FLINT

Fast Library for Number Theory

Version 2.2.0

4 June 2011

William Hart*, Fredrik Johansson†, Sebastian Pancratz‡

* Supported by EPSRC Grant EP/G004870/1
† Supported by Austrian Science Foundation (FWF) Grant Y464-N18
‡ Supported by European Research Council Grant 204083

Contents

§1. Introduction

FLINT is a C library of functions for doing number theory. It is highly optimised and can be compiled on numerous platforms. FLINT also has the aim of providing support for multicore and multiprocessor computer architectures, though we do not yet provide this facility.

FLINT is currently maintained by William Hart of Warwick University in the UK. Its main authors are William Hart, Sebastian Pancratz, Fredrik Johansson, Andy Novocin and David Harvey (no longer active).

FLINT 2 and following should compile on any machine with GCC and a standard GNU toolchain, however it is specially optimised for x86 (32 and 64 bit) machines. As of version 2.0, FLINT required GCC version 2.96 or later, MPIR 2.1.1 or later and MPFR 3.0.0 or later.

FLINT is supplied as a set of modules, `fmpz`, `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 `fmpz_poly_add` located in `fmpz_poly/add.c` has test code in the file `fmpz_poly/test/t-add.c`.

§2. Building and using FLINT

The easiest way to use FLINT is to build a shared library. Simply download the FLINT tarball and untar it on your system.

FLINT requires MPIR version 2.1.1 or later and MPFR 3.0.0 or later.

To configure FLINT you must specify where MPIR and MPFR are on your system. FLINT can work with the libraries installed as usual, e.g. in `/usr` or it can work with the libraries built from source in their standard source trees.

In the case that a library is installed in say `/usr/local` in the `lib` and `include` directories as usual, simply specify the top level location, e.g. `/usr/local` when configuring FLINT. If a library is built in its source tree, specify the top level source directory, e.g. `/home/user1/mpir/`.

To specify the directories where the libraries reside, you must pass the directories as parameters to FLINT's configure, e.g.

```
./configure --with-mpir=/usr/local/
   --with-mpfr=/home/user1/mpfr/
```

If no directories are specified, FLINT assumes it will find the libraries it needs in `/usr`.

Note that FLINT builds static and shared libraries by default. If you do not require one of these then you may pass `--disable-static` or `--disable-shared` to `configure`. When running `make check` a shared library is required.

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.

Once FLINT is configured, in the main directory of the FLINT directory tree simply type:

```
make
make check
```

If you wish to install FLINT, simply type:

```
make install
```

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 and MPIR and MPFR with the options `-lflint -lmpfr -lgmp`.

Note that you may have to set `LD_LIBRARY_PATH` or equivalent for your system to let the linker know where to find these libraries. Please refer to your system documentation for how to do this.

If you have any difficulties with conflicts with system headers on your machine, you can do the following in your code:

```
#undef ulong
#include <stdio.h>
// other system headers
#define ulong unsigned long
```

This prevents FLINT's definition of `ulong` interfering with your system headers.

The FLINT make system responds to the standard commands

```
make
make library
make check
make clean
make distclean
make install
```

In addition, 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 in inverted commas, e.g:

```
make check MOD=ulong_extras
```

If your system supports parallel builds, FLINT will build in parallel, e.g:

```
make -j4 check
```

§3. Test code

Each module of FLINT has an extensive associated test module. We strongly recommend running the test programs before relying on results from FLINT on your system.

To make and run the test programs, simply type:

```
make check
```

in the main FLINT directory after configuring FLINT.

§4. Reporting bugs

The maintainer wishes to be made aware of any and all bugs. Please send an email with your bug report to hart_wb@yahoo.com or report them on the FLINT devel list https://groups.google.com/group/flint-devel?hl=en.

If possible please include details of your system, the version of GCC, the versions of MPIR and MPFR as well as precise details of how to replicate the bug.

Note that FLINT needs to be linked against version 2.1.1 or later of MPIR, version 3.0.0 or later of MPFR and must be compiled with gcc version 2.96 or later.

§5. Contributors

FLINT has been developed since 2007 by a large numnber of people. Initially the library was started by David Harvey and William Hart. Later maintenance of the library was taken over solely by William Hart.

The main authors of FLINT to date have been William Hart, David Harvey (no longer active), Fredrik Johansson, Sebastian Pancratz and Andy Novocin.

Other significant contributions to FLINT have been made by Jason Papadopoulos, Gonzalo Tornaria, David Howden, Burcin Erocal, Tom Boothby, Daniel Woodhouse, Tomasz Lechowski, Richard Howell-Peak and Peter Shrimpton.

Jan Tuitman contributed to the design of the padics module.

Additional research was contributed by Daniel Scott and Daniel Ellam.

Further patches and bug reports have been made by Michael Abshoff, Didier Deshommes, Craig Citro, Timothy Abbot, Carl Witty, Jaap Spies, Kiran Kedlaya, William Stein, Robert Bradshaw, Serge Torres and many others.

Some code (`longlong.h` and `clz_tab.c`) has been used from an LGPL v2+ version of the GMP library. The main author of the GMP library is Torbjorn Granlund.

FLINT 2 was a complete rewrite from scratch which began in about 2010.

§6. Example programs

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

```
make examples
```

The example programs are built in the `build/examples` directory. You must set your `LD_LIBRARY_PATH` or equivalent for the flint, mpir and mpfr libraries. See your operating system documentation to see how to set this.

The current example programs are:

`delta_qexp` Computes the first n terms of the delta function, e.g. `build/examples/delta_qexp 1000000` will compute the first one million terms 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.

§7. FLINT 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 `LONG_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 function `FLINT_BIT_COUNT(x)` returns the number of binary bits required to represent an `unsigned long x`. If `x` is zero, returns 0.

§8. fmpz

Arbitrary precision integers

8.1 Introduction

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/ MPIR'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/ MPIR'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 long 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.

```
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

```
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.

8.2 Simple example

The following example computes the square of the integer 7 and prints the result.

```
#include "fmpz.h"
...
fmpz_t x, y;
fmpz_init(x);
fmpz_init(y);
fmpz_set_ui(x, 7);
fmpz_mul(y, x, x);
fmpz_print(x);
printf("^2 = ");
fmpz_print(y);
printf("\n");
fmpz_clear(x);
fmpz_clear(y);
```

The output is:

```
7^2 = 49
```

We now describe the functions available in the `fmpz` module.

8.3 Memory management

```
void fmpz_init(fmpz_t f)
```

A small `fmpz_t` is initialised, i.e. just a `long`. 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 `long`. 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.

8.4 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.

```
void fmpz_randbits(fmpz_t f, flint_rand_t state,
   mp_bitcnt_t bits)
```

Generates a random signed integer whose absolute value has the given number of bits.

```
void fmpz_randtest(fmpz_t f, flint_rand_t state,
   mp_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,
    mp_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,
    mp_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, 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.

8.5 Conversion

```
ulong fmpz_get_si(const fmpz_t f)
```

Returns f as a `signed long`. The result is undefined if f does not fit into a `long`.

```
ulong fmpz_get_ui(const fmpz_t f)
```

Returns f as an `unsigned long`. The result is undefined if f does not fit into an `unsigned long` or is negative.

```
double fmpz_get_d_2exp(long * exp, const fmpz_t f)
```

Returns f as a normalized `double` along with a 2-exponent `exp`, i.e. if r is the return value then `f = r * 2^exp`, to within 1 ULP.

```
void fmpz_get_mpz(mpz_t x, const fmpz_t f)
```

Sets the `mpz_t` x to the same value as f.

```
char * fmpz_get_str(char * str, int b, const fmpz_t f)
```

Returns the representation of f in base b, which can vary between 2 and 62, inclusive.

If `str` is `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.

```
void fmpz_set_si(fmpz_t f, long val)
```

Sets f to the given `signed long` value.

```
void fmpz_set_ui(fmpz_t f, ulong val)
```

Sets f to the given `unsigned long` value.

```
void fmpz_set_mpz(fmpz_t f, const mpz_t x)
```

Sets f to the given `mpz_t` value.

```
int fmpz_set_str(fmpz_t f, char * str, int b)
```

Sets f to the value given in the null-terminated string `str`, in base b. The base b can vary between 2 and 62, inclusive. Returns 0 if the string contains a valid input and -1 otherwise.

```
void fmpz_set_ui_mod(fmpz_t f, mp_limb_t x, mp_limb_t m)
```

Sets f to the signed remainder $y \equiv x \bmod m$ satisfying $-m/2 \leq y < m/2$, given x which is assumed to satisfy $0 \leq x < m$.

8.6 Input and output

```
int fmpz_read(fmpz_t f)
```

Reads a multiprecision 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, returns a positive 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 MPIR.

```
int fmpz_fread(FILE * file, fmpz_t f)
```

Reads a multiprecision 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, returns a positive 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 MPIR.

```
int fmpz_print(fmpz_t x)
```

Prints the value x to `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.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

This convention is adopted in light of the return values of `printf` from the standard library and `mpz_out_str` from MPIR.

```
int fmpz_fprint(FILE * file, fmpz_t x)
```

Prints the value x to `file`, 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.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

This convention is adopted in light of the return values of `printf` from the standard library and `mpz_out_str` from MPIR.

8.7 Basic properties and manipulation

```
size_t fmpz_sizeinbase(const fmpz_t f, int b)
```

Returns the size of f in base b, measured in numbers of digits. The base b can be between 2 and 62, inclusive.

```
mp_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 is the sign of f is negative, $+1$ if it is positive, otherwise returns 0.

```
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.

8.8 Comparison

```
int fmpz_cmp(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_cmp_ui(const fmpz_t f, ulong 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_equal(const fmpz_t f, const fmpz_t 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.

8.9 Basic arithmetic

```
void fmpz_neg(fmpz_t f1, const fmpz_t f2)
```

Sets f_1 to $-f_2$.

```
void fmpz_abs(fmpz_t f1, const fmpz_t f2)
```

Sets f_1 to the absolute value of f_2.

```
void fmpz_add(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to $g + h$.

```
void fmpz_add_ui(fmpz_t f, const fmpz_t g, ulong x)
```

Sets f to $g + x$ where x is an `unsigned long`.

```
void fmpz_sub(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to $g - h$.

```
void fmpz_sub_ui(fmpz_t f, const fmpz_t g, ulong x)
```

Sets f to $g - x$ where x is an `unsigned long`.

```
void fmpz_mul(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to $g \times h$.

```
void fmpz_mul_si(fmpz_t f, const fmpz_t g, long x)
```

Sets f to $g \times x$ where x is a `signed long`.

```
void fmpz_mul_ui(fmpz_t f, const fmpz_t g, ulong x)
```

Sets f to $g \times x$ where x is an `unsigned long`.

```
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 `unsigned long`.

```
void fmpz_mul_2exp(fmpz_t f, const fmpz_t g, ulong e)
```

Sets f to $g \times 2^e$.

```
void fmpz_addmul(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to $f + g \times h$.

```
void fmpz_addmul_ui(fmpz_t f, const fmpz_t g, ulong x)
```

Sets f to $f + g \times x$ where x is an `unsigned long`.

```
void fmpz_submul(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to $f - g \times h$.

```
void fmpz_submul_ui(fmpz_t f, const fmpz_t g, ulong x)
```

Sets f to $f - g \times x$ where x is an `unsigned long`.

```
void fmpz_cdiv_q(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to the quotient of g by h, rounding up towards infinity. If h is 0 an exception is raised.

```
void fmpz_cdiv_q_si(fmpz_t f, const fmpz_t g, long h)
```

Sets f to the quotient of g by h, rounding up towards infinity. If h is 0 an exception is raised.

```
void fmpz_cdiv_q_ui(fmpz_t f, const fmpz_t g, ulong h)
```

Sets f to the quotient of g by h, rounding up towards infinity. If h is 0 an exception is raised.

```
void fmpz_fdiv_q(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to the quotient of g by h, rounding down towards minus infinity. If h is 0 an exception is raised.

```
void fmpz_fdiv_q_si(fmpz_t f, const fmpz_t g, long h)
```

Set f to the quotient of g by h, rounding down towards minus infinity. If h is 0 an exception is raised.

```
void fmpz_fdiv_q_ui(fmpz_t f, const fmpz_t g, ulong h)
```

Set f to the quotient of g by h, rounding down towards minus infinity. If h is 0 an exception is raised.

```
void fmpz_fdiv_qr(fmpz_t f, fmpz_t s, const fmpz_t g, const
    fmpz_t h)
```

Sets f to the quotient of g by h, rounding down towards minus infinity and s to the remainder. If h is 0 an exception is raised.

```
void fmpz_fdiv_q_2exp(fmpz_t f, const fmpz_t g, ulong exp)
```

Sets f to g divided by `2^exp`, but rounded down.

```
void fmpz_tdiv_q(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to the quotient of g by h, rounding down towards zero. If h is 0 an exception is raised.

```
void fmpz_tdiv_q_si(fmpz_t f, const fmpz_t g, long h)
```

Set f to the quotient of g by h, rounding down towards zero. If h is 0 an exception is raised.

```
void fmpz_tdiv_q_ui(fmpz_t f, const fmpz_t g, ulong h)
```

Set f to the quotient of g by h, rounding down towards zero. If h is 0 an exception is raised.

```
void fmpz_divexact(fmpz_t f, const fmpz_t g, const fmpz_t 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_divexact_si(fmpz_t f, const fmpz_t g, long 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_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.

```
void fmpz_mod(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to the remainder of g divided by h. The remainder is always taken to be positive.

```
ulong fmpz_mod_ui(fmpz_t f, const fmpz_t g, ulong x)
```

Sets f to g reduced modulo x where x is an `unsigned long`. If x is 0 an exception will result.

```
ulong fmpz_fdiv_ui(const fmpz_t g, ulong x)
```

Returns the remainder of g modulo x where x is an `unsigned long`, without changing g. If x is 0 an exception will result.

```
void fmpz_pow_ui(fmpz_t f, const fmpz_t g, ulong x)
```

Sets f to g^x where x is an `unsigned long`. If x is 0 and g is 0, then f will be set to 1.

```
void fmpz_powm_ui(fmpz_t f, const fmpz_t g, ulong e, const
    fmpz_t m)
```

Sets f to $g^e \bmod m$. If $e = 0$, sets f to 1.

Assumes that $m \neq 0$, raises an `abort` signal otherwise.

```
void fmpz_powm(fmpz_t f, const fmpz_t g, const fmpz_t e,
    const fmpz_t m)
```

Sets f to $g^e \bmod m$. If $e = 0$, sets f to 1.

Assumes that $m \neq 0$, raises an `abort` signal otherwise.

```
int fmpz_sqrtmod(fmpz_t b, const fmpz_t a, const fmpz_t p)
```

Returns whether a is a quadratic residue or zero modulo p and sets b to a square root of a if this is the case.

```
void fmpz_sqrt(fmpz_t f, const 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 fmpz_sqrtrem(fmpz_t f, fmpz_t r, const 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.

```
void fmpz_fac_ui(fmpz_t f, ulong n)
```

Sets f to $n!$ where n is an `unsigned long`.

```
void fmpz_bin_uiui(fmpz_t f, ulong n, ulong k)
```

Sets f to the binomial coefficient $\binom{n}{k}$.

8.10 Greatest common divisor

```
void fmpz_gcd(fmpz_t f, const fmpz_t g, 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 fmpz_lcm(fmpz_t f, const fmpz_t g, 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.

8.11 Modular arithmetic

```
long _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 `finv` is the precomputed `double` inverse of f whenever f is a small integer.

Does not support aliasing.

```
long fmpz_remove(fmpz_t rop, const fmpz_t op, const fmpz_t
    f)
```

Remove all occurrences of the factor $f > 1$ from the integer `op` and sets `rop` to the resulting integer.

If `op` is zero, sets `rop` to `op` and return 0.

Returns an `abort` signal if any of the assumptions are violated.

```
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.

8.12 Bit packing and unpacking

```
int fmpz_bit_pack(mp_limb_t * arr, mp_bitcnt_t shift,
    mp_bitcnt_t bits, fmpz_t coeff, int negate, int borrow)
```

Shifts the given coefficient to the left by `shift` bits and adds it to the integer in `arr` in a field of the given number of bits.

```
shift  bits  --------------

X X X C C C C 0 0 0 0 0 0 0
```

An optional borrow of 1 can be subtracted from `coeff` before it is packed. If `coeff` is negative after the borrow, then a borrow will be returned by the function.

The value of `shift` is assumed to be less than `FLINT_BITS`. All but the first `shift` bits of `arr` are assumed to be zero on entry to the function.

The value of `coeff` may also be optionally (and notionally) negated before it is used, by setting the `negate` parameter to -1.

```
int fmpz_bit_unpack(fmpz_t coeff, mp_limb_t * arr,
    mp_bitcnt_t shift, mp_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, mp_bitcnt_t shift, mp_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`.

8.13 Chinese remaindering

The following functions can be used to reconstruct an integer from its residues modulo a set of small (word-size) prime numbers. The first two functions, `fmpz_CRT_ui` and `fmpz_CRT_ui_unsigned`, 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.

```
void fmpz_CRT_ui_unsigned(fmpz_t out, fmpz_t r1, fmpz_t m1,
    ulong r2, ulong m2)
```

Uses the Chinese Remainder Theorem to compute the unique integer $0 \leq x < M$ 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 integers greater than 1 and coprime, and that $0 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$. The result x is stored in `out`.

```
void fmpz_CRT_ui(fmpz_t out, fmpz_t r1, fmpz_t m1, ulong
    r2, ulong m2)
```

Uses the Chinese Remainder Theorem to compute the unique integer $-M/2 \leq x < M/2$ 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 integers greater than 1 and coprime, and that $-m_1 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$. The result x is stored in `out`.

```
void fmpz_multi_mod_ui(mp_limb_t * out, const fmpz_t in,
    const fmpz_comb_t comb, fmpz_comb_temp_t temp)
```

Reduces the multiprecision integer `in` modulo each of the primes stored in the `comb` structure. The array `out` will be filled with the residues modulo these primes. The structure `temp` is temporary space which must be provided by `fmpz_comb_temp_init` and cleared by `fmpz_comb_temp_clear`.

```
void fmpz_multi_CRT_ui_unsigned(fmpz_t output, const
    mp_limb_t * residues, const fmpz_comb_t comb,
    fmpz_comb_temp_t temp)
```

This function takes a set of residues modulo the list of primes contained in the `comb` structure and reconstructs the unique unsigned multiprecision integer modulo the product of the primes which has these residues modulo the corresponding primes. The structure `temp` is temporary space which must be provided by `fmpz_comb_temp_init` and cleared by `fmpz_comb_temp_clear`.

```
void fmpz_multi_CRT_ui(fmpz_t output, const mp_limb_t *
    residues, const fmpz_comb_t comb, fmpz_comb_temp_t temp)
```

This function takes a set of residues modulo the list of primes contained in the `comb` structure and reconstructs a signed 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 `out` is normalised to be in the range $[-(N-1)/2, N/2]$. The array `temp` is temporary space which must be provided by `fmpz_comb_temp_init` and cleared by `fmpz_comb_temp_clear`.

```
void fmpz_comb_init(fmpz_comb_t comb, mp_limb_t * primes,
    long num_primes)
```

Initialises a `comb` structure for multimodular reduction and recombination. The array `primes` is assumed to contain `num_primes` primes each of `FLINT_BITS - 1` bits. Modular reductions and recombinations will be done modulo this list of primes. The `primes` array must not be `free`'d until the `comb` structure is no longer required and must be cleared by the user.

```
void fmpz_comb_temp_init(fmpz_comb_temp_t temp, const
    fmpz_comb_t comb)
```

Creates temporary space to be used by multimodular and CRT functions based on an initialised `comb` structure.

```
void fmpz_comb_clear(fmpz_comb_t comb)
```

Clears the given `comb` structure, releasing any memory it uses.

```
void fmpz_comb_temp_clear(fmpz_comb_temp_t temp)
```

Clears temporary space `temp` used by multimodular and CRT functions using the given `comb` structure.

§9. fmpz_vec

Vectors over **Z**

9.1 Memory management

```
fmpz * _fmpz_vec_init(long len)
```

Returns an initialised vector of `fmpz`'s of given length.

```
void _fmpz_vec_clear(fmpz * vec, long len)
```

Clears the entries of `(vec, len)` and frees the space allocated for `vec`.

9.2 Randomisation

```
void _fmpz_vec_randtest(fmpz * f, flint_rand_t state, long
    len, mp_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(fmpz * f, flint_rand_t
    state, long len, mp_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.

9.3 Bit sizes

```
long _fmpz_vec_max_bits(const fmpz * vec, long 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.

```
ulong _fmpz_vec_max_limbs(const fmpz * vec, long 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.

9.4 Input and output

```
int _fmpz_vec_fread(FILE * file, fmpz ** vec, long * 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, long * 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, long
    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, long len)
```

Prints the vector of given length to `stdout`.

For further details, see `_fmpz_vec_fprint()`.

9.5 Conversions

```
void _fmpz_vec_get_nmod_vec(mp_ptr res, const fmpz * poly,
    long 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,
    long 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 $[-n/2, n/2)$.

9.6 Assignment and basic manipulation

```
void _fmpz_vec_set(fmpz * vec1, const fmpz * vec2, long
    len2)
```

Makes a copy of (vec2, len2) into vec1.

```
void _fmpz_vec_swap(fmpz * vec1, fmpz * vec2, long len2)
```

Swaps the integers in (vec1, len2) and (vec2, len2).

```
void _fmpz_vec_zero(fmpz * vec, long len)
```

Zeros the entries of (vec, len).

```
void _fmpz_vec_neg(fmpz * vec1, const fmpz * vec2, long
    len2)
```

Negates (vec2, len2) and places it into vec1.

9.7 Comparison

```
int _fmpz_vec_equal(const fmpz * vec1, const fmpz * vec2,
    long 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, long len)
```

Returns 1 if (vec, len) is zero, and 0 otherwise.

9.8 Sorting

```
void _fmpz_vec_sort(fmpz * vec, long len)
```

Sorts the coefficients of vec in ascending order.

9.9 Addition and subtraction

```
void _fmpz_vec_add(fmpz * res, const fmpz * vec1,
                            const fmpz * vec2, long len2)
```

Sets (res, len2) to the sum of (vec1, len2) and (vec2, len2).

```
void _fmpz_vec_sub(fmpz * res, const fmpz * vec1,
                            const fmpz * vec2, long len2)
```

Sets (res, len2) to (vec1, len2) minus (vec2, len2).

9.10 Scalar multiplication and division

```
void _fmpz_vec_scalar_mul_fmpz(fmpz * vec1, const fmpz *
    vec2, long len2, const fmpz_t x)
```

Sets (vec1, len2) to (vec2, len2) multiplied by c, where c is an fmpz_t.

```
void _fmpz_vec_scalar_mul_si(fmpz * vec1, const fmpz *
    vec2, long len2, long c)
```

Sets (vec1, len2) to (vec2, len2) multiplied by c, where c is a signed long.

```
void _fmpz_vec_scalar_mul_ui(fmpz * vec1, const fmpz *
    vec2, long len2, ulong c)
```

Sets (vec1, len2) to (vec2, len2) multiplied by c, where c is an unsigned long.

```
void _fmpz_vec_scalar_mul_2exp(fmpz * vec1, const fmpz *
    vec2, long len2, ulong exp)
```

Sets (vec1, len2) to (vec2, len2) multiplied by 2^exp.

```
void _fmpz_vec_scalar_divexact_fmpz(fmpz * vec1, const fmpz
    * vec2, long 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.

```
void _fmpz_vec_scalar_divexact_si(fmpz * vec1, const fmpz *
    vec2, long len2, long c)
```

Sets (vec1, len2) to (vec2, len2) divided by x, where the division is assumed to be exact for every entry in vec2.

```
void _fmpz_vec_scalar_divexact_ui(fmpz * vec1, const fmpz *
    vec2, ulong len2, ulong c)
```

Sets (vec1, len2) to (vec2, len2) divided by x, where the division is assumed to be exact for every entry in vec2.

```
void _fmpz_vec_scalar_fdiv_q_fmpz(fmpz * vec1, const fmpz *
    vec2, long 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.

```
void _fmpz_vec_scalar_fdiv_q_si(fmpz * vec1, const fmpz *
    vec2, long len2, long c)
```

Sets (vec1, len2) to (vec2, len2) divided by c, rounding down towards minus infinity whenever the division is not exact.

```
void _fmpz_vec_scalar_fdiv_q_ui(fmpz * vec1, const fmpz *
    vec2, long len2, ulong c)
```

Sets (vec1, len2) to (vec2, len2) divided by c, rounding down towards minus infinity whenever the division is not exact.

```
void _fmpz_vec_scalar_fdiv_q_2exp(fmpz * vec1, const fmpz *
    vec2, long len2, ulong exp)
```

Sets (vec1, len2) to (vec2, len2) divided by 2^exp, rounding down towards minus infinity whenever the division is not exact.

```
void _fmpz_vec_scalar_tdiv_q_fmpz(fmpz * vec1, const fmpz *
    vec2, long 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, long len2, long 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, long 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_addmul_fmpz(fmpz * vec1, const fmpz *
    vec2, long len2, const fmpz_t c)
```

Adds `(vec2, len2)` times c to `(vec1, len2)`, where c is a `fmpz_t`.

```
void _fmpz_vec_scalar_addmul_si(fmpz * vec1, const fmpz *
    vec2, long len2, long c)
```

Adds `(vec2, len2)` times c to `(vec1, len2)`, where c is a `signed long`.

```
void _fmpz_vec_scalar_addmul_si_2exp(fmpz * vec1, const
    fmpz * vec2, long len2, long c, ulong exp)
```

Adds `(vec2, len2)` times `c * 2^exp` to `(vec1, len2)`, where c is a `signed long`.

```
void _fmpz_vec_scalar_submul_fmpz(fmpz * vec1, const fmpz *
    vec2, long 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, long len2, long c)
```

Subtracts `(vec2, len2)` times c from `(vec1, len2)`, where c is a `signed long`.

```
void _fmpz_vec_scalar_submul_si_2exp(fmpz * vec1, const
    fmpz * vec2, long len2, long c, ulong e)
```

Subtracts `(vec2, len2)` times $c \times 2^e$ from `(vec1, len2)`, where c is a `signed long`.

9.11 Gaussian content

```
void _fmpz_vec_content(fmpz_t res, const fmpz * vec, long
    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_lcm(fmpz_t res, const fmpz * vec, long 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.

§10. fmpz_mat

Matrices over **Z**

10.1 Introduction

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.

10.2 Simple example

The following example creates the 2×2 matrix A with value $2i+j$ at row i and column j, computes $B = A^2$, and prints both matrices.

```
#include "fmpz.h"
#include "fmpz_mat.h"
...
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);
printf("A = \n");
fmpz_mat_print_pretty(A);
printf("A^2 = \n");
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]]
```

10.3 Memory management

```
void fmpz_mat_init(fmpz_mat_t mat, long rows, long 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.

10.4 Random matrix generation

```
void fmpz_mat_randbits(fmpz_mat_t mat, flint_rand_t state,
    mp_bitcnt_t bits)
```

Sets the entries of `mat` to random signed integers whose absolute values have the given number of binary bits.

```
void fmpz_mat_randtest(fmpz_mat_t mat, flint_rand_t state,
    mp_bitcnt_t bits)
```

Sets the entries of `mat` to random signed integers whose absolute values have a random number of bits up to the given number of bits inclusive.

```
void fmpz_mat_randintrel(fmpz_mat_t mat, flint_rand_t
    state, mp_bitcnt_t bits)
```

Sets `mat` to be a random *integer relations* matrix, with signed entries up to the given number of bits.

The number of columns of `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 fmpz_mat_randsimdioph(fmpz_mat_t mat, flint_rand_t
    state, mp_bitcnt_t bits, mp_bitcnt_t bits2)
```

Sets `mat` to a random *simultaneous diophantine* matrix.

The matrix must be square. The top left entry is set to `2^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^bits` with all remaining entries zero.

```
void fmpz_mat_randntrulike(fmpz_mat_t mat, flint_rand_t
    state, mp_bitcnt_t bits, ulong q)
```

Sets a square matrix `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 q times the identity matrix. Finally the top right submatrix has the following format. A random vector h of length $r/2$ is created, with random signed entries of the given number of bits. Then entry (i, j) of the submatrix is set to $h[i+j \bmod r/2]$.

```
void fmpz_mat_randntrulike2(fmpz_mat_t mat, flint_rand_t
    state, mp_bitcnt_t bits, ulong q)
```

Sets a square matrix `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 h of length $r/2$ is created, with random signed entries of the given number of bits. Then entry (i, j) of the submatrix is set to $h[i + j \bmod r/2]$.

```
void fmpz_mat_randajtai(fmpz_mat_t mat, flint_rand_t state,
    double alpha)
```

Sets a square matrix `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 $b = \lfloor (2r - i)^\alpha \rfloor$ for some double parameter α. 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 fmpz_mat_randpermdiag(fmpz_mat_t mat, flint_rand_t
    state, const fmpz * diag, long n)
```

Sets `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 fmpz_mat_randrank(fmpz_mat_t mat, flint_rand_t state,
    long rank, mp_bitcnt_t bits)
```

Sets `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 `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,
    long 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.

10.5 Basic assignment and manipulation

```
void fmpz_mat_set(fmpz_mat_t mat1, 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, fmpz_mat_t src)
```

Initialises the matrix `mat` to the same size as `src` and sets it to a copy of `src`.

```
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.

```
fmpz * fmpz_mat_entry(fmpz_mat_t mat, long i, long 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.

10.6 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().

10.7 Comparison

```
int fmpz_mat_equal(fmpz_mat_t mat1, 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(fmpz_mat_t mat)
```

Returns a non-zero value if all entries mat are zero, and otherwise returns zero.

```
int fmpz_mat_is_empty(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(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.

10.8 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.

10.9 Modular reduction and reconstruction

```
void fmpz_mat_get_nmod_mat(nmod_mat_t Amod, 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_poly_CRT_ui(fmpz_poly_t res, const fmpz_poly_t
    poly1, const fmpz_t m, const nmod_poly_t poly2)
```

Given mat1 with entries modulo m and mat2 with modulus n, sets res to the CRT reconstruction modulo mn with signed entries satisfying $-mn/2 <= c < mn/2$.

```
void fmpz_poly_CRT_ui_unsigned(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_t m1, const nmod_poly_t
    poly2)
```

Given mat1 with entries modulo m and mat2 with modulus n, sets res to the CRT reconstruction modulo mn with signed entries satisfying $0 <= c < mn$.

10.10 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.

10.11 Matrix-scalar arithmetic

```
void fmpz_mat_scalar_mul_si(fmpz_mat_t B, const fmpz_mat_t
    A, long 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 A = B*c where B is an fmpz_mat_t and c is a scalar respectively of type long, unsigned long, 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, long 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 A = A + B*c where B is an fmpz_mat_t and c is a scalar respectively of type long, unsigned long, 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, long 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 A = A - B*c where B is an fmpz_mat_t and c is a scalar respectively of type long, unsigned long, or fmpz_t. The dimensions of A and B must be compatible.

```
void fmpz_mat_scalar_addmul_nmod_mat_fmpz(fmpz_mat_t B,
    const nmod_mat_t A, const fmpz_t c)
```

```
void fmpz_mat_scalar_addmul_nmod_mat_ui(fmpz_mat_t B, const
    nmod_mat_t A, ulong c)
```

Set A = A + B*c where B is an nmod_mat_t and c is a scalar respectively of type unsigned long 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, long 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 long, unsigned long, or fmpz_t, which is assumed to divide all elements of B exactly.

10.12 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_multi_mod(fmpz_mat_t C, fmpz_mat_t A,
    fmpz_mat_t B, long bits)
```

```
void fmpz_mat_mul_multi_mod(fmpz_mat_t C, fmpz_mat_t A,
    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 bits parameter is a bound for the bit size of largest element of C, or twice the absolute value of the largest element if any elements of C are negative. 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.

10.13 Inverse

```
void fmpz_mat_inv(fmpz_mat_t B, fmpz_t d, const fmpz_mat_t
    A)
```

Sets `(B, d)` to the inverse matrix of the square matrix A, i.e. computes an integer matrix B and an integer d such that $AB = BA = dI$, the identity matrix.

If A is singular, d will be set to zero and the value of B will be undefined. In general, `(B, d)` will not be reduced to lowest terms. A and B are allowed to be the same object.

10.14 Determinant

```
void fmpz_mat_det(fmpz_t det, const fmpz_mat_t A)
```

Sets `det` to the determinant of the square matrix A. This function automatically chooses between `fmpz_mat_det_cofactor`, `fmpz_mat_det_bareiss` and `fmpz_mat_det_multi_mod` (with `proved` $= 1$), depending on the size of the matrix.

The matrix of dimension 0×0 is defined to have determinant 1.

```
void fmpz_mat_det_cofactor(fmpz_t det, const fmpz_mat_t A)
```

Sets `det` to the determinant of the square matrix A computed using direct cofactor expansion. This function only supports matrices up to size 4×4.

```
void fmpz_mat_det_bareiss(fmpz_t det, const fmpz_mat_t A)
```

Sets `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.

```
void fmpz_mat_det_multi_mod(fmpz_t det, const fmpz_mat_t A,
    int proved)
```

Sets `det` to the determinant of the square matrix A (if `proved` $= 1$), or a probabilistic value for the determinant (`proved` $= 0$), computed using a multimodular algorithm.

The determinant is computed modulo several small primes and reconstructed using the Chinese Remainder Theorem. With `proved` $= 1$, sufficiently many primes are chosen to satisfy the bound computed by `fmpz_mat_det_bound`. With `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_bound(fmpz_t bound, const fmpz_mat_t A)
```

Sets `bound` to a nonnegative integer B such that $|\det(A)| \le B$. Assumes A to be a square matrix. The bound is computed from the Hadamard inequality $|\det(A)| \le \prod \|a_i\|_2$ where the product is taken over the rows a_i of A.

10.15 Rank

```
long 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.

10.16 Nonsingular solving

The following functions allow solving matrix-vector equations $Ax = b$ or 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: all functions compute an integer vector or matrix $\hat{x}$ and a separate denominator d (`den`) such that $A(\hat{x}/d) = b$, or equivalently such that $A\hat{x} = bd$ holds over the integers.

No guarantee is made that the numerators and denominator are reduced to lowest terms. 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.

```
void fmpz_mat_solve(fmpz * x, fmpz_t den, const fmpz_mat_t
    A, const fmpz * b)
```

Solves the matrix-vector equation $Ax = b$ where A is a nonsingular square matrix of size m and b is an integer vector of length m. This function calls `fmpz_mat_solve_cramer` for small matrices and `fmpz_mat_solve_fraction_free_LU` otherwise.

```
void fmpz_mat_solve_cramer(fmpz * x, fmpz_t den, const
    fmpz_mat_t A, const fmpz * b)
```

Solves the matrix-vector equation $Ax = b$ using Cramer's rule. Only systems of size up to 3×3 are allowed.

```
void fmpz_mat_solve_fraction_free_LU(fmpz * x, fmpz_t den,
    const fmpz_mat_t A, const fmpz * b)
```

Solves the matrix-vector equation $Ax = b$ by computing a fraction-free LU decomposition of A and then solving the triangular systems $Ly = b$, $Ux = y$ using fraction-free forward and backward substitution.

```
void fmpz_mat_solve_mat(fmpz_mat_t X, fmpz_t den, const
    fmpz_mat_t A, const fmpz_mat_t B)
```

Solves $AX = B$ for the m-by-n matrix X, where A is an m-by-m integer matrix and B is an m-by-n integer matrix. Equivalently, solves $Ax = b$ for each respective column vector x in X and b in B.

This function computes a fraction-free LU decomposition of A and reuses the LU decomposition to solve the column equations one by one.

```
void _fmpz_mat_solve_fraction_free_LU_precomp(fmpz * b,
    const fmpz_mat_t LU)
```

Solves the numerator part of the matrix-vector equation $Ax = b$, given a precomputed fraction-free LU decomposition of A. `b` is overwritten with the solution vector in-place. If the rows of the system matrix were permuted during the LU decomposition, replace `b[i]` with `b[perm[i]]` in the input.

```
void 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 numerators and denominators in $A^{-1}B$ are bounded by N and D respectively.

```
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 $AX = B \bmod M$ and such that all the reduced numerators and denominators of the elements $x = p/q$ in the full solution satisfy $2|p|q < B$. As such, the explicit rational solution matrix can be recovered uniquely by passing the output of this function to `fmpq_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.

10.17 Kernel

```
long fmpz_mat_kernel(fmpz_mat_t B, const fmpz_mat_t A)
```

Computes a basis for the right kernel (or null space) of A and returns the dimension of the kernel. B is set to 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 column of A).

10.18 Echelon form

```
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 `den` such that `B / 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 `perm` unless this argument is `NULL`. That is, row `i` of the output matrix will correspond to row `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.

10.19 Internal functions for row reduction

```
int fmpz_mat_pivot(long * perm, fmpz_mat_t mat, long r,
    long 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.

```
long _fmpz_mat_rowreduce(long * perm, fmpz_mat_t mat, int
    options)
```

Row reduces the matrix in-place using fraction-free Gaussian elimination. The number of rows m and columns n may be arbitrary.

This function effectively implements the algorithms FFGE, FFGJ and FFLU given in [19], but with pivoting. The `options` parameter is a bitfield which may be set to any combination of the following flags; use `options = 0` to disable all, resulting in FFLU:

- `ROWREDUCE_FAST_ABORT`

  If set, the function immediately aborts and returns 0 when (if) the matrix is detected to be rank-deficient (singular). In this event, the state of the matrix will be undefined.

- `ROWREDUCE_FULL`

  If set, performs fraction-free Gauss-Jordan elimination (FFGJ), i.e. eliminates the elements above each pivot element as well as those below. If not set, regular Gaussian elimination is performed and only the elements below pivots are eliminated.

- `ROWREDUCE_CLEAR_LOWER`

  If set, clears (i.e. zeros) elements below the pivots (FFGE). If not set, the output becomes the fraction-free LU decomposition of the matrix with L in the lower triangular part.

Pivoting (to avoid division by zero entries) is performed by permuting the vector of row pointers in-place. The matrix entries themselves retain their original order in memory.

The permutation order will also be written to `perm` unless this argument is `NULL`. That is, row `i` of the output matrix will correspond to row `perm[i]` of the input matrix.

The return value r is the rank of the matrix, multiplied by a sign indicating the parity of row interchanges. If $r = 0$, the matrix has rank zero, unless `ROWREDUCE_FAST_ABORT` is set, in which case $r = 0$ indicates any deficient rank. Otherwise, the leading nonzero entries of $a[0], a[1], \ldots, a[|r| - 1]$ will point to the successive pivot elements. If $|r| = m = n$, the determinant of the matrix is given by $\operatorname{sgn}(r) \times a\big[|r| - 1\big]\big[|r| - 1\big]$.

§11. fmpz_poly

Polynomials over **Z**

11.1 Introduction

The fmpz_poly_t data type represents elements of $\mathbf{Z}[x]$. 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.

11.2 Simple example

The following example computes the square of the polynomial $5x^3 - 1$.

```
#include "fmpz_poly.h"
...
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); printf("\n");
fmpz_poly_print(y); printf("\n");
fmpz_poly_clear(x);
fmpz_poly_clear(y);
```

The output is:

```
4  -1 0 0 5
7  1 0 0 -10 0 0 25
```

11.3 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.

We now describe the functions available in `fmpz_poly`.

11.4 Memory management

```
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, long 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, long 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, long 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, long newlen)
```

Demotes the coefficients of `poly` beyond `newlen` and sets the length of `poly` to `newlen`.

11.5 Polynomial parameters

```
long fmpz_poly_length(const fmpz_poly_t poly)
```

Returns the length of `poly`. The zero polynomial has length zero.

```
long fmpz_poly_degree(const fmpz_poly_t poly)
```

Returns the degree of `poly`, which is one less than its length.

```
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.

```
long 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.

11.6 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, long 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`.

```
void fmpz_poly_set_mpz(fmpz_poly_t poly, const mpz_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 `str` represents a valid polynomial returns 1, otherwise returns 0.

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.

```
char * _fmpz_poly_get_str(const fmpz * poly, long len)
```

Returns the plain FLINT string representation of the polynomial `(poly, len)`.

```
char * fmpz_poly_get_str(const fmpz_poly_t poly)
```

Returns the plain FLINT string representation of the polynomial `poly`.

```
char * _fmpz_poly_get_str_pretty(const fmpz * poly, long
    len, const char * x)
```

Returns a pretty representation of the polynomial `(poly, len)` using the null-terminated string `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 `poly` using the null-terminated string `x` as the variable name.

```
void fmpz_poly_zero(fmpz_poly_t poly)
```

Sets `poly` to the zero polynomial.

```
void fmpz_poly_one(fmpz_poly_t poly)
```

Sets `poly` to the constant polynomial one.

```
void fmpz_poly_zero_coeffs(fmpz_poly_t poly, long i, long j)
```

Sets the coefficients of $x^i, \dots, x^{j-1}$ to zero.

```
void fmpz_poly_swap(fmpz_poly_t poly1, fmpz_poly_t poly2)
```

Swaps `poly1` and `poly2`. This is done efficiently without copying data by swapping pointers, etc.

```
void _fmpz_poly_reverse(fmpz * res, const fmpz * poly, long
    len, long n)
```

Sets `(res, n)` to the reverse of `(poly, n)`, where `poly` is in fact an array of length `len`. Assumes that `0 < len <= n`. Supports aliasing of `res` and `poly`, but the behaviour is undefined in case of partial overlap.

```
void fmpz_poly_reverse(fmpz_poly_t res, const fmpz_poly_t
    poly, long n)
```

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.

```
void fmpz_poly_truncate(fmpz_poly_t poly, long newlen)
```

If the current length of poly is greater than newlen, it is truncated to have the given length. Discarded coefficients are not necessarily set to zero.

11.7 Randomisation

```
void fmpz_poly_randtest(fmpz_poly_t f, flint_rand_t state,
    long len, mp_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. One must call flint_randinit() before calling this function.

```
void fmpz_poly_randtest_unsigned(fmpz_poly_t f,
    flint_rand_t state, long len, mp_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. One must call flint_randinit() before calling this function.

```
void fmpz_poly_randtest_not_zero(fmpz_poly_t f,
    flint_rand_t state, long len, mp_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. One must call flint_randinit() before calling this function.

11.8 Getting and setting coefficients

```
void fmpz_poly_get_coeff_fmpz(fmpz_t x, const fmpz_poly_t
    poly, long n)
```

Sets x to the nth 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.

```
long fmpz_poly_get_coeff_si(const fmpz_poly_t poly, long n)
```

Returns coefficient n of poly as a long. The result is undefined if the value does not fit into a long. Coefficient numbering is from zero and if n is set to a value beyond the end of the polynomial, zero is returned.

```
ulong fmpz_poly_get_coeff_ui(const fmpz_poly_t poly, long n)
```

Returns coefficient n of poly as a ulong. The result is undefined if the value does not fit into a ulong. Coefficient numbering is from zero and if n is set to a value beyond the end of the polynomial, zero is returned.

```
fmpz * fmpz_poly_get_coeff_ptr(const fmpz_poly_t poly, long
    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.

```
fmpz * fmpz_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, long 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, long n, long
    x)
```

Sets coefficient n of `poly` to the `long` 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, long n, ulong
    x)
```

Sets coefficient n of `poly` to the `unsigned long` 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.

11.9 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_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 is the polynomial is the constant polynomial ± 1, and 0 otherwise.

11.10 Addition and subtraction

```
void _fmpz_poly_add(fmpz * res, const fmpz * poly1, long
    len1, const fmpz * poly2, long 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_sub(fmpz * res, const fmpz * poly1, long
    len1, const fmpz * poly2, long 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_neg(fmpz_poly_t res, const fmpz_poly_t poly)
```

Sets res to -poly.

11.11 Scalar multiplication and division

```
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, fmpz_poly_t
    poly2, long x)
```

Sets poly1 to poly2 times the signed long x.

```
void fmpz_poly_scalar_mul_ui(fmpz_poly_t poly1, fmpz_poly_t
    poly2, ulong x)
```

Sets poly1 to poly2 times the unsigned long x.

```
void fmpz_poly_scalar_addmul_fmpz(fmpz_poly_t poly1, const
    fmpz_poly_t poly2, const fmpz_t x)
```

Sets poly to poly1 + x * poly2.

```
void fmpz_poly_scalar_submul_fmpz(fmpz_poly_t poly1, const
    fmpz_poly_t poly2, const fmpz_t x)
```

Sets poly 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,
    fmpz_poly_t poly2, long x)
```

Sets poly1 to poly2 divided by the long x, rounding coefficients down toward $-\infty$.

```
void fmpz_poly_scalar_fdiv_ui(fmpz_poly_t poly1,
    fmpz_poly_t poly2, ulong x)
```

Sets poly1 to poly2 divided by the unsigned long x, 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,
    fmpz_poly_t poly2, long x)
```

Sets poly1 to poly2 divided by the long x, rounding coefficients toward 0.

```
void fmpz_poly_scalar_tdiv_ui(fmpz_poly_t poly1,
    fmpz_poly_t poly2, ulong x)
```

Sets poly1 to poly2 divided by the unsigned long 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 coefficient is exact for every coefficient.

```
void fmpz_poly_scalar_divexact_si(fmpz_poly_t poly1,
    fmpz_poly_t poly2, long x)
```

Sets poly1 to poly2 divided by the long x, assuming the coefficient is exact for every coefficient.

```
void fmpz_poly_scalar_divexact_ui(fmpz_poly_t poly1,
    fmpz_poly_t poly2, ulong x)
```

Sets poly1 to poly2 divided by the unsigned long x, assuming the coefficient is exact for every coefficient.

11.12 Bit packing

```
void _fmpz_poly_bit_pack(mp_ptr arr, const fmpz * poly,
    long len, mp_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, long len, mp_srcptr
    arr, mp_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 ± 1 should be added at position len of poly.

```
void _fmpz_poly_bit_unpack_unsigned(fmpz * poly, long len,
    mp_srcptr_t arr, mp_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_t f, const fmpz_poly_t poly,
    mp_bitcnt_t bit_size)
```

Packs poly into bitfields of size bit_size, writing the result to f. If 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_t
    f, mp_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, mp_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.

11.13 Multiplication

```
void _fmpz_poly_mul_classical(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long 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, long len1, const fmpz * poly2, long len2, long 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, long n)
```

Sets `res` to the first n coefficients of `poly1 * poly2`.

```
void _fmpz_poly_mulhigh_classical(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long len2, long
    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, long 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, long len1, const fmpz * poly2, long 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, long len1, const fmpz * poly2, long 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.

```
void _fmpz_poly_mullow_karatsuba_n(fmpz * res, const fmpz *
    poly1, const fmpz * poly2, long 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, long 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, long 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, long 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, long
    len1, const fmpz * poly2, long 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,
    long len1, const fmpz * poly2, long len2, long 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, long n)
```

Sets `res` to the lowest n coefficients of the product of `poly1` and `poly2`.

```
void _fmpz_poly_mul(fmpz * res, const fmpz * poly1, long
    len1, const fmpz * poly2, long 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.

```
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, long
    len1, const fmpz * poly2, long len2, long 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, long 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, long 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.

11.14 Powering

```
void _fmpz_poly_pow_multinomial(fmpz * res, const fmpz *
    poly, long 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 $e\ell$ 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 [15, 21]. 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 $1 \leq k \leq mn$,

$$a(n, k) = (kp_0)^{-1} \sum_{i=1}^{m} p_i \big((n + 1)i - k\big) a(n, k - i).$$

```
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, long 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,
    long len, ulong e)
```

Sets `res = poly^e` using left-to-right binary exponentiation as described in [15, p. 461].

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,
    long 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, long
    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, long 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, long 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.

11.15 Shifting

```
void _fmpz_poly_shift_left(fmpz * res, const fmpz * poly,
    long len, long 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, long n)
```

Sets `res` to `poly` shifted left by n coeffs. Zero coefficients are inserted.

```
void _fmpz_poly_shift_right(fmpz * res, const fmpz * poly,
    long len, long 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, long 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.

11.16 Norms

```
void _fmpz_poly_2norm(fmpz_t res, const fmpz * poly, long
    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, long 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.

11.17 Greatest common divisor

```
void _fmpz_poly_gcd_subresultant(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long 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 [4, Algorithm 3.3.1].

```
int _fmpz_poly_gcd_heuristic(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long 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,
    long len1, const fmpz * poly2, long 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.

```
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, long
    len1, const fmpz * poly2, long 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.

Assumes that res has space for len2 coefficients. Aliasing between res, poly1 and poly2 is not supported.

```
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, long len1, const fmpz * g, long 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 len2 and the length of t will be no greater than len1 (both are zero padded if necessary).

It is assumed that len1 >= len2 > 0. No aliasing of inputs and outputs is permitted.

Uses a multimodular algorithm. The resultant is first computed and extended GCD's 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.

```
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.

Uses the multimodular algorithm.

```
void _fmpz_poly_xgcd(fmpz_t r, fmpz * s, fmpz * t, const
    fmpz * f, long len1, const fmpz * g, long 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 len2 and the length of t will be no greater than len1 (both are zero padded if necessary).

It is assumed that len1 >= len2 > 0. No aliasing of inputs and outputs is permitted.

```
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`.

```
void _fmpz_poly_resultant(fmpz_t res, const fmpz * poly1,
    long len1, const fmpz * poly2, long len2)
```

Sets `res` to the resultant of `(poly1, len1)` and `(poly2, len2)`, assuming that `len1 >= len2 > 0`.

```
void fmpz_poly_resultant(fmpz_t res, const fmpz_poly_t
    poly1, 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 [4, Algorithm 3.3.7].

11.18 Gaussian content

```
void _fmpz_poly_content(fmpz_t res, const fmpz * poly, long
    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, long 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.

11.19 Euclidean division

```
void _fmpz_poly_divrem_basecase(fmpz * Q, fmpz * R, const
    fmpz * A, long lenA, const fmpz * B, long lenB)
```

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 $\mathbf{Q}$.

Assumes that $\mathrm{len}(A), \mathrm{len}(B) > 0$. 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 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 $\mathrm{len}(B) - 1$ 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 $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_divrem_divconquer_recursive(fmpz * Q, fmpz
    * BQ, fmpz * W, const fmpz * A, const fmpz * B, long
    lenB)
```

Computes `(Q, lenB)`, `(BQ, 2 lenB - 1)` such that $BQ = B \times Q$ and $A = BQ + R$ where each coefficient of R beyond $\mathrm{len}(B) - 1$ is reduced modulo the leading coefficient of B. We assume that $\mathrm{len}(A) = 2\,\mathrm{len}(B) - 1$. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$.

Assumes $\mathrm{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Requires a temporary array `(W, 2 lenB - 1)`. No aliasing of input and output operands is allowed.

This function does not read the bottom $\mathrm{len}(B) - 1$ coefficients from A, which means that they might not even need to exist in allocated memory.

```
void _fmpz_poly_divrem_divconquer(fmpz * Q, fmpz * R, const
    fmpz * A, long lenB, const fmpz * B, long lenB)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ + R$ and each coefficient of R beyond $\mathrm{len}(B) - 1$ 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 as division over $\mathbf{Q}$.

Assumes $\mathrm{len}(A) \geq \mathrm{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fmpz_poly_divrem_divconquer(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 $\mathrm{len}(B) - 1$ 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 as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_divrem(fmpz * Q, fmpz * R, const fmpz * A,
    long lenA, const fmpz * B, long lenB)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ + R$ and each coefficient of R beyond $\mathrm{len}(B) - 1$ 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 $\mathbf{Q}$.

Assumes $\mathrm{len}(A) \geq \mathrm{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fmpz_poly_divrem(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 $\operatorname{len}(B) - 1$ 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 as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_div_basecase(fmpz * Q, fmpz * R, const fmpz
    * A, long lenA, const fmpz * B, long lenB)
```

Computes the quotient `(Q, lenA - lenB + 1)` of `(A, lenA)` divided by `(B, lenB)`.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ 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 as division over $\mathbf{Q}$.

Assumes $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Requires a temporary array R of size at least the (actual) length of A. For convenience, R may be `NULL`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fmpz_poly_div_basecase(fmpz_poly_t Q, const
    fmpz_poly_t A, const fmpz_poly_t B)
```

Computes the quotient Q of A divided by Q.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ 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 as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_divremlow_divconquer_recursive(fmpz * Q,
    fmpz * BQ, const fmpz * A, const fmpz * B, long lenB)
```

Divide and conquer division of `(A, 2 lenB - 1)` by `(B, lenB)`, computing only the bottom $\operatorname{len}(B) - 1$ coefficients of BQ.

Assumes $\operatorname{len}(B) > 0$. Requires BQ to have length at least $2\operatorname{len}(B) - 1$, although only the bottom $\operatorname{len}(B) - 1$ coefficients will carry meaningful output. Does not support any aliasing. Allows zero-padding in A, but not in B.

```
void _fmpz_poly_div_divconquer_recursive(fmpz * Q, fmpz *
    temp, const fmpz * A, const fmpz * B, long lenB)
```

Recursive short division in the balanced case.

Computes the quotient `(Q, lenB)` of `(A, 2 lenB - 1)` upon division by `(B, lenB)`. Requires $\operatorname{len}(B) > 0$. Needs a temporary array `temp` of length $2\operatorname{len}(B) - 1$. Does not support any aliasing.

For further details, see [18].

```
void _fmpz_poly_div_divconquer(fmpz * Q, const fmpz * A,
    long lenA, const fmpz * B, long lenB)
```

Computes the quotient `(Q, lenA - lenB + 1)` of `(A, lenA)` upon division by `(B, lenB)`. Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Does not support aliasing.

```
fmpz_poly_div_divconquer(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 ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_div(fmpz * Q, const fmpz * A, long lenA,
    const fmpz * B, long lenB)
```

Computes the quotient `(Q, lenA - lenB + 1)` of `(A, lenA)` divided by `(B, 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 ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Aliasing of input and output operands is not allowed.

```
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 ± 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_basecase(fmpz * R, const fmpz * A, long
    lenA, const fmpz * B, long lenB)
```

Computes the remainder `(R, lenA)` of `(A, lenA)` upon division by `(B, 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 ± 1 or the division is exact, this is the same thing as division over $\mathbf{Q}$.

Assumes that $\text{len}(A), \text{len}(B) > 0$. 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 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 ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$. 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, lenA)` of `(A, lenA)` upon division by `(B, 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 ± 1 or the division is exact, this is the same thing as division over $\mathbf{Q}$.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in `(A, 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 ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$. An exception is raised if B is zero.

11.20 Divisibility testing

```
int _fmpz_poly_divides(fmpz * Q, const fmpz * A, long lenA,
    const fmpz * B, long lenB)
```

Returns 1 if `(B, lenB)` divides `(A, 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.

Aliasing of Q with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```
int fmpz_poly_divides(fmpz_poly_t Q, const fmpz_poly_t A,
    const fmpz_poly_t B)
```

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.

11.21 Power series division

```
void _fmpz_poly_inv_series_newton(fmpz * Qinv, const fmpz *
    Q, long n)
```

Computes the first n terms of the inverse power series of Q using Newton iteration.

Assumes that $n \geq 1$, that Q has length at least n and constant term 1. Does not support aliasing.

```
void fmpz_poly_inv_series_newton(fmpz_poly_t Qinv, const
    fmpz_poly_t Q, long n)
```

Computes the first n terms of the inverse power series of Q using Newton iteration, assuming that Q has constant term 1 and $n \geq 1$.

```
void _fmpz_poly_inv_series(fmpz * Qinv, const fmpz * Q,
    long n)
```

Computes the first n terms of the inverse power series of Q.

Assumes that $n \geq 1$, that Q has length at least n and constant term 1. Does not support aliasing.

```
void fmpz_poly_inv_series(fmpz_poly_t Qinv, const
    fmpz_poly_t Q, long n)
```

Computes the first n terms of the inverse power series of Q, assuming Q has constant term 1 and $n \geq 1$.

```
void _fmpz_poly_div_series(fmpz * Q, const fmpz * A, const
    fmpz * B)
```

Divides (A, n) by (B, n) as power series over **Z**, assuming B has constant term 1 and $n \geq 1$.

Only supports aliasing of (Q, n) and (B, n).

```
void fmpz_poly_div_series(fmpz_poly_t Q, const fmpz_poly_t
    A, const fmpz_poly_t B, long n)
```

Performs power series division in $\mathbf{Z}[[x]]/(x^n)$. 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 constant term 1 and $n \geq 1$.

11.22 Pseudo division

```
void _fmpz_poly_pseudo_divrem_basecase(fmpz * Q, fmpz * R,
    ulong * d, const fmpz * A, long lenA, const fmpz * B,
    long lenB)
```

If ℓ 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 **Q**.

Assumes that $\mathrm{len}(A) \geq \mathrm{len}(B) > 0$. Assumes that Q can fit $\mathrm{len}(A) - \mathrm{len}(B) + 1$ coefficients, and that R can fit $\mathrm{len}(A)$ coefficients. Supports aliasing of (R, lenA) and (A, lenA). But other than this, no aliasing of the inputs and outputs is suppported.

```
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 ℓ 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 **Q**.

```
void _fmpz_poly_pseudo_divrem_divconquer(fmpz * Q, fmpz *
    R, ulong * d, const fmpz * A, long lenB, const fmpz * B,
    long lenB)
```

Computes (Q, lenA - lenB + 1), (R, lenA) such that $\ell^d A = BQ + R$, only setting the bottom $\mathrm{len}(B) - 1$ coefficients of R to their correct values. The remaining top coefficients of (R, lenA) may be arbitrary.

Assumes $\mathrm{len}(A) \geq \mathrm{len}(B) > 0$. Allows zero-padding in (A, lenA). No aliasing of input and output operands is allowed.

```
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 ℓ 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, long lenA, const fmpz * B, long lenB)
```

Assumes that $\mathrm{len}(A) \geq \mathrm{len}(B) > 0$. Assumes that Q can fit $\mathrm{len}(A) - \mathrm{len}(B) + 1$ coefficients, and that R can fit $\mathrm{len}(A)$ coefficients. Supports aliasing of (R, lenA) and (A, 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 $\ell^d A = BQ+R$. However, the value of d is fixed at $\max\{0, \operatorname{len}(A) - \operatorname{len}(B) + 1\}$.

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 $\operatorname{len}(B) < 32$.

```
void _fmpz_poly_pseudo_rem_cohen(fmpz * R, const fmpz * A,
    long lenA, const fmpz * B, long lenB)
```

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Assumes that R can fit $\operatorname{len}(A)$ coefficients. Supports aliasing of `(R, lenA)` and `(A, lenA)`. But other than this, no aliasing of the inputs and outputs is supported.

```
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 $\ell^d A = BQ + R$, but only returns R. However, the value of d is fixed at $\max\{0, \operatorname{len}(A) - \operatorname{len}(B) + 1\}$.

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 $\operatorname{len}(B) < 32$.

This function uses the algorithm described in [4, Algorithm 3.1.2].

```
void _fmpz_poly_pseudo_divrem(fmpz * Q, fmpz * R, ulong *
    d, const fmpz * A, long lenA, const fmpz * B, long lenB)
```

If ℓ is the leading coefficient of B, then computes `(Q, lenA - lenB + 1)`, `(R, lenB - 1)` and d such that $\ell^d A = BQ + R$. This function is used for simulating division over $\mathbf{Q}$.

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Assumes that Q can fit $\operatorname{len}(A) - \operatorname{len}(B) + 1$ coefficients, and that R can fit $\operatorname{len}(A)$ coefficients, although on exit only the bottom $\operatorname{len}(B)$ coefficients will carry meaningful data.

Supports aliasing of `(R, lenA)` and `(A, lenA)`. But other than this, no aliasing of the inputs and outputs is suppported.

```
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 $\ell^d A = BQ + R$.

```
void _fmpz_poly_pseudo_div(fmpz * Q, ulong * d, const fmpz
    * A, long lenA, const fmpz * B, long lenB)
```

Pseudo-division, only returning the quotient.

```
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.

```
void _fmpz_poly_pseudo_rem(fmpz * R, ulong * d, const fmpz
    * A, long lenA, const fmpz * B, long lenB)
```

Pseudo-division, only returning the remainder.

```
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.

11.23 Derivative

```
void _fmpz_poly_derivative(fmpz * rpoly, const fmpz * poly,
    long 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`.

11.24 Evaluation

```
void _fmpz_poly_evaluate_divconquer_fmpz(fmpz_t res, const
    fmpz * poly, long len, const fmpz_t a)
```

Evaluates the polynomial (`poly, 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 poly, const fmpz_t a)
```

Evaluates the polynomial `poly` at the integer a using a divide and conquer approach.

Aliasing between `res` and `a` is supported, however, `res` may not be part of `poly`.

```
void _fmpz_poly_evaluate_horner_fmpz(fmpz_t res, const fmpz
    * f, long len, const fmpz_t a)
```

Evaluates the polynomial (`f, len`) at the integer a using Horner's rule, 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_horner_fmpz(fmpz_t res, const
    fmpz_poly_t f, 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,
    long len, const fmpz_t a)
```

Evaluates the polynomial (`f, len`) 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_horner_mpq(fmpz_t rnum, fmpz_t
    rden, const fmpz * f, long len, const fmpz_t anum, const
    fmpz_t aden)
```

Evaluates the polynomial (f, len) 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_mpq(mpq_t res, const
    fmpz_poly_t f, const mpq_t a)
```

Evaluates the polynomial f at the rational a using Horner's rule, and sets res to the result.

```
void _fmpz_poly_evaluate_mpq(fmpz_t rnum, fmpz_t rden,
    const fmpz * f, long len, const fmpz_t anum, const
    fmpz_t aden)
```

Evaluates the polynomial (f, len) 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_mpq(mpq_t res, const fmpz_poly_t f,
    const mpq_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, long
    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, long n)
```

Evaluates f at the n values given in the vector f, writing the results to res.

11.25 Interpolation

```
void fmpz_poly_interpolate_fmpz_vec(fmpz_poly_t poly, const
    fmpz * xs, const fmpz * ys, long 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_u in xs and ys, assuming that this polynomial has integer coefficients.

If an interpolating polynomial with integer coefficients does not exist, the result is undefined.

It is assumed that the x values are distinct.

11.26 Composition

```
void _fmpz_poly_compose_horner(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long len2)
```

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `res` has space for `(len1-1)*(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, long len1, const fmpz * poly2, long 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)* (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 $f(t) = g(h(t))$.

```
void _fmpz_poly_compose(fmpz * res, const fmpz * poly1,
    long len1, const fmpz * poly2, long len2)
```

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `res` has space for `(len1-1)*(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(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 $f(t) = g(h(t))$.

11.27 Signature

```
void _fmpz_poly_signature(long * r1, long * r2, fmpz *
    poly, long len)
```

Computes the signature (r_1, r_2) of the polynomial `(poly, len)`. Assumes that the polynomial is squarefree over $\mathbf{Q}$.

```
void fmpz_poly_signature(long * r1, long * r2, fmpz_poly_t
    poly)
```

Computes the signature (r_1, r_2) of the polynomial `poly`, which is assumed to be square-free over $\mathbf{Q}$. 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 [4, Algorithm 4.1.11].

11.28 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 carot `^` 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:

```
5*x^3+7*x-4
x^2+3
-x^4+2*x-1
x+1
5
```

```
int _fmpz_poly_print(const fmpz * poly, long 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, long 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, long
    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, long len, 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, 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.

The parser is implemented via a finite state machine as follows:

```
state   event     next state
----------------------------
  0      '-'         1
         D           2
         V0          3
  1      D           2
         V0          3
  2      D           2
         '*'         4
         '+', '-'    1
  3      V           3
         '^'         5
         '+', '-'    1
  4      V0          3
  5      D           6
  6      D           6
         '+', '-'    1
```

Here, `D` refers to any digit, `V0` to any character which is allowed as the first character in the variable name (an alphetic character), and `V` to any character which is allowed in the remaining part of the variable name (an alphanumeric character or underscore).

Once we encounter a character which does not fit into the above pattern, we stop.

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.

11.29 Modular reduction and reconstruction

```
void fmpz_poly_get_nmod_poly(nmod_poly_t Amod, 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 `Amod` 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 `Amod` 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, long len1, const fmpz_t m1, mp_srcptr poly2, long
    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 $m_1 m_2$ 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 $m_1 m_2$, 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 $0 <= r < m_1 m_2$ are computed, while if `sign` = 1, residues $-m_1 m_2/2 <= r < m_1 m_2/2$ 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, long
    len1, const fmpz_t m1, mp_srcptr poly2, long 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 $m_1 m_2$ 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)
```

Given `poly1` with coefficients modulo `m` and `poly2` with modulus n, sets `res` to the CRT reconstruction modulo mn with signed coefficients satisfying $-mn/2 \leq c < mn/2$.

```
void fmpz_poly_CRT_ui_unsigned(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_t m1, const nmod_poly_t
    poly2)
```

Given `poly1` with coefficients modulo `m` and `poly2` with modulus n, sets `res` to the CRT reconstruction modulo mn with unsigned coefficients satisfying $0 \leq c < mn$.

11.30 Products

```
void _fmpz_poly_product_roots_fmpz_vec(fmpz * poly, const
    fmpz * xs, long po)
```

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, long , )
```

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`.

§12. fmpq

Arbitrary-precision rational numbers

12.1 Introduction

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.

12.2 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.

12.3 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.

12.4 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_zero(fmpq_t res)
```

Sets the value of `res` to 0.

```
void fmpq_one(fmpq_t res)
```

Sets the value of `res` to 1.

12.5 Comparison

```
int fmpq_is_zero(fmpq_t res)
```

Returns nonzero if `res` has value 0, and returns zero otherwise.

```
int fmpq_is_one(fmpq_t res)
```

Returns nonzero if `res` has value 1, and returns zero otherwise.

```
int fmpq_equal(const fmpq_t x, const fmpq_t y)
```

Returns nonzero if `x` and `y` are equal, and zero otherwise. Assumes that `x` and `y` are both in canonical form.

```
int fmpq_sgn(const fmpq_t x)
```

Returns the sign of the rational number x.

```
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.

```
mp_bitcnt_t fmpq_height_bits(const fmpq_t x)
```

Returns the number of bits in the height of x.

12.6 Conversion

```
void fmpq_set_si(fmpq_t res, long p, ulong q)
```

Sets `res` to the canonical form of the fraction `p / q`.

```
void _fmpq_set_si(fmpz_t rnum, fmpz_t rden, long 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`.

```
void fmpq_get_mpq(mpq_t dest, const fmpq_t src)
```

Sets the value of `dest`

12.7 Input and output

```
void fmpq_print(const fmpq_t x)
```

Prints `x` as a fraction. The numerator and denominator are printed verbatim as integers, with a forward slash (/) printed in between.

```
void _fmpq_print(fmpz_t num, fmpz_t den)
```

Does the same thing as `fmpq_print`, but for numerator and denominator given explicitly as `fmpz_t` variables.

12.8 Random number generation

```
void fmpq_randtest(fmpq_t res, flint_rand_t state,
    mp_bitcnt_t bits)
```

Sets `res` to a random value, with numerator and denominator having up to `bits` bits. The fraction will be in canonical form. This function has an increased probability of generating special values which are likely to trigger corner cases.

```
void _fmpq_randtest(fmpz_t num, fmpz_t den, flint_rand_t
    state, mp_bitcnt_t bits)
```

Does the same thing as `fmpq_randtest`, but for numerator and denominator given explicitly as `fmpz_t` variables. Aliasing of `num` and `den` is not allowed.

```
void fmpq_randbits(fmpq_t res, flint_rand_t state,
    mp_bitcnt_t bits)
```

Sets `res` to a random value, with numerator and denominator both having exactly `bits` bits before canonicalisation, and then puts `res` in canonical form. Note that as a result of the canonicalisation, the resulting numerator and denominator can be slightly smaller than `bits` bits.

```
void _fmpq_randbits(fmpz_t num, fmpz_t den, flint_rand_t
    state, mp_bitcnt_t bits)
```

Does the same thing as `fmpq_randbits`, but for numerator and denominator given explicitly as `fmpz_t` variables. Aliasing of `num` and `den` is not allowed.

12.9 Arithmetic

```
void fmpq_add(fmpq_t res, const fmpq_t op1, const fmpq_t
    op2)
```

```
void fmpq_sub(fmpq_t res, const fmpq_t op1, const fmpq_t
    op2)
```

```
void fmpq_mul(fmpq_t res, const fmpq_t op1, const fmpq_t
    op2)
```

```
void fmpq_div(fmpq_t res, const fmpq_t op1, const fmpq_t
    op2)
```

Sets `res` respectively to `op1 + op2`, `op1 - op2`, `op1 * op2`, or `op1 / op2`. Assumes that the inputs are in canonical form, and produces output in canonical form. 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 `(op1num, op1den)`. Aliasing between any combination of the variables is allowed, as long as no numerator is aliased with a denominator.

```
void fmpq_addmul(fmpq_t res, const fmpq_t op1, const fmpq_t
    op2)
```

```
void fmpq_submul(fmpq_t res, const fmpq_t op1, const fmpq_t
    op2)
```

Sets `res` to `res + op1 * op2` or `res - op1 * op2` respectively, placing the result in canonical form. Aliasing between any combination of the variables is allowed.

```
void _fmpq_addmul(fmpz_t rnum, fmpz_t rden, const fmpz_t
    op1num, const fmpz_t op1den, const fmpz_t op2num, const
    fmpz_t op2den)
```

```
void _fmpq_submul(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 fraction (rnum, rden) + (op1num, op1den) * (op1num, op1den) or (rnum, rden) - (op1num, op1den) * (op1num, op1den) respectively. Aliasing between any combination of the variables is allowed, as long as no numerator is aliased with a denominator.

```
void fmpq_inv(fmpq_t dest, const fmpq_t src)
```

Sets dest to 1 / src. The result is placed in canonical form, assuming that src is already in canonical form.

```
void fmpq_mul_fmpz(fmpq_t res, const fmpq_t op, const
    fmpz_t x)
```

Sets res to the product of the rational number op and the integer x.

```
void fmpq_div_fmpz(fmpq_t res, const fmpq_t op, const
    fmpz_t x)
```

Sets res to the quotient of the rational number op and the integer x.

12.10 Modular reduction and rational reconstruction

```
int _fmpq_mod_fmpz(fmpz_t res, fmpz_t num, fmpz_t den,
    fmpz_t mod)
```

```
int fmpq_mod_fmpz(fmpz_t res, const fmpq_t x, const fmpz_t
    mod)
```

Sets the integer res to the residue a of $x = n/d =$ (num, den) modulo the positive integer $m =$ mod, defined as the $0 \le a < m$ satisfying $n \equiv ad \pmod{m}$. If such an a exists, 1 will be returned, otherwise 0 will be returned.

```
int _fmpq_reconstruct_fmpz(fmpz_t num, fmpz_t den, 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 res = (num, den) from its residue a modulo m. This is essentially the inverse operation of fmpq_mod_fmpz.

More precisely, assuming $m > 1$ and $0 \le a < m$ this function either finds the unique rational number n/d with $|n|, d \le \sqrt{m/2}$ and $n \equiv ad \pmod{m}$ and returns 0, or returns 1 if no such rational number exists.

12.11 Rational enumeration

```
void _fmpq_next_minimal(fmpz_t rnum, fmpz_t rden, const
    fmpz_t num, const fmpz_t den)
```

```
void fmpq_next_minimal(fmpq_t res, const fmpq_t x)
```

Given x which is 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 $1 \le p < q$ in order and generating both p/q and q/p, but skipping all $\gcd(p, q) \ne 1$. Starting with zero, this generates every nonnegative rational number once and only once, with the first few entries being:

$$0, 1, 1/2, 2, 1/3, 3, 2/3, 3/2, 1/4, 4, 3/4, 4/3, 1/5, 5, 2/5, \ldots.$$

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.

```
void _fmpq_next_signed_minimal(fmpz_t rnum, fmpz_t rden,
    const fmpz_t num, const fmpz_t den)
```

```
void fmpq_next_signed_minimal(fmpq_t res, const fmpq_t x)
```

Given a signed rational number x 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:

$$0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, \ldots.$$

Starting with zero, this generates every rational number once and only once, in order of minimal height.

```
void _fmpq_next_calkin_wilf(fmpz_t rnum, fmpz_t rden, const
    fmpz_t num, const fmpz_t den)
```

```
void fmpq_next_calkin_wilf(fmpq_t res, const fmpq_t x)
```

Given x 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:

$$0, 1, 1/2, 2, 1/3, 3/2, 2/3, 3, 1/4, 4/3, 3/5, 5/2, 2/5, \ldots.$$

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.

```
void _fmpq_next_signed_calkin_wilf(fmpz_t rnum, fmpz_t
    rden, const fmpz_t num, const fmpz_t den)
```

```
void fmpq_next_signed_calkin_wilf(fmpq_t res, const fmpq_t
    x)
```

Given a signed rational number x assumed to be in canonical form, sets `res` to the next element in the Calkin-Wilf sequence with negative numbers interleaved:

$$0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, \ldots.$$

Starting with zero, this generates every rational number once and only once, but not in order of minimal height.

§13. fmpq_mat

Matrices over **Q**

13.1 Introduction

The `fmpq_mat_t` data type represents matrices over **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.

13.2 Memory management

```
void fmpq_mat_init(fmpq_mat_t mat, long rows, long cols)
```

Initialises a matrix with the given number of rows and columns for use.

```
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.

13.3 Entry access

```
MACRO fmpq_mat_entry(mat,i,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.

```
MACRO fmpq_mat_entry_num(mat,i,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.

```
MACRO fmpq_mat_entry_den(mat,i,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.

13.4 Basic assignment

```
void fmpq_mat_set(fmpq_mat_t dest, const fmpq_mat_t src)
```

Sets the entries in `dest` to the same values as in `src`, assuming the two matrices have the same dimensions.

```
void fmpq_mat_zero(fmpq_mat_t mat)
```

Sets `mat` to the zero matrix.

```
void fmpq_mat_one(fmpq_mat_t mat)
```

Let m be the minimum of the number of rows and columns in the matrix `mat`. This function sets the first $m \times m$ block to the identity matrix, and the remaining block to zero.

13.5 Addition, scalar multiplication

```
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.

```
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.

```
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.

```
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`.

```
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`.

13.6 Input and output

```
void fmpq_mat_print(fmpq_mat_t mat)
```

Prints the matrix mat to standard output.

13.7 Random matrix generation

```
void fmpq_mat_randbits(fmpq_mat_t mat, flint_rand_t state,
    mp_bitcnt_t bits)
```

This is equivalent to applying fmpq_randbits to all entries in the matrix.

```
void fmpq_mat_randtest(fmpq_mat_t mat, flint_rand_t state,
    mp_bitcnt_t bits)
```

This is equivalent to applying fmpq_randtest to all entries in the matrix.

13.8 Special matrices

```
void fmpq_mat_hilbert_matrix(fmpq_mat_t mat)
```

Sets mat to a Hilbert matrix of the given size. That is, the entry at row i and column j is set to $1/(i+j+1)$.

13.9 Basic comparison and properties

```
int fmpq_mat_equal(const fmpq_mat_t mat1, const fmpq_mat_t
    mat2)
```

Returns nonzero if mat1 and mat2 have the same shape and all their entries agree, and returns zero otherwise. Assumes the entries in both mat1 and mat2 are in canonical form.

```
int fmpq_mat_is_integral(const fmpq_mat_t mat)
```

Returns nonzero if all entries in mat are integer-valued, and returns zero otherwise. Assumes that the entries in mat are in canonical form.

```
int fmpq_mat_is_zero(const fmpq_mat_t mat)
```

Returns nonzero if all entries in mat are zero, and returns zero otherwise.

```
int fmpq_mat_is_empty(fmpq_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 fmpq_mat_is_square(fmpq_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.

13.10 Integer matrix conversion

```
int fmpq_mat_get_fmpz_mat(fmpz_mat_t dest, const fmpq_mat_t
    mat)
```

Sets dest to mat and returns nonzero if all entries in mat are integer-valued. If not all entries in mat are integer-valued, sets dest to an undefined matrix and returns zero. Assumes that the entries in mat are in canonical form.

```
void fmpq_mat_get_fmpz_mat_entrywise(fmpz_mat_t num,
    fmpz_mat_t den, const fmpq_mat_t mat)
```

Sets the integer matrices `num` and `den` respectively to the numerators and denominators of the entries in `mat`.

```
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`.

13.11 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)
```

Set `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.

13.12 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.

13.13 Determinant

```
void fmpq_mat_det(fmpq_t det, 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.

13.14 Nonsingular solving

```
int fmpq_mat_solve_fraction_free(fmpq_mat_t X, const
    fmpq_mat_t A, const fmpq_mat_t B)
```

Solves `AX = B` for nonsingular A by clearing denominators and solving the rescaled system over the integers using a fraction-free algorithm. This is usually the fastest algorithm for small systems. Returns nonzero if X is nonsingular or if the right hand side is empty, and zero otherwise.

```
int fmpq_mat_solve_dixon(fmpq_mat_t X, const fmpq_mat_t A,
    const fmpq_mat_t B)
```

Solves `AX = B` for nonsingular A by clearing denominators and solving the rescaled system over the integers using Dixon's algorithm. The rational solution matrix is generated using rational reconstruction. This is usually the fastest algorithm for large systems. Returns nonzero if X is nonsingular or if the right hand side is empty, and zero otherwise.

13.15 Inverse

```
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.

13.16 Echelon form

```
int fmpq_mat_pivot(long * perm, fmpq_mat_t mat, long r,
    long 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.

```
long 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.

```
long 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.

```
long 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.

§14. fmpq_poly

Polynomials over $\mathbf{Q}$

14.1 Introduction

The fmpq_poly_t data type represents elements of $\mathbf{Q}[x]$. 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 long, 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.

14.2 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, long alloc)
```

Initialises the polynomial with space for at least `alloc` coefficients and set the length to zero. The `alloc` coefficients are all set to zero.

```
void fmpq_poly_realloc(fmpq_poly_t poly, long 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, long 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, long 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, long
    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, long 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.

14.3 Polynomial parameters

```
long fmpq_poly_degree(fmpq_poly_t poly)
```

Returns the degree of `poly`, which is one less than its length, as a `long`.

```
long fmpq_poly_length(fmpq_poly_t poly)
```

Returns the length of `poly`.

14.4 Accessing the numerator and denominator

```
fmpz * fmpq_poly_numref(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 `fmpq_poly_fit_length()` or `fmpq_poly_realloc()`.

This function is implemented as a macro returning `(poly)->coeffs`.

```
fmpz_t fmpq_poly_denref(fmpq_poly_t poly)
```

Returns a reference to the denominator as a `fmpz_t`. The integer is guaranteed to be properly initialised.

This function is implemented as a macro returning `(poly)->den`.

14.5 Random testing

The functions `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, `COEFF_MAX`, and `LONG_MAX`. On a polynomial level, the integer numerator has a reasonable chance to have a non-trivial content.

```
void fmpq_poly_randtest(fmpq_poly_t f, flint_rand_t state,
    long len, mp_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 `flint_randinit()` before calling this function.

```
void fmpq_poly_randtest_unsigned(fmpq_poly_t f,
    flint_rand_t state, long len, mp_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 `flint_randinit()` before calling this function.

```
void fmpq_poly_randtest_not_zero(fmpq_poly_t f,
    flint_rand_t state, long len, mp_bitcnt_t bits)
```

As for `fmpq_poly_randtest()` except that `len` and `bits` may not be zero and the polynomial generated is guaranteed not to be the zero polynomial. One must call `flint_randinit()` before calling this function.

14.6 Assignment, swap, negation

```
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, long 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_mpz(fmpq_poly_t poly, const mpz_t x)
```

Sets poly to the integer x.

```
void fmpq_poly_set_mpq(fmpq_poly_t poly, const mpq_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_array_mpq(fmpz * poly, fmpz_t den,
    const mpq_t * a, long n)
```

Sets (poly, den) to the polynomial given by the first $n \geq 1$ coefficients in the array a, from lowest degree to highest degree.

The result is only guaranteed to be in lowest terms if all input coefficients are given in lowest terms.

```
void fmpq_poly_set_array_mpq(fmpq_poly_t poly, const mpq_t
    * a, long n)
```

Sets poly to the polynomial with coefficients as given in the array a of length $n \geq 0$, from lowest degree to highest degree.

The result is only guaranteed to be in canonical form if all input coefficients are given in lowest terms.

```
int _fmpq_poly_set_str(fmpz * poly, fmpz_t den, const char
    * str)
```

Sets (poly, den) to the polynomial specified by the null-terminated string str.

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 a non-zero value, 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 result is only guaranteed to be in canonical for if all coefficients in the input string are in lowest terms.

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.

```
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 fmpq_poly_zero(fmpq_poly_t poly)
```

Sets `poly` to zero.

```
void fmpq_poly_one(fmpq_poly_t poly)
```

Sets `poly` to the constant polynomial 1.

```
void fmpq_poly_neg(fmpq_poly_t poly1, const fmpq_poly_t
    poly2)
```

Sets `poly1` to the additive inverse of `poly2`.

```
void fmpq_poly_inv(fmpq_poly_t poly1, const fmpq_poly_t
    poly2)
```

Sets `poly1` to the multiplicative inverse of `poly2` if possible. Otherwise, if `poly2` is not a unit, leaves `poly1` unmodified and calls `abort()`.

```
void fmpq_poly_swap(fmpq_poly_t poly1, fmpq_poly_t poly2)
```

Efficiently swaps the polynomials `poly1` and `poly2`.

```
void fmpq_poly_truncate(fmpq_poly_t poly, long n)
```

If the current length of `poly` is greater than n, it is truncated to the given length. Discarded coefficients are demoted, but they are not necessarily set to zero.

14.7 Setting and getting coefficients

```
void fmpq_poly_get_coeff_fmpq(fmpq_t x, const fmpq_poly_t
    poly, long n)
```

Retrieves the nth coefficient of `poly`, in lowest terms.

```
void fmpq_poly_get_coeff_mpq(mpq_t x, const fmpq_poly_t
    poly, long n)
```

Retrieves the nth coefficient of `poly`, in lowest terms.

```
void fmpq_poly_set_coeff_si(fmpq_poly_t poly, long n, long
    x)
```

Sets the nth coefficient in `poly` to the integer x.

```
void fmpq_poly_set_coeff_ui(fmpq_poly_t poly, long n, ulong
    x)
```

Sets the nth coefficient in `poly` to the integer x.

```
void fmpq_poly_set_coeff_fmpz(fmpq_poly_t poly, long n,
    const fmpz_t x)
```

Sets the nth coefficient in `poly` to the integer x.

```
void fmpq_poly_set_coeff_fmpq(fmpq_poly_t poly, long n,
    const fmpq_t x)
```

Sets the nth coefficient in `poly` to the rational x.

```
void fmpq_poly_set_coeff_mpz(fmpq_poly_t rop, long n, const
    mpz_t x)
```

Sets the nth coefficient in `poly` to the integer x.

```
void fmpq_poly_set_coeff_mpq(fmpq_poly_t rop, long n, const
    mpq_t x)
```

Sets the nth coefficient in `poly` to the rational x, which is expected to be provided in lowest terms.

14.8 Comparison

```
int fmpq_poly_equal(const fmpq_poly_t poly1, const
    fmpq_poly_t poly2)
```

Returns 1 if `poly1` is equal to `poly2`, otherwise returns 0.

```
int _fmpq_poly_cmp(const fmpz * lpoly, const fmpz_t lden,
    const fmpz * rpoly, const fmpz_t rden, long 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_cmp(const fmpq_poly_t left, const fmpq_poly_t
    right)
```

Compares the two polynomials `left` and `right`.

Compares the two polynomials `left` and `right`, returning -1, 0, or 1 as `left` is less than, equal to, or greater than `right`. The comparison is first done by the degree, and then, in case of a tie, by the individual coefficients from highest to lowest.

```
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.

14.9 Addition and subtraction

```
void _fmpq_poly_add(fmpz * rpoly, fmpz_t rden, const fmpz *
    poly1, const fmpz_t den1, long len1, const fmpz * poly2,
    const fmpz_t den2, long 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(fmpq_poly_t res, fmpq_poly poly1,
    fmpq_poly poly2)
```

Sets `res` to the sum of `poly1` and `poly2`, using Henrici's algorithm.

```
void _fmpq_poly_sub(fmpz * rpoly, fmpz_t rden, const fmpz *
    poly1, const fmpz_t den1, long len1, const fmpz * poly2,
    const fmpz_t den2, long 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(fmpq_poly_t res, fmpq_poly poly1,
    fmpq_poly poly2)
```

Sets `res` to the difference of `poly1` and `poly2`, using Henrici's algorithm.

14.10 Scalar multiplication and division

```
void _fmpq_poly_scalar_mul_si(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long len, long c)
```

Sets `(rpoly, rden, len)` to the product of c of `(poly, den, 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 `(rpoly, den)` and `(poly, den)`.

```
void _fmpq_poly_scalar_mul_ui(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long len, ulong c)
```

Sets `(rpoly, rden, len)` to the product of c of `(poly, den, 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 `(rpoly, den)` and `(poly, den)`.

```
void _fmpq_poly_scalar_mul_fmpz(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long len, const
    fmpz_t c)
```

Sets `(rpoly, rden, len)` to the product of c of `(poly, den, 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 `(rpoly, den)` and `(poly, den)`.

```
void _fmpq_poly_scalar_mul_mpq(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long len, const
    fmpz_t r, const fmpz_t s)
```

Sets `(rpoly, rden)` to the product of r/s and `(poly, den, len)`, in lowest terms.

Assumes that `(poly, den, len)` and r/s are provided in lowest terms. Assumes that `rpoly` is an array of length `len`. Supports aliasing of `(rpoly, den)` and `(poly, den)`. The `fmpz_t`'s r and s may not be part of `(rpoly, rden)`.

```
void fmpq_poly_scalar_mul_si(fmpq_poly_t rop, const
    fmpq_poly_t op, long c)
```

Sets `rop` to c times `op`.

```
void fmpq_poly_scalar_mul_ui(fmpq_poly_t rop, const
    fmpq_poly_t op, ulong c)
```

Sets `rop` to c times `op`.

```
void fmpq_poly_scalar_mul_fmpz(fmpq_poly_t rop, const
    fmpq_poly_t op, const fmpz_t c)
```

Sets `rop` to c times `op`. Assumes that the `fmpz_t c` is not part of `rop`.

```
void fmpq_poly_scalar_mul_mpz(fmpq_poly_t rop, const
    fmpq_poly_t op, const mpz_t c)
```

Sets `rop` to c times `op`.

```
void fmpq_poly_scalar_mul_mpq(fmpq_poly_t rop, const
    fmpq_poly_t op, const mpq_t c)
```

Sets `rop` to c times `op`.

```
void _fmpq_poly_scalar_div_fmpz(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long len, const
    fmpz_t 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)`. Assumes that c is not part of `(rpoly, rden)`.

```
void _fmpq_poly_scalar_div_si(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long len, long 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, long 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_mpq(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long 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, long c)
```

```
void fmpq_poly_scalar_div_ui(fmpq_poly_t rop, const
    fmpq_poly_t op, ulong c)
```

```
void fmpq_poly_scalar_div_fmpz(fmpq_poly_t rop, const
    fmpq_poly_t op, const fmpz_t c)
```

```
void fmpq_poly_scalar_div_mpz(fmpq_poly_t rop, const
    fmpq_poly_t op, const mpz_t c)
```

```
void fmpq_poly_scalar_div_mpq(fmpq_poly_t rop, const
    fmpq_poly_t op, const mpq_t c)
```

14.11 Multiplication

```
void _fmpq_poly_mul(fmpz * rpoly, fmpz_t rden, const fmpz *
    poly1, const fmpz_t den1, long len1, const fmpz * poly2,
    const fmpz_t den2, long 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, long len1, const fmpz *
    poly2, const fmpz_t den2, long len2, long 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, long 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, 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, fmpq_poly_t op2)
```

Subtracts the product of `op1` and `op2` from `rop`.

14.12 Powering

```
void _fmpq_poly_pow(fmpz * rpoly, fmpz_t rden, const fmpz *
    poly, const fmpz_t den, long 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)`.

```
void fmpq_poly_pow(fmpq_poly_t res, const fmpq_poly_t poly,
    ulong e)
```

Sets `res` to `pow^e`, where the only special case 0^0 is defined as 1.

14.13 Shifting

```
void fmpz_poly_shift_left(fmpz_poly_t res, const
    fmpz_poly_t poly, long n)
```

Set `res` to `poly` shifted left by n coefficients. Zero coefficients are inserted.

```
void fmpz_poly_shift_right(fmpz_poly_t res, const
    fmpz_poly_t poly, long 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.

14.14 Euclidean division

```
void _fmpq_poly_divrem(fmpz * Q, fmpz_t q, fmpz * R, fmpz_t
    r, const fmpz * A, const fmpz_t a, long lenA, const fmpz
    * B, const fmpz_t b, long lenB)
```

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.

```
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`.

```
void _fmpq_poly_div(fmpz * Q, fmpz_t q, const fmpz * A,
    const fmpz_t a, long lenA, const fmpz * B, const fmpz_t
    b, long lenB)
```

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.

```
void fmpq_poly_div(fmpq_poly_t Q, 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.

```
void _fmpq_poly_rem(fmpz * R, fmpz_t r, const fmpz * A,
    const fmpz_t a, long lenA, const fmpz * B, const fmpz_t
    b, long lenB)
```

Finds the remainder (R, r) of the Euclidean division of (A, a) by (B, b).

Assumes that lenA >= lenB > 0. Supports no aliasing between the inputs and the outputs.

```
void fmpq_poly_rem(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.

14.15 Power series division

```
void _fmpq_poly_inv_series_newton(fmpz * rpoly, fmpz_t
    rden, const fmpz * poly, const fmpz_t den, long n)
```

Computes the first n terms of the inverse power series of poly using Newton iteration.

Assumes that $n \geq 1$, that poly has length at least n and non-zero constant term. Does not support aliasing.

```
void fmpq_poly_inv_series_newton(fmpq_poly_t res, const
    fmpq_poly_t poly, long )
```

Computes the first n terms of the inverse power series of poly using Newton iteration, assuming that poly has non-zero constant term and $n \geq 1$.

```
void _fmpq_poly_inv_series(fmpz * rpoly, fmpz_t rden, const
    fmpz * poly, const fmpz_t den, long n)
```

Computes the first n terms of the inverse power series of poly.

Assumes that $n \geq 1$, that poly has length at least n and non-zero constant term. Does not support aliasing.

```
void fmpq_poly_inv_series(fmpq_poly_t res, const
    fmpq_poly_t poly, long n)
```

Computes the first n terms of the inverse power series of poly, assuming that poly has non-zero constant term and $n \geq 1$.

```
void _fmpq_poly_div_series(fmpz * Q, fmpz_t denQ, const
    fmpz * A, const fmpz_t denA, const fmpz * B, const
    fmpz_t denB, long n)
```

Divides (A, denA, n) by (B, denB, n) as power series over $\mathbf{Q}$, assuming B has non-zero constant term and $n \geq 1$.

Supports no aliasing other than that of (Q, denQ, n) and (B, denB, n).

This function does not ensure that the numerator and denominator are coprime on exit.

```
void fmpq_poly_div_series(fmpq_poly_t Q, const fmpq_poly_t
    A, const fmpq_poly_t B, long n)
```

Performs power series division in $\mathbf{Q}[[x]]/(x^n)$. 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 $n \geq 1$.

14.16 Derivative and integral

```
void _fmpq_poly_derivative(fmpz * rpoly, fmpz_t rden, const
    fmpz * poly, const fmpz_t den, long 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_integral(fmpz * rpoly, fmpz_t rden, const
    fmpz * poly, const fmpz_t den, long len)
```

Sets `(rpoly, rden, len)` to the integral of `(poly, den, len - 1)`. Assumes `len >= 0`. Supports aliasing between the two polynomials.

```
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.

14.17 Square roots

```
void _fmpq_poly_sqrt_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, long n)
```

Sets `(g, gden, n)` to the series expansion of the square root of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` 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, long 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, long n)
```

Sets `(g, gden, n)` to the series expansion of the inverse square root of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` 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, long 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.

14.18 Transcendental functions

```
void _fmpq_poly_log_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, long n)
```

Sets `(g, gden, n)` to the series expansion of the logarithm of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` 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, long 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, long n)
```

Sets `(g, gden, n)` to the series expansion of the exponential function of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_exp_series(fmpq_poly_t res, const
    fmpq_poly_t h, long n)
```

Sets `res` to the series expansion of the exponential function of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_atan_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, long n)
```

Sets `(g, gden, n)` to the series expansion of the inverse tangent of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` 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, long 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, long n)
```

Sets `(g, gden, n)` to the series expansion of the inverse hyperbolic tangent of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` 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, long 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, long n)
```

Sets `(g, gden, n)` to the series expansion of the inverse sine of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` 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, long 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, long n)
```

Sets (g, gden, n) to the series expansion of the inverse hyperbolic sine of (f, fden, n). Assumes n > 0 and that (f, fden, n) 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, long 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 * h, const fmpz_t hden, long n)
```

Sets (g, gden, n) to the series expansion of the tangent function of (f, fden, n). Assumes n > 0 and that (f, fden, n) 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 h, long 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, long n)
```

Sets (g, gden, n) to the series expansion of the sine of (f, fden, n). Assumes n > 0 and that (f, fden, n) 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, long 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, long n)
```

Sets (g, gden, n) to the series expansion of the cosine of (f, fden, n). Assumes n > 0 and that (f, fden, n) 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, long 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_sinh_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, long n)
```

Sets (g, gden, n) to the series expansion of the hyperbolic sine of (f, fden, n). Assumes n > 0 and that (f, fden, n) 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, long 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, long n)
```

Sets `(g, gden, n)` to the series expansion of the hyperbolic cosine of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` 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, long 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_tanh_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, long n)
```

Sets `(g, gden, n)` to the series expansion of the hyperbolic tangent of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` 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, long n)
```

Sets `res` to the series expansion of the hyperbolic tangent of `f` to order `n > 0`. Requires `f` to have constant term 0.

14.19 Evaluation

```
void _fmpq_poly_evaluate_fmpz(fmpz_t rnum, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long 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(mpq_t res, const fmpq_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_mpq(fmpz_t rnum, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long 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_mpq(mpq_t res, const fmpq_poly_t
    poly, const mpq_t a)
```

Evaluates the polynomial `poly` at the rational a and sets `res` to the result.

14.20 Interpolation

```
void _fmpq_poly_interpolate_fmpz_vec(fmpz * poly, fmpz_t
    den, const fmpz * xs, const fmpz * ys, long 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. Aliasing of `poly` or `den` with any other argument is not allowed.

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.

```
fmpq_poly_interpolate_fmpz_vec(fmpq_poly_t poly, const fmpz
    * xs, const fmpz * ys, long 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.

14.21 Composition

```
void _fmpq_poly_compose(fmpz * res, fmpz_t den, const fmpz
    * poly1, const fmpz_t den1, long len1, const fmpz *
    poly2, const fmpz_t den2, long 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, long 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 fmpz_poly_rescale(fmpq_poly_t res, const fmpq_poly_t
    poly, const mpq_t a)
```

Sets `res` to `poly` with the indeterminate rescaled by a.

14.22 Gaussian content

```
void _fmpq_poly_content(mpq_t res, const fmpz * poly, const
    fmpz_t den, long len)
```

Sets `res` to the content of `(poly, den, len)`. If `len == 0`, sets `res` to zero.

```
void fmpq_poly_content(mpq_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, long 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, long 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, long 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.

14.23 Square-free

```
int _fmpq_poly_is_squarefree(const fmpz * poly, const
    fmpz_t den, long len)
```

Returns whether the polynomial (poly, den, len) is 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.

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.

14.24 Input and output

```
int _fmpq_poly_print(const fmpz * poly, const fmpz_t den,
    long 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 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 case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fmpq_poly_fprint(FILE * file, const fmpz * poly, const
    fmpz_t den, long 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(FILE * file, const fmpq_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 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.

§15. fmpz_poly_q

Rational functions over **Q**

15.1 Introduction

The module fmpz_poly_q provides functions for performing arithmetic on rational functions in $\mathbf{Q}(t)$, 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 canonicalise the rational function using the function fmpz_poly_q_canonicalise() 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 [13]. Differentiation is implemented in a way slightly improving on the algorithm described in [14].

15.2 Simple example

The following example computes the product of two rational functions and prints the result:

```
#include "fmpz_poly_q.h"
...
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");
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:

```
(3*t+1)/2 * 3/(7*t+2) = (9*t+3)/(14*t+4)
```

15.3 Memory management

We represent a rational function over **Q** as the quotient of two coprime integer polynomials of type `fmpz_poly_t`, enforcing that the leading coefficient of the denominator is positive. The zero function is represented as 0/1.

```
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.

15.4 Randomisation

```
void fmpz_poly_q_randtest(fmpz_poly_q_t poly, flint_rand_t
    state, long len1, mp_bitcnt_t bits1, long len2,
    mp_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, long len1, mp_bitcnt_t bits1, long
    len2, mp_bitcnt_t bits2)
```

Sets poly to a random non-zero rational function.

15.5 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, long op)
```

Sets the element rop to the value given by the long 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.

15.6 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.

15.7 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.

15.8 Scalar multiplication and division

```
void fmpz_poly_q_scalar_mul_si(fmpz_poly_q_t rop, const
    fmpz_poly_q_t op, long x)
```

Sets rop to the product of the rational function op and the long integer x.

```
void fmpz_poly_q_scalar_mul_mpz(fmpz_poly_q_t rop, const
    fmpz_poly_q_t op, const mpz_t x)
```

Sets rop to the product of the rational function op and the mpz_t integer x.

```
void fmpz_poly_q_scalar_mul_mpq(fmpz_poly_q_t rop, const
    fmpz_poly_q_t op, const mpq_t x)
```

Sets rop to the product of the rational function op and the mpq_t rational x.

```
void fmpz_poly_q_scalar_div_si(fmpz_poly_q_t rop, const
    fmpz_poly_q_t op, long x)
```

Sets rop to the quotient of the rational function op and the long integer x.

```
void fmpz_poly_q_scalar_div_mpz(fmpz_poly_q_t rop, const
    fmpz_poly_q_t op, const mpz_t x)
```

Sets rop to the quotient of the rational function op and the mpz_t integer x.

```
void fmpz_poly_q_scalar_div_mpq(fmpz_poly_q_t rop, const
    fmpz_poly_q_t op, const mpq_t x)
```

Sets rop to the quotient of the rational function op and the mpq_t rational x.

15.9 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.

15.10 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 $0^0 = 1$.

15.11 Derivative

```
void fmpz_poly_q_derivative(fmpz_poly_q_t rop, const
    fmpz_poly_q_t op)
```

Sets rop to the derivative of op.

15.12 Evaluation

```
int fmpz_poly_q_evaluate(mpq_t rop, const fmpz_poly_q_t f,
    const mpq_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 $f(a)$.

15.13 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 $a_0 + a_1X + \cdots + a_nX^n$ of length $n+1$ is represented by the string "n+1 a_0 a_1 ... 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.

```
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`.

§16. fmpz_poly_mat

Matrices over $\mathbf{Z}[x]$

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.

16.1 Simple example

The following example constructs the matrix $\begin{pmatrix} 2x+1 & x \\ 1-x & -1 \end{pmatrix}$ and computes its determinant.

```
#include "fmpz_poly.h"
#include "fmpz_poly_mat.h"
...
fmpz_poly_mat_t A;
fmpz_poly_t P;

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
```

16.2 Memory management

```
void fmpz_poly_mat_init(fmpz_poly_mat_t mat, long rows,
    long 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.

16.3 Basic assignment and manipulation

```
MACRO fmpz_poly_mat_entry(mat,i,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.

16.4 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`.

16.5 Random matrix generation

```
void fmpz_poly_mat_randtest(fmpz_poly_mat_t mat,
    flint_rand_t state, long len, mp_bitcnt_t bits)
```

This is equivalent to applying `fmpz_poly_randtest` to all entries in the matrix.

16.6 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.

16.7 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_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.

16.8 Norms

```
long 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.

```
long fmpz_poly_mat_max_length(const fmpz_poly_mat_t A)
```

Returns the maximum polynomial length among all the entries in A.

16.9 Evaluation

```
void fmpz_poly_mat_evaluate_fmpz(fmpz_mat_t B, const
    fmpz_poly_mat_t A, const fpz_t x)
```

Sets the fmpz_mat_t B to A evaluated entrywise at the point x.

16.10 Arithmetic

```
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.

16.11 Row reduction

```
long fmpz_poly_mat_rowreduce(long * perm, fmpz_poly_mat_t
    B, fmpz_poly_t den, const fmpz_poly_mat_t A, int options)
```

Sets B to the row reduced form of A as computed using fraction-free Gaussian elimination, and returns the rank. The denominator is written to den.

If perm is non-NULL, the permutation of rows in the matrix will also be applied to perm.

This function accepts the option flags ROWREDUCE_FAST_ABORT for aborting if the matrix does not have full rank, ROWREDUCE_FULL to obtain a (fraction-free) reduced row echelon form, and ROWREDUCE_CLEAR_LOWER to zero entries below the pivots (otherwise a fraction-free LU decomposition is computed, with the L matrix stored in the lower part).

```
int fmpz_poly_mat_pivot(long * perm, fmpz_poly_mat_t A,
    long r, long c)
```

Internal function for pivoting during row reduction.

16.12 Determinants

```
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. The determinant is computed by performing row reduction on a temporary copy of A, except for matrices of size 2 or smaller which are handled directly.

```
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 determing 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.

§17. nmod_vec

Vectors over $\mathbf{Z}/n\mathbf{Z}$ for word-sized moduli

17.1 Memory management

```
mp_ptr _nmod_vec_init(long len)
```

Returns a vector of the given length. The entries are not necessarily zero.

```
void _nmod_vec_free(mp_ptr vec)
```

Frees the memory used by the given vector.

17.2 Modular reduction and arithmetic

```
void nmod_init(nmod_t * mod, mp_limb_t n)
```

Initialises the given `nmod_t` structure for reduction modulo n with a precomputed inverse.

```
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`.

```
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.

```
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`.

```
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`.

```
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.

17.3 Random functions

```
void _nmod_vec_randtest(mp_ptr vec, flint_rand_t state,
    long len, nmod_t mod)
```

Sets `vec` to a random vector of the given length with entries reduced modulo `mod.n`.

17.4 Basic manipulation and comparison

```
void _nmod_vec_set(mp_ptr res, mp_srcptr vec, long len)
```

Copies `len` entries from the vector `vec` to `res`.

```
void _nmod_vec_zero(mp_ptr vec, long len)
```

Zeros the given vector of the given length.

```
void _nmod_vec_swap(mp_ptr a, mp_ptr b, long 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, long len,
    nmod_t mod)
```

Reduces the entries of `(vec, len)` modulo `mod.n` and set `res` to the result.

```
mp_bitcnt_t _nmod_vec_max_bits(mp_srcptr vec, long len)
```

Returns the maximum number of bits of any entry in the vector.

```
int _nmod_vec_equal(mp_ptr vec, mp_srcptr vec2, long len)
```

Returns 1 if (vec, len) is equal to (vec2, len), otherwise returns 0.

17.5 Arithmetic operations

```
void _nmod_vec_add(mp_ptr res, mp_srcptr vec1,
                   mp_srcptr vec2, long len, nmod_t mod)
```

Sets (res, len) to the sum of (vec1, len) and (vec2, len).

```
void _nmod_vec_sub(mp_ptr res, mp_srcptr vec1,
                   mp_srcptr vec2, long len, nmod_t mod)
```

Sets (res, len) to the difference of (vec1, len) and (vec2, len).

```
void _nmod_vec_neg(mp_ptr res, mp_srcptr vec, long len,
   nmod_t mod)
```

Sets (res, len) to the negation of (vec, len).

```
void _nmod_vec_scalar_mul_nmod(mp_ptr res, mp_srcptr vec,
                                long len, mp_limb_t c,
   nmod_t mod)
```

Sets (res, len) to (vec, len) multiplied by c.

```
void _nmod_vec_scalar_addmul_nmod(mp_ptr res, mp_srcptr
   vec,                            long len, mp_limb_t c,
   nmod_t mod)
```

Adds (vec, len) times c to the vector (res, len).

§18. nmod_poly

Polynomials over $\mathbf{Z}/n\mathbf{Z}$ for word-sized moduli

18.1 Introduction

The nmod_poly_t data type represents elements of $\mathbf{Z}/n\mathbf{Z}[x]$ 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.

18.2 Simple example

The following example computes the square of the polynomial $5x^3 + 6$ in $\mathbf{Z}/7\mathbf{Z}[x]$.

```
#include "nmod_poly.h"
...
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_si(x, 0, 6);
nmod_poly_mul(y, x, x);
nmod_poly_print(x); printf("\n");
nmod_poly_print(y); 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
```

18.3 Definition of the nmod_poly_t type

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.

It is recommended that users do not access the fields of an `nmod_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose, detailed below.

Functions in `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.

We now describe the functions available in `nmod_poly`.

18.4 Memory management

```
void nmod_poly_init(nmod_poly_t poly, mp_limb_t n)
```

Initialises `poly`. It will have coefficients modulo n.

```
void nmod_poly_init_preinv(nmod_poly_t poly, mp_limb_t n,
    mp_limb_t ninv)
```

Initialises `poly`. It will have coefficients modulo n. The caller supplies a precomputed inverse limb generated by `n_preinvert_limb()`.

```
void nmod_poly_init2(nmod_poly_t poly, mp_limb_t n, long
    alloc)
```

Initialises `poly`. It will have coefficients modulo n. Up to `alloc` coefficients may be stored in `poly`.

```
void nmod_poly_init2_preinv(nmod_poly_t poly, mp_limb_t n,
    mp_limb_t ninv, long alloc)
```

Initialises `poly`. It will have coefficients modulo n. The caller supplies a precomputed inverse limb generated by `n_preinvert_limb()`. Up to `alloc` coefficients may be stored in `poly`.

```
void nmod_poly_realloc(nmod_poly_t poly, long alloc)
```

Reallocates `poly` to the given length. If the current length is less than `alloc`, the polynomial is truncated and normalised. If `alloc` is zero, the polynomial is cleared.

```
void 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 nmod_poly_fit_length(nmod_poly_t poly, long alloc)
```

Ensures `poly` has space for at least `alloc` coefficients. This function only ever grows the allocated space, so no data loss can occur.

```
void _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.

18.5 Polynomial properties

```
long nmod_poly_length(const nmod_poly_t poly)
```

Returns the length of the polynomial `poly`. The zero polynomial has length zero.

```
long nmod_poly_degree(const nmod_poly_t poly)
```

Returns the degree of the polynomial `poly`. The zero polynomial is deemed to have degree -1.

```
mp_limb_t nmod_poly_modulus(const nmod_poly_t poly)
```

Returns the modulus of the polynomial `poly`. This will be a positive integer.

```
mp_bitcnt_t nmod_poly_max_bits(const nmod_poly_t poly)
```

Returns the maximum number of bits of any coefficient of `poly`.

18.6 Assignment and basic manipulation

```
void nmod_poly_set(nmod_poly_t a, const nmod_poly_t b)
```

Sets `a` to a copy of `b`.

```
void nmod_poly_swap(nmod_poly_t poly1, nmod_poly_t poly2)
```

Efficiently swaps `poly1` and `poly2` by swapping pointers internally.

```
void nmod_poly_zero(nmod_poly_t res)
```

Sets `res` to the zero polynomial.

```
void nmod_poly_truncate(nmod_poly_t poly, long len)
```

Truncates `poly` to the given length and normalises it. If `len` is greater than the current length of `poly`, then nothing happens.

```
void _nmod_poly_reverse(mp_ptr output, mp_srcptr input,
    long len, long m)
```

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 nmod_poly_reverse(nmod_poly_t output, const
    nmod_poly_t input, long m)
```

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.

18.7 Randomisation

```
void nmod_poly_randtest(nmod_poly_t poly, flint_rand_t
    state, long len)
```

Generates a random polynomial with up to the given length.

18.8 Getting and setting coefficients

```
ulong nmod_poly_get_coeff_ui(const nmod_poly_t poly, ulong
    j)
```

Returns the coefficient of `poly` at index `j`, where coefficients are numbered with zero being the constant coefficient, and returns it as an `unsigned long`. If `j` refers to a coefficient beyond the end of `poly`, zero is returned.

```
void nmod_poly_set_coeff_ui(nmod_poly_t poly, ulong 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.

18.9 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 `free()`.

```
int nmod_poly_set_str(const char * s, nmod_poly_t poly)
```

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_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 a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned. If an error occurs whilst writing to the file, an error message is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
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.

18.10 Comparison

```
int nmod_poly_equal(const nmod_poly_t a, const nmod_poly_t
    b)
```

Returns 1 if the polynomials are equal, otherwise 0.

```
int nmod_poly_is_zero(const nmod_poly_t poly)
```

Returns 1 if the polynomial `poly` is the zero polynomial, otherwise returns 0.

18.11 Shifting

```
void _nmod_poly_shift_left(mp_ptr res, mp_srcptr poly, long
    len, long k)
```

Sets `(res, len + k)` to `(poly, len)` shifted left by `k` coefficients. Assumes that `res` has space for `len + k` coefficients.

```
void nmod_poly_shift_left(nmod_poly_t res, const
    nmod_poly_t poly, long k)
```

Sets `res` to `poly` shifted left by `k` coefficients, i.e. multiplied by x^k.

```
void _nmod_poly_shift_right(mp_ptr res, mp_srcptr poly,
    long len, long 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.

```
void nmod_poly_shift_right(nmod_poly_t res, const
    nmod_poly_t poly, long k)
```

Sets `res` to `poly` shifted right by `k` coefficients, i.e. divide by x^k and throws away the remainder. If `k` is greater than or equal to the length of `poly`, the result is the zero polynomial.

18.12 Addition and subtraction

```
void _nmod_poly_add(mp_ptr res, mp_srcptr poly1, long len1,
    mp_srcptr poly2, long len2, nmod_t mod)
```

Sets `res` to the sum of `(poly1, len1)` and `(poly2, len2)`. There are no restrictions on the lengths.

```
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`.

```
void _nmod_poly_sub(mp_ptr res, mp_srcptr poly1, long len1,
    mp_srcptr poly2, long len2, nmod_t mod)
```

Sets `res` to the difference of `(poly1, len1)` and `(poly2, len2)`. There are no restrictions on the lengths.

```
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`.

```
void nmod_poly_neg(nmod_poly_t res, const nmod_poly_t poly)
```

Sets `res` to the negation of `poly`.

18.13 Scalar multiplication and division

```
void nmod_poly_scalar_mul_nmod(nmod_poly_t res, const
    nmod_poly_t poly, ulong c)
```

Sets `res` to `(poly, len)` multiplied by c, where c is reduced modulo the modulus of `poly`.

```
void _nmod_poly_make_monic(mp_ptr output, mp_srcptr input,
    long 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`.

```
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`.

18.14 Bit packing and unpacking

```
void _nmod_poly_bit_pack(mp_ptr res, mp_srcptr poly, long
    len, mp_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^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`.

```
void _nmod_poly_bit_unpack(mp_ptr res, long 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.

18.15 Multiplication

```
void _nmod_poly_mul_classical(mp_ptr res, mp_srcptr poly1,
    long len1, mp_srcptr poly2, long len2, nmod_t mod)
```

Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`. Assumes `len1 >= 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, long len1, mp_srcptr poly2, long len2, long
    trunc, nmod_t mod)
```

Sets `res` to the lower `trunc` coefficients of the product of `(poly1, len1)` and `(poly2, len2)`. Assumes that `len1 >= len2 > 0` and `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, long trunc)
```

Sets `res` to the lower `trunc` coefficients of the product of `poly1` and `poly2`.

```
void _nmod_poly_mulhigh_classical(mp_ptr res, mp_srcptr
    poly1, long len1, mp_srcptr poly2, long len2, long
    start, nmod_t mod)
```

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 nmod_poly_mulhigh_classical(nmod_poly_t res, const
    nmod_poly_t poly1, const nmod_poly_t poly2, long start)
```

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 _nmod_poly_mul_KS(mp_ptr out, mp_srcptr in1, long
    len1, mp_srcptr in2, long len2, mp_bitcnt_t bits, nmod_t
    mod)
```

Sets `res` to the product of `poly1` and `poly2` assuming the output coefficients are at most the given number of bits wide. If `bits` is set to 0 an appropriate value is computed automatically. Assumes that `len1 >= len2 > 0`.

```
void nmod_poly_mul_KS(nmod_poly_t res, const nmod_poly_t
    poly1, const nmod_poly_t poly2, mp_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 `bits` is set to 0 an appropriate value is computed automatically.

```
void _nmod_poly_mullow_KS(mp_ptr out, mp_srcptr in1, long
    len1, mp_srcptr in2, long len2, mp_bitcnt_t bits, long
    n, nmod_t mod)
```

Sets `out` to the low n coefficients of `in1` of length `len1` times `in2` of length `len2`. The output must have space for `n` coefficients. We assume that `len1 >= len2 > 0` and that `0 < n <= len1 + len2 - 1`.

```
void nmod_poly_mullow_KS(nmod_poly_t res, const nmod_poly_t
    poly1, const nmod_poly_t poly2, mp_bitcnt_t bits, long n)
```

Set `res` to the low n coefficients of `in1` of length `len1` times `in2` of length `len2`.

```
void _nmod_poly_mul(mp_ptr res, mp_srcptr poly1, long len1,
    mp_srcptr poly2, long len2, nmod_t mod)
```

Sets `res` to the product of `poly1` of length `len1` and `poly2` of length `len2`. Assumes `len1 >= len2 > 0`. No aliasing is permitted between the inputs and the output.

```
void nmod_poly_mul(nmod_poly_t res, const nmod_poly_t poly,
    const nmod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _nmod_poly_mullow(mp_ptr res, mp_srcptr poly1, long
    len1, mp_srcptr poly2, long len2, long n, nmod_t mod)
```

Sets `res` to the first `n` coefficients of the product of `poly1` of length `len1` and `poly2` of length `len2`. It is assumed that `0 < n <= len1 + len2 - 1` and that `len1 >= len2 > 0`. No aliasing of inputs and output is permitted.

```
void nmod_poly_mullow(nmod_poly_t res, const nmod_poly_t
    poly1, const nmod_poly_t poly2, long trunc)
```

Sets `res` to the first `trunc` coefficients of the product of `poly1` and `poly2`.

```
void _nmod_poly_mulhigh(mp_ptr res, mp_srcptr poly1, long
    len1, mp_srcptr poly2, long len2, long n, nmod_t mod)
```

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 other coefficients being arbitrary. It is assumed that `len1 >= len2 > 0` and that `0 < n <= len1 + len2 - 1`. Aliasing of inputs and output is not permitted.

```
void nmod_poly_mulhigh(nmod_poly_t res, const nmod_poly_t
    poly1, const nmod_poly_t poly2, long n)
```

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.

18.16 Powering

```
void _nmod_poly_pow_binexp(mp_ptr res, mp_srcptr poly, long
    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_binexp(nmod_poly_t res, const
    nmod_poly_t poly, ulong e)
```

Raises `poly` to the power `e` and sets `res` to the result. Uses the binary exponentiation method.

```
void _nmod_poly_pow(mp_ptr res, mp_srcptr poly, long 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.

```
void nmod_poly_pow(nmod_poly_t res, const nmod_poly_t poly,
    ulong e)
```

Raises poly to the power e and sets res to the result.

```
void _nmod_poly_pow_trunc_binexp(mp_ptr res, mp_srcptr
    poly, ulong e, long 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_trunc_binexp(nmod_poly_t res, const
    nmod_poly_t poly, ulong e, long trunc)
```

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 _nmod_poly_pow_trunc(mp_ptr res, mp_srcptr poly, ulong
    e, long 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.

```
void nmod_poly_pow_trunc(nmod_poly_t res, const nmod_poly_t
    poly, ulong e, long trunc)
```

Sets res to the low trunc coefficients of poly to the power e. This is equivalent to doing a powering followed by a truncation.

18.17 Division

```
void _nmod_poly_divrem_basecase(mp_ptr Q, mp_ptr R, mp_ptr
    W, mp_srcptr A, long A_len, mp_srcptr B, long 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 NMOD_DIVREM_BC_ITCH(A_len, B_len, mod) coefficients.

```
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_div_basecase(mp_ptr Q, mp_ptr W, mp_srcptr
    A, long A_len, mp_srcptr B, long B_len, nmod_t mod)
```

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. We require that W is temporary space of NMOD_DIV_BC_ITCH(A_len, B_len, mod) coefficients.

```
void nmod_poly_div_basecase(nmod_poly_t Q, const
    nmod_poly_t A, const nmod_poly_t B)
```

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 _nmod_poly_divrem_divconquer_recursive(mp_ptr Q,
    mp_ptr BQ, mp_ptr W, mp_ptr V, mp_srcptr A, mp_srcptr B,
    long lenB, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `2 * lenB - 1` and `B` is of length `lenB`. Sets `BQ` to the low `lenB - 1` coefficients of `B * Q`. We require that `Q` have space for `lenB` coefficients, that `W` be temporary space of size `lenB - 1` and `V` be temporary space for a number of coefficients computed by `NMOD_DIVREM_DC_ITCH(lenB, mod)`.

```
void _nmod_poly_divrem_divconquer(mp_ptr Q, mp_ptr R,
    mp_srcptr A, long lenA, mp_srcptr B, long lenB, nmod_t
    mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`. We require that `Q` have space for `lenA - lenB + 1` coefficients.

```
void nmod_poly_divrem_divconquer(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_divrem(mp_ptr Q, mp_ptr R, mp_srcptr A,
    long lenA, mp_srcptr B, long lenB, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`. We require that `Q` have space for `lenA - lenB + 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_divconquer_recursive(mp_ptr Q, mp_ptr
    W, mp_ptr V, mp_srcptr A, mp_srcptr B, long lenB, nmod_t
    mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `2 * lenB - 1` and `B` is of length `lenB`. We require that `Q` have space for `lenB` coefficients and that `W` be temporary space of size `lenB - 1` and `V` be temporary space for a number of coefficients computed by `NMOD_DIV_DC_ITCH(lenB, mod)`.

```
void _nmod_poly_div_divconquer(mp_ptr Q, mp_srcptr A, long
    lenA, mp_srcptr B, long lenB, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but returns only `Q`. We require that `Q` have space for `lenA - lenB + 1` coefficients.

```
void nmod_poly_div_divconquer(nmod_poly_t Q, const
    nmod_poly_t A, const nmod_poly_t B)
```

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q.

```
void _nmod_poly_div(mp_ptr Q, mp_srcptr A, long lenA,
    mp_srcptr B, long lenB, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\mathrm{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but returns only `Q`. We require that `Q` have space for `lenA - lenB + 1` coefficients.

```
void nmod_poly_div(nmod_poly_t Q, const nmod_poly_t A,
    const nmod_poly_t B)
```

Notionally computes Q and R such that $A = BQ + R$ with $\mathrm{len}(R) < \mathrm{len}(B)$, but returns only Q.

```
void _nmod_poly_inv_series_basecase(mp_ptr Qinv, mp_srcptr
    Q, long n, nmod_t mod)
```

Given `Q` of length `n` 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 * 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, long n)
```

Given `Q` of length at least `n` find `Qinv` of length `n` such that the top `n` coefficients of the product `Q * 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_srcptr Q,
    long n, nmod_t mod)
```

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`. 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, long n)
```

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`. This function can be viewed as inverting a power series via Newton iteration.

```
void _nmod_poly_inv_series(mp_ptr Qinv, mp_srcptr Q, long
    n, nmod_t mod)
```

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`. This function can be viewed as inverting a power series.

```
void nmod_poly_inv_series(nmod_poly_t Qinv, const
    nmod_poly_t Q, long n)
```

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`. This function can be viewed as inverting a power series.

```
void _nmod_poly_div_series(mp_ptr Q, mp_srcptr A, mp_srcptr
    B, long n, nmod_t mod)
```

Given polynomials `A` and `B` of length `n`, finds the polynomial `Q` of length `n` such that `Q * 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.

```
void nmod_poly_div_series(nmod_poly_t Q, const nmod_poly_t
    A, const nmod_poly_t B, long n)
```

Given polynomials A and B considered modulo n, finds the polynomial Q of length at most n such that Q * B = A modulo x^n. We assume n > 0 and that the constant coefficient of B is invertible modulo the modulus. An exception is raised if n == 0 or the constant coefficient of B is zero.

```
void _nmod_poly_div_newton(mp_ptr Q, mp_srcptr A, long
    Alen, mp_srcptr B, long Blen, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than lenB, 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. The algorithm used is to reverse the polynomials and divde the resulting power series, then reverse the result.

```
void nmod_poly_div_newton(nmod_poly_t Q, const nmod_poly_t
    A, const nmod_poly_t B)
```

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q. The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void _nmod_poly_divrem_newton(mp_ptr Q, mp_ptr R, mp_srcptr
    A, long Alen, mp_srcptr B, long Blen, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than lenB, where A is of length lenA and B is of length lenB. We require that Q have space for lenA - lenB + 1 coefficients. The algorithm used is to call div_newton() and then multiply out and compute the remainder.

```
void nmod_poly_divrem_newton(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 $\text{len}(R) < \text{len}(B)$. The algorithm used is to call div_newton() and then multiply out and compute the remainder.

18.18 Derivative and integral

```
void _nmod_poly_derivative(mp_ptr x_prime, mp_srcptr x,
    long len, nmod_t mod)
```

Sets the first len - 1 coefficients of x_prime to the derivative of x which is assumed to be of length len. It is assumed that len > 0.

```
void nmod_poly_derivative(nmod_poly_t x_prime, const
    nmod_poly_t x)
```

Sets x_prime to the derivative of x.

```
void _nmod_poly_integral(mp_ptr x_int, mp_srcptr x, long
    len, nmod_t mod)
```

Set the first len coefficients of x_int to the integral of x which is assumed to be of length len - 1. The constant term of x_int is set to zero. It is assumed that len > 0. The result is only well-defined if the modulus is a prime number strictly larger than the degree of x.

```
void nmod_poly_integral(nmod_poly_t x_int, const
    nmod_poly_t x)
```

Set x_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.

18.19 Evaluation

```
mp_limb_t _nmod_poly_evaluate_nmod(mp_srcptr poly, long
    len, mp_limb_t c, nmod_t mod)
```

Evaluates poly at the value c and reduces modulo the given modulus of poly. The value c should be reduced modulo the modulus. The algorithm used is Horner's method.

```
mp_limb_t nmod_poly_evaluate_nmod(nmod_poly_t poly,
    mp_limb_t c)
```

Evaluates poly at the value c and reduces modulo the modulus of poly. The value c should be reduced modulo the modulus. The algorithm used is Horner's method.

18.20 Composition

```
void _nmod_poly_compose_horner(mp_ptr res, mp_srcptr poly1,
    long len1, mp_srcptr poly2, long 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, long len1, mp_srcptr poly2, long 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, long
    len1, mp_srcptr poly2, long 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`.

18.21 GCD

```
long _nmod_poly_gcd_euclidean(mp_ptr G, mp_srcptr A, long
    lenA, mp_srcptr B, long lenB, nmod_t mod)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= 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 `lenB` coefficients.

```
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.

```
long _nmod_poly_gcd(mp_ptr G, mp_srcptr A, long lenA,
    mp_srcptr B, long lenB, nmod_t mod)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= 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 `lenB` coefficients.

```
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.

```
long _nmod_poly_xgcd_euclidean(mp_ptr G, mp_ptr S, mp_ptr
    T, mp_srcptr A, long A_len, mp_srcptr B, long B_len,
    nmod_t mod)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= 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 `lenB` coefficients.

The polynomials `S` and `T` are set such that `S*A + B*T = G`. The length of `S` will be `lenB` and the length of `T` will be `lenA` (both zero padded if required).

No aliasing of input and output operands is permitted.

```
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 `lenB` and the length of `T` will be at most `lenA`.

```
long _nmod_poly_xgcd(mp_ptr G, mp_ptr S, mp_ptr T,
    mp_srcptr A, long A_len, mp_srcptr B, long B_len, nmod_t
    mod)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= 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 `lenB` coefficients.

Polynomials `S` and `T` are also computed such that `S*A + B*T = G`. The length of `S` will be `lenB` and the length of `T` will be `lenA` (both zero padded if required).

No aliasing of input and output operands is permitted.

```
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 `lenB` and the length of `T` will be at most `lenA`.

18.22 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, long
    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 and that h is zero-padded as necessary to length n. Aliasing is not permitted.

```
void nmod_poly_invsqrt_series(nmod_poly_t g, const
    nmod_poly_t h, long 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, long 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 and that h is zero-padded as necessary to length n. Aliasing is not permitted.

```
void nmod_poly_sqrt_series(nmod_poly_t g, const nmod_poly_t
    h, long n)
```

Set g to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

18.23 Transcendental functions

The elementary transcendental functions of a formal power series h are defined as

$$\exp(h(x)) = \sum_{k=0}^{\infty} \frac{(h(x))^k}{k!}$$

$$\log(h(x)) = \int_0^x \frac{h'(t)}{h(t)} dt$$

$$\operatorname{atan}(h(x)) = \int_0^x \frac{h'(t)}{1 + (h(t))^2} dt$$

$$\operatorname{atanh}(h(x)) = \int_0^x \frac{h'(t)}{1 - (h(t))^2} dt$$

$$\operatorname{asin}(h(x)) = \int_0^x \frac{h'(t)}{\sqrt{1 - (h(t))^2}} dt$$

$$\operatorname{asinh}(h(x)) = \int_0^x \frac{h'(t)}{\sqrt{1 + (h(t))^2}} dt$$

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_monomial_ui(mp_ptr g, mp_limb_t
    c, ulong r, long n, nmod_t mod)
```

Set $g = \log(1 + cx^r) + O(x^n)$. Assumes $n > 0$, $r > 0$, and that the coefficient is reduced by the modulus. Works efficiently in linear time.

```
void nmod_poly_log_series_monomial_ui(nmod_poly_t g,
    mp_limb_t c, ulong r, long n)
```

Set $g = \log(1 + cx^r) + O(x^n)$. Works efficiently in linear time.

```
void _nmod_poly_log_series(mp_ptr g, mp_srcptr h, long n,
    nmod_t mod)
```

Set $g = \log(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_log_series(nmod_poly_t g, const nmod_poly_t
    h, long 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_monomial_ui(mp_ptr g, mp_limb_t
    c, ulong r, long n, nmod_t mod)
```

Set $g = \exp(cx^r) + O(x^n)$. Assumes $n > 0$, $r > 0$, and that the coefficient is reduced by the modulus. Works efficiently in linear time.

```
void nmod_poly_exp_series_monomial_ui(nmod_poly_t g,
    mp_limb_t c, ulong r, long n)
```

Set $g = \exp(cx^r) + O(x^n)$. Works efficiently in linear time.

```
void _nmod_poly_exp_series_basecase(mp_ptr g, mp_srcptr h,
    long hlen, long n, nmod_t mod)
```

Set $g = \exp(h) + O(x^n)$ using a simple $O(n^2)$ algorithm. Assumes $n > 0$ and $\text{hlen} > 0$. Only the first hlen coefficients of h will be read. Aliasing of f and h is allowed.

```
void nmod_poly_exp_series_basecase(nmod_poly_t g, const
    nmod_poly_t h, long n)
```

Set $g = \exp(h) + O(x^n)$ using a simple $O(n^2)$ algorithm.

```
void _nmod_poly_exp_series(mp_ptr g, mp_srcptr h, long n,
    nmod_t mod)
```

Set $g = \exp(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is not allowed.

Uses Newton iteration (the version given in [11]). For small n, falls back to the basecase algorithm.

```
void nmod_poly_exp_series(nmod_poly_t g, const nmod_poly_t
    h, long 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 g, mp_srcptr h, long n,
    nmod_t mod)
```

Set $g = \operatorname{atan}(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_atan_series(nmod_poly_t g, const nmod_poly_t
    h, long n)
```

Set $g = \operatorname{atan}(h) + O(x^n)$.

```
void _nmod_poly_atanh_series(mp_ptr g, mp_srcptr h, long n,
    nmod_t mod)
```

Set $g = \operatorname{atanh}(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_atanh_series(nmod_poly_t g, const
    nmod_poly_t h, long n)
```

Set $g = \operatorname{atanh}(h) + O(x^n)$.

```
void _nmod_poly_asin_series(mp_ptr g, mp_srcptr h, long n,
    nmod_t mod)
```

Set $g = \operatorname{asin}(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_asin_series(nmod_poly_t g, const nmod_poly_t
    h, long n)
```

Set $g = \operatorname{asin}(h) + O(x^n)$.

```
void _nmod_poly_asinh_series(mp_ptr g, mp_srcptr h, long n,
    nmod_t mod)
```

Set $g = \operatorname{asinh}(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_asinh_series(nmod_poly_t g, const
    nmod_poly_t h, long n)
```

Set $g = \operatorname{asinh}(h) + O(x^n)$.

```
void _nmod_poly_sin_series(mp_ptr g, mp_srcptr h, long n,
    nmod_t mod)
```

Set $g = \sin(h) + O(x^n)$. 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, long n)
```

Set $g = \sin(h) + O(x^n)$.

```
void _nmod_poly_cos_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \cos(h) + O(x^n)$.

```
void _nmod_poly_tan_series(mp_ptr g, mp_srcptr h, long 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 not allowed. Uses Newton iteration to invert the atan function.

```
void nmod_poly_tan_series(nmod_poly_t g, const nmod_poly_t
    h, long n)
```

Set $g = \tan(h) + O(x^n)$.

```
void _nmod_poly_sinh_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \sinh(h) + O(x^n)$.

```
void _nmod_poly_cosh_series(mp_ptr g, mp_srcptr h, long 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 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, long n)
```

Set $g = \cosh(h) + O(x^n)$.

```
void _nmod_poly_tanh_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \tanh(h) + O(x^n)$.

§19. nmod_mat

Matrices over $\mathbf{Z}/n\mathbf{Z}$ for word-sized moduli

19.1 Introduction

An nmod_mat_t represents a matrix of integers modulo n, for any non-zero n that fits in a single limb, up to $2^{32} - 1$ or $2^{64} - 1$. The implementation uses functions and data types of the nmod_vec module for low-level operations.

One or both dimensions of a matrix may be zero.

Except where otherwise noted, it is assumed all entries in input data are already reduced in the range $[0, n)$. It is also assumed that all arguments have the same modulus.

Functions that require the modulus to be a prime number document this requirement explicitly.

19.2 Memory management

```
void nmod_mat_init(nmod_mat_t mat, long rows, long 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, 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, nmod_mat_t src)
```

Sets the elements of mat to those of src. It is assumed that mat and src have identical dimensions, but the moduli need not be the same. The elements will be reduced to the modulus of mat.

```
int nmod_mat_equal(nmod_mat_t mat1, nmod_mat_t mat2)
```

Returns nonzero if mat1 and mat2 have the same dimensions and elements, and zero otherwise. The moduli are ignored.

19.3 Printing

```
void nmod_mat_print_pretty(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:

```
<2 x 3 integer matrix mod 2903>
[   0    0 2607]
[ 622    0    0]
```

19.4 Random matrix generation

```
void nmod_mat_randtest(nmod_mat_t mat, flint_rand_t state)
```

Sets the elements to uniformly random numbers between 0 and $m-1$ inclusive, where m is the modulus of `mat`.

```
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, mp_limb_t * diag,
    long n, flint_rand_t state)
```

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, long rank,
    flint_rand_t state)
```

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(fmpz_mat_t mat, long 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.

19.5 Matrix arithmetic

```
void nmod_mat_transpose(nmod_mat_t B, 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. Entries will be reduced to the modulus of B.

```
void nmod_mat_add(nmod_mat_t C, nmod_mat_t A, nmod_mat_t B)
```

Computes $C = A + B$. Dimensions must be identical. Entries will be reduced to the modulus of C.

```
void nmod_mat_mul(nmod_mat_t C, nmod_mat_t A, nmod_mat_t B)
```

Computes the matrix product $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Entries will be reduced to the modulus of C.

```
void nmod_mat_mul_classical(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

Performs matrix multiplication using the classical algorithm, by calling `_nmod_mat_mul_1()` etc. The interface is identical to `nmod_mat_mul()`.

```
void nmod_mat_mul_strassen(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

Performs matrix multiplication respectively using Strassen multiplication. The interface is identical to `nmod_mat_mul()`.

```
void _nmod_mat_mul_1(nmod_mat_t C, nmod_mat_t A, nmod_mat_t
    B)
```

```
void _nmod_mat_mul_2(nmod_mat_t C, nmod_mat_t A, nmod_mat_t
    B)
```

```
void _nmod_mat_mul_3(nmod_mat_t C, nmod_mat_t A, nmod_mat_t
    B)
```

```
void _nmod_mat_mul_transpose_1(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

```
void _nmod_mat_mul_transpose_2(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

```
void _nmod_mat_mul_transpose_3(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

Computes $C = AB$ or $C = AB^T$, respectively, using classical matrix multiplication. That is, for each i and j, C_{ij} is set to the scalar product of row i of A with column j of B (or row j of B when transposed).

The `_X` version uses a register `X` limbs wide to accumulate the scalar product, and a modular reduction is performed only at the end. The caller must ensure that elements are sufficiently small for overflow not to occur.

Let k be the scalar product size, i.e. the number of columns of A, and let a and b the largest absolute values of the elements in A and B, respectively. Then the following conditions are sufficient to guarantee correctness:

```
1  :  k*a*b  <  2^FLINT_BITS
2  :  k*a*b  <  2^(2*FLINT_BITS)
3  :  always
```

Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B.

19.6 Row reduction, solving

```
long _nmod_mat_rowreduce(nmod_mat_t mat, int options)
```

Row reduces the m-by-n matrix `mat` using standard in-place Gaussian elimination with pivoting. The modulus `mat->mod.n` must be a prime number.

The `options` parameter is a bitfield which may be set to any combination of the following flags, where using `options = 0` to disable all and perform in-place LU decomposition.

- `ROWREDUCE_FAST_ABORT`.

  If set, the function immediately aborts and returns 0 if the matrix is detected to be rank-deficient, i.e. singular. In this event, the state of the matrix will be undefined.

- `ROWREDUCE_FULL`.

  If set, performs Gauss–Jordan elimination, i.e. eliminates the elements above each pivot element as well as those below. If not set, regular Gaussian elimination is performed and only the elements below pivots are eliminated.

- `ROWREDUCE_CLEAR_LOWER`.

  If set, clears, i.e. zeros, elements below the pivots.

  If not set, the output becomes the LU decomposition of the input matrix. That is, the input matrix A is overwritten with L, U such that $A = PLU$ where P is a permutation matrix. U is stored in the upper triangular part (including the main diagonal), and L is stored with an implicit unit main diagonal in the lower triangular part.

Pivoting (to avoid division by zero entries) is performed by permuting the vector of row pointers in-place so that they point to the successive pivot rows. The matrix entries themselves retain their original order in memory.

The return value r is the rank of the matrix, multiplied by a sign indicating the parity of row interchanges. If $r = 0$, the matrix has rank zero, unless `ROWREDUCE_FAST_ABORT` is set, in which case $r = 0$ indicates any deficient rank. Otherwise, the leading non-zero entries of $a[0], a[1], \ldots, a[|(|r) - 1]$ will point to the successive pivot elements. If $|r| = m = n$, the determinant of the matrix is given by $\operatorname{sgn}(r)$ times the product of the entries on the main diagonal.

```
mp_limb_t nmod_mat_det(nmod_mat_t A)
```

Returns the determinant of A. The modulus of A must be a prime number.

```
long nmod_mat_rank(nmod_mat_t A)
```

Returns the rank of A. The modulus of A must be a prime number.

```
void _nmod_mat_solve_lu_precomp(mp_limb_t * b, mp_limb_t **
    LU, long n, nmod_t mod)
```

Transforms b to the solution x of $LUx = b$ where `LU` points to the rows of a precomputed LU factorisation of a nonsingular n-by-n matrix with modulus `mod.n`. The modulus must be a prime number.

```
int nmod_mat_solve(mp_limb_t * x, nmod_mat_t A, mp_limb_t *
    b)
```

Solves the matrix-vector equation $Ax = b$ over $\mathbf{Z}/p\mathbf{Z}$ where p is the modulus of A which must be a prime number.

Returns 0 if A has full rank; otherwise returns 1 and sets the elements of x to undefined values.

```
int nmod_mat_solve_mat(nmod_mat_t X, nmod_mat_t A,
    nmod_mat_t B)
```

Solves the matrix-matrix equation $AX = B$ over $\mathbf{Z}/p\mathbf{Z}$ where p is the modulus of X which must be a prime number. X, A, and B should have the same moduli.

Returns 0 if A has full rank; otherwise returns 1 and sets the elements of X to undefined values.

```
int nmod_mat_inv(nmod_mat_t B, 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.

§20. padic

p-Adic numbers in $\mathbf{Q}_p$

20.1 Introduction

The `padic_t` data type represents elements of $\mathbf{Q}_p$, stored in the form $x = p^v u$ with $u, v \in \mathbf{Z}$. Arithmetic operations can be carried out with respect to a context containing the prime number p and precision N.

Independent of the context, we consider a p-adic number $x = up^v$ to be in *canonical form* whenever either $p \nmid u$ or $u = v = 0$.

With a given context, i.e. a prime p and a precision N, in mind, we say a p-adic number $x = up^v$ is *reduced* if either $u = v = 0$ or $p \nmid u$ and $u \in (0, p^N)$.

The main idea behind the treatment of the precision is that where possible p-adic numbers that are input arguments to a function are interpreted as exact p-adic numbers and the precision of the context object is only used as the precision to which the output is to be computed.

20.2 Data structures

```
fmpz * 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 `fmpz` functions.

Note that this function is implemented as a macro.

```
long 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 *lvalue* and an *rvalue*.

20.3 Context

At the bare minimum, a context object for p-adic arithmetic contains the prime number p, the precision N and the printing mode.

In addition, various other useful objects may be stored in the context, such as a precomputed `double` inverse of the prime p or various powers of p near p^N.

```
void padic_ctx_init(padic_ctx_t ctx, const fmpz_t p, long
    N, enum padic_print_mode mode)
```

Initialises the context `ctx` with prime p, precision N, and printing mode.

Assumes that p is a prime and that printing mode is one of `PADIC_TERSE`, `PADIC_SERIES`, or `PADIC_VAL_UNIT`.

```
void padic_ctx_clear(padic_ctx_t ctx)
```

Clears all memory that has been allocated as part of the context.

```
void _padic_ctx_pow_ui(fmpz *rop, int *alloc, ulong e,
    const padic_ctx_t ctx)
```

Sets `*rop` to p^e as efficiently as possible. If `*alloc` is set to a non-zero value on return then it is the responsibility of the caller to clear the returned integer.

N.B. Expects `rop` to be an uninitialised `fmpz_t`.

20.4 Memory management

```
void padic_init(padic_t rop, const padic_ctx_t ctx)
```

Initialises the p-adic number `rop`.

```
void padic_clear(padic_t rop, const padic_ctx_t ctx)
```

Clears all memory used by the p-adic number `rop`.

```
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 $p \nmid u$. This operation does not reduce the precision of the number.

```
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.

```
void padic_reduce(padic_t rop, const padic_ctx_t ctx)
```

Ensures that the p-adic number `rop` is reduced with respect to the given context.

That is to say, ensures that $v < N$, and that $0 \leq u < p^{N-v}$, and that u, if non-zero, is not divisible by p. If $v \geq N$, sets $u = 0$, $v = 0$. If $u = 0$, sets $v = 0$. Thus, the unique value of zero is $(u, v) = (0, 0)$.

20.5 Randomisation

```
void padic_randtest(padic_t rop, flint_rand_t state, const
    padic_ctx_t ctx)
```

Sets `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.

```
void padic_randtest_not_zero(padic_t rop, flint_rand_t
    state, const padic_ctx_t ctx)
```

Sets `rop` to a random non-zero p-adic number modulo p^N, where the range of the valuation is as for the function `padic_randtest()`.

20.6 Assignment

```
void _padic_set(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets `rop` to a copy of `op`.

N.B. No reduction takes place.

```
void padic_set(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets `rop` to the value of `op` reduced modulo p^N.

```
void _padic_set_si(padic_t rop, long op, const padic_ctx_t
    ctx)
```

Sets the p-adic number `rop` to the `long` integer `op`.

```
void padic_set_si(padic_t rop, long op, const padic_ctx_t
    ctx)
```

Sets the p-adic number `rop` to the `long` integer `op` reduced modulo p^N.

```
void _padic_set_ui(padic_t rop, ulong op, const padic_ctx_t
    ctx)
```

Sets the p-adic number `rop` to the `unsigned long` integer `op`.

```
void padic_set_ui(padic_t rop, ulong op, const padic_ctx_t
    ctx)
```

Sets the p-adic number `rop` to the `unsigned long` integer `op` reduced modulo p^N.

```
void _padic_set_fmpz(padic_t rop, const fmpz_t op, const
    padic_ctx_t ctx)
```

Sets the p-adic number `rop` to the integer `op`.

```
void padic_set_fmpz(padic_t rop, const fmpz_t op, const
    padic_ctx_t ctx)
```

Sets the p-adic number `rop` to the integer `op` reduced modulo p^N.

```
void padic_set_fmpq(padic_t rop, const fmpq_t op, const
    padic_ctx_t ctx)
```

Sets `rop` to the rational `op` reduced modulo p^N.

```
void _padic_set_mpz(padic_t rop, const mpz_t op, const
    padic_ctx_t ctx)
```

Sets the p-adic number `rop` to the MPIR integer `op`.

The unit part is reduced modulo p^N if and only if `op` is negative.

```
void padic_set_mpz(padic_t rop, const mpz_t op, const
    padic_ctx_t ctx)
```

Sets the p-adic number `rop` to the MPIR integer `op` reduced modulo p^N.

```
void padic_set_mpq(padic_t rop, const mpq_t op, const
    padic_ctx_t ctx)
```

Sets `rop` to the MPIR rational `op` reduced modulo p^N.

```
void _padic_get_fmpz(fmpz_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets the integer `rop` to the p-adic integer `op`.

If `op` is not a p-adic integer, sets `rop` to zero.

N.B. No reduction takes place.

```
void padic_get_fmpz(fmpz_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets the integer `rop` to the p-adic integer `op` reduced modulo p^N.

If `op` is not a p-adic integer, sets `rop` to zero.

```
void _padic_get_fmpq(fmpq_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets the rational `rop` to the p-adic integer `op`.

N.B. No reduction takes place.

```
void padic_get_fmpq(fmpq_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets the rational `rop` to the p-adic integer `op` reduced modulo p^N.

```
void _padic_get_mpz(mpz_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets the MPIR integer `rop` to the p-adic integer `op`.

If `op` is not a p-adic integer, sets `rop` to zero.

N.B. No reduction takes place.

```
void padic_get_mpz(mpz_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets the MPIR integer `rop` to the p-adic integer `op`, reduced modulo p^N.

If `op` is not a p-adic integer, sets `rop` to zero.

```
void _padic_get_mpq(mpq_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets the MPIR rational `rop` to the value of `op`.

N.B. No reduction takes place.

```
void padic_get_mpq(mpq_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets the MPIR rational `rop` to the value of `op`, reduced modulo p^N.

```
void padic_swap(padic_t op1, padic_t op2, const padic_ctx_t
    ctx)
```

Swaps the two p-adic numbers op1 and op2.

N.B. No reduction takes place.

```
void _padic_zero(padic_t rop)
```

Sets the p-adic number rop to zero.

```
void padic_zero(padic_t rop, const padic_ctx_t ctx)
```

Sets the p-adic number rop to zero.

```
void _padic_one(padic_t rop)
```

Sets the p-adic number rop to one.

```
void padic_one(padic_t rop, const padic_ctx_t ctx)
```

Sets the p-adic number rop to one, reduced modulo p^N.

20.7 Arithmetic operations

```
void _padic_add(padic_t rop, const padic_t op1, const
    padic_t op2, const padic_ctx_t ctx)
```

Sets rop to the sum of op1 and op2.

```
void padic_add(padic_t rop, const padic_t op1, const
    padic_t op2, const padic_ctx_t ctx)
```

Sets rop to the sum of op1 and op2, reduced modulo p^N.

```
void _padic_sub(padic_t rop, const padic_t op1, const
    padic_t op2, const padic_ctx_t ctx)
```

Sets rop to the difference of op1 and op2.

```
void padic_sub(padic_t rop, const padic_t op1, const
    padic_t op2, const padic_ctx_t ctx)
```

Sets rop to the difference of op1 and op2, reduced modulo p^N.

```
void _padic_neg(padic_t rop, const padic_t op)
```

Sets rop to the additive inverse of op.

```
void padic_neg(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Sets rop to the additive inverse of op, reduced modulo p^N.

```
void _padic_mul(padic_t rop, const padic_t op1, const
    padic_t op2)
```

Sets rop to the product of op1 and op2.

```
void padic_mul(padic_t rop, const padic_t op1, const
    padic_t op2, const padic_ctx_t ctx)
```

Sets rop to the product of op1 and op2, reduced modulo p^N.

```
void padic_shift(padic_t rop, const padic_t op, long v,
    const padic_ctx_t ctx)
```

Sets rop to the product of op and p^v, reduced modulo p^N.

```
void padic_div(padic_t rop, const padic_t op1, const
    padic_t op2, const padic_ctx_t ctx)
```

Sets rop to the quotient of op1 and op2, reduced modulo p^N.

```
void _padic_inv(fmpz_t rop, const fmpz_t op, const fmpz_t
    p, long N)
```

Sets rop to the inverse of op modulo p^N, assuming that op is a unit and $N \geq 1$.

In the current implementation, allows aliasing, but this might change in future versions.

```
void padic_inv(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Computes the inverse of op modulo p^N.

Suppose that op is given as $x = up^v$. Raises an abort signal if $v < -N$. Otherwise, computes the inverse of u modulo p^{N+v}.

```
void _padic_inv_naive(fmpz_t rop, const fmpz_t op, const
    fmpz_t p, long N)
```

Sets rop to the inverse of op modulo p^N, assuming that op is a unit and $N \geq 1$.

In the current implementation, allows aliasing, but this might change in future versions.

```
void padic_inv_naive(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Computes the inverse of op modulo p^N.

This function naively refers to the function fmpz_invmod, which works for any ring $\mathbf{Z}/m\mathbf{Z}$.

```
void _padic_inv_hensel(fmpz_t rop, const fmpz_t op, const
    fmpz_t p, long N)
```

Sets rop to the inverse of op modulo p^N, assuming that op is a unit and $N \geq 1$.

In the current implementation, allows aliasing, but this might change in future versions.

```
void padic_inv_hensel(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Computes the inverse of op modulo p^N.

This function employs Hensel lifting of an inverse modulo p.

```
int padic_sqrt(padic_rop, const padic_t op, const
    padic_ctx_t ctx)
```

Returns whether op is a p-adic square. If this is the case, sets rop to one of the square roots; otherwise, the value of rop is undefined.

We have the following theorem:

Let $u \in \mathbf{Z}^{\times}$. Then u is a square if and only if u mod p is a square in $\mathbf{Z}/p\mathbf{Z}$, for $p > 2$, or if u mod 8 is a square in $\mathbf{Z}/8\mathbf{Z}$, for $p = 2$.

```
void padic_pow_si(padic_t rop, const padic_t op, long e,
    const padic_ctx_t ctx)
```

Sets `rop` to `op` raised to the power e.

Assumes that some computations involving e and the valuation of `op` do not overflow in the `long` range.

20.8 Comparison

```
int _padic_is_zero(const padic_t op, const padic_ctx_t ctx)
```

Returns whether `op` is zero.

```
int padic_is_zero(const padic_t op, const padic_ctx_t ctx)
```

Returns whether `op` is zero modulo p^N.

```
int _padic_is_one(const padic_t op)
```

Returns whether `op` is one.

```
int padic_is_one(const padic_t op, const padic_ctx_t ctx)
```

Returns whether `op` is one modulo p^N.

```
int _padic_equal(const padic_t op1, const padic_t op2)
```

Returns whether `op1` and `op2` are equal.

```
int padic_equal(const padic_t op1, const padic_t op2, const
    padic_ctx_t ctx)
```

Returns whether `op1` and `op2` are equal modulo p^N.

20.9 Special functions

```
void padic_teichmuller(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Computes the Teichmuller 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.

```
int padic_exp(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Returns whether the p-adic exponential function converges at the p-adic number `op`, and if so sets `rop` 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 $\mathrm{ord}_p(x) > 1/(p-1)$. For elements $x \in \mathbf{Q}_p$, this means that $\mathrm{ord}_p(x) \geq 1$ when $p \geq 3$ and $\mathrm{ord}_2(x) \geq 2$ when $p = 2$.

```
void padic_val_fac(fmpz_t rop, const fmpz_t n, const fmpz_t
    p)
```

Computes the p-adic valuation of $n!$, assuming that $n > 0$.

20.10 Input and output

```
char * padic_get_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.

```
int padic_fprint(FILE * file, const padic_t op, const
    padic_ctx_t ctx)
```

Prints the string representation of the p-adic number `op` to the stream `file`.

In the current implementation, always returns 1.

```
int padic_print(const padic_t op, const padic_ctx_t ctx)
```

Prints the string representation of the p-adic number `op` to the stream `stdout`.

In the current implementation, always returns 1.

```
void padic_debug(const padic_t op, const padic_ctx_t ctx)
```

Prints debug information about `op` to the stream `stdout`.

§21. arith

Arithmetic functions

21.1 Introduction

This module implements arithmetic functions, number-theoretic and combinatorial special number sequences and polynomials.

21.2 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(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.

This currently only uses trial division, falling back to `n_factor()` as soon as the number shrinks to a single limb.

```
void fmpz_unfactor(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.

21.3 Primorials

```
void fmpz_primorial(fmpz_t res, long n)
```

Sets `res` to "n primorial" or $n\#$, the product of all prime numbers less than or equal to n.

21.4 Harmonic numbers

```
void _harmonic_number(fmpz_t num, fmpz_t den, long n)
```

Sets `(num, den)` to the reduced numerator and denominator of the n-th harmonic number $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$. The result is zero if $n \le 0$.

Table lookup is used for H_n whose numerator and denominator fit in single limb. For larger n, the function `mpn_harmonic_odd_balanced()` is used.

```
void harmonic_number(fmpq_t x, long n)
```

Sets `x` to the n-th harmonic number. This function is equivalent to `_harmonic_number` apart from the output being a single `fmpq_t` variable.

21.5 Stirling numbers

```
void stirling_number_1u(fmpz_t s, long n, long k)
```

```
void stirling_number_1(fmpz_t s, long n, long k)
```

```
void stirling_number_2(fmpz_t s, long n, long 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 stirling_number_1u(fmpz * row, long n, long klen)
```

```
void stirling_number_1(fmpz * row, long n, long klen)
```

```
void stirling_number_2(fmpz * row, long n, long klen)
```

Computes the row of Stirling numbers `S(n,0), S(n,1), S(n,2), ..., 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 stirling_number_1u_vec_next(fmpz * row, fmpz * prev,
    long n, long klen)
```

```
void stirling_number_1_vec_next(fmpz * row, fmpz * prev,
    long n, long klen)
```

```
void stirling_number_2_vec_next(fmpz * row, fmpz * prev,
    long n, long klen)
```

Given the vector `prev` containing a row of Stirling numbers `S(n-1,0), S(n-1,1), S(n-1,2), ..., S(n-1,klen-2)`, computes and stores in the row argument `S(n,0), S(n,1), S(n,2), ..., S(n,klen-1)`. It is assumed that `klen` is at most $n+1$.

The `row` and `prev` arguments are permitted to be the same, meaning that the row will be updated in-place.

```
void stirling_number_1u_mat(fmpz_mat_t mat)
```

```
void stirling_number_1_mat(fmpz_mat_t mat)
```

```
void stirling_number_2_mat(fmpz_mat_t mat)
```

For an arbitrary m-by-n matrix, writes the truncation of the infinite Stirling number matrix

```
row 0   : S(0,0)
row 1   : S(1,0), S(1,1)
row 2   : S(2,0), S(2,1), S(2,2)
row 3   : S(3,0), S(3,1), S(3,2), S(3,3)
```

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.

21.6 Bell numbers

```
void bell_number(fmpz_t b, ulong n)
```

Sets `b` to the `n`:th 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.

This function uses a table lookup if B_n fits in a single word, and otherwise evaluates a precise truncation of the series $B_n = e^{-1} \sum_{k=0}^{\infty} \frac{k^n}{k!}$ using binary splitting.

```
void bell_number_vec(fmpz * b, long n)
```

Sets `b` to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive. Automatically switches between the `recursive` and `multi_mod` algorithms depending on the size of n.

```
double bell_number_size(ulong n)
```

Returns b such that $B_n < 2^{\lfloor b \rfloor}$, using the inequality

$$B_n < \left(\frac{0.792n}{\log(n+1)}\right)^n$$

which is given in [2].

```
void _bell_number_vec_recursive(fmpz * b, long n)
```

Sets `b` to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive. This function uses table lookup if B_{n-1} fits in a single word, and a simple triangular recurrence otherwise.

```
void _bell_number_vec_multi_mod(fmpz * b, long n)
```

Sets `b` to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive.

This function evaluates the Bell numbers modulo several limb-size primes using the exponential generating function

$$e^{e^x-1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} x^n$$

and `nmod_poly` arithmetic. A tight bound for the number of needed primes is computed using `bell_number_size`, and the final integer values are recovered using balanced CRT reconstruction.

21.7 Bernoulli numbers and polynomials

```
void _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. As presently implemented, this function simply calls `_bernoulli_number_zeta`.

```
void bernoulli_number(fmpq_t x, ulong n)
```

Sets `x` to the n-th Bernoulli number. This function is equivalent to `_bernoulli_number` apart from the output being a single `fmpq_t` variable.

```
void _bernoulli_number_vec(fmpz * num, fmpz * den, long n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of the Bernoulli numbers $B_0, B_1, B_2, \ldots, B_{n-1}$ inclusive. This function automatically chooses between the `recursive`, `zeta` and `multi_mod` algorithms according to the size of n.

```
void bernoulli_number_vec(fmpq * x, long n)
```

Sets the `x` to the vector of Bernoulli numbers $B_0, B_1, B_2, \ldots, B_{n-1}$ inclusive. This function is equivalent to `_bernoulli_number_vec` apart from the output being a single `fmpq` vector.

```
void 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 bernoulli_number_size(ulong n)
```

Returns b such that $|B_n| < 2^{\lfloor b \rfloor}$, using the inequality

$$|B_n| < \frac{4n!}{(2\pi)^n}$$

and $n! \le (n+1)^{n+1} e^{-n}$. No special treatment is given to odd n. Accuracy is not guaranteed if $n > 10^{14}$.

```
void bernoulli_polynomial(fmpq_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 `bernoulli_number_vec` and then rescales the coefficients efficiently.

```
void _bernoulli_number_zeta(fmpz_t num, fmpz_t den, ulong n)
```

Sets `(num, den)` to the reduced numerator and denominator of the n-th Bernoulli number.

This function first computes the exact denominator and a bound for the size of the numerator. It then computes an approximation of $|B_n| = 2n!\zeta(n)/(2\pi)^n$ as a floating-point number and multiplies by the denominator to to obtain a real number that rounds to the exact numerator. For tiny n, the numerator is looked up from a table to avoid unnecessary overhead.

```
void _bernoulli_number_vec_recursive(fmpz * num, fmpz *
    den, long n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of $B_0, B_1, B_2, \ldots, B_{n-1}$ inclusive.

The first few entries are computed using `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 _bernoulli_number_vec_zeta(fmpz * num, fmpz * den,
    long n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of $B_0, B_1, B_2, \ldots, B_{n-1}$ inclusive. Uses repeated direct calls to `_bernoulli_number_zeta`.

```
void _bernoulli_number_vec_multi_mod(fmpz * num, fmpz *
    den, long n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of $B_0, B_1, B_2, \ldots, B_{n-1}$ 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 [3], saves about half of the time compared to the usual generating function $x/(e^x - 1)$ since the odd terms vanish.

21.8 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

$$\frac{2e^{xt}}{e^t+1} = \sum_{n=0}^{\infty} \frac{E_n(x)}{n!} t^n.$$

```
void euler_number(fmpz_t res, ulong n)
```

Sets `res` to the Euler number E_n. Currently calls `_euler_number_zeta`.

```
void euler_number_vec(fmpz * res, long 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 `euler_number_size`, and the final integer values are recovered using balanced CRT reconstruction.

```
double euler_number_size(ulong n)
```

Returns b such that $|E_n| < 2^{\lfloor b \rfloor}$, 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 $n > 10^{14}$.

```
void euler_polynomial(fmpq_poly_t poly, ulong n)
```

Sets `poly` to the Euler polynomial $E_n(x)$. Uses the formula

$$E_n(x) = \frac{2}{n+1} \left(B_{n+1}(x) - 2^{n+1} B_{n+1}\left(\frac{x}{2}\right) \right),$$

with the Bernoulli polynomial $B_{n+1}(x)$ evaluated once using `bernoulli_polynomial` and then rescaled.

```
void _euler_number_zeta(fmpz_t res, ulong n)
```

Sets `res` to the Euler number E_n. For even n, this function uses the relation

$$|E_n| = \frac{2^{n+2} n!}{\pi^{n+1}} L(n+1)$$

where $L(n+1)$ denotes the Dirichlet L-function with character $\chi = \{0, 1, 0, -1\}$.

21.9 Legendre polynomials

```
void legendre_polynomial(fmpq_poly_t poly, ulong n)
```

Sets `poly` to the n-th Legendre polynomial

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} \left[\left(x^2 - 1 \right)^n \right].$$

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 $\gcd(n!, 2^n) = 2^{\lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \cdots}$.

21.10 Multiplicative functions

```
void fmpz_euler_phi(fmpz_t res, const fmpz_t n)
```

Sets `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.

```
int 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, $\mu(0) = 0$.

```
void fmpz_divisor_sigma(fmpz_t res, const fmpz_t n, ulong k)
```

Sets `res` to $\sigma_k(n)$, the sum of kth powers of all divisors of n.

```
void fmpz_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.

```
void fmpz_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 $f(q) = q \prod_{k \geq 1} \left(1 - q^k\right)^{24}$.

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 `fmpz_poly_ramanujan_tau()`. Thus the speed of `fmpz_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 `fmpz_poly_ramanujan_tau()` directly.

```
void fmpz_poly_ramanujan_tau(fmpz_poly_t res, long n)
```

Sets `res` to the polynomial with coefficients $\tau(0), \tau(1), \ldots, \tau(n-1)$, giving the initial n terms in the series expansion of $f(q) = q \prod_{k \geq 1} \left(1 - q^k\right)^{24}$.

The algorithm is borrowed from the `delta_qexp()` function in Sage, written by William Stein and David Harvey, and consists of expanding the series of the equivalent representation

$$f(q) = q \left(\sum_{k \geq 0} (-1)^k (2k+1) q^{k(k+1)/2} \right)^8.$$

The first squaring is done directly since the polynomial is very sparse at this point.

21.11 Swinnerton-Dyer polynomials

```
void swinnerton_dyer_polynomial(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.

21.12 Partition function

```
void partition_function_vec(fmpz * res, long len)
```

Computes first `len` values of the partition function $p(n)$ starting with $p(0)$. Uses inversion of Euler's pentagonal series.

```
void partition_function_nmod_vec(mp_ptr res, long len,
    nmod_t mod)
```

Computes first `len` values of the partition function $p(n)$ starting with $p(0)$, modulo the modulus defined by `mod`. Uses inversion of Euler's pentagonal series.

21.13 Landau's function

```
void landau_function_vec(fmpz * res, long len)
```

Computes the first `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 [6]. The running time is $O(n^{3/2}/\sqrt{\log n})$.

§22. ulong_extras

Unsigned single limb arithmetic

22.1 Introduction

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 MPIR. 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.

22.2 Simple example

The following example computes $ab \pmod{n}$ using a precomputed inverse, where $a = 12345678, b = 87654321$ and $n = 111111111$.

```
#include <stdio.h>
#include "ulong_extras.h"
...
mp_limb_t r, a, b, n, ninv;

a = 12345678UL;
b = 87654321UL;
n = 111111111UL;
ninv = n_preinvert_limb(n);
```

```
r = n_mulmod2_preinv(a, b, n, ninv);

printf("%lu*%lu mod %lu is %lu\n", a, b, n, r);
```

The output is:

```
12345678*87654321 mod 111111111 is 23456790
```

22.3 Random functions

```
void n_randinit(flint_rand_t state)
```

Initialise a random state for use in random functions. Currently this function does nothing, but must be used for compatibility with future versions of flint. In particular the random functions below are not implemented in a threadsafe manner.

```
void n_randinit(flint_rand_t state)
```

Release any memory used by a random state. Currently this function does nothing, but must be used for compatibility with future versions of flint.

```
mp_limb_t 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 $v_{i+1} = (v_i a + b) \bmod p_j$ for some initial seed v_0, randomly chosen values a and b and `p_0 = 4294967311 = nextprime(2^32)` on a 64-bit machine and `p_0 = nextprime(2^16)` on a 32-bit machine and `p_1 = nextprime(p_0)`.

```
mp_limb_t 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.

```
mp_limb_t n_randint(flint_rand_t state, mp_limb_t 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.

```
mp_limb_t 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 `LONG_MAX` is increased. This random function is mainly used for testing purposes. Warning: this function is not threadsafe and is for use in test code only. It should not be used in library code.

```
mp_limb_t n_randtest_not_zero(flint_rand_t state)
```

As for `n_randtest()`, but does not return 0. Warning: this function is not threadsafe and is for use in test code only. It should not be used in library code.

```
mp_limb_t n_randprime(flint_rand_t state, unsigned long
    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`.

```
mp_limb_t 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.

22.4 Basic arithmetic

```
mp_limb_t n_pow(mp_limb_t n, ulong exp)
```

Returns n^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.

22.5 Miscellaneous

```
ulong n_revbin(ulong in, ulong bits)
```

Returns the binary reverse of in, assuming it is the given number of bits long, e.g. n_revbin(10110, 6) will return 110100.

```
int n_sizeinbase(mp_limb_t 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 $n = 0$.

22.6 Basic arithmetic with precomputed inverses

```
mp_limb_t n_mod_precomp(mp_limb_t a, mp_limb_t n, double
    ninv)
```

Returns a mod n given a precomputed inverse of n computed by n_precompute_inverse(). We require $n < 2^{53}$ and n < 2^(FLINT_BITS-1) and $0 \le a < n^2$.

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 n, the binary representation of a is exact and the mantissa is less than 2, thus we see that m * ninv can be at most one out in the mantissa. We now truncate m * 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. But this is impossible, as $n < 2^{53}$. Thus the quotient we have computed is either exactly what we are after, or one too small.

```
mp_limb_t n_mod2_precomp(mp_limb_t a, mp_limb_t 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 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 have been either negative or positive, so the quotient we compute may be one out in either direction.

```
mp_limb_t n_mod2_preinv(mp_limb_t a, mp_limb_t n, mp_limb_t
    ninv)
```

Returns $a \bmod n$ given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a or on n.

The old version of this function was implemented simply by making use of `udiv_qrnnd_preinv()`.

The new version uses the new algorithm of Granlund and Möller [9]. First n is normalised and a shifted into two limbs to compensate. Then their algorithm is applied verbatim and the result shifted back.

```
mp_limb_t n_divrem2_precomp(mp_limb_t * q, mp_limb_t a,
    mp_limb_t n, double npre)
```

Returns $a \bmod n$ given a precomputed inverse of n computed by `n_precompute_inverse()` and sets q to the quotient. There are no restrictions on a or on n.

This is as for `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.

```
mp_limb_t n_ll_mod_preinv(mp_limb_t a_hi, mp_limb_t a_lo,
    mp_limb_t n, mp_limb_t ninv)
```

Returns $a \bmod n$ given a precomputed inverse of n computed by `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 `udiv_qrnnd_preinv()` could be used.

The new version reduces the top limb modulo n as per `n_mod2_preinv()` and then the algorithm of Granlund and Möller [9] is used again to reduce modulo n.

```
mp_limb_t n_lll_mod_preinv(mp_limb_t a_hi, mp_limb_t a_mi,
    mp_limb_t a_lo, mp_limb_t n, mp_limb_t ninv)
```

Returns $a \bmod n$, where a has three limbs `(a_hi, a_mi, a_lo)`, given a precomputed inverse of n computed by `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 [9] to first reduce the top two limbs modulo n, then does the same on the bottom two limbs.

```
mp_limb_t n_mulmod_precomp(mp_limb_t a, mp_limb_t b,
    mp_limb_t n, double ninv)
```

Returns $ab \bmod n$ given a precomputed inverse of n computed by `n_precompute_inverse()`. We require $n < 2^{53}$ and $0 \le a, b < n$.

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. The product of a and b is computed with error at most half a place. When `a * b` is multiplied by n we find that the exact quotient and computed quotient differ by less than two places. As the quotient is less than 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 `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.

```
n_mulmod2_preinv(mp_limb_t a, mp_limb_t b, mp_limb_t n,
    mp_limb_t ninv)
```

Returns $ab \bmod n$ given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a, b or on n. This is implemented by multiplying using `umul_ppmm()` and then reducing using `n_ll_mod_preinv()`.

22.7 Greatest common divisor

```
mp_limb_t n_gcd(mp_limb_t x, mp_limb_t y)
```

Returns the greatest common divisor g of x and y. We require $x \geq y$.

The algorithm is a slight embelishment of the Euclidean algorithm which uses some branches to avoid most divisions.

One wishes to compute the quotient and remainder of u_3/v_3 without division where possible. This is accomplished when $u_3 < 4v_3$, i.e. the quotient is either 1, 2 or 3.

We first compute $s = u_3 - v_3$. If $s < v_3$, i.e. $u_3 < 2v_3$, we know the quotient is 1, else if $s < 2v_3$, i.e. $u_3 < 3v_3$ we know the quotient is 2. In the remaining cases, the quotient must be 3. When the quotient is 4 or above, we use division. However this happens rarely for generic inputs.

```
mp_limb_t n_gcdinv(mp_limb_t * a, mp_limb_t x, mp_limb_t y)
```

Returns the greatest common divisor g of x and y and computes a such that $0 \leq a < y$ and $ax = \gcd(x, y) \bmod y$, when this is defined. We require $0 \leq x < y$.

This is merely an adaption of the extended Euclidean algorithm with appropriate normalisation.

```
mp_limb_t n_xgcd(mp_limb_t * a, mp_limb_t * b, mp_limb_t x,
    mp_limb_t y)
```

Returns the greatest common divisor g of x and y and unsigned values a and b such that $ax - by = g$. We require $x \geq y$.

We claim that computing the extended greatest common divisor via the Euclidean algorithm always results in cofactor $|a| < x/2$, $|b| < x/2$, with perhaps some small degenerate exceptions.

We proceed by induction.

Suppose we are at some step of the algorithm, with $x_n = qy_n + r$ with $r \geq 1$, and suppose $1 = sy_n - tr$ with $s < y_n/2$, $t < y_n/2$ by hypothesis.

Write $1 = sy_n - t(x_n - qy_n) = (s + tq)y_n - tx_n$.

It suffices to show that $(s + tq) < x_n/2$ as $t < y_n/2 < x_n/2$, which will complete the induction step.

But at the previous step in the backsubstitution we would have had $1 = sr - cd$ with $s < r/2$ and $c < r/2$.

Then $s + tq < r/2 + y_n/2q = (r + qy_n)/2 = x_n/2$.

See the documentation of `n_gcd()` for a description of the branching in the algorithm, which is faster than using division.

22.8 Jacobi and Kronecker symbols

```
int n_jacobi(mp_limb_signed_t x, mp_limb_t y)
```

Computes the Jacobi symbol of $x \bmod y$. Assumes that y is positive and odd, and for performance reasons that $\gcd(x, y) = 1$.

This is just a straightforward application of the law of quadratic reciprocity. For performance, divisions are replaced with some comparisons and subtractions where possible.

22.9 Modular Arithmetic

```
mp_limb_t n_addmod(mp_limb_t a, mp_limb_t b, mp_limb_t n)
```

Returns $(a + b) \bmod n$.

```
mp_limb_t n_submod(mp_limb_t a, mp_limb_t b, mp_limb_t n)
```

Returns $(a - b) \bmod n$.

```
mp_limb_t n_invmod(mp_limb_t x, mp_limb_t y)
```

Returns a value a such that $0 \le a < y$ and $ax = \gcd(x, y) \bmod y$, when this is defined. We require $0 \le x < y$.

Specifically, when x is coprime to y, a is the inverse of x in $\mathbf{Z}/y\mathbf{Z}$.

This is merely an adaption of the extended Euclidean algorithm with appropriate normalisation.

```
mp_limb_t n_powmod_precomp(mp_limb_t a, mp_limb_signed_t
    exp, mp_limb_t n, double npre)
```

Returns `(a^exp) % n` given a precomputed inverse of n computed by `n_precompute_inverse()`. We require $n < 2^{53}$ and $0 \le a < n$. 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.

```
mp_limb_t n_powmod(mp_limb_t a, mp_limb_signed_t exp,
    mp_limb_t n)
```

Returns `(a^exp) % n`. We require `n < 2^FLINT_D_BITS` and $0 \le a < n$. There are no restrictions on `exp`, i.e. it can be negative.

This is implemented by precomputing an inverse and calling the `precomp` version of this function.

```
mp_limb_t n_powmod2_preinv(mp_limb_t a, mp_limb_signed_t
    exp, mp_limb_t n, mp_limb_t ninv)
```

Returns `(a^exp) % n` given a precomputed inverse of n computed by `n_preinvert_limb()`. We require $0 \le a < n$, 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.

```
mp_limb_t n_powmod2(mp_limb_t a, mp_limb_signed_t exp,
    mp_limb_t n)
```

Returns `(a^exp) % 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 `preinv` version of this function.

```
mp_limb_t n_sqrtmod(mp_limb_t a, mp_limb_t p)
```

Computes a square root of a modulo p.

Assumes that p is a prime and that a is reduced modulo p. Returns 0 if a is a quadratic non-residue modulo p.

22.10 Prime number generation and counting

```
void n_compute_primes(ulong num_primes)
```

Precomputes `num_primes` primes and their `double` precomputed inverses and stores them in `flint_primes` and `flint_prime_inverse`, respectively.

The algorithm is a simple sieve of Eratosthenes with the constant array of primes `flint_small_primes` as a starting point.

The sieve works by marking all multiples of small primes in the sieve, but the sieve does not contain entries for numbers below the current cutoff (in case the function may have already been called before).

One only needs to start sieving with p^2 as all smaller multiples of p have already been marked off.

At first p^2 may be less than the start of the sieve (the old cutoff), so this case is dealt with separately, but for all primes p beyond that all multiples of p starting at p^2 are marked off in the sieve.

As the small prime cutoff is currently 1030, primes can be computed up to almost $n = 2^{20}$, in fact $\lfloor n/\log_2(n) \times 0.7 \rfloor = 74898$ primes which actually takes us to 949937.

```
mp_limb_t n_nextprime(mp_limb_t n, int proved)
```

Returns the next prime after n. Assumes the result will fit in an `mp_limb_t`. If proved is 0, i.e. false, the prime is not proven prime, otherwise it is.

```
ulong n_prime_pi(mp_limb_t 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_prime_pi(n_nth_prime(n))== n` holds, or `n_prime_pi(flint_primes[n-1])== n`, where `flint_primes` is indexed from zero.

Currently, this function simply extends `flint_primes` up to an upper limit and then performs a binary search.

```
void n_prime_pi_bounds(ulong *lo, ulong *hi, mp_limb_t n)
```

Calculates lower and upper bounds for the value of the prime counting function `lo <= pi(n)<= hi`. If `lo` and `hi` point to the same location, the high value will be stored.

The upper approximation is $1.25506n/\ln n$, and the lower is $n/\ln n$. These bounds are due to Rosser and Schoenfeld [20] 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.

```
mp_limb_t n_nth_prime(ulong n)
```

Returns the nth prime number p_n, using the mathematical indexing convention $p_1 = 2, p_2 = 3, \ldots$.

This function simply ensures that `flint_primes` is large enough and then looks up `flint_primes[n-1]`.

```
void n_nth_prime_bounds(mp_limb_t *lo, mp_limb_t *hi, ulong
    n)
```

Calculates lower and upper bounds for the nth 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 [7], and the last is due to Massias and Robin [17]. 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.

22.11 Primality testing

```
int n_is_oddprime_small(mp_limb_t 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(mp_limb_t n)
```

This function performs a simple binary search through `flint_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(mp_limb_t 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 [12] 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 `~0L` 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 http://mathworld.wolfram.com/PocklingtonsTheorem.html for a description of the algorithm.

```
int n_is_prime_pseudosquare(mp_limb_t n)
```

Tests if n is a prime according to [16, Theorem 2.7].

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 http://research.att.com/~njas/sequences/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.

```
int n_is_prime(mp_limb_t n)
```

Tests if n is a prime. Up to 10^{16} this simply calls `n_is_probabprime()` which is a primality test up to that limit. Beyond that point it calls `n_is_probabprime()` and returns 0 if n is composite, then it calls `n_is_prime_pocklington()` which proves the primality of n in most cases. As a fallback, `n_is_prime_pseudosquare()` is called, which will unconditionally prove the primality of n.

```
int n_is_strong_probabprime_precomp(mp_limb_t n, double
    npre, mp_limb_t a, mp_limb_t 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 $n < 2^{53}$, a to be reduced modulo n and not 0 and n to be odd.

If we write $n-1 = 2^s d$ 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: http://mathworld.wolfram.com/StrongPseudoprime.html

```
int n_is_strong_probabprime2_preinv(mp_limb_t n, mp_limb_t
    ninv, mp_limb_t a, mp_limb_t 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 not 0 and n to be odd.

If we write $n-1 = 2^s d$ 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: http://mathworld.wolfram.com/StrongPseudoprime.html

```
int n_is_probabprime_fermat(mp_limb_t n, mp_limb_t 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(mp_limb_t n)`

Let F_j be the jth element of the Fibonacci sequence $0, 1, 1, 2, 3, 5, \ldots$, 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 [5, pp. 142].

We require that n is not divisible by 2 or 5.

`int n_is_probabprime_BPSW(mp_limb_t n)`

Implements the Bailey–Pomerance–Selfridge–Wagstaff probable primality test. There are no known counterexamples to this being a primality test. For further details, see [5].

`int n_is_probabprime_lucas(mp_limb_t n)`

For details on Lucas pseudoprimes, see [5, pp. 143].

We implement a variant of the Lucas pseudoprime test as described by Baillie and Wagstaff [1].

`int n_is_probabprime(mp_limb_t 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 10^{16} it uses a number of strong probable prime tests, `n_is_strong_probabprime_precomp()`, 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`.

Beyond that point the BPSW probabilistic primality test is used, by calling the function `n_is_probabprime_BPSW()`. There are no known counterexamples, but it may well declare some composites to be prime.

22.12 Square root and perfect power testing

`mp_limb_t n_sqrt(mp_limb_t a)`

Computes the integer truncation of the square root of a. The integer itself can be represented exactly as a 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. 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 so long as the square root itself can be represented in a single float, i.e. for $a < 281474976710656 = 2^{46}$.

`mp_limb_t n_sqrtrem(mp_limb_t * r, mp_limb_t a)`

Computes the integer truncation of the square root of a. The integer itself can be represented exactly as a 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. 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 so long as 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.

```
int n_is_square(mp_limb_t 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(mp_limb_t 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.

22.13 Factorisation

```
int n_remove(mp_limb_t * n, mp_limb_t p)
```

Removes the highest possible power of p from n, replacing n with the quotient. The return value is that highest power of p that divided n. Assumes n is not 0.

For $p = 2$ trailing zeroes are counted. For other primes p is 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(mp_limb_t * n, mp_limb_t p, double
    ppre)
```

Removes the highest possible power of p from n, replacing n with the quotient. The return value is that 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, mp_limb_t p,
    ulong exp)
```

Inserts the given prime power factor `p^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^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.

```
mp_limb_t n_factor_trial_range(n_factor_t * factors,
    mp_limb_t n, ulong start, ulong num_primes)
```

Trial factor n with the first `num_primes` primes, but starting at the prime in `flint_primes` with index start.

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 `mp_limb_t`'s containing the distinct prime factors, `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.

```
mp_limb_t n_factor_trial(n_factor_t * factors, mp_limb_t 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.

```
mp_limb_t n_factor_power235(ulong *exp, mp_limb_t n)
```

Returns 0 if n is not a perfect square, cube or fifth power. Otherwise it returns the root and sets `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.

```
mp_limb_t n_factor_one_line(mp_limb_t n, ulong iters)
```

This implements Bill Hart's one line factoring algorithm [12]. 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}.

```
mp_limb_t n_factor_lehman(mp_limb_t 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.

```
mp_limb_t n_factor_SQUFOF(mp_limb_t n, ulong iters)
```

Attempts to split n using the given number of iterations of SQUFOF. Simply set `iters` to `~0L` for maximum persistence.

The version of SQUFOF imlemented here is as described by Gower and Wagstaff [8].

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 `iters` large enough this should never happen.

```
void n_factor(n_factor_t * factors, mp_limb_t n, int proved)
```

Factors n with no restrictions on n. If the prime factors are required to be certified prime, one may set `proved` to 1, otherwise set it to 0, and they will only be probable primes (with no known counterexamples to the conjecture that they are in fact all prime).

For details on the `n_factor_t` structure, see `n_factor_trial()`.

This function first tries trial factoring with a number of primes specified by the constant `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 `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 `FLINT_FACTOR_ONE_LINE_MAX` then `n_factor_one_line()` is called with `FLINT_FACTOR_ONE_LINE_ITERS` to try and split the factor. If that fails or the factor is too large for `n_factor_one_line()` then `n_factor_SQUFOF()` is called, with `FLINT_FACTOR_SQUFOF_ITERS`. If that fails an error results and the program aborts. However this should not happen in practice.

```
mp_limb_t n_factor_trial_partial(n_factor_t * factors,
    mp_limb_t n, mp_limb_t * prod, ulong num_primes,
    mp_limb_t limit)
```

Attempts trial factoring of n with the first `num_primes primes`, but stops when the product of prime factors so far exceeds `limit`.

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 `mp_limb_t`'s containing the distinct prime factors, `exp` an array containing the corresponding exponents.

The return value is the unfactored cofactor after trial factoring is done. The value `prod` will be set to the product of the factors found.

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.

```
mp_limb_t n_factor_partial(n_factor_t * factors, mp_limb_t
    n, mp_limb_t limit, int proved)
```

Factors n, but stops when the product of prime factors so far exceeds `limit`.

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`.

On exit, `num` will contain the number of distinct prime factors found. The field p is an array of `mp_limb_t`'s containing the distinct prime factors, `exp` an array containing the corresponding exponents.

The return value is the unfactored cofactor after factoring is done.

The factors are proved prime if `proved` is 1, otherwise they are merely probably prime.

22.14 Arithmetic functions

```
int n_moebius_mu(mp_limb_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, $\mu(0) = 0$.

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 `n_factor()` and count the factors.

```
void n_moebius_mu_vec(int * mu, ulong len)
```

Computes $\mu(n)$ for `n = 0, 1, ..., 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.

```
int n_is_squarefree(mp_limb_t 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.

```
mp_limb_t n_euler_phi(mp_limb_t 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.

22.15 Factorials

```
mp_limb_t n_factorial_mod2_preinv(ulong n, mp_limb_t p,
    mp_limb_t pinv)
```

Returns $n!$ mod p given a precomputed inverse of p as computed by `n_preinvert_limb()`.

§23. long_extras

Signed single limb arithmetic

23.1 Random functions

```
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`, `LONG_MAX` and `LONG_MIN` is increased.

This random function is mainly used for testing purposes.

```
mp_limb_signed_t z_randtest_not_zero(flint_rand_t state)
```

As for `z_randtest(state)`, but does not return 0.

```
mp_limb_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 `LONG_MAX`, it is interpreted as `LONG_MAX`.

§24. longlong.h

64-bit arithmetic

24.1 Auxiliary asm macros

```
umul_ppmm(high_prod, low_prod, multipler, multiplicand)
```

Multiplies two single limb integers `MULTIPLER` and `MULTIPLICAND`, and generates a two limb product in `HIGH_PROD` and `LOW_PROD`.

```
smul_ppmm(high_prod, low_prod, multipler, multiplicand)
```

As for `umul_ppmm()` but the numbers are signed.

```
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.

```
sdiv_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 $[-2^63, 2^63)$.

```
count_leading_zeros(count, x)
```

Counts 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 count is undefined.

```
count_trailing_zeros(count, x)
```

As for `count_leading_zeros()`, but counts from the least significant end. If `x` is zero then count is undefined.

```
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.

```
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.

```
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.

```
invert_limb(invxl, xl)
```

Computes an approximate inverse `invxl` of the limb `xl`, with an implicit leading 1. More formally it computes

```
invxl = (B^2 - B*x - 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 [10, Lemma 8.1]:

Let d be normalised, $d < B$, i.e. it fits in a word, and suppose that $md < B^2 \le (m+1)d$. Let $0 \le n \le Bd - 1$. Write $n = n_2 B + n_1 B/2 + n_0$ with $n_1 = 0$ or 1 and $n_0 < B/2$. Suppose $q_1 B + q_0 = n_2 B + (n_2 + n_1)(m - B) + n_1(d - B/2) + n_0$ and $0 \le q_0 < B$. Then $0 \le q_1 < B$ and $0 \le n - q_1 d < 2d$.

In the theorem, m is the inverse of d. If we let `m = invxl + B` and $d = x$ we have $md = B^2 - 1 < B^2$ and $(m+1)x = B^2 + d - 1 \ge B^2$.

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 $n_2 B + n_2(m - B)$. But note that $(m - B)$ is precisely our precomputed inverse `invxl`. If we write $q_1 B + q_0 = n_2 B + n_2(m - B)$, then from the theorem, we have $0 \le n - q_1 d < 3d$, i.e. the quotient is out by at most 2 and is always either correct or too small.

```
udiv_qrnnd_preinv(q, r, nh, nl, d, di)
```

As for `udiv_qrnnd()` but takes a precomputed inverse `di` as computed by `invert_limb()`. The algorithm, in terms of the theorem above, is:

```
nadj = n1*(d-B/2) + n0
xh, xl = (n2+n1)*(m-B)
xh, xl += nadj + n2*B ( xh, xl = n2*B + (n2+n1)*(m-B) +
    n1*(d-B/2) + n0 )
_q1 = B - xh - 1
xh, xl = _q1*d + nh, nl - B*d = nh, nl - q1*d - d so that
    xh = 0 or -1
r = xl + xh*d where xh is 0 if q1 is off by 1, otherwise -1
q = xh - _q1 = xh + 1 + n2
```

§25. mpn_extras

25.1 Macros

```
MACRO MPN_NORM(a, an)
```

Normalise `(a, an)` so that either `an` is zero or `a[an - 1]` is nonzero.

```
MACRO MPN_SWAP(a, an, b, bn)
```

Swap `(a, an)` and `(b, bn)`, i.e. swap pointers and sizes.

25.2 Utility functions

```
void mpn_debug(mp_srcptr x, mp_size_t xsize)
```

Prints debug information about `(x, xsize)` to `stdout`. In particular, this will print binary representations of all the limbs.

```
int mpn_zero_p(mp_srcptr x, mp_size_t xsize)
```

Returns 1 if all limbs of `(x, xsize)` are zero, otherwise 0.

25.3 Divisibility

```
int mpn_divisible_1_p(x, xsize, 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 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 mpn_remove_2exp(mp_ptr x, mp_size_t xsize,
    mp_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 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 mpn_factor_trial(mp_srcptr x, mp_size_t xsize, long
    start, long stop)
```

Searches for a factor of (`x, xsize`) among the primes in positions `start, ..., 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 >= 1`.

25.4 Division

```
int 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 `limbs1 >= limbsg > 0`.

25.5 GCD

```
mp_size_t mpn_gcd_full(mp_ptr arrayg, mp_ptr array1,
    mp_size_t limbs1, mp_ptr array2, mp_size_t limbs2)
```

Sets (`arrayg, retvalue`) to the gcd of (`array1, limbs1`) and (`array1, limbs2`).

The only assumption is that neither `limbs1` or `limbs2` is zero.

25.6 Special numbers

```
void mpn_harmonic_odd_balanced(mp_ptr t, mp_size_t * tsize,
    mp_ptr v, mp_size_t * vsize, long a, long b, long n, int
    d)
```

Computes (`t,tsize`) and (`v,vsize`) such that $t/v = H_n = 1 + 1/2 + \cdots + 1/n$. The computation is performed using recursive balanced summation over the odd terms. The resulting fraction will not generally be normalized. At the top level, this function should be called with $n > 0$, $a = 1$, $b = n$, and $d = 1$.

Enough space should be allocated for t and v to fit the entire sum $1 + 1/2 + \cdots + 1/n$ computed without normalization; i.e. t and v should have room to fit $n!$ plus one extra limb.

§26. profiler

26.1 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);

printf("cpu = %ld ms  wall = %ld ms\n", t0->cpu, t0->wall);
```

```
void start_clock(int n)
```

```
void stop_clock(int n)
```

```
double get_clock(int n)
```

Gives time based on cycle counter.

First one must ensure the processor speed in cycles per second is set correctly in `profiler.h`, in the macro definition `#define FLINT_CLOCKSPEED`.

One can access the cycle counter directly by `get_cycle_counter()` which returns the current cycle counter as a `double`.

A sample usage of clocks is:

```
init_all_clocks();

start_clock(n);

// do something

stop_clock(n);

printf("Time in seconds is %f.3\n", get_clock(n));
```

where `n` is a clock number (from 0-19 by default). The number of clocks can be changed by altering `FLINT_NUM_CLOCKS`. One can also initialise an individual clock with `init_clock(n)`.

26.2 Framework for repeatedly sampling a single target

```
void prof_repeat(double *min, double *max, profile_target_t
    target, ulong count)
```

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

Suppose one has a function one wishes to profile:

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

One creates a struct for passing arguments to our function:

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

a sample function:

```
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++)
    {
        prof_start();
        myfunc(a, b);
        prof_stop();
    }
}
```

Then we do the profile

```
double min, max;

myfunc_t params;

params.a = 3;
params.b = 4;

prof_repeat(&min, &max, sample_myfunc, &params);

printf("Min time is %lf.3s, max time is %lf.3s\n", min,
    max);
```

If either of the first two parameters to `prof_repeat` are `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`.

References

[1] Robert Baillie and Jr. Wagstaff, Samuel S., *Lucas pseudoprimes*, Mathematics of Computation **35** (1980), no. 152, pp. 1391–1417.

[2] D. Berend and T. Tassa, *Improved bounds on Bell numbers and on moments of sums of random variables*, Probability and Mathematical Statistics **30** (2010), pp. 185–205.

[3] J.P. Buhler, R.E. Crandall, and R.W. Sompolski, *Irregular primes to one million*, Math. Comp. **59** (1992), no. 2000, 717–722.

[4] Henri Cohen, *A course in computational algebraic number theory*, second ed., Springer, 1996.

[5] Richard Crandall and Carl Pomerance, *Prime numbers: A computational perspective*, second ed., Springer, August 2005.

[6] Marc Deleglise, Jean-Louis Niclas, and Paul Zimmermann, *Landau's function for one million billions*, J. Théor. Nombres Bordeaux **20** (2009), no. 3, 625–671.

[7] Pierre Dusart, *The kth prime is greater than $k(\ln k + \ln\ln k - 1)$ for $k \geq 2$*, Math. Comp. **68** (1999), no. 225, 411–415.

[8] Jason E. Gower and Samuel S. Wagstaff, Jr., *Square form factorization*, Math. Comp. **77** (2008), no. 261, 551–588.

[9] Torbjörn Granlund and Niels Möller, *Improved division by invariant integers*, IEEE Transactions on Computers **99** (2010), no. PrePrints, draft version available at http://www.lysator.liu.se/~nisse/archive/draft-division-paper.pdf.

[10] Torbjörn Granlund and Peter L. Montgomery, *Division by invariant integers using multiplication*, SIGPLAN Not. **29** (1994), 61–72.

[11] Guillaume Hanrot and Paul Zimmermann, *Newton iteration revisited*, http://www.loria.fr/~zimmerma/papers/fastnewton.ps.gz, 2004.

[12] William Hart, *A one line factoring algorithm*, http://sage.math.washington.edu/home/wbhart/onelinefactor.pdf, 2009.

[13] Peter Henrici, *A subroutine for computations with rational numbers*, J. ACM **3** (1956), no. 1, 6–9, http://doi.acm.org/10.1145/320815.320818.

[14] Ellis Horowitz, *Algorithms for rational function arithmetic operations*, Annual ACM Symposium on Theory of Computing: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing (Denver) (1972), 108–118, http://doi.acm.org/10.1145/800152.804903.

[15] Donald Knuth, *The art of computer programming vol. 2, seminumerical algorithms*, third ed., Addison–Wesley, Reading, Massachusetts, 1997.

[16] R. F. Lukes, C. D. Patterson, and H. C. Williams, *Some results on pseudosquares*, Math. Comp. **65** (1996), no. 213, 361–372, S25–S27, available at http://www.ams.org/journals/mcom/1996-65-213/S0025-5718-96-00678-3/S0025-5718-96-00678-3.pdf.

[17] Jean-Pierre Massias and Guy Robin, *Bornes effectives pour certaines fonctions concernant les nombres premiers*, J. Théor. Nombres Bordeaux **8** (1996), no. 1, 215–242.

[18] Thom Mulders, *On short multiplications and divisions*, AAECC **11** (2000), 69–88.

[19] George Nakos, Peter Turner, and Robert Williams, *Fraction-free algorithms for linear and polynomial equations*, ACM SIGSAM Bull. **31** (1997), no. 3, 11–19.

[20] J. Barkley Rosser and Lowell Schoenfeld, *Approximate formulas for some functions of prime numbers*, Illinois J. Math. **6** (1962), 64–94.

[21] D. Zeilberger, *The J.C.P. Miller recurrence for exponentiating a polynomial, and its q-analog*, Journal of Difference Equations and Applications **1** (1995), 57–60.