FLINT: Fast Library for Number Theory

Fredrik Johansson

Mathematical Software and
High Performance Algebraic Computing

ENS Lyon
June 26, 2023

What is FLINT?

https://flintlib.org

The goal of FLINT is (roughly) to provide a single C library with efficient implementations of all the basic rings needed in computational number theory and computer algebra.

Used, for instance, in SageMath and Oscar:

```
sage: R.<x> = PolynomialRing(ZZ, "x")
sage: %time a = (1+x)^10000 * (1-x)^10000
CPU times: user 640 ms, sys: 87.8 ms, total: 728 ms
Wall time: 733 ms
sage: len(str(a))
21804542
```

FLINT programming

```
#include "fmpz_poly.h"
...

fmpz_poly_t a, b;
fmpz_poly_init(a);
fmpz_poly_init(b);

fmpz_poly_set_coeff_si(a, 0, 1);
fmpz_poly_set_coeff_si(a, 1, 1);
fmpz_poly_pow(a, a, 10000);
fmpz_poly_set_coeff_si(b, 0, 1);
fmpz_poly_set_coeff_si(b, 1, -1);
fmpz_poly_pow(b, b, 10000);
fmpz_poly_mul(a, a, b);

fmpz_poly_clear(a);
fmpz_poly_clear(b);
```

Types in FLINT

Basic types:

- `fmpz`, `fmpq` - integers, rationals
- `nmod`, `fmpz_mod` - integers mod n (single/multi-word)
- `fq`, `fq_nmod`, `fq_zech` - finite fields
- `arb`, `acb` - real, complex numbers (ball arithmetic)
- `arf`, `acf` - real, complex floating-point numbers
- `nf_elem` - number field elements
- `qqbar` - algebraic numbers (canonical form)
- `ca` - exact algebraic, real or complex numbers

Structural types (examples):

- `fmpz_mat` - matrices over $\mathbb{Z}$
- `fmpz_poly`, `fmpz_mpoly` - elements of $\mathbb{Z}[x]$ and $\mathbb{Z}[x_1, \ldots, x_n]$
- `fmpz_mpoly_q` - elements of $\mathbb{Q}(x_1, \ldots, x_n)$

Data and algorithms in FLINT

- We give users access to all internals (for good or bad)
- Objects are mutable
- Arbitrary-size types inline small values: integers $\leq$ 62 bits, floats $\leq$ 128 bits
- Big integers use GMP, plus a lot of FLINT-specific code
- Specific tricks: multivariate exponents use byte-packed vectors, polynomials over $\mathbb{Q}$ use a single denominator, ...
- Many algorithms use non-canonical intermediate values (delayed GCD reduction, modular reduction, rounding, etc.)
- We use asymptotically fast algorithms, but much effort has also gone into optimizing small/medium input
- Many builtin functions are multithreaded (use `flint_set_num_threads(N)`)

Brief development history

FLINT was started in 2007 by Bill Hart and David Harvey.

FLINT 2: rewrite from scratch done around 2010 by Bill Hart, Sebastian Pancratz and myself.

Dozens of authors (`https://flintlib.org/authors.html`).
Notable sources of funding include:

- Google Summer of Code
- H2020 “OpenDreamKit”
- DFG “Oscar”

Currently, I am the only active developer after the departure of Bill Hart and Daniel Schultz in 2022.

Major changes in FLINT 3.0 (upcoming)

- Arb, Calcium and Antic merged into FLINT
  - Partly due to lack of manpower
  - The libraries were already tightly integrated
  - Can use Arb arithmetic in traditional FLINT code, etc.
- Small-prime FFT (contributed by Daniel Schultz)
  - Much faster arithmetic with huge numbers and polynomials
- Generic rings
  - Entirely in C, efficient, mathematically sound
  - Useful both internally and as an external interface
- Autotools-based build system (contributed by Albin Ahlbäck)

FFT multiplication

- GMP uses the Schönhage-Strassen algorithm (SSA), involving a mod $2^N + 1$ FFT, to multiply large integers.

- FLINT 2 has its own multithreaded SSA. This is also used for multiplying polynomials in $\mathbb{Z}[x]$ with large coefficients (Kronecker substitution is used for small coefficients).

- However, on modern (last 15+ years) hardware with fast single-word multiplication, small-prime FFTs (using $\mathbb{Z}/p\mathbb{Z}$ arithmetic with word-size p) outperform SSA.

  Used, for example, in Victor Shoup's NTL for polynomial multiplication in some ranges.

Daniel Schultz's new FFT in FLINT

- Uses 50-bit moduli, exactly representable in a `double`, with precomputed inverses $\approx 1/p$
- Double-word operations based on fused multiply-adds
- Uses SIMD vectors (currently AVX2 or NEON)
- Multithreaded
- Various data is cached to speed up multiple multiplications
- Integer multiplication uses up to 8 primes, with specialized code for reduction and Chinese remaindering
- Currently requires building FLINT with `--enable-avx2` or similar (this may change)

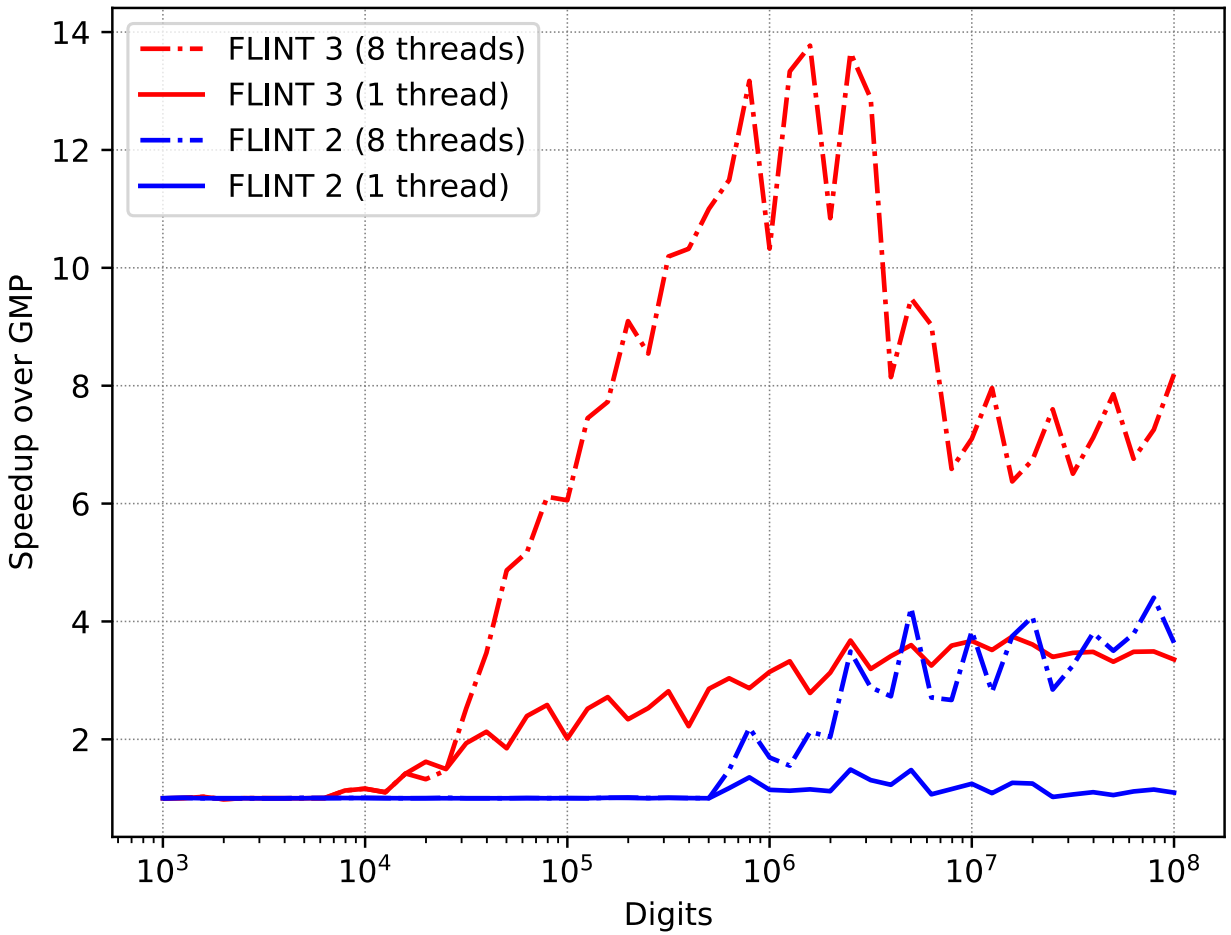
Things that benefit from the new FFT

- Arithmetic with integers, integers mod N, arbitrary-precision floats, etc. (>10,000 digits)
- Division, radix conversion, transcendental functions
- Polynomial arithmetic
- Anything else that FLINT reduces to big-integer or univariate polynomial multiplication internally

Some exceptions:

- Middle product (for optimal division code)
- Integer GCD, XGCD (these still use GMP)

Speedup for integer multiplication



Speedup of `flint_mpn_mul` vs `mpn_mul` (or `fmpz_mul` vs `mpz_mul`)
Machine: 8-core AMD Ryzen 7 PRO 5850U CPU (Zen 3)

Example: computing 100 million digits of π

gmp-chudnovsky.c	62.1 s
MPFR	172.9 s
FLINT 2	69.3 s
FLINT 3	31.6 s
y-cruncher	18.3 s
FLINT 2 (8 threads)	22.4 s
FLINT 3 (8 threads)	9.3 s
y-cruncher (8 threads)	4.5 s

Remarks:

- y-cruncher (closed source) is the world champion π program
- FLINT uses no special tricks for π; other transcendental functions (e.g. modular forms) get similar speedups

Speedup for multiplication in $\mathbb{Z}[x]$

Single-threaded speedup, FLINT 3 vs FLINT 2

	length n								
bits	16	64	256	1024	4096	16384	65536	262144	1048576
16	1.00	1.01	3.28	4.23	6.31	8.61	6.00	3.12	3.39
64	0.95	0.97	2.80	3.92	3.98	3.54	4.13	5.39	5.46
256	1.01	1.15	2.00	2.43	2.44	2.65	3.53	3.10	2.96
1024	0.97	1.01	1.20	1.47	2.30	3.97	3.28	3.08	3.27
4096	2.41	2.31	2.24	2.01	2.24	2.80	3.18	3.26	
16384	2.30	2.18	1.95	1.95	1.75	1.86	3.05		
65536	2.41	2.17	2.06	1.81	1.71	1.73			
262144	2.46	2.33	2.03	1.95	1.97				
1048576	2.83	2.45	2.27	2.28					

Observation: Kronecker substitution now (nearly) always beats Schönhage-Strassen

Speedup for multiplication in $\mathbb{Z}[x]$

Multi-threaded (8 threads)

	length n								
bits	16	64	256	1024	4096	16384	65536	262144	1048576
16	1.07	1.08	3.54	4.49	6.65	11.58	5.09	1.69	2.79
64	1.05	1.06	2.98	5.37	9.52	5.82	2.93	3.14	3.70
256	1.07	1.14	2.95	5.57	4.65	2.88	2.50	2.03	1.99
1024	1.04	1.03	2.62	3.09	2.16	2.68	2.17	2.19	2.20
4096	4.23	5.90	4.05	2.58	0.53	1.40	2.21	2.23	
16384	10.23	13.07	4.47	1.56	0.99	1.02	1.02		
65536	11.22	7.88	2.73	1.62	1.07	1.02			
262144	7.46	4.78	2.49	1.62	1.02				
1048576	6.27	5.23	2.90	1.97					

Generic rings

FLINT is designed around the principle that each structure over each base ring has its own C type:

- `fmpz_poly` for $\mathbb{Z}[x]$
- `fmpq_poly` for $\mathbb{Q}[x]$
- `arb_poly` for $\mathbb{R}[x]$
- ...

This has pros (type safety in C, performance) and cons (code duplication, bloat, API inconsistencies).

Also, generic programming on top of FLINT requires working with a wrapper (in Python, Julia, ...).

Generic rings: the `gr` module in FLINT 3

A ring R is represented by a context object which stores:

- `sizeof(element)`
- Pointer to method table: `init`, `set`, `add`, `mul`, `equal`, ...
- Ring parameters (optional): precision, modulus, matrix size, precomputed inverse, monomial ordering, evaluation settings, ...

Elements $x \in R$ are passed around as pointers.

Vectors of elements are packed together without overhead and can be allocated efficiently on the heap or the stack (compatible with preexisting FLINT code):

```
| element | element | element | ...
```

Generic rings - example

```
#include "gr.h"

int main() {
  int status;                          // error code
  gr_ctx_t ZZ, ZZx;                    // context objects
  gr_ptr a;                            // an element
  gr_ctx_init_fmpz(ZZ);                // ring of fmpz_t integers
  gr_ctx_init_gr_poly(ZZx, ZZ);        // Z[x]

  GR_TMP_INIT(a, ZZx);      // create one element on the stack
  status  = gr_gen(a, ZZx);                  // a = x
  status |= gr_add_ui(a, a, 1, ZZx);         // a += 1
  status |= gr_pow_ui(a, a, 10, ZZx);        // a = a ^ 10
  status |= gr_println(a, ZZx);

  GR_TMP_CLEAR(a, ZZx);
  gr_ctx_clear(ZZx);
  gr_ctx_clear(ZZ);
  return status;
}
```

Generic rings - Python demo

```
>>> from flint_ctypes import *

>>> R = PowerSeriesRing(QQbar, prec=3)
>>> x = R.gen()
>>> (2-x).sqrt()
(Root a = 1.41421 of a^2-2)
    + (Root a = -0.353553 of 8*a^2-1)*x
    + (Root a = -0.0441942 of 512*a^2-1)*x^2
    + O(x^3)

>>> M = Mat(R, 2)
>>> M
Ring of 2 x 2 matrices over Power series over Complex
      algebraic numbers (qqbar) with precision O(x^3)
>>> M([[(2-x).sqrt(), x], [2, (2-x).sqrt()]]).det()
2 - 3*x + O(x^3)
```

Generic rings - Python demo

```
>>> R = PowerSeriesRing(CC, prec=3)
>>> x = R.gen()
>>> (2-x).sqrt()
[1.414213562373095 +/- 2.99e-16]
    + [-0.353553390593274 +/- 5.31e-16]*x
    + [-0.044194173824159 +/- 3.48e-16]*x^2 + O(x^3)

>>> M = Mat(R, 2)
>>> M
Ring of 2 x 2 matrices over Power series over Complex
    numbers (acb, prec = 53) with precision O(x^3)
>>> M([[(2-x).sqrt(), x], [2, (2-x).sqrt()]]).det()
[2.00000000000000 +/- 1.30e-15]
    + [-3.00000000000000 +/- 1.44e-15]*x
    + [+/- 6.53e-16]*x^2 + O(x^3)
```

Why do generics in C in FLINT?

- Sage and Oscar are nice, but only convenient to use from their respective languages (Python/Julia), and there are performance and correctness issues.
- Economical way to implement long-requested features (multivariate polynomials over $\mathbb{R}$, power series types, sparse matrices, ...) without adding 100s of new types
- Chance to reduce bloat in FLINT (copy-pasted algorithms, plus *multiple* previous partial generics solutions) - 30,000 LOC already removed
- Allows adding specializations for performance (fixed-size finite fields, floats, etc.)
- Simplify interfacing with FLINT - can wrap generics once instead of wrapping each individual FLINT type (with subtle API differences between types)

Generic rings - current features

Wrapped base rings:

- Most familiar FLINT types (including Arb, Calcium, Antic)

Generic structures:

- Dense vectors, matrices
- Dense univariate polynomials, power series
- Sparse multivariate polynomials (very rudimentary)

Generic algorithms:

- Basecase and asymptotically fast polynomial operations, standard linear algebra, special functions, ...

Planned:

- Generic fraction fields, sparse matrices, etc.
- Fixed-size finite fields, floats, etc.

Correctness & error handling

Methods perform error handling uniformly, returning flags:

- `DOMAIN` (e.g. divide by zero)
- `UNABLE` (e.g. overflow, not implemented, undecidable)

Predicates return `TRUE`, `FALSE` or `UNKNOWN`.

Rings have enclosure semantics for inexact elements. For example, we distinguish between two kinds of power series:

- $2 - 3x + O(x^3)$ is an enclosure in $R[[x]]$
- $2 - 3x \pmod{x^3}$ is an exact element in $R[[x]]/\langle x^3 \rangle$

The Arb-based real field $\mathbb{R}$ is actually a field: it does not contain the element $+\infty$ (but admits the enclosure $(-\infty, +\infty)$).

Some current weaknesses of FLINT

- Lacks a lot of the higher level functionality found in Pari/GP, Magma, etc. (generic rings may or may not help with this)
- Many algorithms implemented years ago are now out of date
- Linear algebra can be hit-or-miss (no sparse linear algebra; some functions are poorly optimized)
- Limited SIMD use; no GPU acceleration; no distributed parallel computation
- Lacks tuning for specific architectures
- Sometimes memory-hungry (overallocation, caching); no support for out-of-RAM computation