News about FLINT

Fredrik Johansson

2024-06-17
MPFR/MPC/MPFI/ARB Developers Meeting
Bordeaux

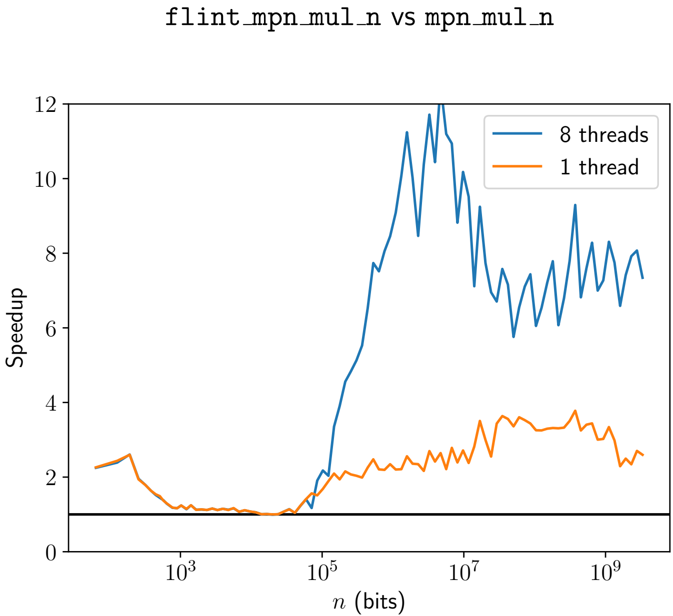
FLINT 2020 - present

- Bill Hart and Daniel Schultz leaving (2022)
- Albin Ahlbäck joining (2021)
- Big 3.0 release (2023)
- Merged Arb, Calcium, Antic
- Generic rings
- Small-prime FFT
- SIMD, assembly and multithreading optimizations
- Build and test system overhaul
- Interfaces (e.g. Python-flint)
- Many new functions and improvements

Workshops

Kaiserslautern (October 2023), Bordeaux (March 2024)

Integer multiplication: FLINT vs GMP



Generics in FLINT 3: motivation

Original FLINT philosophy: one ring $\leftrightarrow$ one C type

- `fmpq` - $\mathbb{Q}$
- `arb` - $\mathbb{R}$
- `arb_poly` - $\mathbb{R}[x]$

Drawbacks:

- 100 types $\times$ 100 methods $\approx$ 10 000 methods
- Hard to optimize versatile types (e.g. `arb`) for every use case

With generics, we can have:

- Generic polynomials, matrices, power series, etc. that work with any coefficient type
- Unified interface to all FLINT types and methods
- More specialized, efficient base types

Implementing rings

A ring R is defined by a context object `ctx` which contains:

- `sizeof(element)`
  - Elements will be packed contiguously in vectors
- Parameters and settings specific to a ring
- A method table
  - Memory management: `init`, `clear`, `swap`, ...
  - Assignment: `zero`, `one`, `set`, `set_si`, `set_other`, ...
  - Arithmetic: `neg`, `add`, `sub`, `mul`, `div`, ...
  - Predicates: `is_zero`, `equal`, ...
  - I/O: `write`, `set_str`, randomization: `randtest`
  - Ring predicates: `is_field`, `is_commutative_ring`, ...
  - Optional overloads for speed: `vec_add`, `mat_mul`, `poly_mul`, ...

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 $R[[x]]/\langle x^3 \rangle$

Example

Two implementations of real numbers:

```
>>> from flint_ctypes import *

>>> RR_ca("(1 + 1/3)^(1/2)")
1.15470 {(2*a)/3 where a = 1.73205 [a^2-3=0]}

>>> RR("(1 + 1/3)^(1/2)")
[1.154700538379251 +/- 6.94e-16]
```

Floating-point approximations, with the same interface:

```
>>> RF("(1 + 1/3)^(1/2)")
1.154700538379251
```

```
>>> R=RR

>>> A = Mat(R)([[R.cos(1), R.sin(1)], [R.sin(-1), R.cos(1)]])
>>> A
[[[0.540302305868140 +/- 4.59e-16], [0.841470984807897 +/- 6.08e-16]],
[[-0.841470984807897 +/- 6.08e-16], [0.540302305868140 +/- 4.59e-16]]]

>>> A.det()
[1.00000000000000 +/- 5.90e-16]

>>> A.det() == R("0.999")
False

>>> A.det() == 1
Traceback (most recent call last):
  ...
flint_ctypes.Undecidable: unable to decide x == y for
    x = [1.00000000000000 +/- 5.90e-16], y = 1 over
    Real numbers (arb, prec = 53)
```

```
>>> R = RR_ca
>>> A = Mat(R)([[R.cos(1), R.sin(1)], [R.sin(-1), R.cos(1)]])
>>> A
[[0.540302 - 0e-24*I {(a^2+1)/(2*a) where
    a = 0.540302 + 0.841471*I [Exp(1.00000*I {b})], ...

>>> A.det()
1
>>> A.det() == 1
True
```

```
>>> R = RF
>>> A = Mat(R)([[R.cos(1), R.sin(1)], [R.sin(-1), R.cos(1)]])
>>> A
[[0.5403023058681398, 0.8414709848078965],
[-0.8414709848078965, 0.5403023058681398]]

>>> A.det()
1.00000000000000
```

The Arb-based implementation of $\mathbb{R}$ does not contain the element ∞ but admits the enclosure $(-\infty, +\infty)$.

```
>>> 1 / RR(0)
  ...
FlintDomainError: x / y is not an element of
    {Real numbers (arb, prec = 53)} for {x = 1}, {y = 0}
```

```
>>> 1 / RR("0 +/- 0.001")
  ...
FlintUnableError: failed to compute x / y in
    {Real numbers (arb, prec = 53)} for {x = 1}, {y = [+/- 1.01e-3]}
```

```
>>> RR("+/- 1e100").exp()
[+/- inf]
```

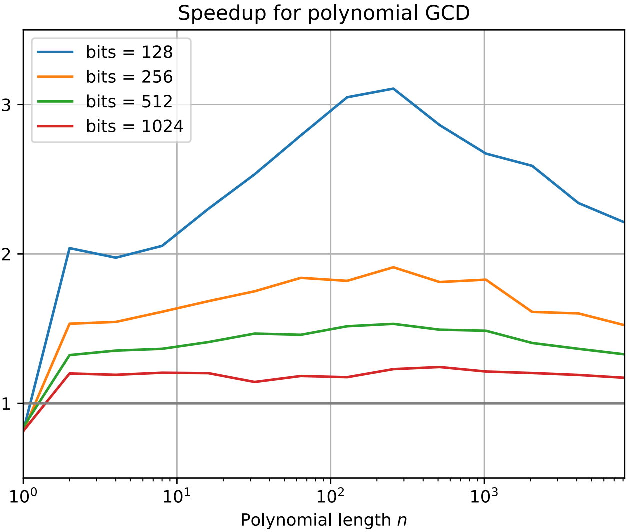
Specializations: few-word arithmetic

`mpn_mod`: $\mathbb{Z}/m\mathbb{Z}$ for n-limb moduli m

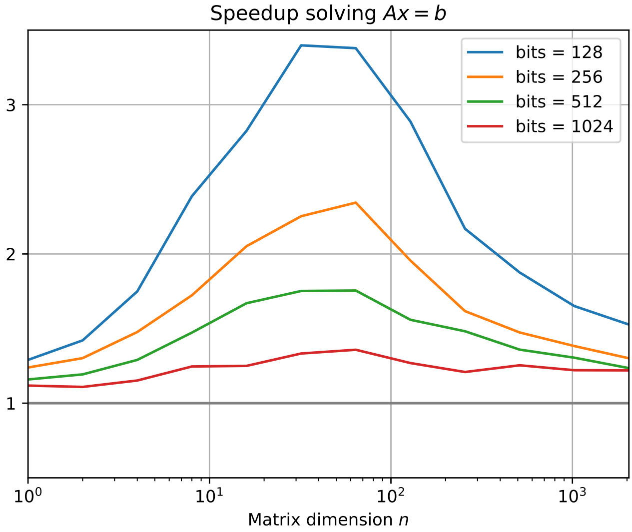
The modulus m is a parameter of the ring. Elements are represented by n contiguous limbs $a_0, \ldots, a_{n-1}$.

No pointers or memory allocation overhead; elements can be allocated on the stack, copied, and placed contiguously in vectors

Specialization: $\mathbb{Z}/m\mathbb{Z}$ for n-limb moduli m



Specialization: $\mathbb{Z}/m\mathbb{Z}$ for n-limb moduli m



Few-word floating-point numbers

`nfloat`: floating-point number with n-limb precision

- `nfloat64`
- `nfloat128`
- `nfloat192`
- `...`
- `nfloat1024`
- `...`

A vector of L elements $a, b, \ldots$ is simply a vector of $(n+2)L$ limbs:

$$\{a_{\text{exp}}, a_{\text{sgn}}, a_0, \ldots, a_{n-1}, b_{\text{exp}}, b_{\text{sgn}}, b_0, \ldots, b_{n-1}, \ldots\}$$

Don't bother with correct rounding: 2 ulps error is fine.

Example

Time in seconds to solve a random 100×100 linear system $Ax = b$.

prec	mpf	mpfr	arf	**nfloat**	dd/qd
64	0.015	0.013	0.00356	0.00221	-
128	0.0154	0.0183	0.00425	0.00253	0.00193
192	0.0163	0.0225	0.00921	0.0036	-
256	0.0177	0.0243	0.0101	0.00435	0.0223
512	0.0255	0.0311	0.0163	0.00943	-
1024	0.0551	0.0546	0.044	0.00278	-
2048	0.15	0.115	0.0961	0.082	-

Example

Time to isolate all the complex roots of $f \in \mathbb{Z}[x]$:

Polynomial	Degree	FLINT 3.1	FLINT 3.2-dev	Speedup
$x^{50} + (100x+1)^5$	50	0.278 s	0.102 s	2.73x
$W_{100}(x)$	100	1.30 s	0.52 s	2.50x
$T_{300}(x)$	150	3.75 s	1.44 s	2.60x
$\sum_{i=0}^{256} x^i / i!$	256	8.397 s	2.845 s	2.95x
$\Phi_{777}(x)$	432	28.0 s	0.65 s	43.1x
$B_{640}(x)$	640	114.1 s	20.2 s	5.65x
$\sum_{i=0}^{1000} (i+1)x^i$	1000	4134 s	4.31 s	959x

To do

- Similar optimizations for ball arithmetic
- Classical interval arithmetic
- Faster elementary functions (see paper with Joris van der Hoeven)
- Generic code for transcendental functions with guaranteed accuracy for any floating-point output format (partially implemented)
- Machine precision arithmetic