004
_fmpz_poly_rem (C function), 195
_fmpz_poly_rem_basecase (C function), 194
_fmpz_poly_rem_powers_precomp (C function),

_fmpz_poly_remove_content_2exp (C function),

_fmpz_poly_resultant (C function), 190
_fmpz_poly_resultant_euclidean (C function),

_fmpz_poly_resultant_modular (C function),

_fmpz_poly_reverse (C function), 176
_fmpz_poly_reverse_series (C function), 205
_fmpz_poly_scale_2exp (C function), 180
_fmpz_poly_set_length (C function), 175
_fmpz_poly_set_str (C function), 175
_fmpz_poly_shift_left (C function), 187
_fmpz_poly_shift_right (C function), 187
_fmpz_poly_signature (C function), 207
_fmpz_poly_sqrt (C function), 184
_fmpz_poly_sqrt_KS (C function), 184
_fmpz_poly_sqrt_classical (C function), 184
_fmpz_poly_sqrt_karatsuba (C function), 184
_fmpz_poly_sqrt_KS (C function), 185

_fmpz_poly_sqrtrem_conquer (C function), 205
_fmpz_poly_sqrtrem_divconquer (C function), 205
_fmpz_poly_sqrtrem_divconquer (C function),

_fmpz_poly_sqrtrem_conquer (C function), 205
_fmpz_poly_sqrtrem_divconquer (C function),

_fmpz_poly_sqrtrem_conquer (C function),

_fmpz_poly_theta_qexp (C function), 215
_fmpz_poly_xgcd (C function), 189
_fmpz_poly_xgcd_modular (C function), 188

_fmpz_size (C function), 739
_fmpz_sub_small (C function), 739
_fmpz_vec_add (C function), 144
_fmpz_vec_clear (C function), 141
_fmpz_vec_content (C function), 146
_fmpz_vec_content_chained (C function), 146

_fmpz_vec_dot (C function), 146
_fmpz_vec_dot_general (C function), 146
_fmpz_vec_dot_general_naive (C function), 146
_fmpz_vec_equal (C function), 143
_fmpz_vec_fprint (C function), 142
_fmpz_vec_fread (C function), 142
_fmpz_vec_get_d_vec_2exp (C function), 143
_fmpz_vec_get_fft (C function), 143
_fmpz_vec_get_mmod_vec (C function), 143
_fmpz_vec_height (C function), 142
_fmpz_vec_height_index (C function), 142
_fmpz_vec_init (C function), 141
_fmpz_vec_is_zero (C function), 143
_fmpz_vec_lcm (C function), 146
_fmpz_vec_max (C function), 143
_fmpz_vec_max_bits (C function), 142
_fmpz_vec_max_bits_ref (C function), 142
_fmpz_vec_max_inplace (C function), 143
_fmpz_vec_max_limb (C function), 142
_fmpz_vec_neg (C function), 143
_fmpz_vec_print (C function), 142
_fmpz_vec_prod (C function), 145
Flint Documentation, Release 3.1.0-dev

 fq_zech_poly_make_monic (C function), 961
 fq_zech_poly_mul (C function), 965
 fq_zech_poly_mul KS (C function), 964
 fq_zech_poly_mul_classical (C function), 964
 fq_zech_poly_mul_reorder (C function), 964
 fq_zech_poly_mulhigh (C function), 966
 fq_zech_poly_mulhigh_classical (C function), 965
 fq_zech_poly_mullow (C function), 965
 fq_zech_poly_mullow KS (C function), 965
 fq_zech_poly_mullow_classical (C function), 965
 fq_zech_poly_mulmod (C function), 966
 fq_zech_poly_mulmod_preinv (C function), 966
 fq_zech_poly_neg (C function), 963
 fq_zech_poly_normalise (C function), 960
 fq_zech_poly_normalise2 (C function), 960
 fq_zech_poly_pow (C function), 967
 fq_zech_poly_pow_trunc (C function), 969
 fq_zech_poly_pow_trunc_binexp (C function), 969
 fq_zech_poly_powmod_fmpz_binexp (C function), 968
 fq_zech_poly_powmod_fmpz_binexp_preinv (C function), 968
 fq_zech_poly_powmod_fmpz_sliding_preinv (C function), 968
 fq_zech_poly_powmod_ui_binexp (C function), 967
 fq_zech_poly_powmod ui_binexp (C function), 967
 fq_zech_poly_powmod ui_binexp_preinv (C function), 967
 fq_zech_poly_powmod x_fmpz_preinv (C function), 968
 fq_zech_poly_premcompe_matrix (C function), 976
 fq_zech_poly_print (C function), 978
 fq_zech_poly_print_pretty (C function), 977
 fq_zech_poly_reduce_matrix_mod_poly (C function), 976
 fq_zech_poly_rem (C function), 970
 fq_zech_poly_reverse (C function), 960
 fq_zech_poly_scalar_addmul_fq_zech (C function), 963
 fq_zech_poly_scalar_div_fq_zech (C function), 964
 fq_zech_poly_scalar_mul_fq_zech (C function), 963
 fq_zech_poly_scalar_submul_fq_zech (C function), 963
 fq_zech_poly_set (C function), 961
 fq_zech_poly_set_length (C function), 960
 fq_zech_poly_shift_left (C function), 969
 fq_zech_poly_shift_right (C function), 969
 fq_zech_poly_sqr (C function), 967
 fq_zech_poly_sqr KS (C function), 967
 fq_zech_poly_sqr_classical (C function), 966
 fq_zech_poly_sqr (C function), 974
 fq_zech_poly_sqrt_series (C function), 978
 1128 Index
 967
 969
 946
 961
 965
 969
 952
 966
 966
 966
 966
 966
 966
 966
 966
 966
 966
 966
 966
 966
 966
 966
 966
gr_poly_revert_series_lagrange_fast (C function), 93
gr_poly_revert_series_newton (C function), 93
gr_poly_rsqrtn_series (C function), 91
gr_poly_rsqrtn_series_basecase (C function), 91
gr_poly_rsqrtn_series_miller (C function), 91
gr_poly_set_length (C function), 86
gr_poly_shift_left (C function), 87
gr_poly_shift_right (C function), 87
gr_poly_sin_cos_series_basecase (C function), 97
gr_poly_sin_cos_series_tangent (C function), 97
gr_poly_taylor_shift (C function), 92
gr_poly_taylor_shift_convolution (C function), 92
gr_poly_taylor_shift_divconquer (C function), 92
gr_poly_taylor_shift_horner (C function), 92
gr_poly_tree_alloc (C function), 92
gr_poly_tree_build (C function), 92
gr_poly_tree_free (C function), 92
gr_poly_xgcd_euclidean (C function), 95
gr_poly_xgcd_hgcd (C function), 95
gr_scalar_add_vec (C function), 69
gr_scalar_add2_vec (C function), 69
gr_scalar_div_exact_vec (C function), 69
gr_scalar_mul_vec (C function), 69
gr_scalar_other_add_vec (C function), 70
gr_scalar_other_div_vec (C function), 70
gr_scalar_other_divexact_vec (C function), 70
gr_scalar_other_mul_vec (C function), 70
gr_scalar_other_pow_vec (C function), 70
gr_scalar_other_sub_vec (C function), 70
gr_scalar_pow_vec (C function), 69
gr_scalar_sub_vec (C function), 69
gr_vec_add (C function), 69
gr_vec_add_other (C function), 69
gr_vec_add_scalar (C function), 69
gr_vec_add2_scalar_fmpz (C function), 70
gr_vec_add2_scalar_fmpq (C function), 70
gr_vec_add2_scalar_si (C function), 71
gr_vec_addmul_scalar (C function), 71
gr_vec_addmul_scalar_fmpz (C function), 70
gr_vec_addmul_scalar_fmpq (C function), 70
gr_vec_addmul_scalar_si (C function), 71
gr_vec_clear (C function), 68
gr_vec_div (C function), 69
gr_vec_div_other (C function), 69
gr_vec_div_scalar (C function), 69
gr_vec_div_scalar_fmpz (C function), 70
gr_vec_div_scalar_fmpq (C function), 70
gr_vec_div_scalar_other (C function), 70
gr_vec_div_scalar_si (C function), 70
gr_vec_div_scalar_ui (C function), 70
gr_vec_divexact (C function), 69
gr_vec_divexact_other (C function), 69
gr_vec_divexact_scalar (C function), 69
gr_vec_divexact_scalar_fmpz (C function), 70
gr_vec_divexact_scalar_fmpq (C function), 70
gr_vec_divexact_scalar_si (C function), 70
gr_vec_divexact_scalar_ui (C function), 70
gr_vec_dot (C function), 71
gr_vec_dot_fmpz (C function), 71
gr_vec_dot_fmpq (C function), 71
gr_vec_dot_rev (C function), 71
gr_vec_dot_si (C function), 71
gr_vec_equal (C function), 69
gr_vec_init (C function), 68
gr_vec_is_zero (C function), 69
gr_vec_mul (C function), 69
gr_vec_mul_other (C function), 69
gr_vec_mul_scalar (C function), 69
gr_vec_mul_scalar_2exp_si (C function), 71
gr_vec_mul_scalar_fmpz (C function), 70
gr_vec_mul_scalar_fmpq (C function), 70
gr_vec_mul_scalar_other (C function), 70
gr_vec_mul_scalar_si (C function), 70
gr_vec_mul_scalar_ui (C function), 70
gr_vec_normalise (C function), 69
gr_vec_normalise_weak (C function), 69
gr_vec_pow_other (C function), 69
gr_vec_pow_scalar_other (C function), 69
gr_vec_pow_scalar_ui (C function), 69
gr_vec_pow_scalar_fmpq (C function), 70
gr_vec_pow_scalar_fmpz (C function), 70
gr_vec_pow_scalar_si (C function), 70
gr_vec_pow_scalar_ui (C function), 70
gr_vec_reciprocals (C function), 72
gr_vec_set (C function), 68
gr_vec_set_powers (C function), 72
gr_vec_step (C function), 72
gr_vec_sub (C function), 69
gr_vec_sub_other (C function), 69
gr_vec_sub_scalar (C function), 69
gr_vec_sub_scalar_fmpz (C function), 70
gr_vec_sub_scalar_fmpq (C function), 70
gr_vec_addmul_scalar (C function), 71
nmod_poly_sinh_series (C function), 375
nmod_poly_sqrt (C function), 372
nmod_poly_sqrt_series (C function), 372
nmod_poly_sub (C function), 352
nmod_poly_tan_series (C function), 375
nmod_poly_tanh_series (C function), 375
nmod_poly_taylor_shift (C function), 365
nmod_poly_taylor_shift_convolution (C function), 364
nmod_poly_taylor_shift_horner (C function), 364
nmod_poly_tree_alloc (C function), 376
nmod_poly_tree_build (C function), 376
nmod_poly_tree_free (C function), 376
nmod_poly_xgcd (C function), 369
nmod_poly_xgcd_euclidean (C function), 368
nmod_poly_xgcd_hgcd (C function), 369
nmod_sub (C function), 334
nmod_vec_add (C function), 336
nmod_vec_clear (C function), 335
nmod_vec_dot (C function), 337
nmod_vec_dot_bound_limbs (C function), 336
nmod_vec_dot_ptr (C function), 337
nmod_vec_dot_rev (C function), 337
nmod_vec_equal (C function), 335
nmod_vec_fprint (C function), 336
nmod_vec_fprint_pretty (C function), 336
nmod_vec_init (C function), 335
nmod_vec_max_bits (C function), 335
nmod_vec_neg (C function), 336
nmod_vec_print (C function), 336
nmod_vec_print_pretty (C function), 336
nmod_vec_randtest (C function), 335
nmod_vec_reduce (C function), 335
nmod_vec_scalar_addmul_nmod (C function), 335
nmod_vec_scalar_mul_nmod (C function), 336
nmod_vec_scalar_mul_nmod_shoup (C function), 336
nmod_vec_set (C function), 335
nmod_vec_sub (C function), 336
nmod_vec_swap (C function), 335
nmod_vec_zero (C function), 335
nadic_canonicalise (C function), 986
nadic_ctx_pow_ui (C function), 986
nadic_exp (C function), 989
nadic_exp_balanced (C function), 989
nadic_exp_balanced (C function), 989
nadic_exp_bound (C function), 989
nadic_exp_rectangular (C function), 989
nadic_fprintf (C function), 991
nadic_inv (C function), 989
nadic_inv_clear (C function), 988
nadic_inv_precomp (C function), 988
nadic_inv_precompute (C function), 988
nadic_lifts_exp (C function), 988
nadic_lifts_pows (C function), 988
nadic_log (C function), 990
nadic_log_balanced (C function), 990
nadic_log_bound (C function), 990
nadic_log_rectangular (C function), 990
nadic_log_satoh (C function), 990
nadic_mat_add (C function), 1002
nadic_mat_canonicalise (C function), 1000
nadic_mat_mul (C function), 1003
nadic_mat_neg (C function), 1002
nadic_mat_reduce (C function), 1000
nadic_mat_scalar_mul_fmpz (C function), 1002
nadic_mat_scalar_mul_nmod (C function), 1002
nadic_mat_sub (C function), 1002
nadic_poly_add (C function), 995
nadic_poly_canonicalise (C function), 993
nadic_poly_compose (C function), 997
nadic_poly_compose_pow (C function), 997
nadic_poly_derivative (C function), 997
nadic_poly_evaluate_padic (C function), 997
nadic_poly_fprintf (C function), 998
nadic_poly_fprintf_pretty (C function), 998
nadic_poly_is_canonical (C function), 999
nadic_poly_is_reduced (C function), 999
nadic_poly_log_rectangular (C function), 999
nadic_poly_mul (C function), 996
nadic_poly_normalise (C function), 992
nadic_poly_pow (C function), 996
nadic_poly_print (C function), 998
nadic_poly_print_pretty (C function), 998
nadic_poly_scalar_mul_padic (C function), 995
nadic_poly_set_length (C function), 992
nadic_poly_sub (C function), 995
nadic_poly_print (C function), 992
nadic_print (C function), 992
nadic_reduce (C function), 986
nadic_teachmuller (C function), 991
perm_clear (C function), 455
perm_compose (C function), 455
perm_one (C function), 455
perm_inv (C function), 455
qadic_exp (C function), 1007
qadic_exp_balanced (C function), 1007
qadic_exp_rectangular (C function), 1007
qadic_frobenius (C function), 1008
qadic_frobenius_a (C function), 1008
qadic_inv (C function), 1006
qadic_log (C function), 1008
qadic_log_balanced (C function), 1008
qadic_log_rectangular (C function), 1007
qadic_norm (C function), 1009
qadic_pow (C function), 1006
qadic_teachmuller (C function), 1009
qadic_trace (C function), 1009
qqbar_acb_lindep (C function), 491
qqbar_enclosure_raw (C function), 491
qqbar_evaluate_fmpz_poly (C function), 485
_qqbar_get_fmpz (C function), 480
_qqbar_validate_existence_uniqueness (C function), 490
_qqbar_vec_clear (C function), 479
_qqbar_vec_init (C function), 479
 unity_zp (C type), 249
 unity_zp_pow_select_k (C function), 250
 unity_zp_reduce_cyclotomic (C function), 251
 unity_zp_reduce_cyclotomic_divmod (C function), 251
 unity_zpq (C type), 249
 unity_zpq_mul_unity_p (C function), 252

A

Abs (C macro), 826
acb_abs (C function), 566
acb_acos (C function), 571
acb_acosh (C function), 572
acb_add (C function), 567
acb_add_arb (C function), 567
acb_add_error_arb (C function), 563
acb_add_error_arf (C function), 563
acb_add_error_mag (C function), 563
acb_add_fmpz (C function), 567
acb_add_si (C function), 566
acb_add_ui (C function), 566
acb_addmul (C function), 567
acb_addmul_arb (C function), 567
acb_addmul_fmpz (C function), 567
acb_addmul_si (C function), 567
acb_addmul_ui (C function), 567
acb_agm (C function), 575
acb_agm1 (C function), 575
acb_agm1_cpx (C function), 575
acb_allocated_bytes (C function), 562
acb_approx_dot (C function), 568
acb_arg (C function), 566
acb_asin (C function), 571
acb_asinh (C function), 572
acb_atan (C function), 571
acb_atanh (C function), 572
acb_barnes_g (C function), 574
acb_beroulli_poly_ui (C function), 574
acb_bits (C function), 566
acb_calc_cauchy_bound (C function), 727
acb_calc_func_t (C type), 723
acb_calc_integrate (C function), 724
acb_calc_integrate_gl_auto_deg (C function), 726
acb_calc_integrate_opt_init (C function), 726
acb_calc_integrate_opt_struct (C type), 725
acb_calc_integrate_opt_t (C type), 725
acb_calc_integrate_opt_t_deg_limit (C member), 725
acb_calc_integrate_opt_t_depth_limit (C member), 726
acb_calc_integrate_opt_t_eval_limit (C member), 725
acb_calc_integrate_opt_t_use_heap (C member), 726
acb_calc_integrate_opt_t_verbose (C member), 726
acb_chebyshev_t2_ui (C function), 575
acb_chebyshev_t_ui (C function), 575
acb_chebyshev_u2_ui (C function), 575
acb_chebyshev_u_ui (C function), 575
acb_clear (C function), 562
acb_conj (C function), 566
acb_const_pi (C function), 569
acbcontains (C function), 565
acbcontains_fmpz (C function), 565
acbcontains_fmpq (C function), 565
acbcontains_int (C function), 565
acbcontains_interior (C function), 565
acbcontains_zero (C function), 565
acb_cos (C function), 570
acb_cos_pi (C function), 571
acb_cosh (C function), 571
acb_cot (C function), 570
acb_cot_pi (C function), 571
acb_coth (C function), 571
acb_csc (C function), 571
acb_csc_pi (C function), 571
acb_csch (C function), 571
acb_csgn (C function), 566
acb_cube (C function), 567
acb_dft (C function), 613
acb_dft_bluestein (C function), 616
acb_dft_bluestein_clear (C function), 616
acb_dft_bluestein_init (C function), 616
acb_dft_bluestein_precomp (C function), 616
acb_dft_bluestein_struct (C type), 616
acb_dft_bluestein_t (C type), 616
acb_dft_convol (C function), 614
acb_dft_convol_naive (C function), 614
acb_dft_convol_rad2 (C function), 614
acb_dft_crt (C function), 615
acb_dft_crt_clear (C function), 615
acb_dft_crt_precomp (C function), 615
acb_dft_crt_struct (C type), 615
acb_dft_crt_t (C type), 615
acb_dft_cyc (C function), 615
acb_dft_cyc_clear (C function), 615
acb_dft_cyc_init (C function), 615
acb_dft_cyc_precomp (C function), 615
acb_dft_cyc_struct (C type), 615
acb_dft_cyc_t (C type), 615
acb_dft_inverse (C function), 613
acb_dft_inverse_precomp (C function), 613
acb_dft_inverse_rad2 (C function), 616
acb_dft_naive (C function), 613
acb_dft_naive_clear (C function), 615
acb_dft_naive_init (C function), 615
acb_dft_naive_precomp (C function), 615
acb_dft_naive_struct (C type), 615
acb_dft_naive_t (C type), 615
acb_dft_pre_struct (C type), 613
acb_dft_pre_t (C type), 613
acb_dft_precomp (C function), 613
acb_dft_precomp_clear (C function), 613
acb_dft_precomp_init (C function), 613
acb_dft_prod_clear (C function), 614
acb_dft_prod_init (C function), 614
acb_dft_prod_struct (C type), 613
acb_dft_prod_t (C type), 613
acb_dft_rad2 (C function), 616
acb_dft_rad2_clear (C function), 616
acb_dft_rad2_init (C function), 616
acb_dft_rad2_precomp (C function), 616
acb_dft_rad2_struct (C type), 616
acb_digamma (C function), 573
acb_dirichlet_backlund_s (C function), 713
acb_dirichlet_backlund_s_bound (C function), 714
acb_dirichlet_backlund_s_gram (C function), 714
acb_dirichlet_chi (C function), 707
acb_dirichlet_chi_theta Arb (C function), 708
acb_dirichlet_chi_vec (C function), 707
acb_dirichlet_dft (C function), 710
acb_dirichlet_dft_conrey (C function), 709
acb_dirichlet_dft_prod (C function), 613
acb_dirichlet_dft_prod_precomp (C function), 614
acb_dirichlet_eta (C function), 704
acb_dirichlet_gauss_sum (C function), 708
acb_dirichlet_gauss_sum_factor (C function), 707
acb_dirichlet_gauss_sum naïve (C function), 707
acb_dirichlet_gauss_sum_order2 (C function), 707
acb_dirichlet_gauss_sum_theta (C function), 708
acb_dirichlet_gram_point (C function), 712
acb_dirichlet_hardy_theta (C function), 712
acb_dirichlet_hardy_theta_series (C function), 712
acb_dirichlet_hardy_z (C function), 712
acb_dirichlet_hardy_z_series (C function), 712
acb_dirichlet_hardy_z_zero (C function), 713
acb_dirichlet_hardy_z_zeros (C function), 713
acb_dirichlet_hurwitz (C function), 705
acb_dirichlet_hurwitz_precomp_bound (C function), 706
acb_dirichlet_hurwitz_precomp_choose_param (C function), 706
acb_dirichlet_hurwitz_precomp_clear (C function), 706
acb_dirichlet_hurwitz_precomp_eval (C function), 706
acb_dirichlet_hurwitz_precomp_init (C function), 706
acb_dirichlet_hurwitz_precomp_num (C function), 706
acb_dirichlet_hurwitz_precomp_struct (C type), 706
acb_dirichlet_hurwitz_precomp_t (C type), 706
acb_dirichlet_isolate_hardy_z_zero (C function), 713
acb_dirichlet_jacobi_sum (C function), 708
acb_dirichlet_jacobi_sum_factor (C function), 708
acb_dirichlet_jacobi_sum_gauss (C function), 708
acb_dirichlet_jacobi_sum naïve (C function), 708
acb_dirichlet_jacobi_sum ui (C function), 708
acb_dirichlet_l (C function), 711
acb_dirichlet_l euler_product (C function), 711
acb_dirichlet_l fmpq (C function), 711
acb_dirichlet_l fmpq afe (C function), 711
acb_dirichlet_l hurwitz (C function), 710
acb_dirichlet_l jet (C function), 711
acb_dirichlet_l series (C function), 711
acb_dirichlet_l vec hurwitz (C function), 711
acb_dirichlet_lerch phi (C function), 706
acb_dirichlet_lerch phi direct (C function), 706
acb_dirichlet_lerch phi integral (C function), 706
acb_dirichlet_pairing (C function), 707
acb_dirichlet_pairing char (C function), 707
acb_dirichlet_platt hardy z zeros (C function), 715
acb_dirichlet_platt_local hardy z zeros (C function), 715
acb_dirichlet_platt multieval (C function), 714
acb_dirichlet_platt multieval threaded (C function), 714
acb_dirichlet_platt scaled lambda (C function), 714
acb_dirichlet_platt scaled lambda vec (C function), 714
acb_dirichlet_platt ws interpolation (C function), 714
acb_dirichlet_platt zeta zeros (C function), 715
acb_dirichlet_powsum sieved (C function), 704
acb_dirichlet_powsum smooth (C function), 704
acb_dirichlet_powsum term (C function), 704
acb_dirichlet qseries arb powers naïve (C function), 709
acb_dirichlet_qseries_arb_powers_smallorder (C function), 709
acb_dirichlet_root (C function), 703
acb_dirichlet_root_number (C function), 710
acb_dirichlet_root_number_theta (C function), 710
acb_dirichlet_roots_clear (C function), 703
acb_dirichlet_roots_init (C function), 703
acb_dirichlet_roots_t (C type), 703
acb_dirichlet_stieltjes (C function), 707
acb_dirichlet_theta_length (C function), 709
acb_dirichlet_turing_method_bound (C function), 713
acb_dirichlet_ui_theta_arb (C function), 708
acb_dirichlet_zeta (C function), 704
acb_dirichlet_zeta_bound (C function), 704
acb_dirichlet_zeta_deriv_bound (C function), 704
acb_dirichlet_zeta_jet (C function), 704
acb_dirichlet_zeta_jet_rs (C function), 705
acb_dirichlet_zeta_nzeros (C function), 713
acb_dirichlet_zeta_nzeros_gram (C function), 714
acb_dirichlet_zeta_rs (C function), 705
acb_dirichlet_zeta_rs_bound (C function), 705
acb_dirichlet_zeta_rs_d_coeffs (C function), 705
acb_dirichlet_zeta_rs_f_coeffs (C function), 705
acb_dirichlet_zeta_zero (C function), 705
acb_div (C function), 568
acb_div Arb (C function), 568
acb_div_fmpz (C function), 568
acb_div_onei (C function), 567
acb_div_si (C function), 568
acb_div_ui (C function), 568
acb_dot (C function), 568
acb_dot_fmpz (C function), 568
acb_dot_precise (C function), 568
acb_dot_si (C function), 568
acb_dot_simple (C function), 568
acb_dot_siui (C function), 568
acb_dot_ui (C function), 568
acb_dot_uiui (C function), 568
acb_elliptic_e (C function), 665
acb_elliptic_e_inc (C function), 665
acb_elliptic_f (C function), 665
acb_elliptic_invc (C function), 668
acb_elliptic_invariants (C function), 668
acb_elliptic_k (C function), 665
acb_elliptic_k_jet (C function), 665
acb_elliptic_k_series (C function), 665
acb_elliptic_p (C function), 668
acb_elliptic_p_jet (C function), 668
acb_elliptic_p_prime (C function), 668
acb_elliptic_p_series (C function), 668
acb_elliptic_pi (C function), 665
acb_elliptic_pi_inc (C function), 666
acb_elliptic_rcl (C function), 667
acb_elliptic_rf (C function), 666
acb_elliptic_rg (C function), 667
acb_elliptic_ri (C function), 667
acb_elliptic_ri_carlson (C function), 667
acb_elliptic_ri_integral (C function), 667
acb_elliptic_roots (C function), 668
acb_elliptic_sigma (C function), 668
acb_elliptic_zeta (C function), 668
acb_eq (C function), 565
acb_equal (C function), 564
acb_equal_si (C function), 564
acb_exp (C function), 570
acb_exp_invers (C function), 570
acb_exp_pi_i (C function), 570
acb_expr (C function), 570
acb_fprint (C function), 563
acb_fprintn (C function), 563
acb_fprintln (C function), 563
acb_fprintf (C function), 563
acb_gaussian (C function), 573
acb_get_abs (bound), 564
acb_get_abs_inbound (arf), 565
acb_get_abs_ubound (arf), 565
acb_get_imag (C function), 566
acb_get_mag (C function), 565
acb_get_mag_lower (C function), 565
acb_get_mid (C function), 563
acb_get_rad (bound), 565
acb_get_real (C function), 566
acb_get_unique (fmpz), 566
acb_hurwitz_zeta (C function), 574
acb_hypergeom_0f1 (C function), 643
acb_hypergeom_0f1_asympt (C function), 643
acb_hypergeom_0f1_direct (C function), 643
acb_hypergeom_1f1 (C function), 643
acb_hypergeom_2f1 (C function), 651
acb_hypergeom_2f1_choose (C function), 651
acb_hypergeom_2f1_continuation (C function), 651
acb_hypergeom_2f1_corner (C function), 651
acb_hypergeom_2f1_direct (C function), 651
acb_hypergeom_2f1_series_direct (C function), 651
acb_hypergeom_2f1_transform (C function), 651
acb_hypergeom_2f1_transform_limit (C function), 651
acb_hypergeom_airy (C function), 646
acb_hypergeom_airy_asympt (C function), 646
acb_hypergeom_airy_bound (C function), 646
acb_hypergeom_airy_direct (C function), 646
acb_hypergeom_airy_jet (C function), 647
acb_hypergeom_airy_series (C function), 647
acb_hypergeom_bessel_i (C function), 645
acb_hypergeom_bessel_i_0f1 (C function), 645
acb_hypergeom_bessel_i_asympt (C function), 645
acb_hypgeom_bessel_i_scaled (C function), 645
acb_hypgeom_bessel_j (C function), 644
acb_hypgeom_bessel_j_0f1 (C function), 644
acb_hypgeom_bessel_j_asymp (C function), 644
acb_hypgeom_bessel_jy (C function), 645
acb_hypgeom_bessel_k (C function), 646
acb_hypgeom_bessel_k_0f1 (C function), 645
acb_hypgeom_bessel_k_0f1_series (C function), 645
acb_hypgeom_bessel_k_asym (C function), 645
acb_hypgeom_bessel_k_scaled (C function), 646
acb_hypgeom_bessel_y (C function), 645
acb_hypgeom_beta_lower (C function), 649
acb_hypgeom_beta_lower_series (C function), 649
acb_hypgeom_chebyshev_t (C function), 652
acb_hypgeom_chebyshev_u (C function), 652
acb_hypgeom_chi (C function), 650
acb_hypgeom_chi_2f3 (C function), 650
acb_hypgeom_chebyshev_u (C function), 650
acb_hypgeom_dilog_zero_taylor (C function), 651
acb_hypgeom_dilog_zero (C function), 651
acb_hypgeom_dilog_transform (C function), 654
acb_hypgeom_dilog_continuation (C function), 654
acb_hypgeom_dilog_continue (C function), 654
acb_hypgeom_dilog_asymp (C function), 654
acb_hypgeom_dilog (C function), 654
acb_hypgeom_log_rising_ui (C function), 639
acb_hypgeom_log_rising_ui_series (C function), 639
acb_hypgeom_log_rising_ui_jet (C function), 639
acb_hypgeom_m_asymp (C function), 643
acb_hypgeom_m (C function), 643
acb_hypgeom_pfq_choose_n (C function), 642
acb_hypgeom_pfq (C function), 642
acb_hypgeom_pfq_bound_factor (C function), 640
acb_hypgeom_pfq_choose_n (C function), 640
acb_hypgeom_pfq_direct (C function), 641
acb_hypgeom_pfq_series_direct (C function), 641
acb_hypgeom_pfq_series_sum (C function), 641
acb_hypgeom_pfq_series_sum_rs (C function), 641
acb_hypgeom_pfq_series_sum_forward (C function), 641
acb_hypgeom_pfq_series_sum (C function), 641
acb_hypgeom_pfq_sum (C function), 640
acb_hypgeom_pfq_sum_bs (C function), 640
acb_hypgeom_pfq_sum_bs_inzv (C function), 641
acb_hypgeom_pfq_sum_fme (C function), 640
acb_hypgeom_pfq_sum_forward (C function), 640
acb_hypgeom_pfq_series_sum (C function), 641
acb_hypgeom_pfq_sum bs (C function), 640
acb_hypgeom_pfq_sum rs (C function), 640
acb_hypgeom_rgamma (C function), 640
acb_hypgeom_gegenbauer_c (C function), 644
acb_hypgeom_gamma_upper_asymp (C function), 648
acb_hypgeom_hermite_h (C function), 649
acb_hypgeom_bessel_jy (C function), 655
acb_hypgeom_gamma_asym (C function), 650
acb_hypgeom_beta_lower (C function), 649
acb_hypgeom_beta_lower_series (C function), 649
acb_hypgeom_gamma_lower_series (C function), 648
acb_hypgeom_gamma_stirling (C function), 639
acb_hypgeom_gamma_stirling_sum_horner (C function), 639
acb_hypgeom_gamma_stirling_sum_improved (C function), 639
acb_hypgeom_gamma_taylor (C function), 639
acb_hypgeom_gamma_upper (C function), 648
acb_hypgeom_gamma_upper_1f1a (C function), 648
acb_hypgeom_gamma_upper_1f1b (C function), 648
acb_hypgeom_gamma_upper_asym (C function), 648
acb_hypgeom_gamma_upper_series (C function), 648
acb_hypgeom_gamma_upper_singular (C function), 648
acb_hypgeom_gegenbauer_c (C function), 652
acb_hypgeom_hermite_h (C function), 653
acb_hypgeom_jacobi_p (C function), 652
acb_hypgeom_laguerre_1 (C function), 652
acb_hypgeom_legendre_p (C function), 653
acb_hypgeom_legendre_p uiui_rec (C function), 653
acb_hypgeom_legendre_q (C function), 653
acb_hypgeom_lgamma (C function), 640
acb_hypgeom_li (C function), 650
acb_hypgeom_jacobi_w (C function), 653
acb_hypgeom_jacobi_u (C function), 653
acb_hypgeom_jacobi_z (C function), 653
acb_hypgeom_li_series (C function), 651
acb_hypgeom_log_rising_ui (C function), 639
acb_hypgeom_log_rising_ui_series (C function), 639
acb_hypgeom_log_rising_ui_jet (C function), 639
acb_hypgeom_m (C function), 643
acb_hypgeom_m_asymp (C function), 643
ac
acb_hypgeom_rising (C function), 639
acb_hypgeom_rising_ui (C function), 639
acb_hypgeom_rising_ui_bs (C function), 639
acb_hypgeom_rising_ui_jet (C function), 639
acb_hypgeom_rising_ui_jet_forward (C function), 639
acb_hypgeom_rising_ui_jet_powsum (C function), 639
acb_hypgeom_rising_ui_jet_RS (C function), 639
acb_hypgeom_rising_ui_rec (C function), 639
acb_hypgeom_shi (C function), 650
acb_hypgeom_u (C function), 642
acb_hypgeom_u_1f1 (C function), 642
acb_hypgeom_u_1f1_series (C function), 642
acb_hypgeom_u_asym (C function), 642
acb_hypgeom_u_use_asym (C function), 642
acb_imagref (C macro), 562
acb_indeterminate (C function), 566
acb_init (C function), 562
acb_inv (C function), 568
acb_is_exact (C function), 564
acb_is_finite (C function), 564
acb_is_int (C function), 564
acb_is_int_2exp_si (C function), 564
acb_is_one (C function), 564
acb_is_real (C function), 566
acb_is_zero (C function), 564
acb_lambertw (C function), 572
acb_lambertw_asym (C function), 572
acb_lambertw_bound_deriv (C function), 572
acb_lambertw_check_deriv (C function), 572
acb_lgamma (C function), 573
acb_log (C function), 570
acb_loglp (C function), 570
acb_log_analytic (C function), 570
acb_log_barnes_g (C function), 574
acb_log_sin_pi (C function), 573
acb_mat_add (C function), 631
acb_mat_add_error_mag (C function), 636
acb_mat_allocated_bytes (C function), 628
acb_mat_approx_eig QR (C function), 636
acb_mat_approx_inv (C function), 634
acb_mat_approx_lu (C function), 634
acb_mat_approx_mul (C function), 631
acb_mat_approx_solve (C function), 634
acb_mat_approx_solve_lu_precomp (C function), 634
acb_mat_approx_solve_tril (C function), 634
acb_mat_approx_solve_triu (C function), 634
acb_mat_bound_frobenius_norm (C function), 631
acb_mat_bound_inf_norm (C function), 631
acb_mat_charpoly (C function), 635
acb_mat_clear (C function), 628
acb_mat_companion (C function), 635
acb_mat_conjugate (C function), 630
acb_mat_conjugate_transpose (C function), 630
acb_mat_contains (C function), 629
acb_mat_contains_fmpz_mat (C function), 629
acb_mat_contains_fmpz_mat (C function), 629
acb_mat_det (C function), 634
acb_mat_det_lu (C function), 634
acb_mat_det_precond (C function), 634
acb_mat_dft (C function), 630
acb_mat_diag_prod (C function), 635
acb_mat_eig_enclosure_rump (C function), 637
acb_mat_eig_global_enclosure (C function), 636
acb_mat_eig_multiple (C function), 638
acb_mat_eig_multiple_rump (C function), 638
acb_mat_eig_simple (C function), 637
acb_mat_eig_simple_rump (C function), 637
acb_mat_eig_simple_vdhoeven_mourrain (C function), 637
acb_mat_entry (C macro), 628
acb_mat_eq (C function), 629
acb_mat_equal (C function), 629
acb_mat_exp (C function), 635
acb_mat_exp_taylor_sum (C function), 635
acb_mat_fprintd (C function), 629
acb_mat_frobenius_norm (C function), 631
acb_mat_get_imag (C function), 628
acb_mat_get_mid (C function), 636
acb_mat_get_real (C function), 636
acb_mat_indeterminate (C function), 630
acb_mat_init (C function), 628
acb_mat_inv (C function), 634
acb_mat_is_diag (C function), 630
acb_mat_is_empty (C function), 629
acb_mat_is_exact (C function), 629
acb_mat_is_finite (C function), 630
acb_mat_is_real (C function), 629
acb_mat_is_square (C function), 629
acb_mat_is_tril (C function), 630
acb_mat_is_triu (C function), 630
acb_mat_is_zero (C function), 630
acb_mat_lu (C function), 633
acb_mat_lu_classical (C function), 633
acb_mat_lu_recursive (C function), 633
acb_mat_mul (C function), 631
acb_mat_mul_classical (C function), 631
acb_mat_mul_entrywise (C function), 631
acb_mat_mul_reorder (C function), 631
acb_mat_mul_threaded (C function), 631
acb_mat ncols (C macro), 628
acb_mat ne (C function), 629
acb_mat_neg (C function), 631
acb_mat_nrows (C macro), 628
acb_mat_one (C function), 630
acb_mat_onei (C function), 630
acb_mat_ones (C function), 630
acb_mat_overlaps (C function), 629
acb_mat_pow_ui (C function), 631
acb_mat_printd (C function), 629
acb_mat_randtest (C function), 629
acb_mat_randtest_eig (C function), 629
acb_mat_scalar_addmul_acb (C function), 632
acb_mat_scalar_addmul_2exp_si (C function), 632
acb_mat_scalar_addmul_fmpz (C function), 632
acb_mat_scalar_addmul_si (C function), 632
acb_mat_scalar_addmul_arb (C function), 632
acb_mat_scalar_div_fmpz (C function), 632
acb_mat_scalar_mul_acb (C function), 632
acb_mat_scalar_mul_2exp_si (C function), 632
acb_mat_scalar_mul_fmpz (C function), 632
acb_mat_scalar_mul_si (C function), 632
acb_mat_set (C function), 628
acb_mat_set_arb_mat (C function), 628
acb_mat_set_fmpz_mat (C function), 628
acb_mat_set_fmpq_mat (C function), 628
acb_mat_set_real_imag (C function), 628
acb_mat_set_round_fmpz_mat (C function), 628
acb_mat_set_round_arb_mat (C function), 628
acb_mat_solve (C function), 633
acb_mat_solve_lu (C function), 633
acb_mat_solve_lu_precomp (C function), 633
acb_mat_solve_precond (C function), 633
acb_mat_solve_tril (C function), 633
acb_mat_solve_tril_classical (C function), 633
acb_mat_solve_tril_recursive (C function), 633
acb_mat_solve_triu (C function), 633
acb_mat_solve_triu_classical (C function), 633
acb_mat_solve_triu_recursive (C function), 633
acb_mat_sqr (C function), 631
acb_mat_sqr_classical (C function), 631
acb_mat_struct (C type), 628
acb_mat_sub (C function), 631
acb_mat_t (C type), 628
acb_mat_trace (C function), 635
acb_mat_transpose (C function), 630
acb_mat_vector_mul_col (C function), 632
acb_mat_vector_mul_row (C function), 632
acb_mat_window_clear (C function), 628
acb_mat_window_init (C function), 628
acb_mat_zero (C function), 630
acb_modular_addseq_eta (C function), 674
acb_modular_addseq_theta (C function), 672
acb_modular_delta (C function), 675
acb_modular_eisenstein (C function), 675
acb_modular_elliptic_e (C function), 675
acb_modular_elliptic_k (C function), 675
acb_modular_elliptic_k_cpx (C function), 675
acb_modular_elliptic_p (C function), 675
acb_modular_elliptic_p_cpx (C function), 675
acb_modular_epsilon_arg (C function), 674
acb_modular_eta (C function), 674
acb_modular_eta_sum (C function), 674
acb_modular_fill_addseq (C function), 671
acb_modular_fundamental_domain_approx (C function), 670
acb_modular_fundamental_domain_approx_arf (C function), 670
acb_modular_fundamental_domain_approx_d (C function), 670
acb_modular_hilbert_class_poly (C function), 675
acb_modular_is_in_fundamental_domain (C function), 670
acb_modular_j (C function), 674
acb_modular_lambda (C function), 674
acb_modular_theta (C function), 673
acb_modular_theta_const_sum (C function), 673
acb_modular_theta_const_sum_basecase (C function), 673
acb_modular_theta_const_sum_basecase (C function), 673
acb_modular_theta_series (C function), 674
acb_modular_theta_sum (C function), 672
acb_modular_transform (C function), 671
acb_mul (C function), 567
acb_mul_2exp_fmpz (C function), 567
acb_mul_2exp_si (C function), 567
acb_mul_arb (C function), 567
acb_mul_fmpz (C function), 567
acb_mul_i_pow_si (C function), 567
acb_mul_one (C function), 567
acb_mul_onei (C function), 567
acb_mul_si (C function), 567
acb_mul_ui (C function), 567
acb_ne (C function), 565
acb_neg (C function), 566
acb_neg_round (C function), 566
acb_one (C function), 562
acb_onei (C function), 562
acb_overlaps (C function), 565
acb_poly_add (C function), 597
acb_poly_add_series (C function), 597
acb_poly_add_si (C function), 597
acb_poly_agm1_series (C function), 608
acb_poly_allocated_bytes (C function), 594
acb_poly_atan_series (C function), 604
acb_poly_binomial_transform (C function), 602
acb_poly_binomial_transform_basecase (C function), 602
acb_poly_binomial_transform_convolution (C function), 602
acb_poly_borel_transform (C function), 602
acb_poly_clear (C function), 594
acb_poly_compose (C function), 599
acb_poly_compose_series (C function), 599
acb_poly_contains (C function), 596
acb_poly_contains_fmpz_poly (C function), 596
acb_poly_contains_fmpq_poly (C function), 596
acb_poly_cos_pi_series (C function), 605
acb_poly_cot_pi_series (C function), 605
acb_poly_degree (C function), 594
acb_poly_derivative (C function), 602
acb_poly_div_series (C function), 598
acb_poly_divrem (C function), 599
acb_poly_elliptic_k_series (C function), 609
acb_poly_elliptic_p_series (C function), 609
acb_poly_equal (C function), 596
acb_poly_erf_series (C function), 608
acb_poly_evaluate (C function), 600
acb_poly_evaluate2 (C function), 600
acb_poly_evaluate2_horner (C function), 600
acb_poly_evaluate2_rectangular (C function), 600
acb_poly_evaluate_horner (C function), 600
acb_poly_evaluate_rectangular (C function), 600
acb_poly_evaluate_vec_fast (C function), 601
acb_poly_evaluate_vec_iter (C function), 601
acb_poly_exp_i_series (C function), 604
acb_poly_exp_series (C function), 604
acb_poly_exp_series_basecase (C function), 604
acb_poly_find_roots (C function), 609
acb_poly_fit_length (C function), 594
acb_poly_fprintf (C function), 596
acb_poly_gamma_series (C function), 606
acb_poly_get_coeff_acb (C function), 595
acb_poly_get_coeff_ptr (C macro), 595
acb_poly_get_unique_fmpz_poly (C function), 596
acb_poly_graeffe_transform (C function), 602
acb_poly_init (C function), 594
acb_poly_integral (C function), 602
acb_poly_interpolate_barycentric (C function), 601
acb_poly_interpolate_fast (C function), 601
acb_poly_interpolate_neutron (C function), 601
acb_poly_inv_borel_transform (C function), 602
acb_poly_inv_series (C function), 598
acb_poly_is_one (C function), 595
acb_poly_is_real (C function), 596
acb_poly_is_x (C function), 595
acb_poly_is_zero (C function), 594
acb_poly_lambertw_series (C function), 606
acb_poly_length (C function), 594
acb_poly_lgamma_series (C function), 606
acb_poly_loglp_series (C function), 604
acb_poly_log_series (C function), 604
acb_poly_majorant (C function), 597
acb_poly_mul (C function), 598
acb_poly_mullow (C function), 598
acb_poly_mullow_classical (C function), 598
acb_poly_mullow_transpose (C function), 598
acb_poly_mullow_transpose_gauss (C function), 598
acb_poly_neg (C function), 597
acb_poly_nth_derivative (C function), 602
acb_poly_one (C function), 595
acb_poly_overlaps (C function), 596
acb_poly_polylog_series (C function), 608
acb_poly_pow_acb_series (C function), 603
acb_poly_pow_series (C function), 603
acb_poly_pow_ui (C function), 603
acb_poly_pow_ui_trunc_binexp (C function), 603
acb_poly_printd (C function), 596
acb_poly_product_roots (C function), 600
acb_poly_randtest (C function), 596
acb_poly_revert_series (C function), 599
acb_poly_riemann_zeta_series (C function), 606
acb_poly_rising_ui_series (C function), 606
acb_poly_root_bound_fujiwara (C function), 609
acb_poly_rsqrt_series (C function), 604
acb_poly_scalar_div (C function), 597
acb_poly_scalar_mul (C function), 597
acb_poly_scalar_mul_2exp_si (C function), 597
acb_poly_set2_arb_poly (C function), 595
acb_poly_set_arb_poly (C function), 596
acb_poly_set_fmpz_poly (C function), 596
acb_poly_set2_fmpq_poly (C function), 596
acb_poly_set_acb (C function), 596
acb_poly_set_abspol (C function), 596
acb_poly_set_coeff_acb (C function), 595
acb_poly_set_coeff_fmpz (C function), 595
acb_poly_set_coeff_fmpq (C function), 595
acb_poly_set_coeff_ptr (C function), 595
acb_poly_set_fmpz_poly (C function), 596
acb_poly_set_fmpq_poly (C function), 596
acb_poly_set_mpmq_poly (C function), 596
acb_poly_set_mpmz_poly (C function), 596
acb_poly_set_si (C function), 596
acb_poly_set_trunc (C function), 595
acb_poly_set_trunc_round (C function), 595
acb_poly_shift_left (C function), 595
acb_poly_shift_right (C function), 595
acb_poly_sin_cos_pi_series (C function), 605
acb_poly_sin_cos_series (C function), 604
acb_poly_sin_pi_series (C function), 605
acb_poly_sin_series (C function), 605
acb_poly_sqrt_series (C function), 606
arb_chebyshev_u2_ui (C function), 556
arb_chebyshev_u_ui (C function), 556
arb_clear (C function), 537
arb_const_apery (C function), 553
arb_const_catalan (C function), 552
arb_const_e (C function), 553
arb_const_euler (C function), 552
arb_const_glaisher (C function), 553
arb_const_khinchin (C function), 553
arb_const_reciprocal_fibonacci (C function), 556
arb_const_sqrt_pi (C function), 556
arb_const_log2 (C function), 556
arb_const_log10 (C function), 556
arb_const_glaisher (C function), 556
arb_const_khinchin (C function), 556
arb_const_reciprocal_fibonacci (C function), 556
arb_const_sqrt_pi (C function), 556
arb_equal (C function), 544
arb_equal_si (C function), 544
arb_euler_number_fmpz (C function), 556
arb_euler_number_ui (C function), 556
arb_exp (C function), 550
arb_exp_arf (C function), 560
arb_exp_arf_bb (C function), 558
arb_exp_arf_generic (C function), 560
arb_exp_arf_log_reduction (C function), 559
arb_exp_arf_rs_generic (C function), 558
arb_expinvexp (C function), 550
arb_expm1 (C function), 550
arb_fac_ui (C function), 553
arb_fib_fmpz (C function), 556
arb_fib_ui (C function), 556
arb_floor (C function), 543
arb_fma (C function), 547
arb_fma_arf (C function), 547
arb_fma_fmpz (C function), 547
arb_fma_si (C function), 547
arb_fma_ui (C function), 547
arb_fmpz_div_fmpz (C function), 547
arb_fmpz_euler_number_ui (C function), 556
arb_fmpz_euler_number_ui_multi_mod (C function), 556
arb_fmpz_poly_complex_roots (C function), 611
arb_fmpz_poly_deflate (C function), 611
arb_fmpz_poly_deflation (C function), 611
arb_fmpz_poly_evaluate_acb (C function), 611
arb_fmpz_poly_evaluate_acb_horner (C function), 611
arb_fmpz_poly_evaluate_acb_rectangular (C function), 611
arb_fmpz_poly_evaluate_arb (C function), 611
arb_fmpz_poly_evaluate_arb_horner (C function), 610
arb_fmpz_poly_evaluate_arb_rectangular (C function), 610
arb_fmpz_poly_gauss_period_minpoly (C function), 612
arb_fpwrap_cdouble_agm (C function), 730
arb_fpwrap_cdouble_atanh (C function), 730
arb_fpwrap_cdouble_asinh (C function), 730
arb_fpwrap_cdouble_asin (C function), 730
arb_fpwrap_cdouble_airy_ai_prime (C function), 733
arb_fpwrap_cdouble_airy_ai (C function), 733
arb_fpwrap_cdouble_airy_bi (C function), 733
arb_fpwrap_cdouble_airy_bi_prime (C function), 733
arb_fpwrap_cdouble_sqrt (C function), 733
arb_fpwrap_cdouble_sqrt (C function), 733
arb_fpwrap_cdouble_sqrt (C function), 733
arb_fpwrap_cdouble_sqrt (C function), 733
arb_fpwrap_cdouble_sqrt (C function), 733
arb_fpwrap_cdouble_lgamma (C function), 731
arb_fpwrap_cdouble_lerch_phi (C function), 731
arb_fpwrap_cdouble_log (C function), 729
arb_fpwrap_cdouble_log1p (C function), 729
arb_fpwrap_cdouble_log_barnes_g (C function), 731
arb_fpwrap_cdouble_log_integral (C function), 733
arb_fpwrap_cdouble_modular_delta (C function), 736
arb_fpwrap_cdouble_modular_j (C function), 736
arb_fpwrap_cdouble_modular_lambda (C function), 736
arb_fpwrap_cdouble_polygamma (C function), 731
arb_fpwrap_cdouble_polylog (C function), 731
arb_fpwrap_cdouble_pow (C function), 729
arb_fpwrap_cdouble_rgamma (C function), 731
arb_fpwrap_cdouble_riemann_xi (C function), 731
arb_fpwrap_cdouble_rising (C function), 731
arb_fpwrap_cdouble_rsqrt (C function), 729
arb_fpwrap_cdouble_sec (C function), 729
arb_fpwrap_cdouble_sin (C function), 729
arb_fpwrap_cdouble_sin_integral (C function), 732
arb_fpwrap_cdouble_sin_pi (C function), 730
arb_fpwrap_cdouble_sinc (C function), 730
arb_fpwrap_cdouble_sinc_pi (C function), 730
arb_fpwrap_cdouble_sinh_integral (C function), 732
arb_fpwrap_cdouble_spherical_y (C function), 734
arb_fpwrap_cdouble_sqrt (C function), 729
arb_fpwrap_cdouble_tan (C function), 730
arb_fpwrap_cdouble_tan_pi (C function), 730
arb_fpwrap_cdouble_zeta (C function), 731
arb_fpwrap_cdouble_zeta_zero (C function), 732
arb_fpwrap_cdouble_acos (C function), 730
arb_fpwrap_cdouble_acosh (C function), 730
arb_fpwrap_cdouble_agm (C function), 735
arb_fpwrap_cdouble_airy_ai (C function), 733
arb_fpwrap_cdouble_airy_ai_prime (C function), 733
arb_fpwrap_cdouble_airy_ai_prime_zero (C function), 733
arb_fpwrap_cdouble_airy_ai_zero (C function), 733
arb_fpwrap_cdouble_airy_bi (C function), 733
arb_fpwrap_cdouble_airy_bi_prime (C function), 733
arb_fpwrap_cdouble_airy_bi_prime_zero (C function), 733
arb_fpwrap_cdouble_airy_bi_zero (C function), 733
arb_fpwrap_cdouble_asin (C function), 730
arb_fpwrap_cdouble_asinh (C function), 730
arb_fpwrap_cdouble_atan (C function), 730
arb_fpwrap_cdouble_atan2 (C function), 730
arb.fpwrap_cdouble_atanh (C function), 730
arb_fpwrap_cdouble_bessel_i (C function), 733
arb_fpwrap_cdouble_bessel_j (C function), 733
arb_fpwrap_cdouble_bessel_k (C function), 733
arb_fpwrap_cdouble_bessel_k_scaled (C function), 733
arb_fpwrap_cdouble_bessel_y (C function), 733
arb_fpwrap_cdouble_beta_lower (C function), 732
arb_fpwrap_cdouble_beta_upper (C function), 732
arb.fpwrap_cdouble_cbrt (C function), 729
arb_fpwrap_cdouble_chebyshev_t (C function), 734
arb_fpwrap_cdouble_chebyshev_u (C function), 734
arb_fpwrap_cdouble_cos (C function), 730
arb_fpwrap_cdouble_cos_integral (C function), 732
arb_fpwrap_cdouble_cos_pi (C function), 730
arb_fpwrap_cdouble_cosh (C function), 730
arb.fpwrap_cdouble_dilog (C function), 733
arb_fpwrap_cdouble_digamma (C function), 732
arb.fpwrap_cdouble_erfc (C function), 732
arb.fpwrap_cdouble_erfcinv (C function), 732
arb.fpwrap_cdouble_erf (C function), 732
arb.fpwrap_cdouble_erfci (C function), 732
arb.fpwrap_cdouble_erfcinv (C function), 732
arb.fpwrap_cdouble_erfinv (C function), 732
arb_fpwrap_cdouble_exp (C function), 729
arb_fpwrap_cdouble_exp_integral_e (C function), 732
arb_fpwrap_cdouble_exp_integral_ei (C function), 732
arb.fpwrap_cdouble_exp01 (C function), 729
arb.fpwrap_cdouble_exp1 (C function), 729
arb.fpwrap_cdouble_fresnel_c (C function), 732
arb.fpwrap_cdouble_fresnel_s (C function), 732
arb.fpwrap_cdouble_gamma (C function), 731
arb.fpwrap_cdouble_gamma_lower (C function), 732
arb.fpwrap_cdouble_gamma_upper (C function), 732
arb.fpwrap_cdouble_gegenbauer_c (C function), 734
arb.fpwrap_cdouble_hermite_h (C function), 734
arb.fpwrap_cdouble_hurtwitz_zeta (C function), 731
arb.fpwrap_cdouble_hypgeom_0f1 (C function), 735
arb_fpwrap_double_hypgeom_1f1 (C function), 735
arb_fpwrap_double_hypgeom_2f1 (C function), 735
arb_fpwrap_double_hypgeom_pfq (C function), 735
arb_fpwrap_double_hypgeom_u (C function), 735
arb_fpwrap_double_jacobi_p (C function), 734
arb_fpwrap_double_laguerre_l (C function), 734
arb_fpwrap_double_lambertw (C function), 730
arb_fpwrap_double_legendre_p (C function), 730
arb_fpwrap_double_sinh_integral (C function), 730
arb_fpwrap_double_sqrt (C function), 729
arb_fpwrap_double_zeta (C function), 722
arb_get_abs_lbound_arf (C function), 542
arb_get_abs_ubound_arf (C function), 542
arb_get_int_arf (C function), 542
arb_get_mid_arb (C function), 541
arb_get_rad_arb (C function), 541
arb_get_rand_fmpz (C function), 541
arb_get_str (C function), 538
arb_get_ubound_arf (C function), 541
arb_get_unique_fmpz (C function), 543
arb_gt (C function), 545
arb_hurwitz_zeta (C function), 555
arb_hypgeom_0f1 (C function), 657
arb_hypgeom_1f1 (C function), 657
arb_hypgeom_1f1_integration (C function), 657
arb_hypgeom_2f1 (C function), 657
arb_hypgeom_2f1_integration (C function), 657
arb_hypgeom_airy (C function), 662
arb_hypgeom_airy_jet (C function), 662
arb_hypgeom_airy_series (C function), 662
arb_hypgeom_airy_zero (C function), 662
arb_hypgeom_bessel_i (C function), 661
arb_hypgeom_bessel_i_integration (C function), 661
arb_hypgeom_bessel_i_scaled (C function), 661
arb_hypgeom_bessel_j (C function), 661
arb_hypgeom_bessel_jy (C function), 661
arb_hypgeom_bessel_k (C function), 661
arb_hypgeom_bessel_k_integration (C function), 661
arb_hypgeom_bessel_k_scaled (C function), 661
arb_hypgeom_bessel_y (C function), 661
arb_hypgeom_beta_lower (C function), 659
arb_hypgeom_beta_lower_series (C function), 659
arb_hypgeom_central_bin_ui (C function), 656
arb_hypgeom_chebyshev_t (C function), 663
arb_hypgeom_chebyshev_u (C function), 663
arb_hypgeom_chi (C function), 661
arb_hypgeom_chiseries (C function), 661
arb_hypgeom_ci (C function), 660
arb_hypgeom_ciseries (C function), 661
arb_hypgeom_coulomb (C function), 662
arb_hypgeom_coulomb_jet (C function), 662
arb_hypgeom_coulomb_series (C function), 662
arb_hypgeom_dilog (C function), 664
arb_hypgeom_ei (C function), 660
arb_hypgeom_eiseries (C function), 660
arb_hypgeom_erf (C function), 658
arb_hypgeom_ericf (C function), 658
arb_hypgeom_ericfseries (C function), 658
arb_hypgeom_ericvcinv (C function), 658
arb_hypgeom_ericv (C function), 658
arb_hypgeom_ericvseries (C function), 658
arb_hypgeom_ericvinv (C function), 658
arb_hypgeom_expint (C function), 660
arb_hypgeom_fresnel (C function), 658
arb_hypgeom_fresnels (C function), 658
arb_hypgeom_gamma (C function), 656
arb_hypgeom_gamma_fmpq (C function), 656
arb_hypgeom_gamma_fmpz (C function), 656
arb_hypgeom_gamma_lower (C function), 659
arb_hypgeom_gamma_lower_series (C function), 659
arb_hypgeom_gamma_stirling (C function), 656
arb_hypgeom_gamma_stirling_sum_horner (C function), 656
arb_hypgeom_gamma_stirling_sum_improved (C function), 656
arb_hypgeom_gamma_taylor (C function), 656
arb_hypgeom_gamma_upper (C function), 658
arb_hypgeom_gamma_upper_integration (C function), 658
arb_hypgeom_gamma_upper_series (C function), 658
arb_hypgeom_gegenbauer_c (C function), 663
arb_hypgeom_hermite_h (C function), 663
arb_hypgeom_infsup (C function), 719
arb_hypgeom_jacobi_p (C function), 663
arb_hypgeom_laguerre_l (C function), 663
arb_hypgeom_legendre_p (C function), 663
arb_hypgeom_legendre_p_ui (C function), 663
arb_hypgeom_legendre_p_ui_asympt (C function), 663
arb_hypgeom_legendre_p_ui_deriv_bound (C function), 663
arb_hypgeom_legendre_p_ui_one (C function), 663
arb_hypgeom_legendre_p_ui_rec (C function), 663
arb_hypgeom_legendre_p_ui_root (C function), 663
arb_hypgeom_legendre_p_ui_zero (C function), 663
arb_hypgeom_legendre_q (C function), 663
arb_hypgeom_igamma (C function), 656
arb_hypgeom_li (C function), 661
arb_hypgeom_li_series (C function), 661
arb_hypgeom_m (C function), 657
arb_hypgeom_pdf (C function), 656
arb_hypgeom_rgamma (C function), 656
arb_hypgeom_rising (C function), 655
arb_hypgeom_rising_ui (C function), 655
arb_hypgeom_rising_ui_bs (C function), 655
arb_hypgeom_rising_ui_forward (C function), 655
arb_hypgeom_rising_ui_jet (C function), 655
arb_hypgeom_rising_ui_jet_bs (C function), 655
arb_hypgeom_rising_ui_jet_powsum (C function), 655
arb_hypgeom_rising_ui_jet_rs (C function), 655
arb_hypgeom_rising_ui_rec (C function), 655
arb_hypgeom_rising_ui_rs (C function), 655
arb_hypgeom_shi (C function), 661
arb_hypgeom_shi_series (C function), 661
arb_hypgeom_si (C function), 660
arb_hypgeom_si_series (C function), 660
arb_hypgeom_sum (C function), 719
arb_hypgeom_sum_fmpq_arb (C function), 664
arb_hypgeom_sum_fmpq_arb_forward (C function), 664
arb_hypgeom_sum_fmpq_arb_rs (C function), 664
arb_hypgeom_sum_fmpq_imag_arb (C function), 664
arb_hypgeom_sum_fmpq_imag_arb_bs (C function), 664
arb_hypgeom_sum_fmpq_imag_arb_forward (C function), 664
arb_hypgeom_sum_fmpq_imag_arb_rs (C function), 664
arb_hypgeom_u (C function), 657
arb_hypgeom_u_integration (C function), 657
hypot (C function), 549
indeterminate (C function), 539
init (C function), 537
intersection (C function), 541
inv (C function), 547
ia_exact (C function), 544
iafinite (C function), 544
iaint (C function), 544
ia2exp (C function), 544
ianegative (C function), 544
ianonnegative (C function), 544
ianonpositive (C function), 544
ianonzero (C function), 544
iaone (C function), 544
iap (C function), 544
iazero (C function), 544
lambertw (C function), 553
le (C function), 545
lgamma (C function), 554
load_file (C function), 540
load_str (C function), 540
log (C function), 550
log1 (C function), 550
log (C function), 550
log (C function), 550
log Newton (C function), 550
LOG_PRIME_CACHE_NUM (C macro), 559
log_primes_vec_bsplit (C function), 559
LOG_REDUCTION_DEFAULT_MAXPREC (C macro), 559
log ui (C function), 550
log ui from prev (C function), 550
lt (C function), 545
mat_add (C function), 620
mat_add_error_mag (C function), 627
mat_allocated_bytes (C function), 617
mat_approx_inv (C function), 624
mat_approx lu (C function), 624
arb_poly_pow_ui_trunc_binexp (C function), 588
arb_poly_printd (C function), 580
arb_poly_product_roots (C function), 585
arb_poly_product_roots_complex (C function), 585
arb_poly_randtest (C function), 580
arb_poly_revert_series (C function), 591
arb_poly_riemann_siegel_theta_series (C function), 592
arb_poly_riemann_siegel_z_series (C function), 592
arb_poly_rising_ui_series (C function), 591
arb_poly_root_bound_fujiwara (C function), 593
arb_poly_rsqrt_series (C function), 588
arb_poly_scalar_div (C function), 581
arb_poly_scalar_mul (C function), 581
arb_poly_scalar_mul_2exp_si (C function), 581
arb_poly_set (C function), 579
arb_poly_set_coeff_arb (C function), 579
arb_poly_set_coeff_si (C function), 579
arb_poly_set_fmpq_poly (C function), 580
arb_poly_set_fmpz_poly (C function), 580
arb_poly_set_round (C function), 579
arb_poly_set_si (C function), 580
arb_poly_set_trunc (C function), 579
arb_poly_set_trunc_round (C function), 579
arb_poly_shift_left (C function), 579
arb_poly_shift_right (C function), 579
arb_poly_sin_cos_pi_series (C function), 590
arb_poly_sin_cos_series (C function), 589
arb_poly_sin_pi_series (C function), 590
arb_poly_sinc_pi_series (C function), 591
arb_poly_sinc_series (C function), 591
arb_poly_sin_cosh_series (C function), 590
arb_poly_sin_cosh_series_basecase (C function), 590
arb_poly_sin_cosh_series_exponential (C function), 590
arb_poly_sin_series (C function), 590
arb_poly_sqrt_series (C function), 588
arb_poly_struct (C type), 578
arb_poly_sub (C function), 581
arb_poly_sub_series (C function), 581
arb_poly_swinerton_dyer_ui (C function), 593
arb_poly_t (C type), 578
arb_poly_tan_series (C function), 590
arb_poly_taylor_shift (C function), 583
arb_poly_truncate (C function), 579
arb_poly_valueation (C function), 579
arb_poly_zero (C function), 579
arb_poly_zeta_series (C function), 592
arb_polylog (C function), 556
arb_polylog_si (C function), 556
arb_pos_inf (C function), 539
arb_pow (C function), 549
arb_pow_fmpq (C function), 549
arb_pow_fmpz (C function), 549
arb_pow_fmpz_binexp (C function), 549
arb_pow_ui (C function), 549
arb_power_sum_vec (C function), 555
arb_primorial_nth_ui (C function), 557
arb_primorial_ui (C function), 557
arb_print (C function), 539
arb_printd (C function), 539
arb_printn (C function), 539
arb_ptr (C type), 537
arb_radref (C macro), 537
arb_randtest (C function), 540
arb_randtest_exact (C function), 540
arb_randtest_positive (C function), 540
arb_randtest_precise (C function), 540
arb_randtest_special (C function), 541
arb_randtest_wide (C function), 540
arb_rel_accuracy_bits (C function), 542
arb_rel_error_bits (C function), 542
arb_rel_one_accuracy_bits (C function), 542
arb_rgamma (C function), 554
arb_rising (C function), 553
arb_rising2_ui (C function), 553
arb_rising_fmpq_ui (C function), 553
arb_rising_ui (C function), 553
arb_root (C function), 549
arb_root_ui (C function), 549
arb_rsqrt (C function), 549
arb_rsqrt_ui (C function), 549
arb_sec (C function), 551
arb_sech (C function), 552
arb_set (C function), 538
arb_set_arf (C function), 538
arb_set_d (C function), 538
arb_set_fmpq (C function), 538
arb_set_fmpz (C function), 538
arb_set_fmpz_2exp (C function), 538
arb_set_interval_arf (C function), 542
arb_set_interval_mag (C function), 542
arb_set_interval_mpf (C function), 542
arb_set_interval_neg_pos_mag (C function), 542
arb_set_round (C function), 538
arb_set_round_fmpz (C function), 538
arb_set_round_fmpz_2exp (C function), 538
arb_set_str (C function), 538
arb_set_ui (C function), 538
arb_sgn (C function), 545
arb_sgn_nonzero (C function), 546
arb_si_pow_ui (C function), 549
arb_sin (C function), 550
arb_sin_arf_atan_reduction (C function), 560
arb_sin_cosh_series_exponential (C function), 590
arb_sin_arf_bb (C function), 559
arb_sin_cos_arf_generic (C function), 559
arb_sin_cos_generic (C function), 559
arb_sin_cos_pi (C function), 550
arb_sin_cos_pi_fmpq (C function), 551
arb_sin_cos_wide (C function), 559
arb_sin_pi (C function), 550
arb_sin_pi_fmpq (C function), 551
arb_sinc (C function), 551
arb_sinc_pi (C function), 551
arb_sinh (C function), 552
arb_sinh_cosh (C function), 552
arb_sqrt (C function), 549
arb_sqrtlpm1 (C function), 549
arb_sqrtarf (C function), 548
arb_sqrtfmpz (C function), 548
arb_sqrtui (C function), 548
arb_sqrtropos (C function), 549
arb_srscrepr (C type), 537
arb_struct (C type), 537
arb_sub (C function), 546
arb_subarf (C function), 546
arb_subfmpz (C function), 546
arb_subsi (C function), 546
arb_subui (C function), 546
arb_submul (C function), 547
arb_submularf (C function), 547
arb_submulfmpz (C function), 547
arb_submulsi (C function), 547
arb_submului (C function), 547
arb_swap (C function), 537
arb_t (C type), 537
arb_tan (C function), 550
arb_tanpi (C function), 551
arb_tanh (C function), 552
arb_trim (C function), 543
arb_trunc (C function), 543
arb_ui_div (C function), 547
arb ui_pow ui (C function), 549
arb_union (C function), 541
arb_unit_interval (C function), 539
arb_urandom (C function), 541
arb_zero (C function), 539
arb_zero_pm_inf (C function), 539
arb_zero_pm_one (C function), 539
arb_zeta (C function), 555
arb_zeta ui (C function), 555
arb_zeta ui asympt (C function), 554
arb_zeta ui bernoulli (C function), 554
arb_zeta ui bernoulli_bsp (C function), 554
arb_zeta ui euler_bsp (C function), 554
arb_zeta ui vec (C function), 554
arb_zeta ui vec_bsp (C function), 554
arb_zeta ui vec for Euler (C function), 554
arb_zeta ui vec even (C function), 554
arb_zeta ui vec odd (C function), 554
arf abs (C function), 531
arf abs_bound le 2exp fmpz (C function), 529
arf_abs_bound_lt 2exp_fmpz (C function), 529
arf_abs_bound_lt 2exp_fmpz (C function), 529
arf_add (C function), 531
arf_add fmpz (C function), 531
arf_add fmpz 2exp (C function), 531
arf_add si (C function), 531
arf_add ui (C function), 531
arf_addmul (C function), 532
arf_addmul fmpz (C function), 532
arf_addmul mpz (C function), 532
arf_addmul si (C function), 532
arf_addmul ui (C function), 532
arf_allocated_bytes (C function), 526
arf_approx dot (C function), 533
arf_bits (C function), 529
arf ceil (C function), 528
arf clear (C function), 526
arf cmp (C function), 528
arf cmp 2exp si (C function), 529
arf cmp d (C function), 529
arf cmp si (C function), 528
arf cmp ui (C function), 528
arf cmpabs (C function), 529
arf cmpabs 2exp si (C function), 529
arf cmpabs d (C function), 529
arf cmpabs mag (C function), 529
arf cmpabs ui (C function), 529
arf_complex_mul (C function), 534
arf_complex_mul fallback (C function), 534
arf_complex sqrt (C function), 534
arf debug (C function), 531
arf div (C function), 533
arf div fmpz (C function), 533
arf div si (C function), 533
arf div ui (C function), 533
arf dump file (C function), 531
arf dump str (C function), 531
arf equal (C function), 528
arf equal d (C function), 528
arf equal si (C function), 528
arf equal ui (C function), 528
arf floor (C function), 528
arf fma (C function), 532
arf fmpz div (C function), 533
arf fmpz div fmpz (C function), 533
arf fprintf (C function), 531
arf fprintfd (C function), 531
arf frexp (C function), 527
arf get d (C function), 527
arf get fmpq (C function), 528
arf get fmpz (C function), 528
arf get fmpz 2exp (C function), 527
arf get fmpz fixed fmpz (C function), 528
arf get fmpz fixed si (C function), 528
arf get mag (C function), 529
arf get mag lower (C function), 529
arf get mpfr (C function), 527
arf get si (C function), 528
arf get str (C function), 531
arith_bell_number_nmod_vec_series (C function), 254
arith_bell_number_size (C function), 254
arith_bell_number_vec (C function), 254
arith_bell_number_vec_multi_mod (C function), 254
arith_bell_number_vec_recursive (C function), 254
arith_bernoulli_number (C function), 255
arith_bernoulli_number_denom (C function), 255
arith_bernoulli_number_size (C function), 255
arith_bernoulli_number_vec (C function), 255
arith_bernoulli_polynomial (C function), 255
arith_dedekind_sum_coprime (C function), 257
arith_dedekind_sum_coprime_d (C function), 257
arith_dedekind_sum_coprime_large (C function), 257
arith_dedekind_sum_naive (C function), 257
arith_divisor_sigma (C function), 256
arith_divisors (C function), 256
arith_euler_number (C function), 256
arith_euler_number_size (C function), 256
arith_euler_number_vec (C function), 256
arith_euler_phi (C function), 256
arith_euler_polynomial (C function), 256
arith_eulerian_polynomial (C function), 214
arith_harmonic_number (C function), 252
arith_hrr_expsum_factoried (C function), 257
arith_landau_function_vec (C function), 257
arith_moebius_mu (C function), 256
number_of_partitions (C function), 258
number_of_partitions_mpfr (C function), 258
number_of_partitions_nmod_vec (C function), 257
number_of_partitions_vec (C function), 257
arith_primorial (C function), 252
arith_ramanujan_tau (C function), 256
arith_ramanujan_tau_series (C function), 256
arith_stirling_matrix_1 (C function), 253
arith_stirling_matrix_1lu (C function), 253
arith_stirling_matrix_2 (C function), 253
arith_stirling_number_1 (C function), 253
arith_stirling_number_1_vec (C function), 253
arith_stirling_number_1vec (C function), 253
arith_stirling_number_1_vec_next (C function), 253
arith_stirling_number_1u (C function), 253
arith_stirling_number_1u_vec (C function), 253
arith_stirling_number_1u_vec_next (C function), 253
arith_stirling_number_2 (C function), 253
arith_stirling_number_2_vec (C function), 253
arith_stirling_number_2vec (C function), 253
arith_stirling_number_2_vec_next (C function), 253
arith_sum_of_squares (C function), 259
arith_sum_of_squares_vec (C function), 259
bernoulli_bound_2exp_si (C function), 716
bernoulli_cache (C var), 716
bernoulli_cache_compute (C function), 716
bernoulli_cache_num (C var), 716
bernoulli_fmpq_ui (C function), 716
bernoulli_fmpq_vec_no_cache (C function), 716
bernoulli_mod_p_harvey (C function), 716
bernoulli_rev_clear (C function), 715
bernoulli_rev_init (C function), 715
bernoulli_rev_next (C function), 715
bernoulli_rev_t (C type), 715
BernoulliB (C macro), 828
BernoulliPolynomial (C macro), 828
BernsteinEllipse (C macro), 822
BesselI (C macro), 830
BesselJ (C macro), 830
BesselJZero (C macro), 830
BesselK (C macro), 830
BesselY (C macro), 830
BesselYZero (C macro), 830
BetaFunction (C macro), 829
Binomial (C macro), 829
bool_mat_add (C function), 469
bool_mat_all (C function), 468
bool_mat_all_pairs_longest_walk (C function), 469
bool_mat_any (C function), 468
bool_mat_clear (C function), 467
bool_mat_complement (C function), 469
bool_mat_directed_cycle (C function), 468
bool_mat_directed_path (C function), 468
bool_mat_equal (C function), 468
bool_mat_fprint (C function), 468
bool_mat_get_entry (C function), 467
bool_mat_get_strongly_connected_components (C function), 469
bool_mat_init (C function), 467
bool_mat_is_diagonal (C function), 468
bool_mat_is_empty (C function), 467
bool_mat_is_lower_triangular (C function), 468
bool_mat_is_nilpotent (C function), 468
CA_OPT_GROEBNER_POLY_BITS_LIMIT (C macro), 782
CA_OPT_GROEBNER_POLY_LENGTH_LIMIT (C macro), 782
CA_OPT_LLL_PREC (C macro), 782
CA_OPT_LOW_PREC (C macro), 782
CA_OPT_MPOLL_ORD (C macro), 781
CA_OPT_POW_LIMIT (C macro), 782
CA_OPT_PREC_LIMIT (C macro), 782
CA_OPT_RND)==' (C), 782
CA_OPT_RND)<< (C), 782
CA_OPT_SMOOTH_LIMIT (C macro), 782
CA_OPT_TRIG_FORM.CA_TRIG_DIRECT (C macro), 782
CA_OPT_TRIG_FORM.CA_TRIG_EXPONENTIAL (C macro), 782
CA_OPT_TRIG_FORM.CA_TRIG_SINE_COSINE (C macro), 782
CA_OPT_TRIG_FORM.CA_TRIG_TANGENT (C macro), 782
CA_OPT_USE_GROEBNER (C macro), 782
CA_OPT_VERBOSE (C macro), 782
CA_PRINT_DEFAULT (C macro), 787
CA_PRINT_DEBUG (C macro), 787
CA_PRINT_DIGITS (C macro), 787
CA_PRINT_FIELD (C macro), 787
ca_poly_add (C function), 789
cal_poly_check_is_zero (C function), 789
cal_poly_check_is_one (C function), 789
cal_poly_div (C function), 789
cal_poly_derivative (C function), 789
cal_poly_div_series (C function), 791
cal_poly_divrem (C function), 790
cal_poly_divrem_basecase (C function), 790
cal_poly_evaluate (C function), 790
cal_poly_evaluate_horner (C function), 790
cal_poly_exp_series (C function), 791
cal_poly_factor_squarefree (C function), 792
cal_poly_fit_length (C function), 787
cal_poly_gcd (C function), 791
cal_poly_gcd_euclidean (C function), 791
cal_poly_init (C function), 787
cal_poly_integral (C function), 791
cal_poly_inv_series (C function), 791
cal_poly_is_proper (C function), 789
cal_poly_log_series (C function), 791
cal_poly_make_monic (C function), 789
cal_poly_mul (C function), 789
cal_poly_mul_ca (C function), 790
cal_poly_mullow (C function), 790
cal_poly_neg (C function), 789
cal_poly_one (C function), 788
cal_poly_pow_ui (C function), 790
cal_poly_pow_ui_trunc (C function), 790
cal_poly_print (C function), 788
cal_poly_printn (C function), 788
cal_poly_ranndtest (C function), 788
cal_poly_ranndtest_rational (C function), 788
cal_poly_remi (C function), 790
cal_poly_reverse (C function), 789
cal_poly_roots (C function), 792
cal_poly_set (C function), 788
cal_poly_set_ca (C function), 788
cal_poly_set_coeff_ca (C function), 792
cal_poly_set_fmpz_poly (C function), 788
cal_poly_set_fmpq_poly (C function), 788
cal_poly_set_roots (C function), 792
cal_poly_set_si (C function), 788
cal_poly_shift_left (C function), 789
cal_poly_shift_right (C function), 789
cal_poly_squarefree_part (C function), 792
cal_poly_struct (C type), 787
cal_poly_sub (C function), 789
cal_poly_t (C type), 787
cal_poly_transfer (C function), 788
cal_poly_vec_append (C function), 792
cal_poly_vec_clear (C function), 792
cal_poly_vec_init (C function), 792
cal_poly_vec_set_length (C function), 792
cal_poly_vec_struct (C type), 792
cal_poly_vec_t (C type), 792
cal polyline (C function), 788
cal pow (C function), 788
cal pow_fmpq (C function), 775
cal pow_fmpz (C function), 775
cal pow_si (C function), 775
cal pow_si_arithmetic (C function), 775
cal pow ui (C function), 775
cal print (C function), 767
CA_PRINT_DEBUG (C macro), 766
CA_PRINT_DEFAULT (C macro), 766
CA_PRINT_DIGITS (C macro), 766
CA_PRINT_FIELD (C macro), 766
CA_PRINT_N (C macro), 766
CA_PRINT_REPR (C macro), 766
ca_print (C function), 767
cal_printn (C function), 767
ca_ptr (C type), 765
ca_ranndtest (C function), 769
cal_ranndtest_rational (C function), 769
cal_ranndtest_same_nf (C function), 769
cal_ranndtest_special (C function), 769
ca re (C function), 776
cal rewrite_complex_normal_form (C function), 780
ca_set (C function), 768
cal_set d (C function), 768
ca_set d d (C function), 768
ca_set_fexpr (C function), 766
ca_set_fmpq (C function), 768
Flint Documentation, Release 3.1.0-dev

Index 1159

ca_set_fmpz (C function), 768
ca_set_qbar (C function), 769
ca_set_si (C function), 768
ca_set_ui (C function), 768
ca_sgn (C function), 775
ca_s_i_div (C function), 773
ca_s_i_sub (C function), 773
ca_sin (C function), 778
ca_sin_cos (C function), 777
ca_sin_cos_direct (C function), 777
ca_sin_cos_exponential (C function), 777
ca_sin_cos_tangent (C function), 777
ca_sqr (C function), 775
ca_sqrt (C function), 775
ca_sqrt_factor (C function), 775
ca_sqrt_inert (C function), 777
ca_sqrt_nofactor (C function), 777
ca_sqrt_necessary (C function), 777
ca_swap (C function), 766
ca_t (C type), 765
calculator (C macro), 826
CallIndeterminate (C macro), 826
Cardinality (C macro), 819
CarlsonHypergeometricR (C macro), 832
CarlsonHypergeometricT (C macro), 832
CarlsonRC (C macro), 832
CarlsonRD (C macro), 832
CarlsonRF (C macro), 832
CarlsonRJ (C macro), 832
CartesianPower (C macro), 819
CartesianProduct (C macro), 819
Case (C macro), 818
Cases (C macro), 818
CatalanConstant (C macro), 819
CC (C macro), 821
Ceil (C macro), 826
Characteristic (C macro), 825
ChebyshevT (C macro), 829
ChebyshevU (C macro), 829
ClosedComplexDisk (C macro), 822
ClosedOpenInterval (C macro), 821
COEFF_IS_MPZ (C macro), 122
COEFF_MAX (C macro), 122
COEFF_MIN (C macro), 122
COEFF_TQ_PTR (C function), 122
Coefficient (C macro), 825
Column (C macro), 825
ColumnMatrix (C macro), 825
CommutativeRings (C macro), 826
complex_double (C type), 729
ComplexBranchDerivative (C macro), 824
ComplexDerivative (C macro), 824
ComplexInfinities (C macro), 822
ComplexLimit (C macro), 824
ComplexSignedInfinities (C macro), 822
ComplexSingularityClosure (C macro), 823
ComplexZeroMultiplicity (C macro), 824
Concatenation (C macro), 819
CongruentMod (C macro), 826
Conjugate (C macro), 826
ConreyGenerator (C macro), 832
Cos (C macro), 827
Cosh (C macro), 827
CoshIntegral (C macro), 830
CosIntegral (C macro), 830
Cot (C macro), 827
Coth (C macro), 827
CoulombC (C macro), 831
CoulombF (C macro), 830
CoulombG (C macro), 830
CoulombH (C macro), 830

ca_vec_zero (C function), 729
calcium_write (C function), 763
calcium_write_acb (C function), 763
calcium_write_arb (C function), 763
calcium_write_fmpz (C function), 763
calcium_write_free (C function), 763
calcium_write_si (C function), 763
C function, 768
C function, 768
C function, 768
C function, 775
C function, 773
C function, 773
C function, 773
C function, 775
C function, 775
C function, 775
C function, 775
C function, 778
C function, 778
C function, 778
C function, 778
C function, 768
C function, 785
C function, 784
C function, 784
C function, 784
C function, 785
C function, 785
C function, 785
C function, 784
C function, 784
C function, 784
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
C function, 785
CoulombSigma (C macro), 831
Csc (C macro), 827
Csch (C macro), 827
Csgn (C macro), 826
CurvePath (C macro), 824
Cyclotomic (C macro), 828

D

d_is_nan (C function), 1011
d_lambertw (C function), 1011
d_log2 (C function), 1011
d_mat_approx_equal (C function), 1014
d_mat_clear (C function), 1013
d_mat_entry (C function), 1013
d_mat_entry_ptr (C function), 1013

d_mat_equal (C function), 1014
d_mat_get_entry (C function), 1013
d_mat_init (C function), 1013

d_mat_is_square (C function), 1014
d_mat_mul_classical (C function), 1014

d_mat_print (C function), 1014
d_mat_randtest (C function), 1014

d_mat_set (C function), 1013

d_mat_swap_entrywise (C function), 1013

d_mat_transpose (C function), 1014

d_mat_zero (C function), 1013
d_polyval (C function), 1011

d_randtest (C function), 1011

d_randtest_signed (C function), 1011

d_randtest_special (C function), 1011

Decimal (C macro), 820

DedekindEta (C macro), 832
DedekindEtaEpsilon (C macro), 833

DedekindSun (C macro), 833

Def (C macro), 817

Derivative (C macro), 824

Det (C macro), 825

DiagonalMatrix (C macro), 825

DigammaFunction (C macro), 829

DigammaFunctionZero (C macro), 829

dirichlet_char_clear (C function), 460

dirichlet_char_eq (C function), 461

dirichlet_char_eq_deep (C function), 461

dirichlet_char_exp (C function), 461

dirichlet_char_first_primitive (C function), 460

dirichlet_char_index (C function), 461

dirichlet_char_init (C function), 460

dirichlet_char_is_primitive (C function), 461

dirichlet_char_is_principal (C function), 461

dirichlet_char_is_real (C function), 461

dirichlet_char_lift (C function), 462

dirichlet_char_log (C function), 460

dirichlet_char_lower (C function), 462

dirichlet_char_mul (C function), 462

dirichlet_char_next (C function), 460

dirichlet_char_next_primitive (C function), 461

dirichlet_char_one (C function), 460

dirichlet_char_pow (C function), 462

dirichlet_char_print (C function), 460

dirichlet_char_set (C function), 460

dirichlet_char_struct (C type), 460

dirichlet_char_t (C type), 460

dirichlet_chi (C function), 462

dirichlet_chi_vec (C function), 462

dirichlet_chi_vec_order (C function), 462

dirichlet_conductor_char (C function), 461

dirichlet_conductor_ui (C function), 461

dirichlet_group_clear (C function), 459

dirichlet_group_clear (C function), 460

dirichlet_group_dlog_precompute (C function), 460

dirichlet_group_init (C function), 459

dirichlet_group_num_primitive (C function), 460

dirichlet_group_size (C function), 459

dirichlet_group_struct (C type), 459

dirichlet_group_t (C type), 459

dirichlet_index_char (C function), 461

dirichlet_index_char (C function), 461

dirichlet_index_ui (C function), 461

dirichlet_index_ui (C function), 462

dirichlet_index_ui (C function), 462

dirichlet_domain_char (C function), 461

dirichlet_domain_ui (C function), 461

dirichlet_domain_ui (C function), 459

DirichletCharacter (C macro), 832

DirichletGroup (C macro), 832

DirichletL (C macro), 831

DirichletLambda (C macro), 831

DirichletZero (C macro), 831

DiscreteLog (C macro), 827

Div (C macro), 820

Divides (C macro), 826

DivisorProduct (C macro), 823

DivisorSigma (C macro), 827

DivisorSum (C macro), 823

dlog_bsgs (C function), 465

dlog_bsgs_clear (C function), 465

dlog_bsgs_init (C function), 465

dlog_bsgs_struct (C type), 465

dlog_bsgs_t (C type), 465

dlog_crt (C function), 466

dlog_crt_clear (C function), 466

dlog_crt_init (C function), 466

dlog_crt_struct (C type), 466

dlog_crt_t (C type), 466

dlog_modpe (C function), 465

dlog_modpe_clear (C function), 465

dlog_modpe_init (C function), 465

dlog_modpe_struct (C type), 465

dlog_modpe_t (C type), 465

DLOG_NONE (C macro), 463

dlog_once (C function), 463

dlog_power (C function), 466
dlog_power_clear (C function), 466

dlog_power_init (C function), 466

dlog_power_struct (C type), 466

dlog_power_t (C type), 466

dlog_precomp (C function), 463

dlog_precomp_clear (C function), 463

dlog_precomp_modpe_init (C function), 463

dlog_precomp_n_init (C function), 463

dlog_precomp_p_init (C function), 463

dlog_precomp_small_init (C function), 463

dlog_precomp_struct (C type), 463

dlog_rho_clear (C function), 466

dlog_rho (C function), 466

dlog_rho_clear (C function), 466

dlog_rho_init (C function), 466

dlog_rho_t (C type), 466

dlog_rho_clear (C function), 466

dlog_rho_init (C function), 466

dlog_rho_t (C type), 466

dlog_table (C function), 465

dlog_table_clear (C function), 465

dlog_table_init (C function), 465

dlog_table_struct (C type), 465

dlog_table_t (C type), 465

dlog_vec (C function), 464

dlog_vec_add (C function), 464

dlog_vec_eratos (C function), 464

dlog_vec_eratos_add (C function), 464

dlog_vec_fill (C function), 464

dlog_vec_loop (C function), 464

dlog_vec_loop_add (C function), 464

dlog_vec_set_not_found (C function), 464

dlog_vec_sieve (C function), 464

dlog_vec_sieve_add (C function), 464

do_func_t (C type), 13

DoubleFactorial (C macro), 829

E

EisensteinE (C macro), 833

EisensteinE (C macro), 833

Element (C macro), 818

Ellipsis (C macro), 833

EllipticE (C macro), 832

EllipticK (C macro), 832

EllipticPi (C macro), 832

EllipticRootE (C macro), 833

Enclosure (C macro), 820

Equal (C macro), 817

EqualAndElement (C macro), 818

EqualNearestDecimal (C macro), 821

EqualQSeriesEllipsis (C macro), 826

Equivalent (C macro), 818

Erf (C macro), 830

Erfc (C macro), 830

Erfi (C macro), 830

Euler (C macro), 819

EulerE (C macro), 828

EulerPhi (C macro), 827

EulerPolynomial (C macro), 828

EulerQSeries (C macro), 833

Exists (C macro), 818

Exp (C macro), 827

ExpIntegralE (C macro), 830

ExpIntegralEi (C macro), 830

ExtendedRealNumbers (C macro), 822

F

Factorial (C macro), 829

FallingFactorial (C macro), 829

False (C macro), 817

fermat_to_mpz (C function), 260

expr_add (C function), 814

expr_allocated_bytes (C function), 810

expr_arg (C function), 812

expr_arithmetic_nodes (C function), 814

expr_builtin_length (C function), 816

expr_builtin_lookup (C function), 816

expr_builtin_name (C function), 816

expr_call0 (C function), 813

expr_call1 (C function), 813

expr_call2 (C function), 813

expr_call3 (C function), 813

expr_call4 (C function), 813

expr_call_builtin1 (C function), 813

expr_call_builtin2 (C function), 813

expr_call_vec (C function), 813

expr_clear (C function), 810

expr_cmp_fast (C function), 810

expr_contains (C function), 813

expr_depth (C function), 810

expr_div (C function), 814

expr_equal (C function), 810

expr_equal_si (C function), 810

expr_equal_ui (C function), 810

expr_expanded_normal_form (C function), 815

expr_fit_size (C function), 810

expr_func (C function), 812

expr_get_fmpz (C function), 811

expr_get_fmpz_mpoly_q (C function), 814

expr_get_str (C function), 812

expr_get_str_latex (C function), 812

expr_get_string (C function), 811

expr_get_symbol_str (C function), 811

expr_hash (C function), 810

expr_init (C function), 810

expr_is_any_builtin_call (C function), 813

expr_is_any_builtin_symbol (C function), 811

expr_is_arithmetic_operation (C function), 814

expr_is.atom (C function), 811

expr_is_builtin_call (C function), 813

expr_is_builtin_symbol (C function), 811

expr_is_integer (C function), 811

expr_is_neg_integer (C function), 811

expr_is_string (C function), 811

expr_is_symbol (C function), 811

expr_is_zero (C function), 811
fft_combine_limbs (C function), 259
fft_convolution (C function), 266
fft_convolution_precache (C function), 267
fft_mfa_truncate_sqrt2 (C function), 264
fft_mfa_truncate_sqrt2_inner (C function),
265
fft_mfa_truncate_sqrt2_outer (C function),
265
fft_mulmod_2exp (C function), 266
fft_naive_convo1 (C function), 265
fft_negacyclic (C function), 265
fft_precache (C function), 267
fft_radix2 (C function), 261
fft_radix2_twiddle (C function), 264
fft_split_bits (C function), 259
fft_split_limbs (C function), 259
fft_truncate (C function), 261
fft_truncate1 (C function), 261
fft_truncate1_twiddle (C function), 264
fft_truncate2 (C function), 263
Fibonacci (C macro), 828
FiniteField (C macro), 826
flint_free (C function), 16
flint_fscanf (C function), 18
flint_malloc (C function), 16
flint_mpn_divisible_1_odd (C function), 247
flint_mpn_divisible_1_odd (C function), 247
flint_mpn_gcd_full (C function), 246
flint_mpn_mod_p (C function), 246
flint_mpn_mul (C function), 244
fft_combine_limbs (C function), 259
fft_convolution (C function), 266
fft_convolution_precache (C function), 267
fft_mfa_truncate_sqrt2 (C function), 264
fft_mfa_truncate_sqrt2_inner (C function),
265
fft_mfa_truncate_sqrt2_outer (C function),
265
fft_mulmod_2exp (C function), 266
fft_naive_convo1 (C function), 265
fft_negacyclic (C function), 265
fft_precache (C function), 267
fft_radix2 (C function), 261
fft_radix2_twiddle (C function), 264
fft_split_bits (C function), 259
fft_split_limbs (C function), 259
fft_truncate (C function), 261
fft_truncate1 (C function), 261
fft_truncate1_twiddle (C function), 264
fft_truncate2 (C function), 263
Fibonacci (C macro), 828
FiniteField (C macro), 826
flint_free (C function), 16
flint_fscanf (C function), 18
flint_malloc (C function), 16
flint_mpn_divisible_1_odd (C function), 247
flint_mpn_divisible_1_odd (C function), 247
flint_mpn_gcd_full (C function), 246
flint_mpn_mod_p (C function), 246
flint_mpn_mul (C function), 244
fft_combine_limbs (C function), 259
fft_convolution (C function), 266
fft_convolution_precache (C function), 267
fft_mfa_truncate_sqrt2 (C function), 264
fft_mfa_truncate_sqrt2_inner (C function),
265
fft_mfa_truncate_sqrt2_outer (C function),
265
fft_mulmod_2exp (C function), 266
fft_naive_convo1 (C function), 265
fft_negacyclic (C function), 265
fft_precache (C function), 267
fft_radix2 (C function), 261
fft_radix2_twiddle (C function), 264
fft_split_bits (C function), 259
fft_split_limbs (C function), 259
fft_truncate (C function), 261
fft_truncate1 (C function), 261
fft_truncate1_twiddle (C function), 264
fft_truncate2 (C function), 263
Fibonacci (C macro), 828
FiniteField (C macro), 826
flint_free (C function), 16
flint_fscanf (C function), 18
flint_malloc (C function), 16
flint_mpn_divisible_1_odd (C function), 247
flint_mpn_divisible_1_odd (C function), 247
flint_mpn_gcd_full (C function), 246
flint_mpn_mod_p (C function), 246
flint_mpn_mul (C function), 244
fft_combine_limbs (C function), 259
fft_convolution (C function), 266
fft_convolution_precache (C function), 267
fft_mfa_truncate_sqrt2 (C function), 264
fft_mfa_truncate_sqrt2_inner (C function),
265
fft_mfa_truncate_sqrt2_outer (C function),
265
fft_mulmod_2exp (C function), 266
fft_naive_convo1 (C function), 265
fft_negacyclic (C function), 265
fft_precache (C function), 267
fft_radix2 (C function), 261
fft_radix2_twiddle (C function), 264
fft_split_bits (C function), 259
fft_split_limbs (C function), 259
fft_truncate (C function), 261
fft_truncate1 (C function), 261
fft_truncate1_twiddle (C function), 264
fft_truncate2 (C function), 263
Fibonacci (C macro), 828
FiniteField (C macro), 826
flint_free (C function), 16
flint_fscanf (C function), 18
flint_malloc (C function), 16
flint_mpn_divisible_1_odd (C function), 247
flint_mpn_divisible_1_odd (C function), 247
flint_mpn_gcd_full (C function), 246
flint_mpn_mod_p (C function), 246
flint_mpn_mul (C function), 244
fmpq_mpoly_set_coeff_fmpq_monomial (C function), 316
fmpq_mpoly_set_coeff_fmpq_ui (C function), 317
fmpq_mpoly_set_fmpq (C function), 315
fmpq_mpoly_set_fmpz (C function), 315
fmpq_mpoly_set_si (C function), 315
fmpq_mpoly_set_str_pretty (C function), 315
fmpq_mpoly_set_term_coeff_fmpq (C function), 318
fmpq_mpoly_set_term_exp_fmpz (C function), 318
fmpq_mpoly_set_term_exp_fmpz (C function), 318
fmpq_mpoly_set_term_exp_ui (C function), 318
fmpq_mpoly_set_ui (C function), 315
fmpq_mpoly_sort_terms (C function), 319
fmpq_mpoly_sqrt (C function), 323
fmpq_mpoly_struct (C type), 314
fmpq_mpoly_sub (C function), 320
fmpq_mpoly_sub_fmpz (C function), 320
fmpq_mpoly_sub_fmpq (C function), 320
fmpq_mpoly_sub_fmpz (C function), 320
fmpq_mpoly_sub_fmpq (C function), 320
fmpq_mpoly_sub_si (C function), 320
fmpq_mpoly_swap (C function), 315
fmpq_mpoly_t (C type), 314
fmpq_mpoly_term_content (C function), 322
fmpq_mpoly_term_exp_fits_si (C function), 318
fmpq_mpoly_term_exp_fits_ui (C function), 318
fmpq_mpoly_to_univar (C function), 323
fmpq_mpoly_total_degree_fits_si (C function), 316
fmpq_mpoly_total_degree_fmpz (C function), 316
fmpq_mpoly_total_degree_si (C function), 316
fmpq_mpoly_univar_clear (C function), 323
fmpq_mpoly_univar_degree_fits_si (C function), 324
fmpq_mpoly_univar_get_term_coeff (C function), 324
fmpq_mpoly_univar_get_term_exp_si (C function), 324
fmpq_mpoly_univar_init (C function), 323
fmpq_mpoly_univar_length (C function), 324
fmpq_mpoly_univar_swap (C function), 323
fmpq_mpoly_univar_swap_term_coeff (C function), 324
fmpq_mpoly_used_vars (C function), 316
fmpq_mpoly_zero (C function), 315
fmpq_mpoly_zpoly_ref (C function), 317
fmpq_mpoly_zpoly_term_coeff_ref (C function), 317
fmpq_mul (C function), 276
fmpq_mul_2exp (C function), 277
fmpq_mul_fmpz (C function), 277
fmpq_mul_si (C function), 276
fmpq_mul_ui (C function), 276
fmpq_neg (C function), 272
fmpq_next_calkin_wilf (C function), 278
fmpq_next_signed_calkin_wilf (C function), 278
fmpq_next_signed_minimal (C function), 278
fmpq_numref (C function), 271
fmpq_one (C function), 272
fmpq_poly_add (C function), 295
fmpq_poly_add_can (C function), 295
fmpq_poly_add_series (C function), 296
fmpq_poly_add_series_can (C function), 296
fmpq_poly_addmul (C function), 298
fmpq_poly_asin_series (C function), 305
fmpq_poly_asinh_series (C function), 305
fmpq_poly_atan_series (C function), 305
fmpq_poly_atanh_series (C function), 305
fmpq_poly Canonicalise (C function), 306
fmpq_poly_clear (C function), 291
fmpq_poly_cmp (C function), 295
fmpq_polyCompose (C function), 308
fmpq_poly_compose_series (C function), 309
fmpq_poly_compose_series_brent_kung (C function), 309
fmpq_poly_compose_series_horner (C function), 309
fmpq_poly_content (C function), 311
fmpq_poly_cos_series (C function), 305
fmpq_poly_cosh_series (C function), 306
fmpq_poly_degree (C function), 292
fmpq_poly_denref (C function), 292
fmpq_poly_derivative (C function), 303
fmpq_poly_div (C function), 299
fmpq_poly_div_series (C function), 301
fmpq_poly_divides (C function), 301
fmpq_poly_divrem (C function), 299
fmpq_poly_equal (C function), 295
fmpq_poly_equal_trunc (C function), 295
fmpq_poly_equal (C function), 307
fmpq_poly_exp (C function), 307
fmpq_poly_exp_series (C function), 307
fmpq_poly_exp_expinv_series (C function), 307
fmpq_poly_divides (C function), 304
fmpq_poly_fit_length (C function), 291
fmpq_poly_fprint (C function), 312
fmpq_poly_fprint_pretty (C function), 312
fmpq_poly_fread (C function), 312
fmpq_poly_gcd (C function), 320
fmpq_poly_gegenbauer_c (C function), 307
fmpq_poly_get_coeff_fmpq (C function), 294
fmpq_poly_get_coeff_fmpz (C function), 294
fmpq_poly_get_denominator (C function), 292
fmpq_poly_get_numerator (C function), 292
fmpq_poly_get_slice (C function), 294
fmpq_poly_get_str (C function), 293
fmpq_poly_get_str_pretty (C function), 293
fmpq_poly_init (C function), 291
fmpq_poly_init2 (C function), 291
fmpq_poly_integral (C function), 303
fmpq_poly_interpolate_fmpz_vec (C function), 308
fmpq_poly_inv (C function), 294
fmpq_poly_inv_series (C function), 301
fmpq_poly_inv_series_newton (C function), 301
fmpq_poly_invsqrt_series (C function), 303
fmpq_poly_is_canonical (C function), 291
fmpq_poly_is_gen (C function), 295
fmpq_poly_is_monic (C function), 311
fmpq_poly_is_one (C function), 295
fmpq_poly_is_squarefree (C function), 311
fmpq_poly_is_zero (C function), 295
fmpq_poly_laguerre_l (C function), 307
fmpq_poly_lcm (C function), 302
fmpq_poly_legendre_p (C function), 307
fmpq_poly_length (C function), 292
fmpq_poly_log_series (C function), 304
fmpq_poly_make_monic (C function), 311
fmpq_poly_mul (C function), 298
fmpq_poly_mullog (C function), 298
fmpq_poly_neg (C function), 294
fmpq_poly_nth_derivative (C function), 303
fmpq_poly_numref (C function), 292
fmpq_poly_one (C function), 294
fmpq_poly_pow (C function), 299
fmpq_poly_pow_trunc (C function), 299
fmpq_poly_power_sums (C function), 304
fmpq_poly_power_sums_to_fmpz_poly (C function), 304
fmpq_poly_power_sums_to_poly (C function), 304
fmpq_poly_powers_clear (C function), 300
fmpq_poly_powers_precompute (C function), 300
fmpq_poly_primitive_part (C function), 311
fmpq_poly_print (C function), 311
fmpq_poly_print_pretty (C function), 312
fmpq_poly_randtest (C function), 292
fmpq_poly_randtest_not_zero (C function), 292
fmpq_poly_randtest_unsaged (C function), 292
fmpq_poly_read (C function), 312
fmpq_poly_realloc (C function), 291
fmpq_poly_rem (C function), 300
fmpq_poly_rem_powers_precomp (C function), 300
fmpq_poly_remove (C function), 301
fmpq_poly_rescale (C function), 308
fmpq_poly_resultant (C function), 302
fmpq_poly_resultant_div (C function), 303
fmpq_poly_reverse (C function), 294
fmpq_poly_revert_series (C function), 310
fmpq_poly_revert_series_lagrange (C function), 310
fmpq_poly_revert_series_lagrange_fast (C function), 310
fmpq_poly_revert_series_newton (C function), 310
fmpq_poly_scalar_div_fmpz (C function), 298
fmpq_poly_scalar_div_ui (C function), 298
fmpq_poly_scalar_mul_fmpz (C function), 297
fmpq_poly_scalar_mul_fmpz (C function), 297
fmpq_poly_scalar_mul_si (C function), 297
fmpq_poly_scalar_mul_ui (C function), 297
fmpq_poly_set (C function), 293
fmpq_poly_set_coeff_fmpz (C function), 294
fmpq_poly_set_coeff_fmpq (C function), 294
fmpq_poly_set_coeff_si (C function), 294
fmpq_poly_set_coeff_ui (C function), 294
fmpq_poly_set_fmpz (C function), 293
fmpq_poly_set_fmpq (C function), 293
fmpq_poly_set_fmpz_poly (C function), 293
fmpq_poly_set_mmpoly (C function), 293
fmpq_poly_set_si (C function), 293
fmpq_poly_set_str (C function), 293
fmpq_poly_set_trunc (C function), 294
fmpq_poly_set_ui (C function), 293
fmpq_poly_shift_left (C function), 299
fmpq_poly_shift_right (C function), 299
fmpq_poly_sin_series (C function), 306
fmpq_poly_sinh_cosh_series (C function), 306
fmpq_poly_sin_series (C function), 306
fmpq_poly_sinh_series (C function), 306
fmpq_poly_sqrt_series (C function), 303
fmpq_poly_struct (C type), 290
fmpq_poly_sub (C function), 296
fmpq_poly_sub_can (C function), 296
fmpq_poly_sub_series (C function), 296
fmpq_poly_sub_series_can (C function), 296
fmpq_poly_submul (C function), 298
fmpq_poly_swap (C function), 294
fmpq_poly_t (C type), 290
fmpq_poly_tan_series (C function), 305
fmpq_poly_tanh_series (C function), 306
fmpq_poly_truncate (C function), 294
fmpq_poly_xgcd (C function), 302
fmpq_poly_zero (C function), 293
fmpq_pow_fmpz (C function), 277
fmpq_pow_si (C function), 277
fmpq_print (C function), 275
fmpq_randbits (C function), 275
fmpq_randtest (C function), 275
fmpq_randtest_not_zero (C function), 275
fmpq_reconstruct_fmpz (C function), 278
fmpq_reconstruct_fmpz_2 (C function), 278
fmpq_set (C function), 272
fmpq_set_cfrac (C function), 280
fmpq_set_fmpz_frac (C function), 273
fmpq_set_mmpoly (C function), 273
fmpq_set_si (C function), 273
fmpq_set_str (C function), 273
fmpq_set_ui (C function), 273
fmpq_sgn (C function), 273
fmpq_simplest_between (C function), 279
fmpq_sub (C function), 276
fmpq_sub_fmpz (C function), 276
fmpz_sub_si (C function), 276
fmpz_sub_ui (C function), 276
fmpz_submul (C function), 276
fmpz_swap (C function), 272
fmpz_t (C type), 271
fmpz_zero (C function), 272
fmpz (C type), 122
fmpz_abs (C function), 129
fmpz_abs_fits_ui (C function), 128
fmpz_abs_bound_ui_2exp (C function), 128
fmpz_abs_ubound_ui_2exp (C function), 129
fmpz_add (C function), 129
fmpz_add2_fmpz_si_inline (C function), 739
fmpz_add_inline (C function), 739
fmpz_add_si (C function), 129
fmpz_add_si_inline (C function), 739
fmpz_add_ui (C function), 129
fmpz_add_ui_inline (C function), 739
fmpz_addmul (C function), 130
fmpz_addmul_ui (C function), 130
fmpz_divides (C function), 131
fmpz_divides_mod_list (C function), 135
fmpz_divisible (C function), 131
fmpz_divisible_si (C function), 131
fmpz_divisor_in_residue_class_lenstra (C function), 140
fmpz_divisor_sigma (C function), 141
fmpz_dlog (C function), 132
fmpz_equal (C function), 129
fmpz_equal_si (C function), 129
fmpz_equal_ui (C function), 129
fmpz_euler_phi (C function), 141
fmpz_fac_ui (C function), 133
fmpz_factor (C function), 147
fmpz_factor_clear (C function), 146
fmpz_factor_divisor_sigma (C function), 141
fmpz_factor_ecm (C function), 150
fmpz_factor_ecm_add (C function), 149
fmpz_factor_ecm_addmod (C function), 149
fmpz_factor_ecm_clear (C function), 149
fmpz_factor_ecm_double (C function), 149
fmpz_factor_ecm_init (C function), 149
fmpz_factor_ecm_mul_montgomery_ladder (C function), 149
fmpz_factor_ecm_select_curve (C function), 149
fmpz_factor_ecm_stage_I (C function), 150
fmpz_factor_ecm_stage_II (C function), 150
fmpz_factor_ecm_submod (C function), 149
fmpz_factor_euler_phi (C function), 141
fmpz_factor_expanded_iterative (C function), 147
fmpz_factor_fprint (C function), 148
fmpz_factor_init (C function), 146
fmpz_factor_moebius_mu (C function), 141
fmpz_factor_pollard_brent (C function), 148
fmpz_factor_pollard_brent_single (C function), 148
fmpz_factor_pp1 (C function), 148
fmpz_factor_print (C function), 148
fmpz_factor_refine (C function), 147
fmpz_factor_si (C function), 147
fmpz_factor_smooth (C function), 147
fmpz_factor_struc (C type), 146
fmpz_factor_t (C type), 146
fmpz_factor_trial (C function), 147
fmpz_factor_trial_range (C function), 147
fmpz_factor_trial_range (C function), 147
fmpz_fdiv_q (C function), 130
fmpz_fdiv_q_2exp (C function), 131
fmpz_fdiv_q_ui (C function), 130
fmpz_fdiv_q ui (C function), 131
fmpz_fdiv2_q_ui (C function), 131
fmpz_fdiv2_q ui (C function), 131
fmpz_fits_si (C function), 128
fmpz_flog (C function), 132
fmpz_flog_ui (C function), 132
fmpz_fmma (C function), 130
fmpz_fmmui (C function), 130
fmpz_fprint (C function), 127
fmpz_gcd (C function), 133
fmpz_gcd3 (C function), 133
fmpz_gcd_ui (C function), 133
fmpz_gcdnv (C function), 134
fmpz_get_d (C function), 124
fmpz_get_d_2exp (C function), 125
fmpz_get_mpf (C function), 125
fmpz_get_mpfq (C function), 125
fmpz_get_mpz (C function), 125
fmpz_get_signed_ui (C function), 126
fmpz_get_signed_ui_array (C function), 126
fmpz_get_str (C function), 125
fmpz_get_ui (C function), 124
fmpz_get_ui_array (C function), 126
fmpz_get_uintui (C function), 124
fmpz_init (C function), 123
fmpz_init2 (C function), 123
fmpz_init_set (C function), 123
fmpz_init_set_ro (C function), 127
fmpz_init_set_s (C function), 123
fmpz_is_even (C function), 129
fmpz_is_odd (C function), 129
fmpz_is_one (C function), 129
fmpz_is_perfect_power (C function), 133
fmpz_is_pmi1 (C function), 129
fmpz_is_prime (C function), 139
fmpz_is_prime_moresum (C function), 139
fmpz_is_prime_morsum (C function), 138
fmpz_is_prime_pocklington (C function), 138
fmpz_is_probabprime (C function), 138
fmpz_is_probabprime_BPSW (C function), 138
fmpz_is_probabprime_lucas (C function), 138
fmpz_is_square (C function), 133
fmpz_is_strong_probabprime (C function), 138
fmpz_is_zero (C function), 129
fmpz_jacobi (C function), 135
fmpz_kronecker (C function), 135
fmpz_lcm (C function), 133
fmpz_lld (C function), 173
fmpz_lld_advance_check_babai (C function), 169
fmpz_lld_advance_check_babai_heuristic_d (C function), 170
fmpz_lld_check_babai (C function), 169
fmpz_lld_check_babai_heuristic (C function), 169
fmpz_lld_check_babai_heuristic_d (C function), 169
fmpz_lld_context_init (C function), 168
fmpz_lld_context_init_default (C function), 168
fmpz_lld_d (C function), 170
fmpz_lld_d_heuristic (C function), 170
fmpz_lld_d_heuristic_with_removal (C function), 171
fmpz_lld_d_with_removal (C function), 171
fmpz_lld_d_with_removal_knapsack (C function), 171
fmpz_lld_heuristic_dot (C function), 169
fmpz_lld_is_reduced (C function), 172
fmpz_lld_is_reduced_d (C function), 172
fmpz_lld_is_reduced_d_with_removal (C function), 172
fmpz_lld_is_reduced_mpfr (C function), 172
fmpz_lld_is_reduced_mpfr_with_removal (C function), 172
fmpz_lld_is_reduced_with_removal (C function), 172
fmpz_lld_mpf (C function), 170
fmpz_lld_mpf2 (C function), 170
fmpz_lld_mpf2_with_removal (C function), 171
fmpz_lld_mpf_with_removal (C function), 171
fmpz_lld_randtest (C function), 169
fmpz_lld_shift (C function), 170
fmpz_lld_storjohann_ulll (C function), 172
fmpz_lld_with_removal (C function), 173
fmpz_lld_with_removal_ulll (C function), 172
fmpz_lld_wrapper (C function), 171
fmpz_lld_wrapper_with_removal (C function), 171
fmpz_lld_wrapper_with_removal_knapsack (C function), 171
fmpz_lshift_mpn (C function), 740
fmpz_lucas_chain (C function), 140
fmpz_lucas_chain_add (C function), 140
fmpz_lucas_chain_double (C function), 140
fmpz_lucas_chain_full (C function), 140
fmpz_lucas_chain_mul (C function), 140
fmpz_lucas_chain_VtoU (C function), 140
fmpz_mat_add (C function), 156
fmpz_mat_can_solve (C function), 163
fmpz_mat_can_solve_fflu (C function), 163
fmpz_mat_can_solve_multi_mod_den (C function), 163
fmpz_mat_charpoly (C function), 161
fmpz_mat_charpoly_berkowitz (C function), 160
fmpz_mat_charpoly_modular (C function), 160
fmpz_mat_chol_d (C function), 167
fmpz_mat_clear (C function), 151
fmpz_mat_col_partition (C function), 161
fmpz_mat_concat_horizontal (C function), 155
fmpz_mat_concat_vertical (C function), 155
fmpz_mat_content (C function), 159
fmpz_mat_CRT_ui (C function), 155
fmpz_mat_det (C function), 159
fmpz_mat_det_bareiss (C function), 159
fmpz_mat_det_bound (C function), 160
fmpz_mat_det_bound_nonzero (C function), 160
fmpz_mat_det cofactor (C function), 159
fmpz_mat_det_divisor (C function), 160
fmpz_mat_det_modular (C function), 159
fmpz_mat_det_modular accelerated (C function), 159
fmpz_mat_det_modular_given_divisor (C function), 160
fmpz_mat_entry (C function), 152
fmpz_mat_equal (C function), 154
fmpz_mat_equal_col (C function), 155
fmpz_mat_equal_row (C function), 155
fmpz_mat_fflu (C function), 163
fmpz_mat_find_pivot_any (C function), 163
fmpz_mat_fmpz_vec_mul (C function), 158
fmpz_mat_fmpz_vec_mul_ptr (C function), 158
fmpz_mat_fprint (C function), 154
fmpz_mat_fprint_pretty (C function), 154
fmpz_mat_fread (C function), 154
fmpz_mat_get_d_mat (C function), 167
fmpz_mat_get_d_mat_transpose (C function), 167
fmpz_mat_get_nmod_mat (C function), 155
fmpz_mat_gran (C function), 167
fmpz_mat_hadamard (C function), 167
fmpz_mat_hnf (C function), 165
fmpz_mat_hnf_classical (C function), 165
fmpz_mat_hnf_minors (C function), 166
fmpz_mat_hnf_modular (C function), 165
fmpz_mat_hnf_modular_eldiv (C function), 166
fmpz_mat_hnf_pernet_stein (C function), 166
fmpz_mat_hnf_transform (C function), 165
fmpz_mat_hnf_xgcd (C function), 165
fmpz_mat_howell_form_mod (C function), 164
fmpz_mat_init (C function), 151
fmpz_mat_init_set (C function), 152
fmpz_mat_inv (C function), 159
fmpz_mat_invert_cols (C function), 152
fmpz_mat_invert_rows (C function), 152
fmpz_mat_is_empty (C function), 154
fmpz_mat_is_hadamard (C function), 167
fmpz_mat_is_in_hnf (C function), 166
fmpz_mat_is_in_rref_with_rank (C function), 164
fmpz_mat_is_in_snf (C function), 166
fmpz_mat_is_one (C function), 154
fmpz_mat_is_reduced (C function), 167
fmpz_mat_is_reduced_gram (C function), 167
fmpz_mat_is_reduced_gram_with_removal (C function), 167
fmpz_mat_is_reduced_with_removal (C function), 167
fmpz_mat_is_spd (C function), 167
fmpz_mat_is_square (C function), 155
fmpz_mat_is_zero (C function), 154
fmpz_mat_is_zero_row (C function), 155
fmpz_mat_kronecker_product (C function), 159
fmpz_mat_lil_original (C function), 168
fmpz_mat_lil_storjohann (C function), 168
fmpz_mat_minpoly (C function), 161
fmpz_mat_minpoly_modular (C function), 161
fmpz_mat_mul (C function), 157
fmpz_mat_mul_blas (C function), 158
fmpz_mat_mul_classical (C function), 157
fmpz_mat_mul_fft (C function), 158
fmpz_mat_mul_fmpz_vec (C function), 158
fmpz_mat_mul_fmpz_vec_ptr (C function), 158
fmpz_mat_mul multi_mod (C function), 157
fmpz_mat_mul_strassen (C function), 157
fmpz_mat_mul_waksman (C function), 157
fmpz_mat_multi_CRT_ui (C function), 156
fmpz_mat_multi CRT ui_precomp (C function), 156
fmpz_mat_multi_mod ui (C function), 156
fmpz_mat multi_mod ui_precomp (C function), 155
fmpz_mat ncols (C function), 152
fmpz_mat_neg (C function), 156
fmpz_mat_nrows (C function), 152
fmpz_mat_nullspace (C function), 165
fmpz_mat_one (C function), 152
fmpz_mat_pow (C function), 158
fmpz_mat_print (C function), 154
fmpz_mat_print_pretty (C function), 154
fmpz_mat_rand test (C function), 152
fmpz_mat_randint (C function), 154
fmpz_mat randdet (C function), 154
fmpz_mat_randintrel (C function), 153
fmpz_mat randntrulike (C function), 153
fmpz_mat randntrulike2 (C function), 153
fmpz_mat randops (C function), 154
fmpz_mat randpermdia (C function), 153
fmpz_mat randrank (C function), 153
fmpz_mat randsumdioph (C function), 153
fmpz_mat randtest (C function), 153
fmpz_mat rank (C function), 161
fmpz_mat read (C function), 154
fmpz_mat rref (C function), 163
fmpz_mat rref fflu (C function), 164
fmpz_mat rref fraction_free (C function), 165
fmpz_mat rref mod (C function), 164
fmpz_mat rref mul (C function), 164
fmpz_mat scalar addmul fmpz (C function), 156
fmpz_mat scalar addmul nmod mat fmpz (C function), 157
fmpz_mat scalar addmul nmod ui (C function), 157
fmpz_mat scalar addmul ui (C function), 156
fmpz_mat scalar divexact fmpz (C function), 157
fmpz_mat scalar division fmpz (C function), 155
fmpz_mat scalar inv fmpz (C function), 155
fmpz_mat scalar inv ui (C function), 155
fmpz_mat scalar_iui (C function), 155
fmpz_mat scalar_ui (C function), 155
fmpz_mat scalar ui (C function), 155
fmpz_mat scalar ui transpose (C function), 164
fmpz_mat_scalar tui (C function), 155
fmpz_mat_set (C function), 159
fmpz_mat_set_ui (C function), 159
fmpz_mat_setfmpz (C function), 159
fmpz_mat_set_nth (C function), 160
fmpz_mat_set_str (C function), 159
fmpz_mat_setint (C function), 159
fmpz_mat_setintrel (C function), 153
fmpz_mat_setntrulike (C function), 153
fmpz_mat_setntrulike2 (C function), 153
fmpz_mat_set_permdia (C function), 153
fmpz_mat_setrandrank (C function), 153
fmpz_mat_setrandsumdioph (C function), 153
fmpz_mat_setrandtest (C function), 153
fmpz_mat_set_testsuite (C function), 168
fmpz_mat_setui (C function), 159
fmpz_mat_setui_mod (C function), 159
fmpz_mat_setui_nmod (C function), 159
fmpz_mat_setui_ui (C function), 159
fmpz_mat_setui ui (C function), 159
fmpz_mat_setui ui_transpose (C function), 164
fmpz_mat_setui_transpose (C function), 164
fmpz_mat_setui2ui (C function), 164
fmpz_mat_setui2ui_precomp (C function), 164
fmpz_mat_setui2ui ui (C function), 159
fmpz_mat_setui ui_precomp (C function), 164
fmpz_mat_setui ui transpose (C function), 164
fmpz_mat_setui_transpose (C function), 164
fmpz_mat_setui2ui transpose (C function), 164
fmpz_mat_setui2ui_transpose (C function), 164
fmpz_mat_setui3ui (C function), 165
fmpz_mat_setui3ui_precomp (C function), 165
fmpz_mat_setui3ui ui (C function), 159
fmpz_mat_setui3ui ui_precomp (C function), 165
fmpz_mat_setui3ui ui transpose (C function), 165
fmpz_mat_setui3ui_transpose (C function), 165
fmpz_mat_setui3ui2ui (C function), 165
fmpz_mat_setui3ui2ui_precomp (C function), 165
fmpz_mat_setui3ui2ui ui (C function), 159
fmpz_mat_setui3ui2ui ui_precomp (C function), 165
fmpz_mat_setui3ui2ui ui transpose (C function), 165
fmpz_mat_setui3ui2ui_transpose (C function), 165

fmpz_mod_mat_mul (C function), 405
fmpz_mod_mat_mul_classical_threaded (C function), 405
fmpz_mod_mat_mul_fmpz_vec (C function), 406
fmpz_mod_mat_mul_fmpz_vec_ptr (C function), 406
fmpz_mod_mat ncols (C function), 403
fmpz_mod_mat_neg (C function), 405
fmpz_mod_mat nrows (C function), 403
fmpz_mod_mat one (C function), 403
fmpz_mod_mat_print_pretty (C function), 404
fmpz_mod_mat_randtest (C function), 404
fmpz_mod_mat rref (C function), 406
fmpz_mod_mat scalar_mul_fmpz (C function), 408
fmpz_mod_mat scalar_mul_fmpz (C function), 408
fmpz_mod_mat_set (C function), 404
fmpz_mod_mat_set_fmpz_mat (C function), 404
fmpz_mod_mat solve (C function), 407
fmpz_mod_mat solve_tril (C function), 407
fmpz_mod_mat solve_tril (C function), 407
fmpz_mod_mat sqr (C function), 405
fmpz_mod_mat strong_echelon_form (C function), 406
fmpz_mod_mat struct (C type), 403
fmpz_mod_mat sub (C function), 405
fmpz_mod_mat swap (C function), 403
fmpz_mod_mat swap_entrywise (C function), 403
fmpz_mod_mat t (C type), 403
fmpz_mod_mat trace (C function), 406
fmpz_mod_mat transpose (C function), 404
fmpz_mod_mat window clear (C function), 404
fmpz_mod_mat window init (C function), 404
fmpz_mod_mat zero (C function), 403
fmpz_mod_mpoly add (C function), 447
fmpz_mod_mpoly add_fmpz (C function), 447
fmpz_mod_mpoly add_si (C function), 447
fmpz_mod_mpoly add_ui (C function), 447
fmpz_mod_mpoly clear (C function), 441
fmpz_mod_mpoly cmp (C function), 444
fmpz_mod_mpoly combine_like_terms (C function), 446
fmpz_mod_mpoly compose_fmpz_mod_mpoly (C function), 448
fmpz_mod_mpoly compose_fmpz_mod_mpoly_gen (C function), 448
fmpz_mod_mpoly compose_fmpz_mod_mpoly_geobucket (C function), 448
fmpz_mod_mpoly compose_fmpz_poly (C function), 448
fmpz_mod_mpoly content-vars (C function), 450
fmpz_mod_mpoly ctx clear (C function), 441
fmpz_mod_mpoly ctx_get_modulus (C function), 441
fmpz_mod_mpoly ctx_init (C function), 441
fmpz_mod_mpoly_ctx_nvars (C function), 441
fmpz_mod_mpoly_ctx_ord (C function), 441
fmpz_mod_mpoly_ctx_struct (C type), 440
fmpz_mod_mpoly_ctx_t (C type), 440
fmpz_mod_mpoly_deflate (C function), 452
fmpz_mod_mpoly deflation (C function), 452
fmpz_mod_mpoly_degree_fmpz (C function), 443
fmpz_mod_mpoly_degree si (C function), 443
fmpz_mod_mpoly_degrees_fit si (C function), 443
fmpz_mod_mpoly_degrees_fmpz (C function), 443
fmpz_mod_mpoly_degrees_si (C function), 443
fmpz_mod_mpoly_derivative (C function), 448
fmpz_mod_mpoly_discriminant (C function), 450
fmpz_mod_mpoly_div (C function), 449
fmpz_mod_mpoly divides (C function), 449
fmpz_mod_mpoly_divrem (C function), 449
fmpz_mod_mpoly_divrem ideal (C function), 449
fmpz_mod_mpoly_equal (C function), 442
fmpz_mod_mpoly_equal_fmpz (C function), 442
fmpz_mod_mpoly_equal_si (C function), 442
fmpz_mod_mpoly_equal ui (C function), 442
fmpz_mod_mpoly evaluate_all_fmpz (C function), 448
fmpz_mod_mpoly evaluate_one_fmpz (C function), 448
fmpz_mod_mpoly_factor (C function), 453
fmpz_mod_mpoly_factor_clear (C function), 452
fmpz_mod_mpoly_factor get_base (C function), 453
fmpz_mod_mpoly_factor_get_constant_fmpz (C function), 453
fmpz_mod_mpoly_factor get exp si (C function), 453
fmpz_mod_mpoly_factor get exp si (C function), 453
fmpz_mod_mpoly_factor_init (C function), 452
fmpz_mod_mpoly_factor_length (C function), 453
fmpz_mod_mpoly_factor_length (C function), 453
fmpz_mod_mpoly factor_sort (C function), 453
fmpz_mod_mpoly factor_squarefree (C function), 453
fmpz_mod_mpoly_factorstruct (C type), 452
fmpz_mod_mpoly_factor_swap (C function), 453
fmpz_mod_mpoly_factor_swap_base (C function), 453
fmpz_mod_mpoly factor_t (C type), 452
fmpz_mod_mpoly fprint pretty (C function), 441
fmpz_mod_mpoly_from univar (C function), 451
fmpz_mod_mpoly from univar (C function), 451
fmpz_mod_mpoly gcd (C function), 450
fmpz_mod_mpoly gcd b (C function), 450
fmpz_mod_mpoly gcd cofactors (C function), 450
fmpz_mod_mpoly gcd hensel (C function), 450
fmpz_mod_mpoly gcd_subresultant (C function), 450
fmpz_mod_mpoly gcd_zippel (C function), 450
fmpz_mod_mpoly gcd_zippel2 (C function), 450
fmpz_mod_mpoly gamma (C function), 442
fmpz_mod_mpoly gen (C function), 442
fmpz_poly_divrem (C function), 192
fmpz_poly_divrem_basecase (C function), 191
fmpz_poly_divrem_divconquer (C function), 192
fmpz_poly_divrem_preinv (C function), 195
fmpz_poly_equal (C function), 178
fmpz_poly_equal_trunc (C function), 178
fmpz_poly_eta_fexp (C function), 215
fmpz_poly_divrem_divconquer_fmpz (C function), 200
fmpz_poly_divrem_divconquer_fmpq (C function), 200
fmpz_poly_divrem_fmpz (C function), 201
fmpz_poly_divrem_fmpz_vec (C function), 201
fmpz_poly_divrem_horner_d (C function), 201
fmpz_poly_divrem_horner_d_2exp (C function), 201
fmpz_poly_divrem_horner_fmpq (C function), 201
fmpz_poly_divrem_horner_fmpz (C function), 201
fmpz_poly_gcd_modular (C function), 222
fmpz_poly_gcd_heuristic (C function), 222
fmpz_poly_fread (C function), 222
fmpz_poly_fread_pretty (C function), 222
fmpz_poly_factor (C function), 222
fmpz_poly_factor_init (C function), 222
fmpz_poly_factor_init2 (C function), 222
fmpz_poly_factor_insert (C function), 222
fmpz_poly_factor_malloc (C function), 222
fmpz_poly_factor_malloc (C function), 222
fmpz_poly_factor_squarefree (C function), 223
fmpz_poly_factor_t (C type), 222
fmpz_poly_factor_zassenhaus (C function), 223
fmpz_poly_factor_zassenhaus_recombination (C function), 223
fmpz_poly_fibonacci (C function), 214
fmpz_poly_fit_length (C function), 174
fmpz_poly_fprint (C function), 210
fmpz_poly_fprint_pretty (C function), 210
fmpz_poly_fread (C function), 210
fmpz_poly_gcd (C function), 188
fmpz_poly_gcd_heuristic (C function), 188
fmpz_poly_gcd_modular (C function), 188
fmpz_poly_gcd_subresultant (C function), 188
fmpz_poly_get_coeff_fmpz (C function), 177
fmpz_poly_get_coeff_fmpz (C function), 177
fmpz_poly_get_coeff_fmpz (C function), 177
fmpz_poly_get_coeff_fmpz (C function), 177
fmpz_poly_get_coeff_fmpz (C function), 177
fmpz_poly_get_str (C function), 176
fmpz_poly_get_str_pretty (C function), 176
fmpz_poly_height (C function), 187
fmpz_poly_hensel_build_tree (C function), 207
fmpz_poly_hensel_lift (C function), 207
fmpz_poly_hensel_lift_only_inverse (C function), 208
fmpz_poly_hensel_lift_tree (C function), 208
fmpz_poly_hensel_lift_tree_recursive (C function), 208
fmpz_poly_hensel_lift_without_inverse (C function), 208
fmpz_poly_hermite_h (C function), 214
fmpz_poly_hermite_he (C function), 214
fmpz_poly_inflate (C function), 203
fmpz_poly_init (C function), 174
fmpz_poly_init2 (C function), 174
fmpz_poly_interpolate_fmpz_vec (C function), 202
fmpz_poly_inv_series (C function), 197
fmpz_poly_inv_series_basecase (C function), 197
fmpz_poly_inv_series_newton (C function), 197
fmpz_poly_is_cyclotomic (C function), 213
fmpz_poly_is_gen (C function), 178
fmpz_poly_is_one (C function), 178
fmpz_poly_is_squarefree (C function), 191
fmpz_poly_is_unit (C function), 178
fmpz_poly_is_zero (C function), 178
fmpz_poly_lcm (C function), 189
fmpz_poly_lead (C function), 177
fmpz_poly_legrendre_pt (C function), 213
fmpz_poly_length (C function), 175
fmpz_poly_mat_add (C function), 218
fmpz_poly_mat_clear (C function), 216
fmpz_poly_mat_det (C function), 221
fmpz_poly_mat_det_fflu (C function), 221
fmpz_poly_mat_det_interpolate (C function), 221
fmpz_poly_mat_entry (C function), 217
fmpz_poly_mat_equal (C function), 218
fmpz_poly_mat_mul (C function), 218
fmpz_poly_mat_mullow (C function), 218
fmpz_poly_mat_find_pivot (C function), 220
fmpz_poly_mat_find_pivot_partial (C function), 220
fmpz_poly_mat_find_pivot_any (C function), 220
fmpz_poly_mat_find_pivot_partial (C function), 220
fmpz_poly_mat_init (C function), 216
fmpz_poly_mat_init_set (C function), 216
fmpz_poly_mat_inv (C function), 221
fmpz_poly_mat_is_empty (C function), 218
fmpz_poly_mat_is_one (C function), 218
fmpz_poly_mat_is_square (C function), 218
fmpz_poly_mat_is_zero (C function), 218
fmpz_poly_mat_max_bits (C function), 218
fmpz_poly_mat_max_length (C function), 218
fmpz_poly_mat_mul (C function), 219
fmpz_poly_mat_mul_classical (C function), 219
fmpz_poly_mat_mul KS (C function), 219
fmpz_poly_mat_mullov (C function), 219
fmpz_poly_mat_ncols (C function), 216
fmpz_poly_mat_neg (C function), 219
fmpz_poly_taylor_shift_divconquer (C function), 203
fmpz_poly_taylor_shift_horner (C function), 203
fmpz_poly_taylor_shift_multi_mod (C function), 203
fmpz_poly_theta_qexp (C function), 215
fmpz_poly_truncate (C function), 176
fmpz_poly_xgcd (C function), 130
fmpz_poly_zero (C function), 130
fmpz_pow_ui (C function), 130
fmpz_popcnt (C function), 136
fmpz_powm_ui (C function), 132
fmpz_primorial (C function), 132
fmpz_randtest_not_zero (C function), 132
fmpz_randtest_mod (C function), 132
fmpz_randtest (C function), 132
fmpz_randprime (C function), 132
fmpz_randm (C function), 132
fmpz_print (C function), 132
fmpz_read (C function), 127
fmpz_remove (C function), 135
fmpz_rfac_ui (C function), 133
fmpz_rfac_uiui (C function), 133
fmpz_root (C function), 133
fmpz_set (C function), 128
fmpz_set_d (C function), 125
fmpz_set_d_2exp (C function), 125
fmpz_set_mpfr (C function), 124
fmpz_set_mpn_large (C function), 740
fmpz_set_mpz (C function), 126
fmpz_set_si (C function), 125
fmpz_set_signed_uiui_array (C function), 125
fmpz_set_signed_uiui (C function), 125
fmpz_set_str (C function), 126
fmpz_set_ui (C function), 125
fmpz_set_ui_array (C function), 125
fmpz_set_ui_smod (C function), 126
fmpz_set_uiui (C function), 125
fmpz_set_bit (C function), 128
fmpz_sgn (C function), 128
fmpz_size (C function), 128
fmpz_sizeinbase (C function), 128
fmpz_smod (C function), 132
fmpz_sqrt (C function), 132
fmpz_sqrtmod (C function), 132
fmpz_sqrtrem (C function), 133
fmpz_sub (C function), 130
fmpz_sub_si (C function), 130
fmpz_sub_si_inline (C function), 739
fmpz_sub_ui (C function), 130
fmpz_submul (C function), 130
fmpz_submul_si (C function), 130
fmpz_submul_ui (C function), 130
fmpz_swap (C function), 128
fmpz_t (C type), 122
fmpz_tdiv_q (C function), 130
fmpz_tdiv_q_2exp (C function), 131
fmpz_tdiv_q_si (C function), 130
fmpz_tdiv_q_ui (C function), 131
fmpz_tdiv_qr (C function), 130
fmpz_tdiv_r_2exp (C function), 131
fmpz_tdiv_ui (C function), 131
fmpz tstbit (C function), 128
fmpz_ui_mul_ui (C function), 739
fmpz ui_pow_ui (C function), 132
fmpzval2 (C function), 128
fmpz_xgcd (C function), 134
fmpz_xgcd_canonical_bezout (C function), 134
fmpz_xgcd_partial (C function), 134
fmpz_xor (C function), 136
fmpz_zero (C function), 128
fmpz_add (C function), 477
fmpz_bits (C function), 477
fmpz canonical_unit_i_pow (C function), 477
fmpz canonical_unit_i_pow (C function), 477
fmpz canonical_unit_i_pow (C function), 477
fmpz clear (C function), 476
fmpz conj (C function), 477
fmpz divexact (C function), 477
fmpz divrem (C function), 477
fmpz divrem_approx (C function), 477
fmpz equal (C function), 476
fmpz gcd (C function), 477
fmpz gcd_binary (C function), 477
fmpz gcd euclidean (C function), 477
fmpz gcd euclidean improved (C function), 477
fmpz gcd shortest (C function), 477
fmpz imagref (C macro), 476
fmpz init (C function), 476
fmpz_is_one (C function), 476
fmpz_is_prime (C function), 478
fmpz is probabprime (C function), 478
fmpz is unit (C function), 477
fmpz is zero (C function), 476
fmpz mul (C function), 477
fmpz neg (C function), 477
fmpz norm (C function), 477
fmpz one (C function), 476
fmpz pow_ui (C function), 477
fmpz print (C function), 476
fmpz randtest (C function), 476
fmpz realref (C macro), 476
fmpz remove one plus_i (C function), 477
fmpz set (C function), 476
fmpz set si si (C function), 476
fq_nmod_is_one (C function), 898
fq_nmod_is_primitive (C function), 899
fq_nmod_is_square (C function), 896
fq_nmod_is_zero (C function), 898
fq_nmod_mat_add (C function), 905
fq_nmod_mat_can_solve (C function), 908
fq_nmod_mat_charpoly (C function), 908
fq_nmod_mat_charpoly_danilevsky (C function), 908
fq_nmod_mat_clear (C function), 902
fq_nmod_mat_concat_horizontal (C function), 903
fq_nmod_mat_concat_vertical (C function), 903
fq_nmod_mat_entry (C function), 902
fq_nmod_mat_entry_set (C function), 902
fq_nmod_mat_equal (C function), 904
fq_nmod_mat_fprint (C function), 903
fq_nmod_mat_fprint_pretty (C function), 903
fq_nmod_mat_init (C function), 902
fq_nmod_mat_init_set (C function), 902
fq_nmod_mat_inv (C function), 906
fq_nmod_mat_invert_cols (C function), 903
fq_nmod_mat_invert_rows (C function), 902
fq_nmod_mat_is_empty (C function), 905
fq_nmod_mat_is_one (C function), 904
fq_nmod_mat_is_square (C function), 905
fq_nmod_mat_is_zero (C function), 904
fq_nmod_mat_lu (C function), 906
fq_nmod_mat_lu_classical (C function), 906
fq_nmod_mat_lu_recursive (C function), 906
fq_nmod_mat_mnopoly (C function), 908
fq_nmod_mat_mul (C function), 905
fq_nmod_mat_mul_classical (C function), 905
fq_nmod_mat_mul_KS (C function), 905
fq_nmod_mat_mul_vec (C function), 905
fq_nmod_mat_mul_vec_ptr (C function), 905
fq_nmod_mat_nrows (C function), 902
fq_nmod_mat_neg (C function), 905
fq_nmod_mat_nrows (C function), 902
fq_nmod_mat_one (C function), 902
fq_nmod_mat_print (C function), 903
fq_nmod_mat_print_pretty (C function), 903
fq_nmod_mat_randops (C function), 904
fq_nmod_mat_randperm (C function), 904
fq_nmod_mat_randrank (C function), 904
fq_nmod_mat_randtest (C function), 904
fq_nmod_mat_rantril (C function), 904
fq_nmod_mat_randtril (C function), 904
fq_nmod_mat_reduce_row (C function), 906
fq_nmod_mat_rref (C function), 906
fq_nmod_mat_set (C function), 902
fq_nmod_mat_set_fmpz_mod_mat (C function), 903
fq_nmod_mat_set_nmod_mat (C function), 903
fq_nmod_mat_similarity (C function), 908
fq_nmod_mat_solve (C function), 908
fq_nmod_mat_solve_tril (C function), 907
fq_nmod_mat_solve_tril_classical (C function), 907
fq_nmod_mat_solve_tril_recursive (C function), 907
fq_nmod_mat_solve_triu (C function), 907
fq_nmod_mat_solve_triu_classical (C function), 907
fq_nmod_mat_solve_triu_recursive (C function), 907
fq_nmod_mat_struct (C type), 901
fq_nmod_mat_sub (C function), 905
fq_nmod_mat_submul (C function), 905
fq_nmod_mat_swap (C function), 902
fq_nmod_mat_swap_cols (C function), 902
fq_nmod_mat_swap_entrywise (C function), 902
fq_nmod_mat_swap_rows (C function), 902
fq_nmod_mat_t (C type), 901
fq_nmod_mat_vec_mul (C function), 905
fq_nmod_mat_vec_mul_ptr (C function), 905
fq_nmod_mat_window_clear (C function), 904
fq_nmod_mat_window_init (C function), 904
fq_nmod_mat_zero (C function), 902
fq_nmod_modulus_pow_series_inv (C function), 933
fq_nmod_modulus_pow_series_inv (C function), 933
fq_nmod_mpoly_add (C function), 939
fq_nmod_mpoly_add_fq_nmod (C function), 939
fq_nmod_mpoly_clear (C function), 934
fq_nmod_mpoly_cmp (C function), 937
fq_nmod_mpoly_degrees_fmpz (C function), 938
fq_nmod_mpoly_degrees_fit_si (C function), 938
fq_nmod_mpoly_ctx_t (C function), 933
fq_nmod_mpoly_ctx_clear (C function), 933
fq_nmod_mpoly_ctx_nvars (C function), 933
fq_nmod_mpoly_ctx_init (C function), 933
fq_nmod_mpoly_ctx_size (C function), 933
fq_nmod_mpoly_ctx_t (C function), 933
fq_nmod_mpoly_div (C function), 941
fq_nmod_mpoly_degrees (C function), 941
fq_nmod_mpoly_degrees_si (C function), 941
fq_nmod_mpoly_degrees_fmpz (C function), 936
fq_nmod_mpoly_degrees_fmpz (C function), 936
fq_nmod_mpoly_derivative (C function), 940
fq_nmod_mpoly_discriminant (C function), 942
fq_nmod_mpoly_divrem (C function), 941
fq_nmod_mpoly_divrem_ideal (C function), 941
fq_nmod_mpoly_equal (C function), 935

1188  Index
fq_nmod_mpoly_sub (C function), 939
fq_nmod_mpoly_sub_fq_nmod (C function), 939
fq_nmod_mpoly_swap (C function), 935
fq_nmod_mpoly_clear (C function), 933
fq_nmod_mpoly_content (C function), 941
fq_nmod_mpoly_term_exp_fits_si (C function), 937
fq_nmod_mpoly_term_exp_fits_ui (C function), 937
fq_nmod_mpoly_to_univar (C function), 942
fq_nmod_mpoly_total_degree_fits_si (C function), 936
fq_nmod_mpoly_total_degree_fmpz (C function), 936
fq_nmod_mpoly_totalDegree (C function), 936
fq_nmod_mpoly_univar_clear (C function), 942
fq_nmod_mpoly_univar_degree_fits_si (C function), 942
fq_nmod_mpoly_univar_get_term_coeff (C function), 943
fq_nmod_mpoly_univar_get_term_exp_si (C function), 943
fq_nmod_mpoly_univar_init (C function), 942
fq_nmod_mpoly_univar_length (C function), 943
fq_nmod_mpoly_univar_swap (C function), 942
fq_nmod_mpoly_univar_swap_term_coeff (C function), 943
fq_nmod_mpoly_used_vars (C function), 936
fq_nmod_mpoly_zero (C function), 895
fq_nmod_mul (C function), 895
fq_nmod_mul_fmpz (C function), 895
fq_nmod_mul_si (C function), 895
fq_nmod_one (C function), 899
fq_nmod_multiplicative_order (C function), 899
fq_nmod_neg (C function), 895
fq_nmod_norm (C function), 899
fq_nmod_one (C function), 898
fq_nmod_poly_add (C function), 912
fq_nmod_poly_add_series (C function), 912
fq_nmod_poly_add_si (C function), 912
fq_nmod_poly_clear (C function), 909
fq_nmod_poly_clear (C function), 925
fq_nmod_poly_deflate (C function), 928
fq_nmod_poly_deg (C function), 910
fq_nmod_poly_derivative (C function), 924
fq_nmod_poly_div (C function), 920
fq_nmod_poly_div_newton_n_preinv (C function), 921
fq_nmod_poly_divrem (C function), 922
fq_nmod_poly_divrem (C function), 923
fq_nmod_poly_divrem (C function), 920
fq_nmod_poly_divrem (C function), 920
fq_nmod_poly_divrem (C function), 921
fq_nmod_poly_equal (C function), 912
fq_nmod_poly_equal_fq_nmod (C function), 912
fq_nmod_poly_equal_trunc (C function), 912
fq_nmod_poly_evaluate_fq_nmod (C function), 930
fq_nmod_poly_factor (C function), 930
fq_nmod_poly_factor_berlekamp (C function), 931
fq_nmod_poly_factor_cantor_zassenhaus (C function), 931
fq_nmod_poly_factor_clear (C function), 929
fq_nmod_poly_factor_conc (C function), 929
fq_nmod_poly_factor_distinct_deg (C function), 930
fq_nmod_poly_factor_equal_deg (C function), 930
fq_nmod_poly_factor_equal_deg (C function), 930
fq_nmod_poly_factor_equal_deg (C function), 922
fq_nmod_poly_factor_fit_length (C function), 929
fq_nmod_poly_factor_init (C function), 929
fq_nmod_poly_factor_insert (C function), 929
fq_nmod_poly_factor_kaltofen_shoup (C function), 931
fq_nmod_poly_factor_pow (C function), 929
fq_nmod_poly_factor_print (C function), 929
fq_nmod_poly_factor_print_pretty (C function), 929
fq_nmod_poly_factor_with_berlekamp (C function), 929
fq_nmod_poly_factor_with_conc (C function), 929
fq_nmod_poly_factor_with_split_single (C function), 930
fq_nmod_poly_factor_with_squarefree (C function), 930
fq_nmod_poly_factor_with_struct (C function), 930
fq_nmod_poly_factor_with_t (C type), 929
fq_nmod_poly_factor_with_berlekamp (C function), 931
fq_nmod_poly_factor_with_cantor_zassenhaus (C function), 931
fq_nmod_poly_factor_with_kaltofen_shoup (C function), 931
fq_nmod_poly_fit_length (C function), 909
fq_nmod_poly_print (C function), 928
fq_nmod_poly_print_pretty (C function), 927
fq_nmod_poly_gcd (C function), 922
fq_nmod_rand (C function), 897
fq_nmod_rand_not_zero (C function), 897
fq_nmod_randtest (C function), 897
fq_nmod_randtest_dense (C function), 897
fq_nmod_randtest_not_zero (C function), 897
fq_nmod_reduce (C function), 895
fq_nmod_set (C function), 897
fq_nmod_set_fmpz (C function), 897
fq_nmod_set_nmod_mat (C function), 898
fq_nmod_set_fq (C function), 898
fq_nmod_set_si (C function), 897
fq_nmod_sqrt (C function), 897
fq_nmod_sq (C function), 896
fq_nmod_sqrt (C function), 896
fq_nmod_square (C function), 896
fq_nmod_sub (C function), 895
fq_nmod_sub_one (C function), 895
fq_nmod_swap (C function), 897
fq_nmod_t (C type), 893
fq_nmod_trace (C function), 899
fq_nmod_zero (C function), 898
fq_norm (C function), 840
fq_one (C function), 839
fq_poly_pow (C function), 864
fq_poly_add (C function), 864
fq_poly_add_series (C function), 864
fq_poly_add_si (C function), 864
fq_poly_clear (C function), 861
fq_poly_compose (C function), 875
fq_poly_compose_mod (C function), 876
fq_poly_compose_mod_brent_kung (C function), 876
fq_poly_compose_mod_brent_kung_precomp_preinv (C function), 877
fq_poly_compose_mod_brent_kung_preinv (C function), 876
fq_poly_compose_mod_horner (C function), 875
fq_poly_compose_mod_horner_preinv (C function), 876
fq_poly_compose_mod_preinv (C function), 877
fq_poly_compose_mod_brent_kung_precomp_preinv (C function), 877
fq_poly_compose_mod_preinv (C function), 877
fq_poly_deflate (C function), 879
fq_poly_deflation (C function), 879
fq_poly_degree (C function), 862
fq_poly_derivative (C function), 874
fq_poly_div (C function), 871
fq_poly_div_newton_n_preinv (C function), 872
fq_poly_div_series (C function), 872
fq_poly_divides (C function), 874
fq_poly_divrem (C function), 871
fq_poly_divrem_f (C function), 871
fq_poly_divrem_newton_n_preinv (C function), 872
fq_poly_equal (C function), 863
fq_poly_equal_f (C function), 863
fq_poly_equal_trunc (C function), 863
fq_poly_evaluate_fq (C function), 875
fq_poly_factor (C function), 888
fq_poly_factor_berlekamp (C function), 889
fq_poly_factor_cantor_zassenhaus (C function), 888
fq_poly_factor_clear (C function), 887
fq_poly_factor_concat (C function), 887
fq_poly_factor_distinct_deg (C function), 888
fq_poly_factor_equal_deg (C function), 888
fq_poly_factor_equal_deg_prob (C function), 888
fq_poly_factor_fit_length (C function), 887
fq_poly_factor_init (C function), 887
fq_poly_factor_insert (C function), 887
fq_poly_factor_kaltofen_shoup (C function), 889
fq_poly_factor_pow (C function), 887
fq_poly_factor_print (C function), 887
fq_poly_factor_print_pretty (C function), 887
fq_poly_factor_realloc (C function), 887
fq_poly_factor_set (C function), 887
fq_poly_factor_split_single (C function), 888
fq_poly_factor_squarerfree (C function), 888
fq_poly_factor_struct (C function), 887
fq_poly_factor_t (C type), 887
fq_poly_factor_with_berlekamp (C function), 889
fq_poly_factor_with_cantor_zassenhaus (C function), 889
fq_poly_factor_with_kaltofen_shoup (C function), 889
fq_poly_fit_length (C function), 861
fq_poly_fprint (C function), 878
fq_poly_fprint_pretty (C function), 878
fq_poly_gcd (C function), 873
fq_poly_gcd_euclidean_f (C function), 873
fq_poly_gen (C function), 863
fq_poly_getCoeff (C function), 863
fq_poly_get_str (C function), 878
fq_poly_get_str_pretty (C function), 878
fq_poly_hamming_weight (C function), 871
fq_poly_inflate (C function), 879
fq_poly_init (C function), 861
fq_poly_init2 (C function), 861
fq_poly_inv_series (C function), 872
fq_poly_inv_series_newton (C function), 872
fq_poly_invsqrt_series (C function), 875
fq_poly_is_gen (C function), 863
fq_poly_is_irreducible (C function), 888
fq_poly_is_irreducible_ben_or (C function), 888
fq_poly_is_irreducible_ddf (C function), 888
fq_poly_is_one (C function), 863
fq_poly_is_squarerfree (C function), 888
fq_poly_is_unit (C function), 863
fq_poly_is_zero (C function), 863
fq_poly_iterated_frobenius_preinv (C function), 889
fq_poly_lead (C function), 862
fq_poly_length (C function), 862
fq_poly_make_monic (C function), 863
fq_poly_mul (C function), 866
fq_poly_mul_classical (C function), 865
fq_poly_mul_KS (C function), 866
fq_poly_mul_reorder (C function), 865
fq_poly_mul_univariate (C function), 865
fq_poly_mulhigh (C function), 867
fq_poly_mulhigh_classical (C function), 867
fq_poly_mullow (C function), 866
fq_poly_mullow_classical (C function), 866
fq_poly_mullow_KS (C function), 866
fq_poly_mullow_univariate (C function), 866
fq_poly_mullow_mod (C function), 867
fq_poly_mulmod_preinv (C function), 867
fq_poly_one (C function), 864
fq_poly_neg (C function), 864
fq_poly_one (C function), 862
fq_poly_pow (C function), 868
fq_poly_pow_trunc (C function), 870
fq_poly_pow_trunc_binexp (C function), 870
fq_poly_powmod_fmpz_binexp (C function), 869
fq_poly_powmod_fmpz_binexp_preinv (C function), 869
fq_poly_powmod_fmpz_sliding_preinv (C function), 869
fq_poly_powmod_ui_binexp (C function), 868
fq_poly_powmod_ui_binexp_preinv (C function), 869
fq_poly_powmod_x_fmpz_preinv (C function), 870
fq_poly_precompute_matrix (C function), 877
fq_poly_print (C function), 878
fq_poly_print_pretty (C function), 878
fq_poly_randtest (C function), 862
fq_poly_randtest_irreducible (C function), 862
fq_poly_randtest_monic (C function), 862
fq_poly_randtest_not_zero (C function), 862
fq_poly_realloc (C function), 861
fq_poly_rem (C function), 871
fq_poly_remove (C function), 887
fq_poly_reverse (C function), 861
fq_poly_roots (C function), 889
fq_poly_scalar_addmul_fq (C function), 864
fq_poly_scalar_addmul_ui (C function), 864
fq_poly_scalar_div_fq (C function), 865
fq_poly_scalar_mul_fq (C function), 864
fq_poly_scalar_mulmul_fq (C function), 864
fq_poly_scalar_submul_fq (C function), 864
fq_poly_set (C function), 862
fq_poly_set_coeff (C function), 863
fq_poly_set_coeff_fmpz (C function), 863
fq_poly_set_fmpz_mod_poly (C function), 862
fq_poly_set_fq (C function), 862
fq_poly_set_nmod_poly (C function), 862
fq_poly_set_trunc (C function), 861
fq_poly_shift_left (C function), 870
fq_poly_shift_right (C function), 870
fq_poly_sqrt (C function), 868
fq_poly_sqrt_classical (C function), 868
fq_poly_sqrt_KS (C function), 868
fq_poly_sqrt_reorder (C function), 868
fq_poly_sqrt (C function), 875
fq_poly_sqrt_series (C function), 875
fq_poly_struct (C function), 861
fq_poly_sub (C function), 864
fq_poly_sub_series (C function), 864
fq_poly_swap (C function), 862
fq_poly_t (C type), 861
fq_poly_truncate (C function), 861
fq_poly_xgcd (C function), 873
fq_poly_xgcd_euclidean_f (C function), 874
fq_poly_zero (C function), 862
fq_pow (C function), 837
fq_pow_ui (C function), 838
fq_print (C function), 838
fq_print_pretty (C function), 838
fq_pth_root (C function), 838
fq_rand (C function), 839
fq_rand_not_zero (C function), 839
fq_randtest (C function), 839
fq_randtest_dense (C function), 839
fq_randtest_not_zero (C function), 839
fq_reduce (C function), 837
fq_set (C function), 839
fq_set_fmpz (C function), 839
fq_set_fmpz_mod_mat (C function), 840
fq_set_fmpz_mod_poly (C function), 839
fq_set_fmpz_poly (C function), 839
fq_set_si (C function), 839
fq_set_ui (C function), 839
fq_sqrt (C function), 837
fq_sqrt (C function), 838
fq_sqrt (C function), 838
fq_struct (C type), 835
fq_sub (C function), 837
fq_sub_one (C function), 837
fq_swap (C function), 839
fq_t (C type), 835
fq_trace (C function), 840
fq_zech_add (C function), 947
fq_zech_bit_pack (C function), 951
fq_zech_bit_unpack (C function), 951
fq_zech_clear (C function), 946
fq_zech_ctx_clear (C function), 946
fq_zech_ctx_degree (C function), 946
fq_zech_ctx_fprint (C function), 946
fq_zech_ctx_init_conway_ui (C function), 945
fq_zech_ctx_init_fq_nmod_ctx (C function), 945
fq_zech_ctx_init_modulus (C function), 945
fq_zech_ctx_init_fq_nmod_ctx (C function), 945
fq_zech_ctx_init_fq_nmod_ctx_check (C function), 945
fq_zech_ctx_init_modulus (C function), 945
fq_zech_ctx_init_modulus_check (C function), 945
fq_zech_ctx_init_random_ui (C function), 945
fq_zech_ctx_init_randtest (C function), 945
fq_zech_ctx_init_randtest_redicule (C function), 945
fq_zech_ctx_init_ui (C function), 945
fq_zech_ctx_modulus (C function), 946
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>fq_zech_mul_si</td>
<td>(C function), 947</td>
<td></td>
</tr>
<tr>
<td>fq_zech_mul_ui</td>
<td>(C function), 947</td>
<td></td>
</tr>
<tr>
<td>fq_zech_multiplicative_order</td>
<td>(C function), 950</td>
<td></td>
</tr>
<tr>
<td>fq_zech_neg</td>
<td>(C function), 947</td>
<td></td>
</tr>
<tr>
<td>fq_zech_norm</td>
<td>(C function), 950</td>
<td></td>
</tr>
<tr>
<td>fq_zech_one</td>
<td>(C function), 949</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_add</td>
<td>(C function), 963</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_add_series</td>
<td>(C function), 963</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_add_si</td>
<td>(C function), 963</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_clear</td>
<td>(C function), 960</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_compose</td>
<td>(C function), 975</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_compose_mod</td>
<td>(C function), 975</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_compose_mod_brent_kung</td>
<td>(C function), 975</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_compose_mod_brent_kung_pcomp</td>
<td>(C function), 977</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_compose_mod_brent_kung_preinv</td>
<td>(C function), 976</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_compose_mod_horner</td>
<td>(C function), 975</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_compose_mod_horner_preinv</td>
<td>(C function), 975</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_compose_mod_preinv</td>
<td>(C function), 976</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_deflate</td>
<td>(C function), 978</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_deflation</td>
<td>(C function), 978</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_degree</td>
<td>(C function), 960</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_derivative</td>
<td>(C function), 974</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_div</td>
<td>(C function), 970</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_div_newton_n_preinv</td>
<td>(C function), 971</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_div_series</td>
<td>(C function), 972</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_divides</td>
<td>(C function), 973</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_divrem</td>
<td>(C function), 970</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_divrem_f</td>
<td>(C function), 970</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_divrem_newton_n_preinv</td>
<td>(C function), 971</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_equal</td>
<td>(C function), 962</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_equal_fq_zech</td>
<td>(C function), 962</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_equal_trunc</td>
<td>(C function), 962</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_evaluate_fq_zech</td>
<td>(C function), 974</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_berlekamp</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_cantor_zassenhaus</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_clear</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_concat</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_distinct_deg</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_equal_deg</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_equal_deg_prob</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_fit_length</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_init</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_insert</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_kaltofen_shoup</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_pow</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_print</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_print_pretty</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_realloc</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_set</td>
<td>(C function), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_split_single</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_squarefree</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_struct</td>
<td>(C type), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_t</td>
<td>(C type), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_with_berlekamp</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_with_cantor_zassenhaus</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_with_kaltofen_shoup</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_with_berlekamp</td>
<td>(C type), 979</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_with_kaltofen_shoup</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_with_cantor_zassenhaus</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_with_berlekamp</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_factor_with_kaltofen_shoup</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_gen</td>
<td>(C function), 961</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_get_coeff</td>
<td>(C function), 962</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_get_str</td>
<td>(C function), 978</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_get_str_pretty</td>
<td>(C function), 978</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_hamming_weight</td>
<td>(C function), 970</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_inflate</td>
<td>(C function), 973</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_init</td>
<td>(C function), 959</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_init2</td>
<td>(C function), 959</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_inv_series</td>
<td>(C function), 971</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_inv_series_newton</td>
<td>(C function), 971</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_invsqrt_series</td>
<td>(C function), 974</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_is_gen</td>
<td>(C function), 962</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_is_irreducible</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_is_irreducible_berlekamp</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_is_irreducible_ddf</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_is_one</td>
<td>(C function), 962</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_is_squarefree</td>
<td>(C function), 980</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_is_unit</td>
<td>(C function), 962</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_is_zero</td>
<td>(C function), 962</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_iterated_frobenius_preinv</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_iterated_frobenius_preinv</td>
<td>(C function), 981</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_lead</td>
<td>(C function), 960</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_length</td>
<td>(C function), 960</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_make_monic</td>
<td>(C function), 961</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_mul</td>
<td>(C function), 965</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_mul_classical</td>
<td>(C function), 964</td>
<td></td>
</tr>
<tr>
<td>fq_zech_poly_mul_KS</td>
<td>(C function), 965</td>
<td></td>
</tr>
</tbody>
</table>
fq_zech_poly_mul reorder (C function), 964
fq_zech_poly_mulhigh (C function), 966
fq_zech_poly_mulhigh_classical (C function), 966
fq_zech_poly_mul_low (C function), 965
fq_zech_poly_mul_low_classical (C function), 965
fq_zech_poly_mul_low KS (C function), 965
fq_zech_poly_mulmod (C function), 966
fq_zech_poly_mulmod_preinv (C function), 966
fq_zech_poly_neg (C function), 963
fq_zech_poly_one (C function), 961
fq_zech_poly_pow (C function), 967
fq_zech_poly_pow trunc (C function), 969
fq_zech_poly_pow trunc binexp (C function), 969
fq_zech_poly_powmod fmpz binexp (C function), 968
fq_zech_poly_powmod fmpz binexp preinv (C function), 968
fq_zech_poly_powmod fmpz sliding preinv (C function), 968
fq_zech_poly_powmod ui binexp (C function), 967
fq_zech_poly_powmod ui binexp preinv (C function), 967
fq_zech_poly_powmod x fmpz preinv (C function), 969
fq_zech_poly_precompute matrix (C function), 977
fq_zech_poly_print (C function), 978
fq_zech_poly_print pretty (C function), 977
fq_zech_poly_randtest (C function), 961
fq_zech_poly_randtest irreducible (C function), 961
fq_zech_poly_randtest monic (C function), 961
fq_zech_poly_randtest not zero (C function), 961
fq_zech_poly_realloc (C function), 959
fq_zech_poly_rem (C function), 970
fq_zech_poly_remove (C function), 979
fq_zech_poly_reverse (C function), 960
fq_zech_poly_roots (C function), 982
fq_zech_poly_scalar addmul fq_zech (C function), 963
fq_zech_poly_scalar div fq_zech (C function), 964
fq_zech_poly_scalar mul fq_zech (C function), 963
fq_zech_poly_scalar submul fq_zech (C function), 964
fq_zech_poly_set (C function), 961
fq_zech_poly_set coeff (C function), 962
fq_zech_poly_set coeff fmpz (C function), 962
fq_zech_poly_set fmpz mod poly (C function), 961
fq_zech_poly_set fq_zech (C function), 961
fq_zech_poly_set nmod poly (C function), 961
fq_zech_poly_set trunc (C function), 960
fq_zech_poly_shift left (C function), 969
fq_zech_poly_shift right (C function), 969
fq_zech_poly_sq (C function), 967
fq_zech_poly_sqr (C function), 967
fq_zech_poly_sqr classical (C function), 966
fq_zech_poly_sqr KS (C function), 967
fq_zech_poly_sqrtr (C function), 974
fq_zech_poly_sqrtr series (C function), 974
fq_zech_poly_struct (C type), 959
fq_zech_poly_sub (C function), 963
fq_zech_poly_sub series (C function), 963
fq_zech_poly_swap (C function), 961
fq_zech_poly_t (C type), 959
fq_zech_poly_truncate (C function), 960
fq_zech_poly_xgcd (C function), 972
fq_zech_poly_xgcd Euclidean f (C function), 973
fq_zech_poly_zero (C function), 961
fq_zech_pow (C function), 947
fq_zech_pow ui (C function), 947
fq_zech_print (C function), 948
fq_zech_print pretty (C function), 948
fq_zech_pth root (C function), 948
fq_zech_rand (C function), 948
fq_zech_rand not zero (C function), 949
fq_zech_randtest (C function), 948
fq_zech_randtest dense (C function), 948
fq_zech_randtest not zero (C function), 948
fq_zech_reduce (C function), 946
fq_zech_set (C function), 949
fq_zech_set fmpz (C function), 949
fq_zech_set fmpz mod (C function), 949
fq_zech_set fmpz mod mat (C function), 949
fq_zech_set nmod poly (C function), 949
fq_zech_set si (C function), 949
fq_zech_set ui (C function), 949
fq_zech_sqr (C function), 947
fq_zech_sqrtr (C function), 948
fq_zech_struct (C type), 944
fq_zech_sub (C function), 947
fq_zech_sub one (C function), 947
fq_zech_swap (C function), 949
fq_zech_t (C type), 944
fq_zech_trace (C function), 950
fq_zech_zero (C function), 949
fq_zero (C function), 839
FresnelC (C macro), 830
FresnelS (C macro), 830
Fun (C macro), 817
G
Gamma (C macro), 829
GaussLegendreWeight (C macro), 829
GaussSum (C macro), 832
GCD (C macro), 827
GegenbauerC (C macro), 829
GeneralizedBernoulliB (C macro), 832
GeneralizedRiemannHypothesis (C macro), 832
GenerallinearGroup (C macro), 825
get_clock (C function), 22
get_default_mpz_ctx (C function), 267
get_memory_usage (C function), 23
GlaisherConstant (C macro), 820
GoldenRatio (C macro), 819
gr_abs (C function), 45
gr_acos (C function), 62
gr_acos_pi (C function), 62
gr_acosh (C function), 62
gr_acot (C function), 62
gr_acot_pi (C function), 62
gr_acoth (C function), 62
gr_acsc (C function), 62
gr_acsc_pi (C function), 62
gr_acsch (C function), 62
gr_add (C function), 41
gr_add_fmpz (C function), 41
gr_add_fmpq (C function), 41
gr_add_other (C function), 41
gr_add_si (C function), 41
gr_add_ui (C function), 41
gr_addmul (C function), 42
gr_addmul_fmpz (C function), 42
gr_addmul_fmpq (C function), 42
gr_addmul_other (C function), 42
gr_addmul_si (C function), 42
gr_addmul_ui (C function), 42
gr_agm (C function), 66
gr_agm1 (C function), 66
gr_airy (C function), 65
gr_airy_ai (C function), 65
gr_airy_ai_prime (C function), 65
gr_airy_ai_prime_zero (C function), 65
gr_airy_ai_zero (C function), 65
gr_airy_bi (C function), 65
gr_airy_bi_prime (C function), 65
gr_airy_bi_prime_zero (C function), 65
gr_airy_bi_zero (C function), 65
gr_arg (C function), 45
gr_asec (C function), 62
gr_asec_pi (C function), 62
gr_asinh (C function), 62
gr_asinh (C function), 62
gr_asinh (C function), 62
gr_atan (C function), 62
gr_atan2 (C function), 62
gr_barnes_g (C function), 63
gr_bellnum_fmpz (C function), 64
gr_bellnum_fmpq (C function), 64
gr_bellnum_ui (C function), 64
gr_bellnum_vec (C function), 64
gr_bellnumui_fmpz (C function), 63
gr_bellnumui_fmpq (C function), 63
gr_bellnumui_vec (C function), 63
gr_bessel_i (C function), 65
gr_bessel_i_scaled (C function), 65
gr_bessel_j (C function), 65
gr_bessel_j_scaled (C function), 65
gr_bessel_k (C function), 65
gr_bessel_k_scaled (C function), 65
gr_bessel_y (C function), 65
gr_beta (C function), 63
gr_beta_lower (C function), 64
gr_bin (C function), 62
gr_bin_ui (C function), 62
gr_bin_ui_vec (C function), 62
gr_bin_uiui (C function), 62
gr_bin_vec (C function), 62
gr_carlson_rc (C function), 66
gr_carlson_rd (C function), 66
gr_carlson_rf (C function), 66
gr_catalan (C function), 66
gr_ceil (C function), 45
gr_chebyshev_t (C function), 64
gr_chebyshev_t_fmpz (C function), 64
gr_chebyshev_u (C function), 64
gr_chebyshev_u_fmpz (C function), 64
gr_clear (C function), 38
gr_cmp (C function), 46
gr_cmp_other (C function), 46
gr_cmpabs (C function), 46
gr_cmpabs_other (C function), 46
gr_conj (C function), 45
gr_cos (C function), 61
gr_cos_integral (C function), 64
gr_cos_pi (C function), 61
gr_cosh (C function), 61
gr_cosh_integral (C function), 64
gr_cot (C function), 61
gr_cot_pi (C function), 61
gr_coth (C function), 61
gr_coulomb (C function), 65
gr_coulomb_f (C function), 65
gr_coulomb_g (C function), 65
gr_coulomb_hneg (C function), 65
gr_coulomb_hpos (C function), 65
gr_csc (C function), 61
gr_csc_pi (C function), 61
gr_csch (C function), 61
gr_csgn (C function), 45
gr_ctx_arb_get_prec (C function), 51
gr_ctx_arb_set_prec (C function), 51
gr_ctx_ca_get_option (C function), 52
gr_ctx_ca_set_option (C function), 52
gr_ctx_clear (C function), 38
gr_ctx_cmp_coercion (C function), 50
gr_ctx_fmpz_mod_set_primality (C function), 51
gr_ctx_fq_degree (C function), 46
gr_ctx_fq_order (C function), 46
gr_ctx_fq_prime (C function), 46
gr_generic_other_div_vec (C function), 58
gr_generic_other_divexact_vec (C function), 58
gr_generic_other_mul (C function), 55
gr_generic_other_mul_vec (C function), 58
gr_generic_other_pow (C function), 56
gr_generic_other_pow_vec (C function), 58
gr_generic_other_sub (C function), 55
gr_generic_other_sub_vec (C function), 58
gr_generic_pow_fmpz (C function), 56
gr_generic_pow_fmpz_binexp (C function), 56
gr_generic_pow_fmpz_sliding (C function), 56
gr_generic_pow_other (C function), 56
gr_generic_pow_si (C function), 58
gr_generic_pow_ui (C function), 58
gr_generic_pow_ui_binexp (C function), 58
gr_generic_pow_ui_sliding (C function), 56
gr_generic_randtest (C function), 54
gr_generic_randtest_not_zero (C function), 54
gr_generic_rsqrt (C function), 56
gr_generic_stirling_s1_ui_vec (C function), 57
gr_generic_stirling_s1_uui (C function), 57
gr_generic_stirling_s1u_uiui (C function), 57
gr_generic_stirling_s1u_ui_vec (C function), 57
gr_generic_stirling_s2_uiui (C function), 57
gr_generic_sub (C function), 54
gr_generic_sub_mul_vec (C function), 55
gr_generic_sub_mulz (C function), 55
gr_generic_sub_mulz_vec (C function), 55
gr_generic_sub_other (C function), 55
gr_generic_sub_other_div_vec (C function), 58
gr_generic_sub_other_divexact_vec (C function), 58
gr_generic_sub_other_mul_vec (C function), 58
gr_generic_sub_other_pow_vec (C function), 58
gr_generic_sub_other_sub_vec (C function), 58
gr_generic_sub_u (C function), 55
gr_generic_subul (C function), 55
gr_generic_submul_fmpz (C function), 55
gr_generic_submul_fmpq (C function), 55
gr_generic_submul_other (C function), 55
gr_generic_submul_si (C function), 55
gr_generic_swap (C function), 54
gr_generic_vec_add (C function), 58
gr_generic_vec_add_scalar_fmpz (C function), 58
gr_generic_vec_add_scalar_fmpq (C function), 58
gr_generic_vec_add_scalar_fmpz (C function), 58
gr_generic_vec_add_scalar_fmpq (C function), 58
gr_generic_vec_add_scalar_mul_vec (C function), 56
gr_generic_vec_add_scalar_other (C function), 58
gr_generic_vec_add_scalar_other_div_vec (C function), 58
gr_generic_vec_add_scalar_other_divexact_vec (C function), 58
gr_generic_vec_add_scalar_other_mul_vec (C function), 58
gr_generic_vec_add_scalar_other_pow_vec (C function), 58
gr_generic_vec_add_scalar_sub_vec (C function), 58
gr_generic_vec_add_scalar_vec (C function), 58
gr_generic_vec_add_scalar_fmpz (C function), 58
gr_generic_vec_add_scalar_fmpq (C function), 58
gr_generic_vec_add_scalar_si (C function), 58
gr_generic_vec_add_scalar_other_ui (C function), 58
gr_generic_vec_add_scalar_other_other (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
gr_generic_vec_add_scalar_other_fmpq (C function), 58
gr_generic_vec_add_scalar_other_fmpz (C function), 58
Index

1201

generic_vec_divexact_scalar_ui (C function), 58

generic_vec_dot (C function), 57

generic_vec_dot_mpfr (C function), 58

generic_vec_dot_ui (C function), 58

generic_vec_equal (C function), 57

generic_vec_init (C function), 57

generic_vec_is_zero (C function), 57

generic_vec_mul (C function), 58

generic_vec_mul_other (C function), 58

generic_vec_mul_scalar_ui (C function), 58

generic_vec_mul_scalar_fmpz (C function), 58

generic_vec_mul_scalar_si (C function), 58

generic_vec_mul_scalar_fmpq (C function), 58

generic_vec_normalise (C function), 57

generic_vec_normalise_weak (C function), 57

generic_vec_pow (C function), 58

generic_vec_pow_other (C function), 58

generic_vec_pow_scalar (C function), 58

generic_vec_pow_scalar_si (C function), 58

generic_vec_pow_scalar_fmpz (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58

generic_vec_pow_scalar_fmpq (C function), 58
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>gr_lgamma</td>
<td>63</td>
</tr>
<tr>
<td>gr_log</td>
<td>61</td>
</tr>
<tr>
<td>gr_log10</td>
<td>61</td>
</tr>
<tr>
<td>gr_logip</td>
<td>61</td>
</tr>
<tr>
<td>gr_log2</td>
<td>61</td>
</tr>
<tr>
<td>gr_log_barnes_g</td>
<td>63</td>
</tr>
<tr>
<td>gr_log_integral</td>
<td>64</td>
</tr>
<tr>
<td>gr_log_pi_1</td>
<td>61</td>
</tr>
<tr>
<td>gr_mat_add</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_addmul_scalar</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_adjugate</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_adjugate_charpoly</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_apply_row_similarity</td>
<td>81</td>
</tr>
<tr>
<td>gr_mat_charpoly</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_charpoly_berkowitz</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_charpoly_danilevsky</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_charpoly_faddeev</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_charpoly_faddeev_hags</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_charpoly_from_hessenberg</td>
<td>81</td>
</tr>
<tr>
<td>gr_mat_charpoly_gauss</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_charpoly_householder</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_clear</td>
<td>74</td>
</tr>
<tr>
<td>gr_mat_concat_horizontal</td>
<td>75</td>
</tr>
<tr>
<td>gr_mat_concat_vertical</td>
<td>75</td>
</tr>
<tr>
<td>gr_mat_char</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_char_berkowitz</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_char_cofactor</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_char_fflu</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_char_generic</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_char_generic_field</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_char_generic_integral_domain</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_char_lu</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_diag_mul</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_diagonalization</td>
<td>81</td>
</tr>
<tr>
<td>gr_mat_diagonalization_generic</td>
<td>81</td>
</tr>
<tr>
<td>gr_mat_diagonalization_precomp</td>
<td>81</td>
</tr>
<tr>
<td>gr_mat_div_scalar</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_eigenvalues</td>
<td>81</td>
</tr>
<tr>
<td>gr_mat_eigenvalues_other</td>
<td>81</td>
</tr>
<tr>
<td>gr_MAT_ENTRY</td>
<td>73</td>
</tr>
<tr>
<td>gr_MAT_entry_ptr</td>
<td>73</td>
</tr>
<tr>
<td>gr_mat_equal</td>
<td>74</td>
</tr>
<tr>
<td>gr_mat_exp</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_exp_jordan</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_fflu</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_find_nonzero_pivot</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_find_nonzero_pivot_generic</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_find_nonzero_pivot_large_abs</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_gr_poly_evaluate</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_hadamard</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_hessenberg</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_hessenberg_gauss</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_hessenberg_householder</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_hilbert</td>
<td>83</td>
</tr>
<tr>
<td>gr_mat_init</td>
<td>74</td>
</tr>
<tr>
<td>gr_mat_init_set</td>
<td>74</td>
</tr>
<tr>
<td>gr_mat_inv</td>
<td>80</td>
</tr>
<tr>
<td>gr_mat_invert_cols</td>
<td>75</td>
</tr>
<tr>
<td>gr_mat_invert_rows</td>
<td>75</td>
</tr>
<tr>
<td>gr_mat_is_diagonal</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_is_empty</td>
<td>75</td>
</tr>
<tr>
<td>gr_mat_is_hessenberg</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_is_lower_triangular</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_is_neg_one</td>
<td>74</td>
</tr>
<tr>
<td>gr_mat_is_one</td>
<td>74</td>
</tr>
<tr>
<td>gr_mat_is_scalar</td>
<td>74</td>
</tr>
<tr>
<td>gr_mat_is_square</td>
<td>75</td>
</tr>
<tr>
<td>gr_mat_is_upper_triangular</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_is_zero</td>
<td>74</td>
</tr>
<tr>
<td>gr_mat_jordan_blocks</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_jordan_form</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_jordan_transformation</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_log</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_log_jordan</td>
<td>82</td>
</tr>
<tr>
<td>gr_mat_lu</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_lu_classical</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_lu_recursive</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_minpoly_field</td>
<td>81</td>
</tr>
<tr>
<td>gr_mat_mul</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_mul_classical</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_mul_diag</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_mul_generic</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_mul_scalar</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_mul_strassen</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_ncols</td>
<td>73</td>
</tr>
<tr>
<td>gr_mat_neg</td>
<td>76</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_den</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_den_fflu</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_fflu</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_fflu_precomp</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_fflu_precomp</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_lu</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_lu_precomp</td>
<td>78</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_tril</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_tril_classical</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_tril_recursive</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_triu</td>
<td>77</td>
</tr>
<tr>
<td>gr_mat_nonsingular_solve_triu_classical</td>
<td>77</td>
</tr>
</tbody>
</table>
gr_poly_pow_series_ui (C function), 87
gr_poly_pow_series_ui_binexp (C function), 87
gr_poly_pow_ui (C function), 87
gr_poly_print (C function), 86
gr_poly_randtest (C function), 86
gr_poly_rem (C function), 89
gr_poly_resultant (C function), 95
gr_poly_resultant_euclidean (C function), 95
gr_poly_resultant_hgcd (C function), 95
gr_poly_resultant_small (C function), 95
gr_poly_resultant_sylvestre (C function), 95
gr_poly_reverse (C function), 86
gr_poly_revert_series (C function), 93
gr_poly_revert_series_lagrange (C function), 93
gr_poly_revert_series_lagrange_fast (C function), 93
gr_poly_reverts_series_newton (C function), 93
gr_poly_roots (C function), 96
gr_poly_roots_other (C function), 96
gr_poly_rsqrt_series (C function), 91
gr_poly_rsqrt_series_basecase (C function), 91
gr_poly_rsqrt_series_miller (C function), 91
gr_poly_rsqrt_series_neutron (C function), 91
gr_poly_set (C function), 86
gr_poly_set_coeff_fmpq (C function), 86
gr_poly_set_coeff_fmpz (C function), 86
gr_poly_set_coeff_si (C function), 86
gr_poly_set_fmpz (C function), 86
gr_poly_set_fmpq (C function), 86
gr_poly_set_fmpq_poly (C function), 86
gr_poly_set_fmpz_poly (C function), 86
gr_poly_set_ui (C function), 86
gr_poly_set_si (C function), 86
gr_poly_set_scalar (C function), 86
gr_poly_set_ui (C function), 86
gr_poly_shift_left (C function), 87
gr_poly_shift_right (C function), 87
gr_poly_sin_cos_series_basecase (C function), 97
gr_poly_sin_cos_series_tangent (C function), 97
gr_poly_sqrt_series (C function), 91
gr_poly_sqrt_series_basecase (C function), 91
gr_poly_sqrt_series_miller (C function), 91
gr_poly_sqrt_series_neutron (C function), 91
gr_poly_squarefree_part (C function), 96
gr_poly_struct (C type), 85
gr_poly_sub (C function), 87
gr_poly_swap (C function), 85
gr_poly_t (C type), 85
gr_poly_tan_series (C function), 97
gr_poly_tan_series_basecase (C function), 97
gr_poly_tan_series_neutron (C function), 97
gr_poly_taylor_shift (C function), 92
gr_poly_taylor_shift_convolution (C function), 92
gr_poly_taylor_shift_divconver (C function), 92
gr_poly_taylor_shift_horner (C function), 92
gr_poly_truncate (C function), 86
gr_poly_write (C function), 86
gr_poly_xgcd_euclidean (C function), 95
gr_poly_xgcd_hgcd (C function), 95
gr_poly_zero (C function), 86
gr_polygamma (C function), 66
gr_polylog (C function), 66
gs_pos_inf (C function), 46
gs_pow (C function), 44
gs_pow_fmpq (C function), 44
gs_pow_fmpz (C function), 44
gs_pow_other (C function), 44
gs_pow_si (C function), 44
gs_pow_ui (C function), 44
gs_print (C function), 40
gs_println (C function), 40
gs_ptr (C type), 36
gs_randtest (C function), 39
gs_randtest_not_zero (C function), 39
gs_randtest_small (C function), 39
gs_re (C function), 45
gs_rfac (C function), 62
gs_rfac_fmpz (C function), 62
gs_rfac_ui (C function), 62
gs_rfac_vec (C function), 62
gs_rgamma (C function), 63
gs_riemann_xi (C function), 66
gs_rising (C function), 62
gs_rising_ui (C function), 62
gs_rsqrt (C function), 44
gs_sec (C function), 61
gs_sec_pi (C function), 61
gs_sech (C function), 61
gs_set (C function), 40
gs_set_d (C function), 40
gs_set_f (C function), 40
gs_set_fp (C function), 40
gs_set_fmpz (C function), 40
gs_set_fmpq (C function), 40
gs_set_fmpz (C function), 40
gs_set_fmpz_10exp_fmpz (C function), 40
gs_set_fmpz_2exp_fmpz (C function), 40
gs_set_interval_mid_rad (C function), 46
gs_set_other (C function), 40
gs_set_shallow (C function), 38
gs_set_si (C function), 40
gs_set_str (C function), 40
gs_set_ui (C function), 40
gs_sgn (C function), 45
gs_sin (C function), 61
gs_sin_cos (C function), 61
gs_sin_cos_pi (C function), 61
gs_sin_integral (C function), 64
gs_sin_pi (C function), 61
gs_sinc (C function), 61
ifft_butterfly \( (C\ function) \), 261
ifft_butterfly_sqrt2 \( (C\ function) \), 263
ifft_butterfly_twiddle \( (C\ function) \), 263
ifft_mfa_truncate_sqrt2 \( (C\ function) \), 264
ifft_mfa_truncate_sqrt2_outer \( (C\ function) \), 265
ifft_negacyclic \( (C\ function) \), 265
ifft_radix2 \( (C\ function) \), 262
ifft_radix2_twiddle \( (C\ function) \), 264
ifft_truncate \( (C\ function) \), 262
ifft_truncate1 \( (C\ function) \), 262
ifft_truncate1_twiddle \( (C\ function) \), 264
ifft_truncate_sqrt2 \( (C\ function) \), 263
Im \( (C\ macro) \), 826
Implies \( (C\ macro) \), 818
IncompleteBeta \( (C\ macro) \), 830
IncompleteBetaRegularized \( (C\ macro) \), 830
IncompleteEllipticE \( (C\ macro) \), 832
IncompleteEllipticF \( (C\ macro) \), 832
IncompleteEllipticPi \( (C\ macro) \), 832
IndefiniteIntegralEqual \( (C\ macro) \), 826
Infimum \( (C\ macro) \), 823
Infinity \( (C\ macro) \), 822
IntegersGreaterEqual \( (C\ macro) \), 821
IntegersLessEqual \( (C\ macro) \), 821
Integral \( (C\ macro) \), 824
Intersection \( (C\ macro) \), 819
Interval \( (C\ macro) \), 821
invert_lim \( (C\ macro) \), 243
IsEven \( (C\ macro) \), 826
IsHolomorphicOn \( (C\ macro) \), 824
IsMeromorphicOn \( (C\ macro) \), 824
IsOdd \( (C\ macro) \), 826
IsPrime \( (C\ macro) \), 827
Item \( (C\ macro) \), 818

J
JacobiP \( (C\ macro) \), 829
JacobiSymbol \( (C\ macro) \), 827
JacobiTheta \( (C\ macro) \), 832
JacobiThetaEpsilon \( (C\ macro) \), 833
JacobiThetaPermutation \( (C\ macro) \), 833
JacobiThetaQ \( (C\ macro) \), 832

K
KeiperLiLambda \( (C\ macro) \), 832
KhinchinConstant \( (C\ macro) \), 820
KroneckerDelta \( (C\ macro) \), 826
KroneckerSymbol \( (C\ macro) \), 827

L
LaguerreL \( (C\ macro) \), 829
LambertW \( (C\ macro) \), 828
Landaug \( (C\ macro) \), 828
Lattice \( (C\ macro) \), 822
LCM \( (C\ macro) \), 827
LeftLimit \( (C\ macro) \), 824
LegendreP \( (C\ macro) \), 829
LegendrePolynomialZero \( (C\ macro) \), 829
LegendreSymbol \( (C\ macro) \), 827
Length \( (C\ macro) \), 819
LerchPhi \( (C\ macro) \), 831
Less \( (C\ macro) \), 821
LessEqual \( (C\ macro) \), 821
Limit \( (C\ macro) \), 824
LiouvilleLambda \( (C\ macro) \), 827
List \( (C\ macro) \), 818
Log \( (C\ macro) \), 827
LogBarnesG \( (C\ macro) \), 829
LogBarnesGRemainder \( (C\ macro) \), 829
LogGamma \( (C\ macro) \), 829
Logic \( (C\ macro) \), 833
LogIntegral \( (C\ macro) \), 830
LowerGamma \( (C\ macro) \), 830

M
mag_add \( (C\ function) \), 521
mag_add_2exp_fmpz \( (C\ function) \), 521
mag_add_lower \( (C\ function) \), 521
mag_add_ui \( (C\ function) \), 521
mag_add_ui_2exp_si \( (C\ function) \), 521
mag_add_ui_lower \( (C\ function) \), 521
mag_addmul \( (C\ function) \), 521
mag_allocated_bytes \( (C\ function) \), 518
mag_atan \( (C\ function) \), 523
mag_atan_lower \( (C\ function) \), 523
mag_bernoulli_divFac_ui \( (C\ function) \), 524
mag_bin_ui \( (C\ function) \), 524
mag_binpow_ui ui \( (C\ function) \), 523
mag_clear \( (C\ function) \), 518
mag_cmp \( (C\ function) \), 520
mag_cmp_2exp_si \( (C\ function) \), 520
mag_const_pi \( (C\ function) \), 523
mag_const_pi_lower \( (C\ function) \), 523
mag_cosh \( (C\ function) \), 523
mag_cosh_lower \( (C\ function) \), 523
mag_div \( (C\ function) \), 521
mag_div_fmpz \( (C\ function) \), 521
mag_div_lower \( (C\ function) \), 521
mag_div_ui \( (C\ function) \), 521
mag_dump_file \( (C\ function) \), 520
mag_dump_str \( (C\ function) \), 520
mag_equal \( (C\ function) \), 520
mag_exp \( (C\ function) \), 523
mag_exp_lower \( (C\ function) \), 523
mag_exp_tail \( (C\ function) \), 523
mag_expinv \( (C\ function) \), 523
mag_expinv_lower \( (C\ function) \), 523
mag_expm1 \( (C\ function) \), 523
mag_fast_add_2exp_si \( (C\ function) \), 522
mag_fast_addmul \( (C\ function) \), 522
mag_fast_init_set \( (C\ function) \), 522
mag_fast_init_set_arf \( (C\ function) \), 530
mag_fast_is_zero \( (C\ function) \), 522
mag_fast_mul \( (C\ function) \), 522
mag_fast_mul_2exp_si (C function), 522
mag_fast_zero (C function), 522
mag_fprint (C function), 520
mag_geom_series (C function), 523
mag_get_d (C function), 519
mag_get_d_log2_approx (C function), 519
mag_get_fmpz (C function), 519
mag_get_fmpz_lower (C function), 519
mag_hypot_zeta_uiui (C function), 524
mag_inf (C function), 518
mag_init (C function), 518
mag_init_set (C function), 519
mag_init_set_arf (C function), 529
mag_inv (C function), 521
mag_inv_lower (C function), 521
mag_is_finite (C function), 518
mag_is_inf (C function), 518
mag_is_special (C function), 518
mag_is_zero (C function), 518
mag_load_file (C function), 520
mag_load_str (C function), 520
mag_log (C function), 522
mag_log1p (C function), 522
mag_load_fmpz (C function), 520
mag_log_lower (C function), 523
mag_max (C function), 520
mag_min (C function), 520
mag_mul (C function), 521
mag_mul_2exp_fmpz (C function), 521
mag_mul_2exp_fmpz_lower (C function), 519
mag_mul_fmpz (C function), 521
mag_mul_fmpz_lower (C function), 521
mag_set_d (C function), 519
mag_set_d_2exp_fmpz (C function), 519
mag_set_d_2exp_fmpz_lower (C function), 519
mag_set_d_lower (C function), 519
mag_set_fmpz (C function), 519
mag_set_fmpz_2exp_fmpz (C function), 519
mag_set_fmpz_2exp_fmpz_lower (C function), 519
mag_set_fmpz_lower (C function), 519
mag_set_ui (C function), 519
mag_set_ui_2exp_si (C function), 519
mag_set_ui_lower (C function), 519
mag_sinh (C function), 523
mag_sinh_lower (C function), 523
mag_sqrt (C function), 522
mag_sqrt_lower (C function), 522
mag_struct (C type), 518
mag_sub (C function), 521
mag_sub_lower (C function), 521
mag_swap (C function), 518
mag_t (C type), 518
mag_zero (C function), 518
Matrices (C macro), 825
Matrix (C macro), 825
Matrix2x2 (C macro), 825
Max (C macro), 826
Maximum (C macro), 823
MeromorphicDerivative (C macro), 824
MeromorphicLimit (C macro), 824
Min (C macro), 826
Minimum (C macro), 823
Mod (C macro), 826
ModularGroupAction (C macro), 833
ModularGroupFundamentalDomain (C macro), 833
ModularJ (C macro), 832
ModularLambda (C macro), 832
ModularLambdaFundamentalDomain (C macro), 833
MoebiusMu (C macro), 827
mp_limb_t (C type), 16
mp_ptr (C type), 16
mp_size_t (C type), 16
mp_srcptr (C type), 16
mpfr_mat_clear (C function), 1015
mpfr_mat_entry (C function), 1015
mpfr_mat_equal (C function), 1016
mpfr_mat_init (C function), 1015
mpfr_mat_mul_classical (C function), 1016
mpfr_mat_randtest (C function), 1016
mpfr_mat_set (C function), 1015
mpfr_mat_swap (C function), 1015
mpfr_mat_swap_entrywise (C function), 1015
mpfr_mat_zero (C function), 1015
mpn_addmod_2exppl (C function), 260
mpn_addmod_2exppl1 (C function), 260
mpn_CTX_clear (C function), 267
mpn_CTX_init (C function), 267
mpn_CTX_mpn_mul (C function), 267
mpn_CTX_struct (C type), 267
mpn_CTX_t (C type), 267
mpn_div_2expmod_2exppl (C function), 260
mpn_mul_2expmod_2exppl (C function), 260
mpn_mul_default_mpn_CTX (C function), 267
mpn_negmod_2exppl (C function), 260
Index

n_is_prime (C function), 113
n_is_prime_pocklington (C function), 112
n_is_probabprime (C function), 114
n_is_probabprime_BPSW (C function), 113
n_is_probabprime_fermat (C function), 113
n_is_probabprime_fibonacci (C function), 113
n_is_probabprime_lucas (C function), 114
n_is_square (C function), 114
n_is_squarefree (C function), 119
n_is_strong_probabprime2_preinv (C function), 113
n_is_strong_probabprime_precomp (C function), 113
n_jacobi (C function), 108
n_jacobi_unsigned (C function), 108
n_ll_mod_preinv (C function), 107
n_lll_mod_preinv (C function), 107
n_mod2_precomp (C function), 106
n_mod2_preinv (C function), 106
n_mod_precomp (C function), 106
n_moebius_mu (C function), 119
n_moebius_mu_vec (C function), 119
n_mulmod2 (C function), 107
n_mulmod2_preinv (C function), 107
n_mulmod_precomp (C function), 107
n_mulmod_precomp_shoup (C function), 110
n_mulmod_preinv (C function), 107
n_mulmod_shoup (C function), 110
n_nextprime (C function), 111
n_nth_prime (C function), 111
n_nth_prime_bounds (C function), 111
n_pow (C function), 105
n_powmod (C function), 109
n_powmod2 (C function), 109
n_powmod2_fmpz_preinv (C function), 109
n_powmod2_preinv (C function), 109
n_powmod_ui_preinv (C function), 109
n_powmod_precomp (C function), 109
n_powmod ui precomp (C function), 109
n_precompute_inverse (C function), 106
n_preinvert limb (C function), 105
n_preinvert limb_prenorm (C function), 105
n_prime_inverses_arr_readonly (C function), 111
n_prime_pi (C function), 111
n_prime_pi_bounds (C function), 111
n_primes_arr_readonly (C function), 111
n_primes_clear (C function), 110
n_primes_extend_small (C function), 111
n_primes_init (C function), 110
n_primes_jump_after (C function), 110
n_primes_next (C function), 110
n_primes_sieve_range (C function), 111
n_primitive_root_prime (C function), 120
n_primitive_root_prime_prefactor (C function), 120
n_randbits (C function), 104
n_randint (C function), 104
n_randlimb (C function), 104
n_randprime (C function), 104
n_randtest (C function), 104
n_randtest_bits (C function), 104
n_randtest_not_zero (C function), 104
n_randtest_prime (C function), 104
n_remove (C function), 116
n_remove2_precomp (C function), 116
n_revin (C function), 105
n_rootrem (C function), 115
n_sizeinbase (C function), 105
n_sqrt (C function), 114
n_sqrtmod (C function), 110
n_sqrtmod2pow (C function), 110
n_sqrtmod_primepow (C function), 110
n_sqrtmodn (C function), 110
n_sqrtrem (C function), 114
n_submod (C function), 109
n_urandint (C function), 104
n_xgcd (C function), 108
Neg (C macro), 820
nf_clear (C function), 471
nf_elem_add (C function), 473
nf_elem_canonicalise (C function), 471
nf_elem_clear (C function), 471
nf_elem_coprime_den (C function), 475
nf_elem_coprime_den_signed (C function), 475
nf_elem_div (C function), 474
nf_elem_equal (C function), 473
nf_elem_get_den (C function), 473
nf_elem_get_fmpq_poly (C function), 472
nf_elem_get_fmpz_mat_row (C function), 472
nf_elem_get_fmpz_mod_poly_den (C function), 472
nf_elem_get_fmpz_mod_poly (C function), 472
nf_elem_get_mmod_poly (C function), 472
nf_elem_get_mmod_poly_den (C function), 472
nf_elem_init (C function), 471
nf_elem_inv (C function), 474
nf_elem_is_one (C function), 473
nf_elem_is_zero (C function), 473
nf_elem_norm (C function), 474
nf_elem_norm_div (C function), 474
nf_elem_one (C function), 473
nf_elem_pow (C function), 474
nf_elem_print pretty (C function), 473
nf_elem_randtest (C function), 471
nf_elem_reduce (C function), 472
nf_elem_rep_mat (C function), 475
nf_elem_rep_mat_fmpz_mat_den (C function), 475
nf_elem_set (C function), 473
nf_elem_set_den (C function), 473
nf_elem_set_fmpz_poly (C function), 472
nf_elem_set_fmpz_mat_row (C function), 472
nf_elem_smod_fmpz (C function), 475
nf_elem_smod_fmpz_den (C function), 475
nf_elem_struct (C type), 471
nf_elem_sub (C function), 474
nf_elem_swap (C function), 473
nf_elem_trace (C function), 474
nf_elem_zero (C function), 473
nf_init (C function), 471
nf_struct (C type), 471
nf_t (C type), 471
NMOD2_RED2 (C macro), 333
nmod_add (C function), 334
NMOD_ADDMUL (C macro), 333
nmod_berlekamp_massey_add_point (C function), 378
nmod_berlekamp_massey_add_points (C function), 378
nmod_berlekamp_massey_add_zeros (C function), 378
nmod_berlekamp_massey_clear (C function), 378
nmod_berlekamp_massey_init (C function), 378
nmod_berlekamp_massey_point_count (C function), 378
nmod_berlekamp_massey_points (C function), 378
nmod_berlekamp_massey_R_poly (C function), 379
nmod_berlekamp_massey_reduce (C function), 378
nmod_berlekamp_massey_set_prime (C function), 378
nmod_berlekamp_massey_start_over (C function), 378
nmod_berlekamp_massey_V_poly (C function), 378
NMOD_BITS (C macro), 333
NMOD_CAN_USE_SHOUP (C macro), 333
nmod_discrete_log_pohlig_hellman_clear (C function), 334
nmod_discrete_log_pohlig_hellman_init (C function), 334
nmod_discrete_log_pohlig_hellman_precompute (C function), 334
nmod_discrete_log_pohlig_hellman_primitive_root (C function), 334
nmod_discrete_log_pohlig_hellman_run (C function), 335
nmod_div (C function), 334
nmod_divides (C function), 334
nmod_init (C function), 333
nmod_inv (C function), 334
nmod_mat_add (C function), 341
nmod_mat_addmul (C function), 342
nmod_mat_can_solve (C function), 344
nmod_mat_can_solve_inner (C function), 344
nmod_mat_charpoly (C function), 346
nmod_mat_charpoly_berkowitz (C function), 346
nmod_mat_charpoly_danilevsky (C function), 346
nmod_mat_clear (C function), 338
nmod_mat_concat_horizontal (C function), 339
nmod_mat_concat_vertical (C function), 339
nmod_mat_det (C function), 343
nmod_mat_dhetowell (C function), 343
nmod_mat_entry_ptr (C function), 346
nmod_mat_equal (C function), 340
nmod_mat_fprint (C function), 339
nmod_mat_fprint_query (C function), 339
nmod_mat_get_entry (C function), 338
nmod_mat_howell_form (C function), 346
nmod_mat_init (C function), 338
nmod_mat_init_set (C function), 338
nmod_mat_inv (C function), 343
nmod_mat_invert_cols (C function), 340
nmod_mat_invert_rows (C function), 340
nmod_mat_is_zero (C function), 338
nmod_mat_is_zero_row (C function), 340
nmod_mat_lu (C function), 344
nmod_mat_lu_classical (C function), 344
nmod_mat_lu_classical_delayed (C function), 344
nmod_mat_lu_recursive (C function), 344
nmod_mat_minpoly (C function), 346
nmod_mat_mul (C function), 341
nmod_mat_mul_classical_threaded (C function), 341
nmod_mat_mul_nmod_vec (C function), 342
nmod_mat_mul_nmod_vec_ptr (C function), 342
nmod_mat_mul_strassen (C function), 342
nmod_mat_mul_classical (C function), 341
nmod_mat_nmod_vec_mul (C function), 342
nmod_mat_nmod_vec_mul_ptr (C function), 342
nmod_mat_nullspace (C function), 345
nmod_mat_permute_rows (C function), 340
nmod_mat_print (C function), 339
nmod_mat_print_pretty (C function), 339
nmod_mat_randfull (C function), 339
nmod_mat_randops (C function), 340
nmod_mat_randpermdiag (C function), 339
nmod_mat_randrank (C function), 340
nmod_mat_randtest (C function), 339
nmod_mat_randtril (C function), 340
nmod_mat_randtriu (C function), 340
nmod_mpoly_get_coeff_vars_ui (C function), 391
nmod_mpoly_set_coeff_ui (C function), 389
nmod_mpoly_get_term (C function), 392
nmod_mpoly_get_term_coef_ui (C function), 391
nmod_mpoly_get_term_exp_fmpz (C function), 392
nmod_mpoly_get_term_exp_si (C function), 392
nmod_mpoly_get_term_exp_ui (C function), 392
nmod_mpoly_set_term_coef_ui (C function), 392
nmod_mpoly_set_term_exp_fmpz (C function), 392
nmod_mpoly_set_term_exp_si (C function), 392
nmod_mpoly_set_term_exp_ui (C function), 392
nmod_mpoly_set_term_var_exp_ui (C function), 391
nmod_mpoly_get_term_var_exp_ui (C function), 392
nmod_mpoly_set_term_var_exp_fmpz (C function), 392
nmod_mpoly_set_term_coef_ui (C function), 391
nmod_mpoly_set_term_exp_ui (C function), 392
nmod_mpoly_init (C function), 390
nmod_mpoly_init2 (C function), 388
nmod_mpoly_init3 (C function), 388
nmod_mpoly_is_canonical (C function), 391
nmod_mpoly_is_gen (C function), 389
nmod_mpoly_is_one (C function), 390
nmod_mpoly_is_square (C function), 397
nmod_mpoly_is_ui (C function), 390
nmod_mpoly_is_zero (C function), 390
nmod_mpoly_length (C function), 391
nmod_mpoly_make_monic (C function), 393
nmod_mpoly_mul (C function), 395
nmod_mpoly_mul_array (C function), 395
nmod_mpoly_mul_array_threaded (C function), 395
nmod_mpoly_mul_dense (C function), 395
nmod_mpoly_mul_heap_threads (C function), 395
nmod_mpoly_mul_johnson (C function), 395
nmod_mpoly_neg (C function), 393
nmod_mpoly_one (C function), 390
nmod_mpoly_pow (C function), 395
nmod_mpoly_pow_rmul (C function), 398
nmod_mpoly_pow_ui (C function), 395
nmod_mpoly_print_pretty (C function), 389
nmod_mpoly_push_term_ui_fmpz (C function), 392
nmod_mpoly_push_term_ui_fmpz (C function), 392
nmod_mpoly_set (C function), 389
nmod_mpoly_set_term_coeff_ui_fmpz (C function), 391
nmod_mpoly_set_coeff_ui_monomial (C function), 391
nmod_mpoly_set_coeff_ui_ui (C function), 391
nmod_mpoly_set_term_str_pretty (C function), 389
nmod_mpoly_set_term_coef_ui (C function), 392
nmod_mpoly_set_str_pretty (C function), 390
nmod_mpoly_sort_terms (C function), 392
nmod_mpoly_sqrt (C function), 397
nmod_mpoly_struct (C type), 388
nmod_mpoly_sub (C function), 393
nmod_mpoly_sub_ui (C function), 393
nmod_mpoly_swap (C function), 389
nmod_mpoly_t (C type), 388
nmod_mpoly_term_coefs (C function), 391
nmod_mpoly_term_content (C function), 393
nmod_mpoly_term_exp_fits_si (C function), 397
nmod_mpoly_total_degree_fits_si (C function), 390
nmod_mpoly_total_degree_fits_fmpz (C function), 390
nmod_mpoly_total_degree_si (C function), 390
nmod_mpoly_univar_clear (C function), 397
nmod_mpoly_univar_degree_fits_si (C function), 397
nmod_mpoly_univar_get_term_coeff (C function), 398
nmod_mpoly_univar_get_term_exp_si (C function), 398
nmod_mpoly_univar_get_term_exp_ui (C function), 398
nmod_mpoly_univar_init (C function), 397
nmod_mpoly_univar_length (C function), 397
nmod_mpoly_univar_swap (C function), 397
nmod_mpoly_univar_swap_term_coeff (C function), 398
nmod_mpoly_used_vars (C function), 390
nmod_mpoly_zero (C function), 390
nmod_mul (C function), 334
NMUL_MUL_FULLWORD (C macro), 333
NMUL_MUL_PRESIGNED (C macro), 333
nmod_neg (C function), 334
nmod_poly_add (C function), 352
nmod_poly_add (C function), 352
nmod_poly_add_series (C function), 352
nmod_poly_asin_series (C function), 374
nmod_poly_asinh_series (C function), 374
nmod_poly_atan_series (C function), 374
nmod_poly_atan_series (C function), 374
nmod_poly_bit_pack (C function), 353
nmod_poly_bit_unpack (C function), 353
nmod_poly_clear (C function), 348
nmod_poly_compose (C function), 364
padic_poly_shift_right (C function), 997
padic_poly_sub (C function), 995
padic_poly_swap (C function), 994
padic_poly_truncate (C function), 993
padic_poly_val (C function), 993
padic_poly_zero (C function), 994
padic_pow_si (C function), 989
padic_prec (C function), 985
padic_print (C function), 992
padic_randtest (C function), 987
padic_randtest_int (C function), 987
padic_randtest_not_zero (C function), 987
padic_reduce (C function), 986
padic_set (C function), 987
padic_set_fmpz (C function), 987
padic_set_fmpq (C function), 987
padic_set_mpq (C function), 987
padic_set_ui (C function), 987
padic_shift (C function), 988
padic_sqrt (C function), 989
padic_sub (C function), 988
padic_swap (C function), 987
padic_techmuller (C function), 991
padic_unit (C function), 985
padic_val (C function), 985
padic_val_fac (C function), 991
padic_val_fac_ui (C function), 991
padic_val_fac_ui_2 (C function), 991
padic_zero (C function), 987
Parentheses (C macro), 833
partitions_fmpz_fmpz (C function), 719
partitions_fmpz_ui (C function), 719
partitions_fmpz_ui_using_doubles (C function), 720
partitions_hrr_sum_arb (C function), 719
partitions_leading_fmpz (C function), 720
partitions_rademacher_bound (C function), 719
Partitions (C macro), 828
Path (C macro), 824
Pi (C macro), 819
Pol (C macro), 825
Poles (C macro), 824
PolyLog (C macro), 831
Polynomial (C macro), 825
PolynomialDegree (C macro), 825
PolynomialFractions (C macro), 825
PolynomialRootIndexed (C macro), 820
PolynomialRootNearest (C macro), 820
Polynomials (C macro), 825
Pos (C macro), 820
Pow (C macro), 820
Prime (C macro), 827
PrimePi (C macro), 827
PrimeProduct (C macro), 823
Primes (C macro), 821
PrimeSum (C macro), 823
PrimitiveDirichletCharacters (C macro), 832
PrimitiveReducedPositiveIntegralBinaryQuadraticForms (C macro), 833
Product (C macro), 823
prof_repeat (C function), 22
ProjectiveComplexNumbers (C macro), 822
ProjectiveRealNumbers (C macro), 822
PSL2Z (C macro), 825
psl2z_clear (C function), 669
psl2z_equal (C function), 670
psl2z_fprint (C function), 669
psl2z_init (C function), 669
psl2z_inv (C function), 670
psl2z_is_correct (C function), 670
psl2z_is_one (C function), 669
psl2z_mul (C function), 670
psl2z_one (C function), 669
psl2z_print (C function), 669
psl2z_randtest (C function), 670
psl2z_set (C function), 669
psl2z_struct (C type), 669
psl2z_swap (C function), 669
psl2z_t (C type), 669
PTR_TO_COEFF (C function), 122
Q
qadic_add (C function), 1006
qadic_clear (C function), 1004
qadic_ctx_clear (C function), 1004
qadic_ctx_degree (C function), 1004
qadic_ctx_init (C function), 1003
qadic_ctx_init_conway (C function), 1003
qadic_ctx_print (C function), 1004
qadic_equal (C function), 1005
qadic_exp (C function), 1007
qadic_exp_balanced (C function), 1007
qadic_exp_rectangular (C function), 1007
qadic_fprint_pretty (C function), 1009
qadic_frobenius (C function), 1009
qadic_gen (C function), 1005
qadic_get_padic (C function), 1005
qadic_init (C function), 1004
qadic_init2 (C function), 1004
qadic_inv (C function), 1006
qadic_is_one (C function), 1005
qadic_is_zero (C function), 1005
qadic_log (C function), 1008
qadic_log_balanced (C function), 1008
qadic_log_rectangular (C function), 1008
qadic_mul (C function), 1006
qadic_neg (C function), 1006
qadic_norm (C function), 1009
qadic_norm_analytic (C function), 1009
qadic_norm_resultant (C function), 1009
qadic_one (C function), 1005
qadic_pow (C function), 1005
qadic_prec (C function), 1006
qadic_print_pretty (C function), 1010
qsieve_compute_pre_data (C function), 269
qsieve_do_sieving (C function), 269
qsieve_do_sieving2 (C function), 269
qsieve_evaluate_candidate (C function), 269
qsieve_evaluate_sieve (C function), 269
qsieve_factor (C function), 270
qsieve_get_table_entry (C function), 269
qsieve_init_A0 (C function), 268
qsieve_init_poly_first (C function), 269
qsieve_init_poly_next (C function), 269
qsieve_insert_relation2 (C function), 270
qsieve_knuth_schroeppel (C function), 268
qsieve_merge_relation (C function), 270
qsieve_next_A0 (C function), 269
qsieve_parse_relation (C function), 269
qsieve_primes_init (C function), 268
qsieve_process_relation (C function), 270
qsieve_remove_duplicates (C function), 270
qsieve_write_to_file (C function), 269
QuotientRing (C macro), 826

R
Range (C macro), 821
Re (C macro), 826
RealAbs (C macro), 826
RealAlgebraicNumbers (C macro), 821
RealBall (C macro), 822
RealDerivative (C macro), 824
RealInfinities (C macro), 822
RealLimit (C macro), 824
RealSignedInfinities (C macro), 822
RealSingularityClosure (C macro), 823
Repeat (C macro), 817
Residue (C macro), 824
RiemannHypothesis (C macro), 831
RiemannXi (C macro), 831
RiemannZeta (C macro), 831
RiemannZetaZero (C macro), 831
RightLimit (C macro), 824
Rings (C macro), 826
RisingFactorial (C macro), 829
Root (C macro), 820
RootOfUnity (C macro), 820
Row (C macro), 825
RowMatrix (C macro), 825
RR (C macro), 821

S
Same (C macro), 817
sdiv_qrnd (C macro), 243
Sac (C macro), 827
Sech (C macro), 827
SequenceLimit (C macro), 824
SequenceLimitInferior (C macro), 824
SequenceLimitSuperior (C macro), 824
Ser (C macro), 825
Set (C macro), 818
SetMinus (C macro), 819
Sets (C macro), 819
SHOW_MEMORY_USAGE (C macro), 23
ShowExpandedUsage (C macro), 833
Sign (C macro), 826
signed_mpn_sub_n (C function), 347
SignExtendedComplexNumbers (C macro), 822
Sin (C macro), 827
Sinc (C macro), 828
SingularValues (C macro), 825
Sinh (C macro), 827
SinhIntegral (C macro), 830
SinhIntegral (C macro), 830
SL2Z (C macro), 825
SloaneA (C macro), 828
long (C type), 16
smul_ppmm (C macro), 243
Solutions (C macro), 823
sp2gz_block_diag (C function), 678
sp2gz_decompose (C function), 679
sp2gz_dim (C function), 678
sp2gz_embed (C function), 678
sp2gz_fundamental (C function), 678
sp2gz_inv (C function), 679
sp2gz_is_block_diag (C function), 678
sp2gz_is_correct (C function), 678
sp2gz_is_embedded (C function), 678
sp2gz_is_j (C function), 678
sp2gz_is_trig (C function), 678
sp2gz_j (C function), 678
sp2gz_nb_fundamental (C function), 678
sp2gz_randtest (C function), 679
sp2gz_restrict (C function), 678
sp2gz_set_blocks (C function), 678
sp2gz_trig (C function), 678
SpecialLinearGroup (C macro), 825
Spectrum (C macro), 825
SphericalHarmonicY (C macro), 829
Sqrt (C macro), 820
SquaresR (C macro), 827
start_clock (C function), 22
Step (C macro), 817
StieltjesGamma (C macro), 832
StirlingCycle (C macro), 828
StirlingS1 (C macro), 828
StirlingS2 (C macro), 828
StirlingSeriesRemainder (C macro), 829
stop_clock (C function), 22
Sub (C macro), 820
sub_ddmmmssss (C macro), 243
sub_ddmmss (C macro), 243
Subscript (C macro), 833
Subset (C macro), 819
SubsetEqual (C macro), 819
Subsets (C macro), 819
Sum (C macro), 823
Supremum (C macro), 823
SymmetricPolynomial (C macro), 828
vec8d_set_d (C function), 31
vec8d_set_d8 (C function), 31
vec8d_store (C function), 30
vec8d_store_aligned (C function), 30
vec8d_store_unaligned (C function), 31
vec8d_sub (C function), 32
vec8d_zero (C function), 32
vec8n (C type), 30
vec8n_addmod (C function), 34
vec8n_addmod_limited (C function), 34
vec8n_bit_and (C function), 33
vec8n_bit_shift_right (C function), 33
vec8n_convert_limited_vec8d (C function), 31
vec8n_load_unaligned (C function), 30
vec8n_set_n (C function), 31

W
WeierstrassP (C macro), 833
WeierstrassSigma (C macro), 833
WeierstrassZeta (C macro), 833
Where (C macro), 817

X
XGCD (C macro), 827

Z
z_kronecker (C function), 243
z_mul_checked (C function), 242
z_randint (C function), 242
z_randtest (C function), 242
z_randtest_not_zero (C function), 242
z_sizeinbase (C function), 242
Zero (C macro), 825
ZeroMatrix (C macro), 825
Zeros (C macro), 823
ZZ (C macro), 821