Generic rings and numerical computation in FLINT

Fredrik Johansson

2024-05-23
Journées NuSCAP, LIP6, Paris

The NuSCAP vision

We want a system that allows calculating in $\mathbb{R}$ with three levels of fidelity:

1. Heuristic (e.g. using floating-point approximations)
2. Rigorous (e.g. using ball arithmetic) or exact (e.g. using number field arithmetic)
3. Certified (with a machine-checkable proof)

My goal is for FLINT to provide an efficient and reliable backend for levels 1 and 2.

A few comments about level 3 later.

State of the FLINT project

- Two recent workshops: Kaiserslautern (October 2023) and Bordeaux (March 2024)
- Arb was merged in FLINT 3.0
- Development of FLINT 3.2 is coming along nicely
- The current maintainers are myself and Albin Ahlbäck

Recent & upcoming features

- Theta functions in any dimension (Jean Kieffer)
- Sparse vectors and matrices (Kartik Venkatram)
  - `https://github.com/flintlib/flint/pull/1845`
- Assembly code for x86-64 and arm64 (Albin Ahlbäck)
- New vector-friendly types (FJ)
- More work on generic rings (FJ)
- Python interfaces
  - Revived and improved python-flint (Oscar Benjamin and others) - `https://github.com/flintlib/python-flint`
  - Preliminary Sage interface (Marc Mezzarobba) - `https://github.com/mezzarobba/flint_gr_sage`
  - Unofficial `flint_ctypes` included with FLINT

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 base types specialized for different applications

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

Examples

We have various faithful models of real numbers, with the same interface:

```
>>> 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]
```

Plus floating-point approximations:

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

Examples

Note: the Arb-based real field RR is actually a field. It 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]
```

Examples

```
>>> Mat(RR)([[1, 1], [1, 1]]).inv()
  ...
FlintDomainError: inv(x) is not an element of {Matrices (any shape)
    over Real numbers (arb, prec = 53)} for {x = [[1, 1],
    [1, 1]]}
```

```
>>> Mat(RR)([[1, 1], [1, "1 +/- 0.1"]]).inv()
  ...
FlintUnableError: failed to compute inv(x) in {Matrices (any shape)
    over Real numbers (arb, prec = 53)} for {x = [[1, 1],
    [1, [1e+0 +/- 0.101]]]}
```

Testing ring implementations

Operations in a ring R must satisfy certain laws, for example

$$(a+b)\cdot c = (a\cdot c) + (b\cdot c), \quad \forall a,b,c \in R.$$

Ideally, such properties would be formally verified. Lacking that ability, we can do randomized testing instead.

We also check correctness of generic algorithms (e.g. for linear algebra) by executing them over many randomly generated rings.

Testing rings

```
==============================================================================
Testing Real numbers (arb, prec = 64)
------------------------------------------------------------------------------
ctx_get_str ... PASS   (1 successful, 0 domain, 0 unable, 0 wrong, 0 cp
init/clear ... PASS   (1000 successful, 0 domain, 0 unable, 0 wrong, 0.
equal ... PASS   (1000 successful, 0 domain, 0 unable, 0 wrong, 0.001 c
zero_one ... PASS   (1000 successful, 0 domain, 0 unable, 0 wrong, 0.00
...
get_set_str ... PASS   (1000 successful, 0 domain, 0 unable, 0 wrong, 0
...
add: associative ... PASS   (1000 successful, 0 domain, 0 unable, 0 wro
add: commutative ... PASS   (1000 successful, 0 domain, 0 unable, 0 wro
add: aliasing ... PASS   (1000 successful, 0 domain, 0 unable, 0 wrong,
sub: equal neg add ... PASS   (1000 successful, 0 domain, 0 unable, 0 w
...
div: distributive ... PASS   (803 successful, 115 domain, 82 unable, 0
div: aliasing ... PASS   (740 successful, 174 domain, 94 unable, 0 wron
div: div then mul ... PASS   (806 successful, 124 domain, 70 unable, 0
div: mul then div ... PASS   (813 successful, 128 domain, 59 unable, 0
...
```

Testing approximate rings (work in progress)

An approximate (e.g. floating-point) ring provides the interface of a ring but the methods do not need to satisfy the ring properties.

To test an approximate ring R', we require a faithful implementation of R for reference.

For example, to test $R' =$ 128-bit floats $\approx \mathbb{R}$, we might use $R = \mathbb{R}$ represented by 256-bit balls.

What makes automation complicated is that the appropriate tolerance to check $x' \approx x$ varies with the operation.

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

FLINT originally had two implementations of $\mathbb{Z}/m\mathbb{Z}$

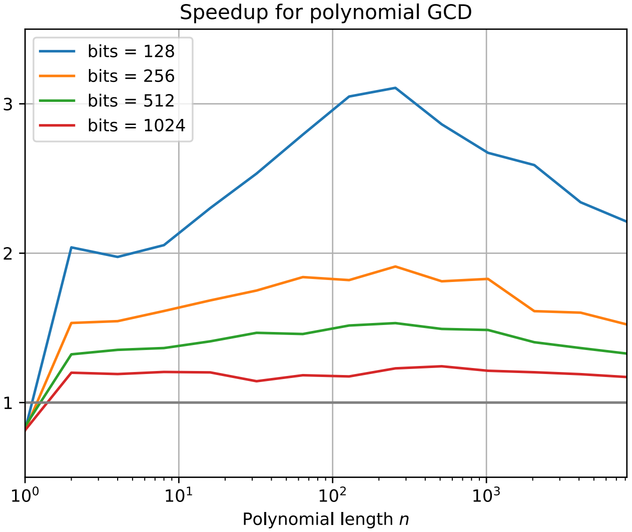
- `nmod` for 1-word moduli
- `fmpz_mod` for arbitrary moduli, with lots of overhead for few-word m

The new generics-based `mpn_mod` format stores elements modulo n-limb integers inline, without indirection or memory management. A vector of L elements $a, b, \ldots$ is simply a vector of nL limbs:

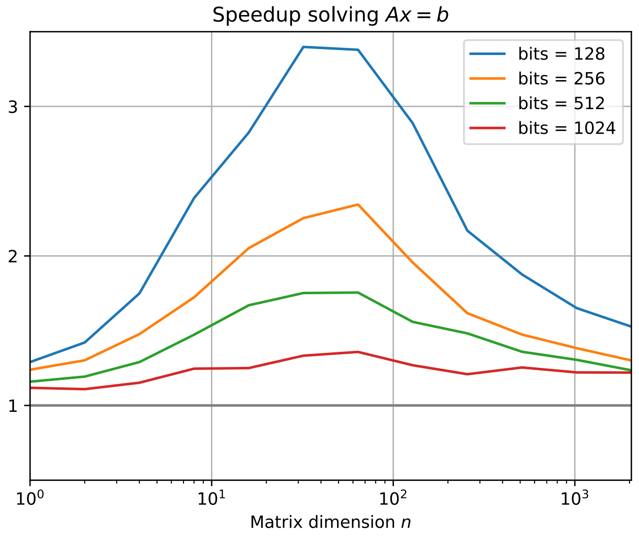
$$\{a_0, ..., a_{n-1}, b_0, ..., b_{n-1}, ...\}$$

This allows implementing basic operations much more efficiently than `fmpz_mod`.

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



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



Specialization: floating-point arithmetic (work in progress)

`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:

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

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

Specialization: floating-point arithmetic (work in progress)

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	-

Specialization: floating-point arithmetic (work in progress)

To do:

- Complex arithmetic
- Ball arithmetic
- Good matrix and polynomial multiplication
- Special functions (currently have wrappers around Arb)

Elementary functions (work in progress)

Joint work with Joris van der Hoeven (see our ARITH 2024 paper).

- CORDIC/BKM-style bitwise or *m*-bitwise argument reduction

$$\exp(x) = \exp\left(x - \sum_i \log(1 + k2^{-i})\right) \prod_i (1 + k2^{-i})$$

- Taylor series improvements

I estimate that one can save a factor 2-4 over current FLINT (Arb) implementations at any precision. Unfortunately, I only have prototype code so far.

What about the certified level?

Possible directions:

- Interfaces between FLINT and theorem provers
- Formal verification or specification of algorithms in FLINT
  - Correctness of algorithms for generic rings
  - Correctness of ring implementations
- Certificate-producing algorithms in FLINT
  - Example: FLINT's factorization code for $\mathbb{Z}[x]$ is extremely complex (using fp LLL, etc.), but could be modified to output a more easily checked certificate.[1] A possible application would be certificate-producing computations in $\overline{\mathbb{Q}}$.

[1] Davenport, Costa, Best and Carneiro, `https://people.bath.ac.uk/masjhd/Slides/JHDatDagstuhlSeminar23401.pdf`

Some more rings I'd like to see in/using FLINT

- More models of $\mathbb{R}$ and $\mathbb{C}$ (including intervals and balls in different formats)
- Analyticity-checking versions of $\mathbb{C}$
- Functions $\mathbb{R} \to \mathbb{R}$ and $\mathbb{C} \to \mathbb{C}$ (Taylor models, Chebyshev models, meromorphic function fields, holonomic functions)

Discussion