FLINT: Fast Library for Number Theory

William B. Hart

September 14, 2007

1 Introduction

FLINT is a C library for doing number theory. It is released under the GPL and we encourage interested people to contribute and/or fork our code.

FLINT will eventually have implementations of algorithms in number theory, specifically algebraic number theory, including p-adics. We have no plans to implement algebraic geometry, group theory or elliptic curve algorithms, but this may change if a suitable maintainer is found who would like to oversee such a project.

FLINT is currently maintained by Bill Hart from Warwick University and David Harvey from Harvard.

Although FLINT is designed as a standalone C library for direct use in C programs by number theorists, parts of FLINT will be made available for use in SAGE, maintained by William Stein.

FLINT 0.5, which includes fast polynomial multiplication (which has been incorporated into the SAGE version of Pari) and fast integer factorisation via the MPQS (multiple polynomial quadratic sieve) have been incorporated into SAGE (the sieve is the qsieve command which appears in SAGE 1.6 and following and the SAGE version of Pari makes use of FLINT from SAGE 2.5.1 onwards).

The intention is to release a series of improvements to Pari and the standalone quadratic sieve starting with FLINT 0.5 and onwards. These releases, FLINT 0.5, 0.6, etc., will be prereleases only.

FLINT 1.0 will be the first version of FLINT which will be a standalone C library with a documented interface which can be used by an end user. Its anticipated release date is July 1st 2007.

1.1 Code Base

FLINT is written entirely in C and all code must conform to the C99 standard. It must compile with the GCC toolset, available on most unix based systems. FLINT should have as few dependencies as possible, but any function from GMP may be used.

The code is maintained at a sourceforge SVN repository. The main development code is available at:

https://svn.sourceforge.net/svnroot/fastlibnt/trunk

Released versions of FLINT are forked from the main trunk and stored in separate folders in the repository, e.g. FLINT 0.5 is at

https://svn.sourceforge.net/svnroot/fastlibnt/flint-0.5

Various experimental branches are held at:

https://svn.sourceforge.net/svnroot/fastlibnt/branches

Programmers who wish to fiddle with some new ideas can start a branch ad libitum and play with FLINT files there without affecting the main development code.

1.2 Website

The FLINT website is found at:

http://www.flintlib.org/

Information about FLINT (pre)releases, progress updates and future directions can be found there. Profiles will also be linked to from the website for comparison with other comparable packages and projects.

In addition, programmers can access the FLINT sourceforge project at:

http://www.sourceforge.org/flintlibnt/

1.3 Development forums

Sourceforge provides us with a development forum. Developers who wish to be added to the FLINT development list can send an email requesting addition to hart_wb@yahoo.com

In addition, on occasion, FLINT developers find it useful to discuss things on IRC. The channel for this is flint-dev on the irc.freenode.net server. A web based IRC client is available at:

http://www.rlscnetwork.com/sharedresources/chat.html

1.4 Performance

The aim is for all FLINT functions to be at least as fast as the comparable functions available in the open source projects of a similar nature. The more elaborate functions will be faster in FLINT than in other open source projects where possible, and sometimes significantly faster.

In particular FLINT will perform as well as or better than NTL, Pari and LiDIA, which seem to be the most popular open source alternatives. FLINT will be regularly profiled and compared against these packages on a function by function basis. The more elaborate functions will have more elaborate profiles.

We also aim to beat MAGMA where possible, however it won't be a condition for a release of FLINT to be made that all functions in FLINT perform better than their MAGMA counterparts.

Profiles comparing FLINT with MAGMA will also be done regularly. However such a comparison is not fair to either FLINT or MAGMA, since MAGMA is an interpreted package, not a C library, and MAGMA is closed source and non-free, whereas FLINT is free and open source.

1.5 Testing

All functions available to an end user in FLINT will have a corresponding test function (to be written by the person who wrote the function, if no one else volunteers to do it for them). Also, all sufficiently sophisticated internal FLINT functions must have a corresponding test function. One line functions, which for example just return the value of some field of a structure, need not have a test function.

The general strategies used for testing FLINT functions are:

1) Send a large amount of random data of varying sizes and parameters to the function where possible.

2) Use the special GMP functions for generating random integers with long strings of 1's and 0's where this is possible.

3) If there is an associated function which should undo the effect of the function being tested (e.g. an addition function and a corresponding subtraction function), test the functions against one another.

4) If possible, get the function to do a standard computation, the result of which can be checked, e.g. check a factoring function by feeding numbers which are the known product of random integers and check the result.

5) If no other form of testing is possible, write a very simple version of the function which performs very poorly perhaps, or which uses a much simpler algorithm but produces the same result and compare the outputs.

6) Always do sufficient "eyeball" tests, i.e. get the function to print its output to the screen and look at the output to see if it looks like it is returning vaguely reasonable looking results to the eye.

7) Check boundary cases and just either side of them.

If it is only possible to test a function in situ (i.e. as part of a larger function which calls it), and a simpler version cannot be implemented to test against, insert checkpoints within each branch of the function and run random data through the function until such time as all branches have been worked. Explicitly check that all branches did what they were supposed to. FLINT_ASSERT's can be used to check that certain conditions were met after the branch executed.

The functions for testing the functions in ssmul.c should all be in a file called ssmul-test.c, etc.

The final version of a test file should take 1-2 seconds to test each function in the file being test, where possible (sometimes a much longer time may be necessary). However, much more extensive tests should be run by the programmer when the function is first written, to ensure that the function works as expected in every conceivable situation, especially if the function is very involved. Such test code should be retained, but need not execute when a user executes a make test.

Each final test function should print which function is being tested and then ok or fail. Examples of easy ways to set up such a test file can be found in the trunk of the development code, e.g. Zpoly_mpn-test.c

1.6 Parallel Processing

FLINT will support parallel processing at the thread level using pthreads. All functions that are sufficiently complicated will allow threads to be used. A global #define USE_THREADS in flint.h specifies whether threads should be used, and flint files using threads should contain

```
#include "flint.h"

#ifdef USE_THREADS
//code that makes use of threads
#else
//code that doesn't use threads
#endif
```

The files flint-threads.h and flint-threads.c will contain a flint thread manager. It will have a function which can be accessed which gives an upper limit on the number of new threads that should be created by a function which wants to create some new threads. All threaded functions should check how many threads it is allowed to create before creating any.

It will also have various other helper functions for implementing more complicated threaded scenarios where threads will be kept hanging around waiting for work and woken up when work is available for them, or for implementing work stealing etc.

1.7 Memory Manager

FLINT has a memory manager. When we were implementing polynomial multiplication, we found that just allocating memory as needed with malloc, was too slow. It is hopelessly bad if the function is recursive.

At the very least, functions should allocate as much of the memory as they need up front, then break it up as needed, rather than allocate lots of small chunks. But even this approach slows some things down. Thus we introduced a memory manager.

The FLINT memory manager is included in files flint-manager.h and flint-manager.c. It is a stack based memory manager (or will be).

Since stack based memory management is not ideal for threaded programs, it is implemented in a slightly strange way. Flint memory allocation functions require a thread number. So if there are numerous threads running within a FLINT function (or indeed a program running multiple threads, each calling different FLINT functions), each thread that is started will have a different stack of memory allocated to it. However the memory manager will be able to transfer blocks of memory from one thread to another if they become available.

However, the implementation details of the memory manager are irrelevant, since it will just work, regardless of how it is implemented. The only constraint in actual programming is that since the memory manager is stack based, any given thread should free memory in the reverse order to what it was allocated in the first place, e.g.

```
mp_limb_t * data1 = (mp_limb_t *) flint_malloc(1000);
mp_limb_t * data2 = (mp_limb_t *) flint_malloc(2000);
mp_limb_t * data3 = (mp_limb_t *) flint_malloc(300);

// intervening code

flint_free(data3);
flint_free(data2);
flint_free(data1);
```

Flint will automatically determine which thread made the call and allocate/deallocate from the correct stack.

1.8 FLINT modules

FLINT is implemented as a series of modules which perform related functions. Examples of modules are Z, Zvec, Zpoly, Zpoly_mpn, Zmod, Zp, etc.

Each module has associated .c and .h files named after it, and an associated test file. E.g. the module Zpoly contains functions for doing arithmetic with polynomials over the integers all of which are contained in Zpoly.c and Zpoly.h. Other files may be associated with Zpoly (e.g. ssmul) and these can be determined by looking at the #includes at the top of Zpoly.h. The test file for Zpoly will be called Zpoly-test.h. Running "make test" will compile and run all test files in FLINT. To run a specific test program, one can just type the name of the module, e.g. ./Zpoly-test, after all the test files have compiled.

Files with names like Zpoly-profile.c are for generating profiles for functions in Zpoly. These must be edited by hand to select which function(s) to spit out profile times for. But all such profile files are similar. To make all the profile files, one types "make profile". To run a specific profile, one types for example ./Zpoly-profile once they have all compiled.

Eventually FLINT will have all of the following modules:
Z - Arithmetic for GMP mpz_t integers
Z_mpn - Arithmetic for integers at the GMP mpn level, but in sign magnitude format
ZLong - Arithmetic for long/unsigned long integers
ZTwosComp - Arithmetic for multi precision integers in twos complement format
ZFermat - Arithmetic for integers modulo a Fermat number $p = 2^n + 1$ where $n = 2^l$
Zmod - Arithmetic for Z/nZ for a multi precision modulus n
ZmodLong - Arithmetic for Z/nZ for a modulus n which fits into an unsigned long
Zp - padic arithmetic
FF - Arithmetic for finite fields
GF2 - Helper functions for arithmetic over GF2

Zpoly - Polynomials over mpz_t integers
Zpoly_mpn - Polynomials over integers in mpn sign magnitude format
ZpolyTwosComp - Polynomial functions for polys over the twos complement format
ZpolyFermat - Polynomial functions for polys mod a Fermat number
ZmodPoly - Polynomials over Z/nZ for multiprecision n
ZmodPolyLong - Polynomials over Z/nZ for n an unsigned long
ZpPoly - Polys over padics
GF2Poly - Polys over GF2

ZMat - Linear Algebra over mpz_t integers
ZMat_mpn - Linear Algebra over integers in mpn sign magnitude format
ZmodMat - Linear Algebra over Z/nZ for multiprecision n
ZmodLongMat - Linear Algebra over Z/nZ for an unsigned long n
ZpMat - Linear Algebra over padics
GF2Poly - Linear Algebra over GF2

Lattice - Functions for lattices, including lattice based reduction (LLL)

QFB - Binary quadratic forms
QNF - Quadratic number fields
QZeta - Cyclotomic number fields
NF - General Number Fields

R - Some basic helper functions for floating point reals
C - Some basic helper functions for multi precision complex numbers
Q - Some basic functions for the rationals

1.9 Introduction to the FLINT C files

The file flint.h contains all the univeral #defines for flint, including ones that specify how many bits per limb the machine has, whether threads should be used and many other useful pieces of information.

2 Zpoly

The Zpoly interface has functions for doing arithmetic with polynomials defined over integers implemented as GMP mpz_t's.

The "alloc" field of the Zpoly_t type specifies the number of coefficients which have been allocated and the "length" field specifies the current length of the polynomial. Alloc must be at least 1 but length can be 0 for the zero polynomial. Alloc should always be less than or equal to length.

The module is divided into two halves. The first half implements functions beginning Zpoly, which manage everything for the user. In particular, if the result of a function returns a polynomial which is too long to fit in the allocated space of the output polynomial the whole output polynomial is reallocated automatically.

The other half of the module implements functions beginning _Zpoly. These functions do not allocate extra space and require the user to do the allocation in advance.

The Zpoly module will contain the following functions:

mpz_t* _Zpoly_get_coeff_ptr(Zpoly_t poly, unsigned long n)

void _Zpoly_get_coeff(mpz_t output, Zpoly_t poly, unsigned long n)

unsigned long _Zpoly_get_coeff_ui(Zpoly_t poly, unsigned long n)

long _Zpoly_get_coeff_si(Zpoly_t poly, unsigned long n)

void _Zpoly_set_coeff(Zpoly_t poly, unsigned long n, mpz_t x)

void _Zpoly_set_coeff_ui(Zpoly_t poly, unsigned long n,

void _Zpoly_set_coeff_si(Zpoly_t poly, unsigned long n, long x)

void _Zpoly_normalise(Zpoly_t poly);

long _Zpoly_get_degree(Zpoly_t poly);

unsigned long _Zpoly_get_length(Zpoly_t poly);

void _Zpoly_set(Zpoly_t output, Zpoly_t input);

void _Zpoly_zero(Zpoly_t output)

void _Zpoly_swap(Zpoly_t x, Zpoly_t y)

int _Zpoly_equal(Zpoly_t input1, Zpoly_t input2);

void _Zpoly_add(Zpoly_t output, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_sub(Zpoly_t output, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_negate(Zpoly_t output, Zpoly_t input);

void _Zpoly_scalar_mul(Zpoly_t poly, mpz_t x);

void _Zpoly_scalar_mul_ui(Zpoly_t poly, unsigned long x);

void _Zpoly_scalar_mul_si(Zpoly_t poly, long x);

void _Zpoly_scalar_div(Zpoly_t poly, mpz_t x);

void _Zpoly_scalar_div_ui(Zpoly_t poly, unsigned long x);

void _Zpoly_mul(Zpoly_t output, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_mul_naive(Zpoly_t output, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_mul_karatsuba(Zpoly_t output, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_sqr(Zpoly_t output, Zpoly_t input);

void _Zpoly_sqr_naive(Zpoly_t output, Zpoly_t input);

void _Zpoly_sqr_karatsuba(Zpoly_t output, Zpoly_t input);

void _Zpoly_left_shift(Zpoly_t output, Zpoly_t input, unsigned long n);

void _Zpoly_right_shift(Zpoly_t output, Zpoly_t input, unsigned long n);

void _Zpoly_div(Zpoly_t quotient, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_rem(Zpoly_t remainder, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_div_rem(Zpoly_t quotient, Zpoly_t remainder, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_gcd(Zpoly_t output, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_xgcd(Zpoly_t a, Zpoly_t b, Zpoly_t output, Zpoly_t input1, Zpoly_t input2);

void _Zpoly_content(mpz_t content, Zpoly_t a);

void Zpoly_init(Zpoly_t poly);

void Zpoly_init2(Zpoly_t poly, unsigned long alloc);

void Zpoly_init3(Zpoly_t poly, unsigned long alloc, unsigned long coeff_bits);

void Zpoly_realloc(Zpoly_t poly, unsigned long alloc);

void Zpoly_ensure_space(Zpoly_t poly, unsigned long alloc)

void Zpoly_clear(Zpoly_t poly);

along with Zpoly versions of all the _Zpoly functions.

3 Zpoly_mpn

The Zpoly_mpn interface has functions for doing arithmetic with polynomials defined over integers implemented as a special flint type which has a sign and magnitude. Each coefficient has a sign limb, followed by zero or more limbs (the number of which is specified by the absolute value of the sign limb) which contain a multiprecision coefficient. If the coefficient is zero, the sign limb is zero. If the sign limb is negative, the coefficient is interpreted to be negative, etc.

However, each coefficient is allocated exactly the same number of limbs (even if not all of them are used in each coefficient). The number of limbs allocated for each limb (excluding the sign limb) is specified in the "limbs" field of the Zpoly_mpn_t type. The length of the polynomial is given by the "length" field and the "alloc" field specifies the number of currently allocated coefficients (length should always be less than or equal to alloc). Alloc must be at least 1 but length can be 0 for the zero polynomial.

The Zpoly_mpn module is divided into two halves. The first half implements functions beginning Zpoly_mpn, which manage everything for the user. In particular, if the result of a function returns a polynomial which is too long to fit in the allocated space of the output polynomial the whole output polynomial is reallocated automatically.

The other half of the module implements functions beginning _Zpoly_mpn. These functions do not allocate extra space and require the user to do the allocation in advance. This includes increasing the number of allocated coefficients and increasing the number of limbs allocated for each coefficient, as necessary.

The useful feature of the _Zpoly_mpn functions is that one can specify a subset of the coefficients of a polynomial and operate on just those coefficients without copying them out to another polynomial first. As such, no such function should modify the "limbs" field of any Zpoly_mpn_t's that are passed to it. These functions should also never even look at the "alloc" field, since it is not even guaranteed to be set.

The Zpoly_mpn module will contain the following functions:

void _Zpoly_mpn_convert_out(Zpoly_t poly_mpz, Zpoly_mpn_t poly_mpn);

void _Zpoly_mpn_convert_in(Zpoly_mpn_t poly_mpn, Zpoly_t poly_mpz);

mp_limb_t _Zpoly_mpn_get_coeff_ptr(Zpoly_mpn_t poly, unsigned long n)

long _Zpoly_mpn_get_coeff(mp_limb_t output, Zpoly_mpn_t poly, unsigned long n)

unsigned long _Zpoly_mpn_get_coeff_ui(Zpoly_mpn_t poly, unsigned long n)

long _Zpoly_mpn_get_coeff_si(Zpoly_mpn_t poly, unsigned long n)

void _Zpoly_mpn_set_coeff(Zpoly_mpn_t poly, unsigned long n, mp_limb_t x, long sign, unsigned long size)

void _Zpoly_mpn_set_coeff_ui(Zpoly_mpn_t poly, unsigned long n, unsigned long x);

void _Zpoly_mpn_set_coeff_si(Zpoly_mpn_t poly, unsigned long n, long x);

void _Zpoly_mpn_normalise(Zpoly_mpn_t poly);

long _Zpoly_mpn_degree(Zpoly_mpn_t poly)

unsigned long _Zpoly_mpn_length(Zpoly_mpn_t poly)

unsigned long _Zpoly_mpn_limbs(Zpoly_mpn_t poly)

long Zpoly_mpn_degree(Zpoly_mpn_t poly);

unsigned long Zpoly_mpn_length(Zpoly_mpn_t poly);

void _Zpoly_mpn_set(Zpoly_mpn_t output, Zpoly_mpn_t input);

void _Zpoly_mpn_zero(Zpoly_mpn_t output)

void _Zpoly_mpn_swap(Zpoly_mpn_t x, Zpoly_mpn_t y);

int _Zpoly_mpn_equal(Zpoly_mpn_p input1, Zpoly_mpn_p input2);

void _Zpoly_mpn_negate(Zpoly_mpn_t output, Zpoly_mpn_t input);

void _Zpoly_mpn_add(Zpoly_mpn_t output, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_sub(Zpoly_mpn_t output, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_scalar_mul(Zpoly_mpn_t output, Zpoly_mpn_t poly, mp_limb_t x);

void _Zpoly_mpn_scalar_mul_ui(Zpoly_mpn_t output, Zpoly_mpn_t poly, unsigned long x);

void _Zpoly_mpn_scalar_mul_si(Zpoly_mpn_t output, Zpoly_mpn_t poly, long x);

void _Zpoly_mpn_scalar_div(Zpoly_mpn_t output, Zpoly_mpn_t poly, mp_limb_t x);

void _Zpoly_mpn_scalar_div_ui(Zpoly_mpn_t output, Zpoly_mpn_t poly, unsigned long x);

void _Zpoly_mpn_scalar_div_si(Zpoly_mpn_t output, Zpoly_mpn_t poly, long x);

void _Zpoly_mpn_scalar_div_exact_ui(Zpoly_mpn_t output, Zpoly_mpn_t poly, unsigned long x);

void _Zpoly_mpn_scalar_div_exact_si(Zpoly_mpn_t output, Zpoly_mpn_t poly, long x);

void _Zpoly_mpn_mul(Zpoly_mpn_t output, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_mul_naive(Zpoly_mpn_t output, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_mul_karatsuba(Zpoly_mpn_t output, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_sqr(Zpoly_mpn_t output, Zpoly_mpn_t input);

void _Zpoly_mpn_sqr_naive(Zpoly_mpn_t output, Zpoly_mpn_t input);

void _Zpoly_mpn_sqr_karatsuba(Zpoly_mpn_t output, Zpoly_mpn_t input);

void _Zpoly_mpn_left_shift(Zpoly_mpn_t output, Zpoly_mpn_t input, unsigned long n);

void _Zpoly_mpn_right_shift(Zpoly_mpn_t output, Zpoly_mpn_t input, unsigned long n);

void _Zpoly_mpn_div(Zpoly_mpn_t quotient, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_rem(Zpoly_mpn_t remainder, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_div_rem(Zpoly_mpn_t quotient, Zpoly_mpn_t remainder, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_gcd(Zpoly_mpn_t output, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_xgcd(Zpoly_mpn_t a, Zpoly_mpn_t b, Zpoly_mpn_t output, Zpoly_mpn_t input1, Zpoly_mpn_t input2);

void _Zpoly_mpn_content(mp_limb_t content, Zpoly_mpn_t a);

void Zpoly_mpn_init(Zpoly_mpn_t poly, unsigned long alloc, unsigned long limbs);

void Zpoly_mpn_realloc(Zpoly_mpn_t poly, unsigned long alloc);

void Zpoly_mpn_clear(Zpoly_mpn_t poly);

along with versions of all the _Zpoly_mpn functions for the Zpoly layer.