FLINT 1.0.2: Fast Library for Number Theory

William B. Hart and David Harvey

December 9, 2007

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 and David Harvey of Harvard University in the US.

As of version 1.0, FLINT compiles on and supports 32 and 64 bit x86 processors, the G5 and Alpha processors, though in theory it compiles on any machine with gcc version 3.4 or later and with GMP version 4.2.1 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, e.g: all the functions in the file `fmpz_poly.c` have test functions in the file `fmpz_poly-test.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.

Next, set the environment variables `FLINT_GMP_LIB_DIR` and `FLINT_GMP_INCLUDE_DIR` to point to your GMP library and include directories respectively.

Next type:

```
source flint_env
```

in the main directory of the FLINT directory tree.

Finally type:

```
make library
```

Move the library file `libflint.so`, `libflint.dll` or `libflint.dylib` (depending on your platform) into your library path and move all the .h files in the main directory of FLINT into your include path.

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 GMP with the options `-lflint -lgmp`.

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 the test programs, simply type:

`make test`

in the main FLINT directory.

The following is a list of the test programs which should be run:

`mpn_extras-test`

`fmpz_poly-test`

`fmpz-test`

`ZmodF-test`

`ZmodF_poly-test`

`mpz_poly-test`

`ZmodF_mul-test`

`long_extras-test`

4 Reporting bugs

The maintainers wish to be made aware of any and all bugs. Please send an email with your bug report to hart_wb@yahoo.com.

If possible please include details of your system, version of gcc, version of GMP and precise details of how to replicate the bug.

Note that FLINT needs to be linked against version 4.2.1 or later of GMP and must be compiled with gcc version 3.4 or later.

5 Example programs

FLINT comes with a number of example programs to demonstrate current and future FLINT features. To make the example programs, type:

`make examples`

The current example programs are:

`delta_qexp` Compute the first n terms of the delta function, e.g. `delta_qexp 1000000` will compute the first one million terms of the q-expansion of delta.

`BPTJCubes` Implements the algorithm of Beck, Pine, Tarrant and Jensen for finding solutions to the equation $x^3 + y^3 + z^3 = k$.

`bernoulli` Compute bernoulli numbers modulo a large number of primes.

`expmod` Computes a very large modular exponentiation.

6 FLINT macros

In the file flint.h are 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.

`FLINT_ABS(x)` returns the absolute value of a `long x`.

`FLINT_MIN(x, y)` returns the minimum of two `long` or two `unsigned long` values `x` and `y`.

`FLINT_MAX(x, y)` returns the maximum of two `long` or two `unsigned long` values `x` and `y`.

`FLINT_BIT_COUNT(x)` retuns the number of binary bits required to represent an `unsigned long x`.

7 The fmpz_poly module

The `fmpz_poly_t` data type represents elements of $\mathbb{Z}[x]$. The `fmpz_poly` module provides routines for memory management, basic arithmetic, and conversions to/from other types.

Each coefficient of an `fmpz_poly_t` is an integer of the FLINT `fmpz_t` type. Each coefficient of an `fmpz_poly_t` has the same number of limbs allocated for it, thus `fmpz_poly_t` polynomials are useful for dense polynomial arithmetic where the coefficients are not wildly different sizes.

Unless otherwise specified, all functions in this section permit aliasing between their input and output arguments.

7.1 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
```

7.2 Definition of the fmpz polynomial 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 limbs field of an fmpz_poly_t called poly1 one writes poly1->limbs.

With this way of thinking, fmpz_poly_t then has four fields:

- mp_limb_t* coeffs. This array contains all the fmpz_t's representing the coefficients of the polynomial, consecutively. The first coefficient represents the constant coefficient of the polynomial.
- unsigned long limbs. The number of limbs allocated for the absolute value of each coefficient. An additional limb per coefficient is also allocated to store the sign/size of the coefficient.
- unsigned long alloc. The maximum number of coefficients which can be stored in coeffs. The total amount of space allocated in coeffs is thus alloc*(limbs+1).
- unsigned long length. The current length of the polynomial, i.e. the number of coefficients which contain actual data. Always length <= alloc. The polynomial is the zero polynomial if and only if length == 0.

An fmpz_poly_t is said to be *normalised* if either length == 0, or if the final coefficient is nonzero. All fmpz_poly functions expect their inputs to be normalised, and unless otherwise specified they produce output that is normalised.

It is permissible to access the coefficients directly by modifying the limbs in coeffs, however if you modify the coefficients in this way, you must ensure that the polynomial is subsequently normalised by calling fmpz_poly_normalise().

7.3 Managed versus unmanaged layer

The module fmpz_poly has two layers, a 'managed' and an 'unmanaged' layer. Functions in the unmanaged layer are differentiated by having a leading underscore, e.g. _fmpz_poly_add.

Functions in the managed layer 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.

As a result of the possible need to reallocate, polynomials modified by functions in the managed layer must have been allocated using the FLINT heap memory manager, i.e. only functions such as fmpz_poly_init, without the leading underscore, can be used to allocate polynomials for use as *outputs* of managed functions.

On the other hand, the unmanaged layer does no memory management for the user. Each polynomial must have its coefficient limb size and maximum length set in advance. Both the memory management functions in the unmanaged and the managed layer can be used to this end. In particular the unmanaged layer offers stack based memory management options, though note that no reallocation can occur if this option is used.

A final benefit of the unmanaged layer is that one can operate on a range of coefficients of a polynomial. Functions are provided for attaching an fmpz_poly_t object to part of an existing polynomial and acting on that part, as though it were a separate polynomial. This can avoid making unnecessary copies of data and increase the performance of code.

Some functions are available in either the managed or unmanaged layer but not in both.

We now describe the functions available in `fmpz_poly`.

7.4 Initialisation and memory management

```
void fmpz_poly_init(fmpz_poly_t poly)
```

Initialise an `fmpz_poly_t` for use. All the fields `alloc`, `length` and `limbs` of `poly` are set 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.

For efficiency reasons, a call to `fmpz_poly_init` does not actually allocate any memory for coefficients. Each of the managed functions will automatically allocate any space needed for coefficients and in fact the easiest way to use the managed layer is to let FLINT do all the allocation automatically.

To this end, a user need only ever make calls to the `fmpz_poly_init` and `fmpz_poly_clear` memory management functions if they so wish. Naturally, more efficient code may result if the other memory management functions are also used.

```
void fmpz_poly_init2(fmpz_poly_t poly, unsigned long alloc,
                                       unsigned long limbs)
```

Initialise an `fmpz_poly_t` for use, allocating space for `alloc` coefficients each with the given number of limbs of space (plus an additional limb for the sign/size limb for each coefficient). The `length` field is set to zero.

This function should be used when the maximum length of the polynomial and the size of the coefficients is roughly known in advance. It may be faster than having FLINT automatically increase the size of the polynomial as the computation proceeds.

```
void fmpz_poly_realloc(fmpz_poly_t poly, unsigned long alloc)
```

Shrink or expand the polynomial so that it has space for precisely `alloc` coefficients. If `alloc` is less than the current length, the polynomial is truncated (and then normalised), otherwise the coefficients and current length remain unaffected.

If the parameter `alloc` is zero, any space currently allocated for coefficients in `poly` is freed. A subsequent call to `fmpz_poly_clear` is still permitted and does nothing.

For performance reasons, if `poly->limbs` is currently zero, this function does not do any allocation. A subsequent call to `fmpz_poly_fit_limbs` will do the actual allocation.

```
void fmpz_poly_fit_length(fmpz_poly_t poly, unsigned long alloc)
```

Expand the polynomial (if necessary) so that it has space for at least `alloc` coefficients. This function will never shrink the memory allocated for coefficients and the contents of the existing coefficients and the current length remain unaffected.

If the `limbs` field of `poly` is currently zero, then for performance reasons this function does not actually allocate any space. A subsequent call to `fmpz_poly_fit_limbs` will do any actual allocation.

```
void fmpz_poly_resize_limbs(fmpz_poly_t poly, unsigned long limbs)
```

Shrink or expand the coefficients so that each of them has space for the given number of limbs. It is required that either the existing coefficients still fit into the new limb size or the parameter `limbs` is set to zero. Given the former, the contents of the existing coefficients and the current length will remain unaffected.

If the parameter `limbs` is zero then any space currently allocated for coefficients in `poly` is freed. A subsequent call to `fmpz_poly_clear` is still permitted and does nothing.

If `poly->alloc` is currently zero, this function does no allocation.

```
void fmpz_poly_fit_limbs(fmpz_poly_t poly, unsigned long limbs)
```

Expand (if necessary) the coefficients so that each of them has space for the given number of limbs. This function will never shrink coefficients, thus existing coefficients and the current length are always preserved.

For performance reasons, no space is allocated if `poly->alloc` is currently zero. A subsequent call to `fmpz_poly_fit_length` will do the actual allocation.

```
void fmpz_poly_clear(fmpz_poly_t poly)
```

Free all memory used by the coefficients of `poly`. The polynomial object `poly` cannot be used again until a subsequent call to an initialisation function is made.

```
void _fmpz_poly_stack_init(fmpz_poly_t poly, unsigned long alloc,
                                             unsigned long limbs)
```

Initialise a polynomial, allocating space for `alloc` coefficients each taking no more than the given number of limbs of space (plus an additional limb for the sign/size).

Space is allocated on the FLINT stack, and can only be released by a corresponding call to `_fmpz_poly_stack_clear`.

Polynomials initialised and cleared in this way can only be used by unmanaged functions (with a leading underscore) and as *inputs* to managed functions. Reallocation or changing the number of limbs per coefficient is not permitted.

The `alloc` and `limbs` parameters of this function may be zero, in which case no memory is actually allocated. A subsequent call to `_fmpz_poly_stack_clear` is still permitted, but does nothing.

```
void _fmpz_poly_stack_clear(fmpz_poly_t poly)
```

Release any space allocated for `poly` back to the stack. The stack based memory manager requires that polynomials be released in the opposite order to that in which they were initialised with `_fmpz_poly_stack_init`.

7.5 Setting/retrieving coefficients

```
void fmpz_poly_get_coeff_mpz(mpz_t x, const fmpz_poly_t poly,
                                                  unsigned long n)
void _fmpz_poly_get_coeff_mpz(mpz_t x, const fmpz_poly_t poly,
                                                  unsigned long n)
```

Retrieve coefficient n as an `mpz_t`.

Coefficients are numbered from zero, starting with the constant coefficient.

The managed version returns zero when $n >=$ `poly->length`.

```
void fmpz_poly_set_coeff_mpz(fmpz_poly_t poly, unsigned long n,
                                                            mpz_t x)
void _fmpz_poly_set_coeff_mpz(fmpz_poly_t poly, unsigned long n,
                                                            mpz_t x)
```

Set coefficient n to the value of the given `mpz_t`.

Coefficients are numbered from zero, starting with the constant coefficient. If n represents a coefficient beyond the current length of `poly`, zero coefficients are added in between the existing coefficients and the new coefficient, if required.

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

Retrieve coefficient n as an `fmpz_t`.

Coefficients are numbered from zero, starting with the constant coefficient.

It is assumed that the `fmpz_t` supplied has already been allocated with sufficient space for the coefficient being retrieved.

The managed version returns zero when $n >=$ `poly->length`.

```
void fmpz_poly_set_coeff_fmpz(fmpz_poly_t poly, unsigned long n,
                                                           fmpz_t x)
void _fmpz_poly_set_coeff_fmpz(fmpz_poly_t poly, unsigned long n,
                                                           fmpz_t x)
```

Set coefficient n to the value of the given `fmpz_t`.

Coefficients are numbered from zero, starting with the constant coefficient. If n represents a coefficient beyond the current length of `poly`, zero coefficients are added in between the existing coefficients and the new coefficient, if required.

```
unsigned long fmpz_poly_get_coeff_ui(const fmpz_poly_t poly,
                                                  unsigned long n)
unsigned long _fmpz_poly_get_coeff_ui(const fmpz_poly_t poly,
                                                  unsigned long n)
```

Return the absolute value of coefficient n as an `unsigned long`.

Coefficients are numbered from zero, starting with the constant coefficient. If the coefficient is longer than a single limb, the first limb is returned.

The managed version returns zero when $n >=$ `poly->length`.

```
void fmpz_poly_set_coeff_ui(fmpz_poly_t poly, unsigned long n,
                                                  unsigned long x)
void _fmpz_poly_set_coeff_ui(fmpz_poly_t poly, unsigned long n,
                                                  unsigned long x)
```

Set coefficient n to the value of the given `unsigned long`.

Coefficients are numbered from zero, starting with the constant coefficient. If n represents a coefficient beyond the current length of `poly`, zero coefficients are added in between the existing coefficients and the new coefficient, if required.

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

Return the value of coefficient n as a `long`.

Coefficients are numbered from zero, starting with the constant coefficient. If the coefficient will not fit into a `long`, i.e. if its absolute value takes up more than `FLINT_BITS - 1` bits then the result is undefined.

The managed version returns zero when $n >=$ `poly->length`.

```
void fmpz_poly_set_coeff_si(fmpz_poly_t poly, unsigned long n, long x)
void _fmpz_poly_set_coeff_si(fmpz_poly_t poly, unsigned long n, long x)
```

Set coefficient n to the value of the given `long`.

Coefficients are numbered from zero, starting with the constant coefficient. If n represents a coefficient beyond the current length of `poly`, zero coefficients are added in between the existing coefficients and the new coefficient, if required.

```
fmpz_t fmpz_poly_get_coeff_ptr(fmpz_poly_t poly, unsigned long n)
fmpz_t _fmpz_poly_get_coeff_ptr(fmpz_poly_t poly, unsigned long n)
```

Return a pointer to coefficient n of `poly`, cast as an `fmpz_t`. This function is provided so that individual coefficients can be operated on by functions in the `fmpz` module.

Coefficients are numbered from zero, starting with the constant coefficient.

The managed version returns NULL when $n >=$ `poly->length`.

7.6 String conversions and I/O

The functions in this section are not intended to be particularly fast. They are intended mainly as a debugging aid.

All of the functions use the same string representation of polynomials. It is given by a sequence of integers, in decimal notation, separated by whitespace. The first integer gives the length of the polynomial; the remaining `length` 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 (provided they fit in the given polynomial).

```
int fmpz_poly_from_string(fmpz_poly_t poly, const char* s)
```

Import a polynomial from a string. If the string represents a valid polynomial the function returns 1, otherwise it returns 0.

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

Convert a polynomial to a string and return a pointer to the string. Space is allocated for the string by this function and must be freed when it is no longer used, by a call to `free`.

```
void fmpz_poly_fprint(const fmpz_poly_t poly, FILE* f)
```

Convert a polynomial to a string and write it to the given stream.

```
void fmpz_poly_print(const fmpz_poly_t poly)
```

Convert a polynomial to a string and write it to `stdout`.

```
void fmpz_poly_fread(fmpz_poly_t poly, FILE* f)
```

Read a polynomial from the given stream. Return 1 if the data from the stream represented a valid polynomial, otherwise return 0.

```
void fmpz_poly_read(fmpz_poly_t poly)
```

Read a polynomial from `stdin`. Return 1 if the data read from `stdin` represented a valid polynomial, otherwise return 0.

7.7 Polynomial parameters (length, degree, limbs, etc.)

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

Return `poly->length - 1`. The zero polynomial is defined to have degree -1.

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

Return `poly->length`. The zero polynomial is defined to have length 0.

```
unsigned long fmpz_poly_limbs(const fmpz_poly_t poly)
unsigned long _fmpz_poly_limbs(const fmpz_poly_t poly)
```

Return `poly->limbs`.

Each coefficient of `poly` is allowed up to this many limbs to store its absolute value, plus an additional limb to store its sign/size. Thus the total memory currently allocated for the storage of coefficients is `poly->alloc*(poly->limbs+1)`.

```
unsigned long fmpz_poly_max_limbs(const fmpz_poly_t poly)
unsigned long _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`. This may be less than `poly->limbs`.

```
long fmpz_poly_max_bits(const fmpz_poly_t poly)
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.

```
long fmpz_poly_max_bits1(const fmpz_poly_t poly)
long _fmpz_poly_max_bits1(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.

The assumption is made that the absolute value of each coefficient fits into an unsigned long. This function will be more efficient than the more general `fmpz_poly_max_bits` in this situation.

7.8 Assignment and basic manipulation

```
void fmpz_poly_set(fmpz_poly_t poly1, const fmpz_poly_t poly2)
void _fmpz_poly_set(fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Set polynomial `x` equal to the polynomial `y`.

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

Efficiently swap two polynomials. The coefficients are not moved in memory, pointers are simply switched. The unmanaged version does not swap the `alloc` fields of the polynomials.

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

Set the polynomial to the zero polynomial.

```
void fmpz_poly_zero_coeffs(fmpz_poly_t poly, unsigned long n)
void _fmpz_poly_zero_coeffs(fmpz_poly_t poly, unsigned long n)
```

Set the first n coefficients of `poly` to zero.

The unmanaged versiom of this function requires that `poly` have space allocated for at least n coefficients.

```
void fmpz_poly_neg(fmpz_poly_t poly)
void _fmpz_poly_neg(fmpz_poly_t poly)
```

Negate the polynomial, i.e. set it to `-poly`.

```
void fmpz_poly_truncate(fmpz_poly_t poly, const unsigned long trunc)
void _fmpz_poly_truncate(fmpz_poly_t poly, const unsigned long trunc)
```

If `trunc` is less than the current length of the polynomial, truncate the polynomial to that length. Note that as the function normalises its output, the eventual length of the polynomial may be less than `trunc`.

```
void fmpz_poly_reverse(fmpz_poly_t output, const fmpz_poly_t poly,
                                            unsigned long length)
void _fmpz_poly_reverse(fmpz_poly_t output, const fmpz_poly_t poly,
                                            unsigned long length)
```

This function considers the polynomial `poly` to be of length n, notionally truncating and zero padding if required, and reverses the result. Since this function normalises its result the eventual length of `output` may be less than `length`.

The unmanaged version of this function requires that `output` have space allocated for at least `length` coefficients and that `output->limbs` is at least `poly->limbs`.

7.9 Subpolynomials

A number of functions are provided for attaching an fmpz_poly_t object to an existing polynomial or to a range of coefficients of an existing polynomial providing an alias for the original polynomial or part thereof.

Each of the functions in this section normalise the aliases.

One must take care when manipulating the alias, since manipulating it may leave the original polynomial unnormalised.

One must also be careful that one does not pass a polynomial and an alias for that polynomial to the same function since that function will have no way to tell it is dealing with aliases of the same polynomial.

```
void _fmpz_poly_attach(fmpz_poly_t output, const fmpz_poly_t poly)
```

Attach the fmpz_poly_t object output to the polynomial poly. Any changes made to the length field of output then do not affect poly.

```
void _fmpz_poly_attach_shift(fmpz_poly_t output,
                            const fmpz_poly_t input, unsigned long n)
```

Attach the fmpz_poly_t object output to poly but shifted to the left by n coefficients. This is equivalent to notionally shifting the original polynomial right (dividing by x^n) then attaching to the result.

```
void _fmpz_poly_attach_truncate(fmpz_poly_t output,
                            const fmpz_poly_t input, unsigned long n)
```

Attach the fmpz_poly_t object output to the first n coefficients of the polynomial poly. This is equivalent to notionally truncating the original polynomial to n coefficients then attaching to the result.

```
void _fmpz_poly_normalise(fmpz_poly_t poly)
```

Normalise the polynomial so that either the polynomial is the zero polynomial or the leading coefficient is not zero.

Since all other functions in fmpz_poly assume that input and output polynomials are normalised, this function is only used when manipulating the internals of a polynomial directly or when using subpolynomials.

7.10 Comparison

```
int fmpz_poly_equal(const fmpz_poly_t poly1, const fmpz_poly_t poly2)
int _fmpz_poly_equal(const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Return 1 if the two polynomials are equal, 0 otherwise.

7.11 Shifting

```
void fmpz_poly_left_shift(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long n)
void _fmpz_poly_left_shift(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long n)
```

Shift poly to the left by n coefficients (multiply by x^n) and write the result to `output`. Zero coefficients are inserted.

The unmanaged version of this function requires that `output` have space allocated for at least `n + poly->length` coefficients.

The parameter n must be non-negative, but can be zero.

```
void fmpz_poly_right_shift(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long n)
void _fmpz_poly_right_shift(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long n)
```

Shift poly to the right by n coefficients (divide by x^n and discard the remainder) and write the result to `output`.

The parameter n must be non-negative, but can be zero. Shifting right by more than the current length of the polynomial results in the zero polynomial.

7.12 Addition/subtraction

```
void fmpz_poly_add(fmpz_poly_t output, const fmpz_poly_t poly1,
                                          const fmpz_poly_t poly2)
void _fmpz_poly_add(fmpz_poly_t output, const fmpz_poly_t poly1,
                                          const fmpz_poly_t poly2)
```

Set the output to the sum of the input polynomials.

Note that if `poly1` and `poly2` have the same length, cancellation may occur (if the leading coefficients have the same absolute values but opposite signs) and so the result may have less coefficients than either of the inputs. However, the unmanaged version of this function requires that the output have space allocated for the number of coefficients of the longest of the input polynomials.

When using the unmanaged version, note that overflow may occur when adding coefficients together and so one additional bit may be required to store the output coefficients than was required in either of the input polynomials. The additional bit is only required in the case that overflow occurs.

```
void fmpz_poly_sub(fmpz_poly_t output, const fmpz_poly_t poly1,
                                          const fmpz_poly_t poly2)
void _fmpz_poly_sub(fmpz_poly_t output, const fmpz_poly_t poly1,
                                          const fmpz_poly_t poly2)
```

Set the output to poly1 - poly2.

Note that if poly1 and poly2 have the same length, cancellation may occur (if the leading coefficients have the same values) and so the result may have less coefficients than either of the inputs. However, the unmanaged version of this function requires that the output have space allocated for the number of coefficients of the longest of the input polynomials.

When using the unmanaged version, note that overflow may occur when subtracting coefficients of opposite signs and so one additional bit may be required to store the output coefficients than was required in either of the input polynomials. The additional bit is only required in the case that overflow occurs.

7.13 Scalar multiplication and division

```
void fmpz_poly_scalar_mul_ui(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long x)
void _fmpz_poly_scalar_mul_ui(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long x)
```

Multiply poly by the unsigned long x and write the result to output.

When using the unmanaged version, the coefficients of output must have space for the largest output coefficient, i.e. the sum of the number of bits of the absolute values of x and the largest coefficient of poly.

```
void fmpz_poly_scalar_mul_si(fmpz_poly_t output,
                                      const fmpz_poly_t poly, long x)
void _fmpz_poly_scalar_mul_si(fmpz_poly_t output,
                                      const fmpz_poly_t poly, long x)
```

Multiply poly by the long x and write the result to output.

When using the unmanaged version, the coefficients of output must have space for the largest output coefficient, i.e. the sum of the number of bits of the absolute values of x and the largest coefficient of poly.

```
void fmpz_poly_scalar_mul_fmpz(fmpz_poly_t output,
                               const fmpz_poly_t poly, const fmpz_t x)
void _fmpz_poly_scalar_mul_fmpz(fmpz_poly_t output,
                               const fmpz_poly_t poly, const fmpz_t x)
```

Multiply poly by the fmpz_t x and write the result to output.

When using the unmanaged version, the coefficients of output must have space for the largest output coefficient, i.e. the sum of the number of bits of the absolute values of x and the largest coefficient of poly.

```
void fmpz_poly_scalar_mul_mpz(fmpz_poly_t output,
                               const fmpz_poly_t poly, const mpz_t x)
```

Multiply poly by the mpz_t x and write the result to output.

```
void fmpz_poly_scalar_div_ui(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long x)
void _fmpz_poly_scalar_div_ui(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long x)
```

Divide poly by the unsigned long x, round quotients towards minus infinity, discard remainders and write the result to output.

When using the unmanaged version, the coefficients of output must have space for the largest input coefficient.

```
void fmpz_poly_scalar_div_si(fmpz_poly_t output,
                                       const fmpz_poly_t poly, long x)
void _fmpz_poly_scalar_div_si(fmpz_poly_t output,
                                       const fmpz_poly_t poly, long x)
```

Divide poly by the long x, round quotients towards minus infinity, discard remainders and write the result to output.

When using the unmanaged version, the coefficients of output must have space for the largest input coefficient.

```
void fmpz_poly_scalar_tdiv_ui(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long x)
void _fmpz_poly_scalar_tdiv_ui(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long x)
```

Divide poly by the unsigned long x, round quotients towards zero, discard remainders and write the result to output.

When using the unmanaged version, the coefficients of output must have space for the largest input coefficient.

```
void fmpz_poly_scalar_tdiv_si(fmpz_poly_t output,
                                       const fmpz_poly_t poly, long x)
void _fmpz_poly_scalar_tdiv_si(fmpz_poly_t output,
                                       const fmpz_poly_t poly, long x)
```

Divide poly by the long x, round quotients towards zero, discard remainders and write the result to output.

When using the unmanaged version, the coefficients of output must have space for the largest input coefficient.

```
void fmpz_poly_scalar_div_exact_ui(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long x)
void _fmpz_poly_scalar_div_exact_ui(fmpz_poly_t output,
                              const fmpz_poly_t poly, unsigned long x)
```

Divide `poly` by the `unsigned long x`. Division is assumed to be exact and the result is undefined otherwise.

When using the unmanaged version, the coefficients of `output` must have space for the largest input coefficient.

```
void fmpz_poly_scalar_div_exact_si(fmpz_poly_t output,
                                        const fmpz_poly_t poly, long x)
void _fmpz_poly_scalar_div_exact_si(fmpz_poly_t output,
                                        const fmpz_poly_t poly, long x)
```

Divide `poly` by the `long x`. Division is assumed to be exact and the result is undefined otherwise.

When using the unmanaged version, the coefficients of `output` must have space for the largest input coefficient.

```
void fmpz_poly_scalar_div_fmpz(fmpz_poly_t output,
                                const fmpz_poly_t poly, const fmpz_t x)
void _fmpz_poly_scalar_div_fmpz(fmpz_poly_t output,
                                const fmpz_poly_t poly, const fmpz_t x)
```

Divide `poly` by the `fmpz_t x`, round quotients towards minus infinity, discard remainders, and write the result to `output`.

When using the unmanaged version, the coefficients of the polynomial `output` must have sufficient space allocated for `limbs1 - limbs2 + 1` limbs, where `limbs1` is the maximum number of limbs of the coefficients in `poly` and `limbs2` is the number of limbs required to store the absolute value of x.

```
void fmpz_poly_scalar_div_mpz(fmpz_poly_t output,
                                const fmpz_poly_t poly, const mpz_t x)
```

Divide `poly` by the `mpz_t x`, round quotients towards minus infinity, discard remainders, and write the result to `output`.

7.14 Polynomial multiplication

```
void fmpz_poly_mul(fmpz_poly_t output, const fmpz_poly_t poly1,
                                        const fmpz_poly_t poly2)
void _fmpz_poly_mul(fmpz_poly_t output, const fmpz_poly_t poly1,
                                        const fmpz_poly_t poly2)
```

Multiply the two given polynomials and return the result in `output`.

When using the unmanaged version, the coefficients of the output polynomial may be as large as `bits1 + bits2 + bits(length2)` where `bits1` is the number of bits of the absolute value of the largest coefficient of `poly1`, `bits2` is the corresponding thing for `poly2`, `bits(length2)` is the number of bits in the binary representation of the length of the shortest polynomial.

The length of the output polynomial will be `poly1->length + poly2->length - 1`.

```
void fmpz_poly_mul_trunc_n(fmpz_poly_t output,
     const fmpz_poly_t poly1, const fmpz_poly_t poly2, unsigned long n)
void _fmpz_poly_mul_trunc_n(fmpz_poly_t output,
     const fmpz_poly_t poly1, const fmpz_poly_t poly2, unsigned long n)
```

Multiply the two given polynomials and truncate the result to n coefficients, storing the result in `output`. This is sometimes known as a short product.

See `_fmpz_poly_mul` for a discussion of how big the output coefficients can be.

The length of the output polynomial will be at most the minimum of n and the value `poly1->length + poly2->length - 1`. It is permissible to set n to any non-negative value, however the function is optimised for n about half of `poly1->length + poly2->length`.

This function is more efficient than multiplying the two polynomials then truncating. It is the operation used when multiplying power series.

```
void fmpz_poly_mul_trunc_left_n(fmpz_poly_t output,
     const fmpz_poly_t poly1, const fmpz_poly_t poly2, unsigned long n)
void _fmpz_poly_mul_trunc_left_n(fmpz_poly_t output,
     const fmpz_poly_t poly1, const fmpz_poly_t poly2, unsigned long n)
```

Multiply the two given polynomials storing the result in `output`. This function guarantees all the coefficients except the first n, which may be arbitrary. This is sometimes known as an opposite short product.

See `_fmpz_poly_mul` for a discussion of how big the output coefficients can be.

The length of the output polynomial will be `poly1->length + poly2->length - 1` unless n is greater than or equal to this value, in which case it will return the zero polynomial. It is permissible to set n to any non-negative value, however the function is optimised for n about half of `poly1->length + poly2->length`.

For short polynomials, this function is more efficient than computing the full product.

7.15 Polynomial division

```
void fmpz_poly_divrem(fmpz_poly_t Q, fmpz_poly_t R,
                               const fmpz_poly_t A, const fmpz_poly_t B)
```

Performs division with remainder in $\mathbb{Z}[x]$. Computes polynomials `Q` and `R` in $\mathbb{Z}[x]$ such that the equation `A = B*Q + R`, holds. All but the final `B->length - 1` coefficients of `R` will be positive and less than the absolute value of the lead coefficient of `B`.

Note that in the special cases where the leading coefficient of `B` is ± 1 or `A = B*Q` for some polynomial `Q`, the result of this function is the same as if the computation had been done over $\mathbb{Q}$.

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

Performs division without remainder in $\mathbb{Z}[x]$. The computation returns the same result as `fmpz_poly_divrem`, but no remainder is computed. This is in general faster than computing quotient and remainder.

Note that in the special cases where the leading coefficient of `B` is ± 1 or `A = B*Q` for some polynomial `Q`, the result of this function is the same as if the computation had been done over $\mathbb{Q}$. In particular it can be used efficiently for exact division in $\mathbb{Z}[x]$.

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

Performs power series division in $\mathbb{Z}[[x]]$. The function considers the polynomials `A` and `B` to be power series of length n starting with the constant terms. The function assumes that `B` is normalised, i.e. that the constant coefficient is ± 1. The result is truncated to length n regardless of the inputs.

7.16 Pseudo division

```
void fmpz_poly_pseudo_divrem(fmpz_poly_t Q, fmpz_poly_t R,
          unsigned long * d, const fmpz_poly_t A, const fmpz_poly_t B)
```

Performs division with remainder of two polynomials in $\mathbb{Z}[x]$, notionally returning the results in $\mathbb{Q}[x]$ (actually in $\mathbb{Z}[x]$ with a single common denominator).

Computes polynomials `Q` and `R` such that `lead(B)^d*A = B*Q + R` where `R` has degree less than that of `B`.

This function may be used to do division of polynomials in $\mathbb{Q}[x]$ as follows. Suppose polynomials `C` and `D` are given in $\mathbb{Q}[x]$.

1) Write `C = d1*A` and `D = d2*B` for some polynomials `A` and `B` in $\mathbb{Z}[x]$ and integers `d1` and `d2`.

2) Use pseudo-division to compute `Q` and `R` in $\mathbb{Z}[x]$ so that `l^d*A = B*Q + R` where `l` is the leading coefficient of `B`.

3) We can now write `C = (d1/d2*D*Q + d1*R)/l^d`.

```
void fmpz_poly_pseudo_div(fmpz_poly_t Q, unsigned long * d,
                          const fmpz_poly_t A, const fmpz_poly_t B)
```

Performs division without remainder of two polynomials in $\mathbb{Z}[x]$, notionally returning the results in $\mathbb{Q}[x]$ (actually in $\mathbb{Z}[x]$ with a single common denominator).

Notionally computes polynomials `Q` and `R` such that `lead(B)^d*A = B*Q + R` where `R` has degree less than that of `B`, but returns only `Q`. This is slightly more efficient than computing the quotient and remainder.

7.17 Powering

```
void fmpz_poly_power(fmpz_poly_t output, const fmpz_poly_t poly,
                                             unsigned long exp)
```

Raises `poly` to the power `exp` and writes the result in `output`.

```
void fmpz_poly_power_trunc_n(fmpz_poly_t output,
          const fmpz_poly_t poly, unsigned long exp, unsigned long n)
```

Notionally raises `poly` to the power `exp`, truncates the result to length n and writes the result in `output`. This is computed much more efficiently than simply powering the polynomial and truncating.

This function can be used to raise power series to a power in an efficient way.

8 The fmpz module

The `fmpz` module is designed for manipulation of the FLINT flat multiprecision integer format `fmpz_t`. An `fmpz_t` is not a struct but merely a pointer to an array of limbs laid out in a certain way.

The first limb is a sign/size limb. If it is 0 the integer represented by the `fmpz_t` is 0. The absolute value of the sign/size limb is the number of subsequent limbs that the absolute value of the integer being represented, takes up. The absolute value of the integer is then stored as limbs, least significant limb first, in the subsequent limbs after the sign/size limb. If the sign/size limb is positive, a positive integer is intended and if the sign/size limb is negative the negative integer with the stored absolute value is intended.

The `fmpz_t` type is not intended as a standalone integer type. It is intended to be used in composite types such as polynomials and matrices which consist of many integer entries. All memory management is then done by the composite type, not by the `fmpz` module itself. Thus, none of the functions in the `fmpz` module do automatic memory management. It is up to the user to ensure that output `fmpz_t`'s have sufficient space allocated for them.

8.1 A simple example

We start with a simple example of the use of the `fmpz` module.

This example sets x to 3 and adds 5 to it.

```
#include "fmpz.h"
 ....
fmpz_t x = fmpz_init(1); // Allocate 1 limb of space
fmpz_set_ui(x, 3);
fmpz_add_ui_inplace(x, 5);
printf("3 + 5 is "); fmpz_print(x); printf("\n");
fmpz_clear(x);
```

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

8.2 Memory management

```
fmpz_t fmpz_init(unsigned long limbs)
```

Allocates space for an `fmpz_t` with the given number of limbs (plus an additional limb for the sign/size) on the heap and return a pointer to the space.

```
fmpz_t fmpz_realloc(fmpz_t f, unsigned long limbs)
```

Reallocate the space used by the `fmpz_t f` so that it has space for the given number of limbs (plus a sign/size limb). The parameter `limbs` must be non-negative. The existing contents of `f` are not altered if they still fit in the new size.

```
void fmpz_clear(const fmpz_t f)
```

Free space used by the `fmpz_t f`.

```
fmpz_t fmpz_stack_init(unsigned long limbs)
```

Allocates space for an `fmpz_t` with the given number of limbs (plus an additional limb for the sign/size) on the stack and return a pointer to the space.

```
void fmpz_stack_clear(const fmpz_t f)
```

Return space used by the `fmpz_t f` to the stack.

8.3 String operations

```
void fmpz_print(const fmpz_t f)
```

Print the multiprecision integer `f`.

8.4 fmpz properties

```
unsigned long fmpz_size(const fmpz_t f)
```

Return the number of limbs used to store the absolute value of the multiprecision integer `f`.

```
unsigned long fmpz_bits(const fmpz_t f)
```

Return the number of bits required to store the absolute value of the multiprecision integer `f`.

```
long fmpz_sgn(const fmpz_t f)
```

Return the sign/size limb of the multiprecision integer `f`. The sign of the sign/size limb is the sign of the multiprecision integer. The absolute value of the sign/size limb is the size in limbs of the absolute value of the multiprecision integer `f`.

8.5 Assignment

```
void fmpz_set_ui(fmpz_t res, unsigned long x)
```

Set the multiprecision integer `res` to the `unsigned long x`.

```
void fmpz_set_si(fmpz_t res, long x)
```

Set the multiprecision integer `res` to the `long x`.

```
void fmpz_set(fmpz_t res, const fmpz_t f)
```

Set the multiprecision integer `res` to equal the multiprecision integer `f`.

8.6 Comparison

```
int fmpz_equal(const fmpz_t f1, const fmpz_t f2)
```

Return 1 if `f1` is equal to `f2`, otherwise return 0.

```
int fmpz_is_one(const fmpz_t f)
```

Return 1 if `f` is one, otherwise return 0.

```
int fmpz_is_zero(const fmpz_t f)
```

Return 1 if `f` is zero, otherwise return 0.

8.7 Conversion

```
void mpz_to_fmpz(fmpz_t res, const mpz_t x)
```

Convert the `mpz_t x` to the `fmpz_t res`.

```
void fmpz_to_mpz(mpz_t res, const fmpz_t f)
```

Convert the `fmpz_t f` to the `mpz_t res`.

8.8 Addition/subtraction

```
void fmpz_add(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to the sum of `f1` and `f2`.

```
void fmpz_add_ui_inplace(fmpz_t res, unsigned long x)
```

Set `res` to the sum of `res` and the `unsigned long x`.

```
void fmpz_add_ui(fmpz_t res, const fmpz_t f, unsigned long x)
```

Set `res` to the sum of `f` and the `unsigned long x`.

```
void fmpz_sub(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to `f1` minus `f2`.

```
void fmpz_sub_ui_inplace(fmpz_t res, unsigned long x)
```

Set `res` to `res` minus the `unsigned long x`.

```
void fmpz_sub_ui(fmpz_t res, const fmpz_t f, unsigned long x)
```

Set `res` to `f` minus the `unsigned long x`.

8.9 Multiplication

```
void fmpz_mul(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to `f1` times `f2`.

```
void fmpz_mul_ui(fmpz_t res, const fmpz_t f1, unsigned long x)
```

Set `res` to `f1` times the `unsigned long x`.

```
void fmpz_addmul(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to `res + f1 * f2`.

8.10 Division

```
void fmpz_tdiv(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to the quotient of `f1` by `f2`. Round the quotient towards zero and discard the remainder.

```
void fmpz_fdiv(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to the quotient of `f1` by `f2`. Round the quotient towards minus infinity and discard the remainder.

```
void fmpz_tdiv_ui(fmpz_t res, const fmpz_t f1, unsigned long x)
```

Set `res` to the quotient of `f1` by the unsigned long `x`. Round the quotient towards zero and discard the remainder.

8.11 Powering

```
void fmpz_pow_ui(fmpz_t res, const fmpz_t f, unsigned long exp)
```

Set `res` to `f` raised to the power `exp`. This requires `exp` to be non-negative.

8.12 Number theoretical

```
void fmpz_binomial_next(fmpz_t next, const fmpz_t prev, long n, long k)
```

Assuming `prev` is set to the binomial coefficient `bin(n, k-1)` this function returns the binomial coefficient `bin(n, k)`. For efficiency reasons, this function requires that `next` has space for one more limb than the size of `prev`.

8.13 Miscellaneous

```
void fmpz_normalise(const fmpz_t f)
```

Normalise the multiprecision integer `f`.

Since all the functions in `fmpz` assume that all inputs are normalised and all outputs are normalised, this function is usually used internally by FLINT or can be used when modifying the internals of an `fmpz_t`.

9 The quadratic sieve

Currently the quadratic sieve is a standalone program which can be built by typing:

`make QS`

in the main FLINT directory.

The program is called `mpQS`. Upon running it, one enters the number to be factored at the prompt.

The quadratic sieve requires that the number entered not be a prime, not be a perfect power and it must not have very small factors. Trial division and the elliptic curve method should be run before making a call to the quadratic sieve, to remove small factors. The sieve may fail silently if the conditions are not met.

10 Large integer multiplication

In the module `mpn_extras` and `mpz_extras` are functions `F_mpn_mul` and `F_mpz_mul` respectively which are drop in replacements for GMP's `mpn_mul` and `mpz_mul` respectively.

These replacement functions are substantially faster than GMP 4.2.1 when multiplying integers which are thousands of limbs in size. For smaller multiplications these functions call their respective GMP counterparts.