FLINT: Fast Library for Number Theory

William B. Hart and David Harvey

September 14, 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.

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.

FLINT is supplied as a set of modules, `mpz_poly`, `fmpz_poly`, etc., each of which can be linked to a C program that wishes to make use of their functionality. There are some dependencies amongst the modules, and these are listed in the introductions to the relevant modules below.

All of the functions in FLINT have a corresponding test function provided in the appropriately named module in the test directory. E.g: all the functions in `mpz_poly.c` have test functions in `mpz_poly-test.c`.

2 The mpz_poly module

The `mpz_poly_t` data type represents elements of $\mathbb{Z}[x]$ by an array of `mpz_t`'s. It provides routines for memory management, basic arithmetic, and conversions to/from other types.

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

2.1 Simple example

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

```
#include "mpz_poly.h"
 ....
mpz_poly_t x, y;
mpz_poly_init(x);
mpz_poly_init(y);
mpz_poly_set_coeff_ui(x, 3, 5);
mpz_poly_set_coeff_si(x, 0, -1);
mpz_poly_mul(y, x, x);
```

```
mpz_poly_print(x); printf("\n");
mpz_poly_print(y); printf("\n");
mpz_poly_clear(x);
mpz_poly_clear(y);
```

Output is:

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

2.2 Definition of `mpz_poly_t`

The `mpz_poly_t` type is actually a typedef for an array of length 1 of `mpz_poly_struct`. This permits passing parameters of type `mpz_poly_t` 'by reference'.

The `mpz_poly_struct` struct has three members:

- `mpz_t* coeffs`. An array of `mpz_t`'s of length `alloc`. All of them are `mpz_init`'d.
- `unsigned long alloc`. Length of `coeffs`. Always `alloc >= 1`.
- `unsigned long length`. The current length of the polynomial. That is, for `n < length`, the coefficient of x^n is `coeffs[n]`, and for `n >= length`, the coefficient of x^n is zero. Always `length <= alloc`. If `length == 0` then this is the zero polynomial.

An `mpz_poly_t` is said to be *normalised* if either `length == 0`, or if `coeffs[length-1]` is nonzero. All `mpz_poly_blah()` functions expect their inputs to be normalised, and unless other specified they produce output that is normalised. If you modify the coefficients yourself, you must ensure that the polynomial is subsequently normalised (for example by using `mpz_poly_normalise()`).

All `mpz_poly_t`'s are allocated on the heap. The reason we don't bother with stack storage is that most of the memory allocation overhead for `mpz_poly_t` is in the coefficients anyway, and providing both stack and heap allocation would just make things unnecessarily complicated.

2.3 Comparison with `fmpz_poly_t`

Advantages of `mpz_poly_t` over `fmpz_poly_t` are:

- GMP's mpz functions may be used directly on the coefficients.
- If the coefficients vary a lot in size, the memory usage will be more efficient. (In fact it might be completely impractical to use `fmpz_poly_t` for such a polynomial.)

Disadvantages compared to `fmpz_poly_t` are:

- `fmpz_poly_t` is more efficient (in both time and space) for dense polynomials with relatively small, equally-sized coefficients, because it has much less memory management overhead.

2.4 Initialisation and memory management

```
void mpz_poly_init(mpz_poly_t poly)
```

Initialises an `mpz_poly_t` object. This function must be called before using the polynomial. The initial allocated size is set to 1. The length is set to zero, so this is the zero polynomial.

This function should not be used twice on the same polynomial without an intervening `mpz_poly_clear()`; this will cause memory leaks.

```
void mpz_poly_clear(mpz_poly_t poly)
```

Frees the resources associated with an `mpz_poly_t` object. The coefficients are `mpz_clear`ed and the polynomial object becomes unusable. To use it again, `mpz_poly_init()` must be called.

```
void mpz_poly_init2(mpz_poly_t poly, unsigned long alloc)
```

Same as `mpz_poly_init()`, but with `alloc` coefficients initially allocated. Must have `alloc >= 1`.

```
void mpz_poly_realloc(mpz_poly_t poly, unsigned long alloc)
```

Reallocates the array of coefficients to length `alloc`. Must have `alloc >= 1`. The value of the polynomial is preserved as far as possible (i.e. up to at most `alloc` coefficients).

```
void mpz_poly_ensure_alloc(mpz_poly_t poly, unsigned long alloc)
```

Ensures that at least `alloc` coefficients are allocated in `poly`, by increasing the number of allocated coefficients if necessary. If more coefficients are required, the number of allocated coefficients is at least doubled. The value of the polynomial is preserved.

2.5 Setting/retrieving coefficients

```
mpz_t* mpz_poly_get_coeff_ptr(mpz_poly_t poly, unsigned long n)
```

Returns a pointer to the coefficient of x^n in `poly`, or `NULL` if n is beyond the current length of the polynomial.

```
void mpz_poly_get_coeff(mpz_t c, mpz_poly_t poly, unsigned long n)
```

Copies the coefficient of x^n in `poly` into `c`. If n is beyond the current length of the polynomial, `c` is set to zero.

```
unsigned long mpz_poly_get_coeff_ui(mpz_poly_t poly, unsigned long n)
```

Returns the absolute value of the coefficient of x^n in `poly` as an `unsigned long`. If it doesn't fit, only the least significant bits are returned. (See GMP's `mpz_get_ui()` function.) If n is beyond the current length of the polynomial, the return value is zero.

```
long mpz_poly_get_coeff_si(mpz_poly_t poly, unsigned long n)
```

Returns the coefficient of x^n in `poly` as a `long`. If it doesn't fit, the return value probably doesn't mean much (but see GMP's `mpz_get_si()` function). If n is beyond the current length of the polynomial, the return value is zero.

```
mpz_t* _mpz_poly_get_coeff_ptr(mpz_poly_t poly, unsigned long n)
void _mpz_poly_get_coeff(mpz_t c, mpz_poly_t poly, unsigned long n)
unsigned long _mpz_poly_get_coeff_ui(mpz_poly_t poly, unsigned long n)
long _mpz_poly_get_coeff_si(mpz_poly_t poly, unsigned long n)
```

These are the same as the functions above, but they are inlined, and do no bounds checking. If n is beyond the current length of the polynomial, the result is undefined.

```
void mpz_poly_set_coeff(mpz_poly_t poly, unsigned long n, mpz_t c)
void mpz_poly_set_coeff_ui(mpz_poly_t poly, unsigned long n,
                           unsigned long c)
void mpz_poly_set_coeff_si(mpz_poly_t poly, unsigned long n, long c)
```

Sets the coefficient of x^n in `poly` to c. If n is beyond the current length of the polynomial, the polynomial is extended and reallocated appropriately.

```
void _mpz_poly_set_coeff(mpz_poly_t poly, unsigned long n, mpz_t c)
void _mpz_poly_set_coeff_ui(mpz_poly_t poly, unsigned long n,
                            unsigned long c)
void _mpz_poly_set_coeff_si(mpz_poly_t poly, unsigned long n, long c)
```

These are the same as the functions above, but they are inlined, and do no bounds checking. If n is beyond the current length of the polynomial, the result is undefined. Additionally, they do not ensure that the result is normalised.

2.6 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`".

```
int mpz_poly_from_string(mpz_poly_t poly, char* s)
```

Converts `s` into a polynomial, stored in `poly`. The return value is 1 if the conversion succeeded. The return value is zero if the string did not represent a valid polynomial, in which case `poly` will be in a legal state, but with an undefined value.

```
char* mpz_poly_to_string(mpz_poly_t poly)
```

Converts the polynomial to a string and returns a character buffer that was allocated by `malloc`. You should call `free` when the string is no longer needed.

```
void mpz_poly_print(mpz_poly_t poly)
```

Prints the given polynomial to standard output.

```
void mpz_poly_fprint(mpz_poly_t poly, FILE* f)
```

Prints the given polynomial to the given stream.

```
int mpz_poly_read(mpz_poly_t poly)
```

Reads a string from standard input and converts it to a polynomial. Return value has the same meaning as for `mpz_poly_from_string()`.

```
int mpz_poly_fread(mpz_poly_t poly, FILE* f)
```

Reads a string from the given stream and converts it to a polynomial. Return value has the same meaning as for `mpz_poly_from_string()`.

2.7 Length and degree

```
unsigned long mpz_poly_length(mpz_poly_t poly)
```

Return the polynomial's length.

```
long mpz_poly_degree(mpz_poly_t poly)
```

Returns the polynomial's degree, which is defined to be `length - 1`. In particular the degree of the zero polynomial is -1.

```
void mpz_poly_normalise(mpz_poly_t poly)
```

Normalises the polynomial; that is, reduces its length until either the length is zero, or the coefficient of $x^{\text{length}-1}$ is nonzero.

```
int mpz_poly_normalised(mpz_poly_t poly)
```

Returns a nonzero value if the polynomial is normalised.

```
void mpz_poly_truncate(mpz_poly_t res, mpz_poly_t poly,
                       unsigned long length)
```

Truncates `poly` to length `length`, puts result in `res`.

```
void mpz_poly_pad(mpz_poly_t poly, unsigned long length)
```

Ensures that the polynomial has length at least `length`, by zero-padding the polynomial if necessary. *The polynomial will not necessarily be normalised after this operation.* The value of the polynomial is preserved.

2.8 Assignment

```
void mpz_poly_set(mpz_poly_t res, mpz_poly_t poly)
```

Copies the value of `poly` into `res`.

```
void mpz_poly_zero(mpz_poly_t poly)
```

Sets `poly` to zero (by setting its length to zero).

```
void mpz_poly_swap(mpz_poly_t poly1, mpz_poly_t poly2)
```

Swaps the contents of `poly1` and `poly2` by pointer swapping. This is much more efficient than going via a temporary.

2.9 Conversions

```
void mpz_poly_to_fmpz_poly(fmpz_poly_t res, mpz_poly_t poly)
```

Converts `poly` into `fmpz_poly_t` format.

```
void fmpz_poly_to_mpz_poly(mpz_poly_t res, fmpz_poly_t poly)
```

Converts `poly` into `mpz_poly_t` format.

2.10 Comparison

```
int mpz_poly_equal(mpz_poly_t poly1, mpz_poly_t poly2)
```

Returns a nonzero value if `poly1` and `poly2` are equal.

2.11 Addition/subtraction

```
void mpz_poly_add(mpz_poly_t res, mpz_poly_t poly1, mpz_poly_t poly2)
```

Sets `res` equal to `poly1` plus `poly2`.

```
void mpz_poly_sub(mpz_poly_t res, mpz_poly_t poly1, mpz_poly_t poly2)
```

Sets `res` equal to `poly1` minus `poly2`.

```
void mpz_poly_neg(mpz_poly_t res, mpz_poly_t poly)
```

Sets `res` equal to the negative of `poly`.

2.12 Shifting

```
void mpz_poly_lshift(mpz_poly_t res, mpz_poly_t poly, unsigned long k)
```

Sets `res` equal to `poly` times x^k. If `res` is the same object as `poly`, this is done efficiently by pointer swapping.

```
void mpz_poly_rshift(mpz_poly_t res, mpz_poly_t poly, unsigned long k)
```

Sets `res` equal to `poly` divided by x^k, with the lower order terms discarded. If `res` is the same object as `poly`, this is done efficiently by pointer swapping.

```
void mpz_poly_shift(mpz_poly_t res, mpz_poly_t poly, long k)
```

Sets `res` equal to `poly` multiplied by x^k, where the semantics are the same as `mpz_poly_lshift()` or `mpz_poly_rshift()`, depending on whether k is non-negative or negative.

2.13 Scalar multiplication and division

```
void mpz_poly_scalar_mul(mpz_poly_t res, mpz_poly_t poly, mpz_t c)
void mpz_poly_scalar_mul_ui(mpz_poly_t res, mpz_poly_t poly,
                            unsigned long c)
void mpz_poly_scalar_mul_si(mpz_poly_t res, mpz_poly_t poly,
                            long c)
```

Sets `res` equal to `poly` times `c`.

```
void mpz_poly_scalar_div(mpz_poly_t res, mpz_poly_t poly, mpz_t c)
void mpz_poly_scalar_div_ui(mpz_poly_t res, mpz_poly_t poly,
                            unsigned long c)
void mpz_poly_scalar_div_si(mpz_poly_t res, mpz_poly_t poly, long c)
```

Sets `res` equal to `poly` divided by c. Rounding is towards zero (similar to the `mpz_tdiv` family in GMP). If c is zero then a division-by-zero is raised.

In the `ui` and `si` cases, in appropriate circumstances some precomputation is performed which is then shared among the coefficients, so this routine will be faster than dividing each coefficient by c separately. Similar functionality is planned for the `mpz_t` case.

```
void mpz_poly_scalar_div_exact(mpz_poly_t res, mpz_poly_t poly,
                               mpz_t c)
void mpz_poly_scalar_div_exact_ui(mpz_poly_t res, mpz_poly_t poly,
                                  unsigned long c)
void mpz_poly_scalar_div_exact_si(mpz_poly_t res, mpz_poly_t poly,
                                  long c)
```

Sets `res` equal to `poly` divided by c, *assuming* that c divides each coefficient exactly. If c does not divide them, the result is undefined. If c is zero then a division-by-zero is raised.

The remarks made above for `mpz_poly_scalar_div` regarding precomputation apply here also.

```
void mpz_poly_scalar_mod(mpz_poly_t res, mpz_poly_t poly, mpz_t c)
void mpz_poly_scalar_mod_ui(mpz_poly_t res, mpz_poly_t poly,
                            unsigned long c)
```

Sets `res` equal to `poly` modulo c, that is, reduces each coefficient into the range $[0, c)$. In the `mpz_t` case, the sign of `c` is ignored.

The remarks made above for `mpz_poly_scalar_div` regarding precomputation apply here also.

2.14 Polynomial multiplication

```
void mpz_poly_mul(mpz_poly_t res, mpz_poly_t poly1, mpz_poly_t poly2)
```

Sets `res` equal to `poly1` times `poly2`. An appropriate multiplication algorithm is selected based on the degree and the maximum size of the coefficients of the input polynomials.

The automatic algorithm selection strategy is based on the assumption that the polynomials are dense and have coefficients whose size does not vary too much. If this assumption is not satisfied, the chosen algorithm may be inappropriate. For example, if the polynomials represent the first few terms of the q-expansion of a modular form, then the coefficients might grow quite rapidly, in which case `mpz_poly_mul` will probably choose an FFT-based algorithm tuned for the largest coefficient; but the naive multiplication algorithm would probably do much better. Another example: if the polynomial is very sparse, then quite possibly FLINT is the wrong tool for the job, since it does not (yet) implement algorithms that can efficiently multiply sparse polynomials.

```
void mpz_poly_mul_naive(mpz_poly_t res, mpz_poly_t poly1,
                        mpz_poly_t poly2)
```

Sets `res` equal to `poly1` times `poly2`, using the 'naive' (classical) algorithm.

```
void mpz_poly_mul_karatsuba(mpz_poly_t res, mpz_poly_t poly1,
                            mpz_poly_t poly2)
```

Sets `res` equal to `poly1` times `poly2`, using Karatsuba's algorithm. This is asymptotically faster than the naive algorithm, but not as fast as FFT-based methods.

```
void mpz_poly_mul_SS(mpz_poly_t res, mpz_poly_t poly1,
                     mpz_poly_t poly2)
```

Sets `res` equal to `poly1` times `poly2`, using a Schönhage–Strassen FFT algorithm [?].

This is asymptotically the fastest multiplication algorithm implemented in FLINT, and is used for very large multiplications (several thousand words or higher). The underlying algorithm is a Schönhage–Strassen FFT operating on a polynomial whose coefficients have about the same number of bits as the degrees of the input polynomials (see the `ZmodF_poly_t` data type). A modification of the truncated Fourier transform [?] is used to improve smoothness of the running time.

To convert the original multiplication to a problem of this type, FLINT either packs coefficients together (in the case that the coefficients are initially too small compared to the degree), or splits them apart (in the case that the coefficients are too large compared to the degree). The first approach is similar to Kronecker segmentation, except that instead of packing all the way into a single integer, we aim directly for the polynomial on which the Schönhage–Strassen FFT operates. This was suggested independently by Paul Zimmerman and David Harvey. The splitting approach for the other case is due to William Hart.

```
void mpz_poly_mul_naive_KS(mpz_poly_t res, mpz_poly_t poly1,
                           mpz_poly_t poly2)
```

Sets `res` equal to `poly1` times `poly2`, using a 'naive Kronecker segmentation' algorithm. This function is provided for testing purposes only; it is never called by `mpz_poly_mul()`. It simply packs the coefficients into a single large integer, and multiplies the integers using GMP. It is asymptotically fast, and less likely to contain bugs than the other functions, as it is based on very mature GMP code.

```
void mpz_poly_sqr(mpz_poly_t res, mpz_poly_t poly)
void mpz_poly_sqr_naive(mpz_poly_t res, mpz_poly_t poly)
void mpz_poly_sqr_karatsuba(mpz_poly_t res, mpz_poly_t poly)
void mpz_poly_sqr_SS(mpz_poly_t res, mpz_poly_t poly)
void mpz_poly_sqr_naive_KS(mpz_poly_t res, mpz_poly_t poly)
```

These functions are the same as the multiplication functions given above, but specialised for squaring. Note that the multiplication functions will automatically call the squaring versions if they are passed two identical inputs.

2.15 Polynomial division

```
void mpz_poly_monic_inverse(mpz_poly_t res, mpz_poly_t poly,
                            unsigned long k)
```

Let n be the degree of `poly`, and assume that `poly` is monic. This function computes a monic polynomial `res` of degree k such that

$$x^{k+n} = \mathrm{res} \cdot \mathrm{poly} + R,$$

where R has degree less than n. In other words it computes an approximate inverse of `poly`, scaled by an appropriate power of x.

For sufficiently small k and sufficiently small input polynomials, this function uses a naive division algorithm (see `mpz_poly_monic_inverse_naive()` below). For larger problems it switches to a divide-and-conquer algorithm, and eventually a Newton iteration method.

```
void mpz_poly_pseudo_inverse(mpz_poly_t res, mpz_poly_t poly,
                             unsigned long k)
```

Let n be the degree of `poly`, and let d be the leading coefficient of `poly` (assumed nonzero). This function computes a polynomial `res` of degree k such that

$$d^{k+1}x^{k+n} = \mathrm{res} \cdot \mathrm{poly} + R,$$

where R has degree less than n. In other words it computes an approximate inverse of `poly`, scaled by an appropriate power of x and d.

The algorithms used are similar to those described above for `mpz_poly_monic_inverse()`, with appropriate modifications to handle $d \neq 1$.

```
void mpz_poly_monic_div(mpz_poly_t quot, mpz_poly_t poly1,
                        mpz_poly_t poly2)
```

This function divides `poly1` by `poly2`, assuming that `poly2` is monic. That is, it computes a polynomial `quot` such that

$$\mathrm{poly1} = \mathrm{quot} \cdot \mathrm{poly2} + \mathrm{rem},$$

where the remainder `rem` has degree less than `poly2`.

```
void mpz_poly_pseudo_div(mpz_poly_t quot, mpz_poly_t poly1,
                         mpz_poly_t poly2)
```

This function pseudo-divides `poly1` by `poly2`. That is, let d be the leading coefficient of `poly2` (assumed nonzero). Let n and m be the degrees of `poly1` and `poly2`. This function computes a polynomial `quot` such that

$$d^{n-m+1}\mathrm{poly1} = \mathrm{quot} \cdot \mathrm{poly2} + \mathrm{rem},$$

where the remainder `rem` has degree less than `poly2`.

```
void mpz_poly_monic_rem(mpz_poly_t rem, mpz_poly_t poly1,
                        mpz_poly_t poly2)
void mpz_poly_pseudo_rem(mpz_poly_t rem, mpz_poly_t poly1,
                         mpz_poly_t poly2)
void mpz_poly_monic_div_rem(mpz_poly_t quot, mpz_poly_t rem,
                            mpz_poly_t poly1, mpz_poly_t poly2)
void mpz_poly_pseudo_div_rem(mpz_poly_t quot, mpz_poly_t rem,
                             mpz_poly_t poly1, mpz_poly_t poly2)
```

The same as the functions above, but compute the remainder, or the quotient and the remainder.

```
void mpz_poly_monic_inverse_naive(mpz_poly_t res, mpz_poly_t poly,
                                  unsigned long k)
void mpz_poly_pseudo_inverse_naive(mpz_poly_t res, mpz_poly_t poly,
                                   unsigned long k)
void mpz_poly_monic_div_naive(mpz_poly_t quot, mpz_poly_t poly1,
                              mpz_poly_t poly2)
void mpz_poly_pseudo_div_naive(mpz_poly_t quot, mpz_poly_t poly1,
                               mpz_poly_t poly2)
void mpz_poly_monic_rem_naive(mpz_poly_t rem, mpz_poly_t poly1,
                              mpz_poly_t poly2)
void mpz_poly_pseudo_rem_naive(mpz_poly_t rem, mpz_poly_t poly1,
                               mpz_poly_t poly2)
void mpz_poly_monic_div_rem_naive(mpz_poly_t quot, mpz_poly_t rem,
                                  mpz_poly_t poly1, mpz_poly_t poly2)
void mpz_poly_pseudo_div_rem_naive(mpz_poly_t quot, mpz_poly_t rem,
                                   mpz_poly_t poly1, mpz_poly_t poly2)
```

The same as the functions above, but they always use a naive division algorithm.

2.16 GCD and extended GCD

```
void mpz_poly_content(mpz_t x, mpz_poly_t poly)
```

Computes the content of `poly` (the non-negative GCD of the coefficients) and stores it in `x`.

```
unsigned long mpz_poly_content_ui(mpz_poly_t poly)
```

Computes the content of `poly`, and returns it as an `unsigned long`. If it doesn't fit, the least significant bits are returned.

```
void mpz_poly_gcd(mpz_poly_t res, mpz_poly_t poly1, mpz_poly_t poly2)
```

(.....)

```
void mpz_poly_xgcd(mpz_poly_t res, mpz_poly_t a, mpz_poly_t b,
                   mpz_poly_t poly1, mpz_poly_t poly2)
```

(.....)

2.17 Miscellaneous

```
unsigned long mpz_poly_max_limbs(mpz_poly_t poly)
unsigned long mpz_poly_max_bits(mpz_poly_t poly)
```

Return the maximum number of limbs (respectively bits) in the coefficients of `poly`. Note that the former is somewhat faster, so it should be used if only a rough upper bound on the size is required.

```
unsigned long mpz_poly_product_max_limbs(mpz_poly_t poly1,
                                          mpz_poly_t poly2)
unsigned long mpz_poly_product_max_bits(mpz_poly_t poly1,
                                         mpz_poly_t poly2)
```

Returns the maximum number of limbs (respectively bits) that the coefficients of the product of `poly1` and `poly2` could possibly have, based on their lengths and coefficient sizes.

Note that `mpz_poly_product_max_limbs()` only examines the limb sizes of each input polynomial, so it's a fairly coarse estimate; it could overshoot the true bound by several limbs. It should not be used in situations where a tight bound is required. On the other hand it is faster than `mpz_poly_product_max_bits()`.