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

\texttt{flint\_mpn\_mul\_n} vs \texttt{mpn\_mul\_n}

![Graph showing speedup comparison between 8 threads and 1 thread](image)

- **x-axis**: $n$ (bits)
- **y-axis**: Speedup
- **Legend**:
  - Blue: 8 threads
  - Orange: 1 thread
Generics in FLINT 3: motivation

Original FLINT philosophy: one ring $\leftrightarrow$ one C type
- \texttt{fmpq} - $\mathbb{Q}$
- \texttt{arb} - $\mathbb{R}$
- \texttt{arb\_poly} - $\mathbb{R}[x]$

Drawbacks:
- 100 types $\times$ 100 methods $\approx$ 10,000 methods
- Hard to optimize versatile types (e.g. \texttt{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 $R[[x]]$
- $2 - 3x \, (\text{mod } x^3)$ is an exact element in $R[[x]] / \langle x^3 \rangle$
Two implementations of real numbers:

```python
>>> from flint_ctypes import *

>>> RR_ca("(1 + 1/3)^(1/2)"
1.15470 \{(2*a)/3 \text{ 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()
1.0000000000000000
The Arb-based implementation of $\mathbb{R}$ does not contain the element $\infty$ but admits the enclosure $(-\infty, +\infty)$.

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

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

```python
>>> RR("+/− 1e100").exp()
[+/− inf]
```
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 \( m \)
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 $100 \times 100$ linear system $Ax = b$.

<table>
<thead>
<tr>
<th>prec</th>
<th>mpf</th>
<th>mpfr</th>
<th>arf</th>
<th>nfloat</th>
<th>dd/qd</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.015</td>
<td>0.013</td>
<td>0.00356</td>
<td>0.00221</td>
<td>-</td>
</tr>
<tr>
<td>128</td>
<td>0.0154</td>
<td>0.0183</td>
<td>0.00425</td>
<td>0.00253</td>
<td>0.00193</td>
</tr>
<tr>
<td>192</td>
<td>0.0163</td>
<td>0.0225</td>
<td>0.00921</td>
<td>0.0036</td>
<td>-</td>
</tr>
<tr>
<td>256</td>
<td>0.0177</td>
<td>0.0243</td>
<td>0.0101</td>
<td>0.00435</td>
<td>0.0223</td>
</tr>
<tr>
<td>512</td>
<td>0.0255</td>
<td>0.0311</td>
<td>0.0163</td>
<td>0.00943</td>
<td>-</td>
</tr>
<tr>
<td>1024</td>
<td>0.0551</td>
<td>0.0546</td>
<td>0.044</td>
<td>0.00278</td>
<td>-</td>
</tr>
<tr>
<td>2048</td>
<td>0.15</td>
<td>0.115</td>
<td>0.0961</td>
<td>0.082</td>
<td>-</td>
</tr>
</tbody>
</table>
Time to isolate all the complex roots of $f \in \mathbb{Z}[x]$: 

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