Arb: a C library for ball arithmetic

Fredrik Johansson

RISC-Linz

ISSAC 2013

Supported by the Austrian Science Fund (FWF) grant Y464-N18

Ball arithmetic

$x \in \mathbb{R}$ is represented by a ball $B = [m - \epsilon, m + \epsilon]$ such that $x \in B$

Floating-point: 3.14159265358979323

Interval: $[3.14159265358979323, 3.14159265358979324]$

Ball: $3.14159265358979323 \; [\pm] \; 8.5 \times 10^{-18}$

Implementations: Mathemagix (J. van der Hoeven), iRRAM (N. Müller)

Arb

- Library for ball arithmetic (ARB = Arbitrary-precision Real Balls)
- Written in ANSI C
- GMP-style interface
- Open source (GPL v2+)
- Code: `https://github.com/fredrik-johansson/arb/`
- Documentation: `http://fredrikj.net/arb/`
- About 35,000 lines of code, developed since April 2012
- Extensively documented and tested ($\sim$ 50% test code)

Extends FLINT, `http://flintlib.org`
(W. Hart, D. Harvey, S. Pancratz, F. Johansson, A. Novocin, and many contributors)

Dependencies

Arb	Arithmetic, polynomials, power series, matrices, special functions over $\mathbb{R}, \mathbb{C}$
FLINT	Arithmetic, polynomials, power series, matrices, special functions over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}, \mathbb{Z}_p$, support and test code
MPFR	Supporting functions, some special functions over $\mathbb{R}$, reference for testing floating-point arithmetic
GMP/MPIR	Low-level arbitrary-precision arithmetic

Types

`fmpr_t`

floating-point numbers $\mathbb{R}_D = \mathbb{Z} \times 2^{\mathbb{Z}} \cup \{-\infty, +\infty, \mathrm{NaN}\}$

`fmprb_t`

real numbers implemented as balls
$\mathbb{R}_B = \{[m - r, m + r] : m, r \in \mathbb{R}_D, r \geq 0\}$

`fmpcb_t`

complex numbers in rectangular form $\mathbb{C}_B = \mathbb{R}_B[i]$

`fmprb_poly_t`, `fmpcb_poly_t`

polynomials (and truncated power series) over $\mathbb{R}_B$, $\mathbb{C}_B$

`fmprb_mat_t`, `fmpcb_mat_t`

matrices over $\mathbb{R}_B$, $\mathbb{C}_B$

Representation of floating-point numbers

Floating-point number: $\mathrm{man} \times 2^{\mathrm{exp}}$

```
typedef struct
{
    fmpz man;       // FLINT integers, single words
    fmpz exp;       // up to 30/62 bits, dynamic allocation
}
fmpr_struct;
```

- ✓ Efficient for error bounds
- ✓ Efficient for integer coefficients (polynomials, binary splitting)
- ✗ Adds overhead at precisions between $31/63$ and ~ 1000 bits
- ✗ Conversions required for calling MPFR functions

Cost of arithmetic

Time compared to MPFR for
* floating-point multiplication $a \times b$
** ball multiplication $(a \pm \varepsilon_1) \times (b \pm \varepsilon_2)$

Bits	mpfr_mul*	fmpr_mul*	fmprb_mul**
32	1.0	0.6	2.3
128	1.0	1.4	2.7
512	1.0	0.9	1.4
2048	1.0	1.1	1.2
8192	1.0	1.2	1.2
32768	1.0	1.2	1.2
131072	1.0	1.1	1.1
524288	1.0	1.0	1.0

Special functions

Goals:

- Support evaluation on all of $\mathbb{R}$ and $\mathbb{C}$
- Support special functions of power series
- Use provably correct error bounds
- Match or beat speed of arbitrary-precision floating-point libraries
- Investigate new algorithms and implementation approaches

Implemented special functions

- Elementary functions over $\mathbb{R}$, $\mathbb{C}$
- Gamma function $\Gamma(z)$, $\log \Gamma(z)$, $\psi(z)$, $z \in \mathbb{R}, \mathbb{C}$
- Hurwitz zeta function $\zeta(s, a)$, $s, a \in \mathbb{C}$ (also derivatives w.r.t. s)
- Hypergeometric series $\sum_{k=0}^{\infty} T(k)$, $T(k+1)/T(k) \in \mathbb{Q}(k)$ (asymptotically fast evaluation using binary splitting, with automatic error bounding)
- Bernoulli numbers (exact or approximate)
- The partition function $p(n)$

Implementation of elementary functions

- MPFR is used for real sqrt, exp, log, sin, cos, atan
  - Error propagation via derivatives
  - Arbitrary-precision exponents (using functional equations)
  - Capped evaluation time (e.g. $\cos(2^{10^{10}}) \to [-1, 1]$)
- Faster code for special values
  - Hypergeometric series for $e, \pi, \log 2, \log 10$
  - Newton iteration for $\cos(a\pi/b)$
- Complex functions use decomposition into real and complex parts
  - Asymptotically stable formulas for tangent, cotangent, ...

Timings for high-precision special functions

MPFR 3.1.1	Pari 2.5.3	Mathematica 8.0	Arb
10^6 digits of $\gamma = 0.577\ldots$ – binary splitting (Brent-McMillan)			
93	$>$ 3600	30	**18**
10^5 digits of $\cos(\pi/31)$ – minimal polynomial root refinement			
6.1	42	12	**0.48**
10^5 digits of ${}_3F_2(\frac{1}{2}, \frac{1}{3}; \frac{1}{4}, \frac{1}{5}, \frac{1}{6}; \frac{1}{7})$ – generic binary splitting			
-	-	1396	**0.45**

Time in seconds

Timings for high-precision special functions

MPFR 3.1.1	Pari 2.5.3	Mathematica 8.0	Arb
10^4 digits of $\Gamma(\sqrt{2})$ – Stirling's series			
60	1.9 (233)	13	**0.21** (**1.3**)
10^4 digits of $\Gamma(\sqrt{2} + i\sqrt{3})$ – Stirling's series			
-	2.9 (235)	5.8 (44)	**0.67** (**1.7**)
10^4 digits of $\zeta(1/2 + 1000i)$ – Euler-Maclaurin summation			
-	24 (1571)	672	**22** (**25**)
10^3 digits of $\zeta(1 + 2i, 3 + 4i)$ – Euler-Maclaurin summation			
-	-	2.4	**0.38**

Time in seconds for repeated evaluation (first evaluation)

Timings for the gamma function

Time in seconds for repeated evaluation (first evaluation) of $\Gamma(x)$, $x = \sqrt{2}$.

Digits	Pari/GP	MPFR	Mathematica	Arb
30	**0.000024**	0.000090	0.000070	0.000032
100	**0.000068**	0.00036	0.00020	0.000090
300	0.00039	0.0028	0.00080	**0.00031**
1000	0.0046	0.046	0.058	**0.0021**
3000	0.12 (6.5)	1.2	0.76	**0.018** (**0.080**)
10000	1.9 (233)	60	13	**0.21** (**1.3**)
30000	13 (6154)	2680	186	**2.4** (**18**)

Improvements from:

- Bernoulli numbers by vector evaluation of $\zeta(n)$
- Argument reduction $x(x+1)\cdots(x+r-1)$ using rectangular splitting

The integer partition function

$$p(n) = \sum_{k=1}^{\infty} \frac{\sqrt{k}\, A_k(n)}{\pi\sqrt{2}} \frac{d}{dn} \left(\frac{\sinh \frac{\pi}{k} \sqrt{\frac{2}{3}\left(n - \frac{1}{24}\right)}}{\sqrt{n - \frac{1}{24}}} \right)$$

$A_k(n)$ is a certain sum involving $2k$-th roots of unity.

n	Mathematica 8.0	FLINT 2.3*	Arb**
10^6	0.328 s	**0.00147 s**	0.00478 s
10^9	23.7 s	**0.142 s**	0.181 s
10^{12}	2458 s	**11.32 s**	11.50 s
10^{15}	307810 s	1109 s	**1097 s**
10^{18}		66738 s	**57102 s**

* using MPFR + hardware doubles (with incomplete error bounds)
** using ball arithmetic throughout to provably determine $p(n)$

Polynomials

Fast multiplication uses the FLINT implementation of $\mathbb{Z}[x]$ (Schönhage-Strassen FFT by Bill Hart)

Fast operations based on multiplication:

- Division with remainder (Newton iteration)
- Composition (divide and conquer)
- Power series division, exp, log (Newton iteration)
- Power series composition (Brent-Kung)
- Multipoint evaluation and interpolation (product trees)

Roots:

- Simple complex root isolation (Durand-Kerner + verification)
- Asymptotically fast real root polishing (Newton iteration)

Multiplying via $\mathbb{Z}[x]$

$P = 31415.9 + 2718.28x + 0.141421x^2 + 1.73205x^3$

- Exact: $31415900000 + 2718280000x + 141421x^2 + 1732050x^3$
  - ✓ Always accurate
  - ✗ Sometimes slow
- Truncating: $314159 + 27182x + 0x^2 + 1x^3$
  - ✓ Always fast
  - ✗ Sometimes inaccurate

A refinement:

- blockwise: $(3141590 + 271828x) + (141421 + 1732050x)x^2$

Further improvements are discussed in (van der Hoeven, 2008).

Implementations of fast polynomial multiplication

- MPFRCX (A. Enge)
  - Floating-point Toom-Cook and FFT, no error bounds
- Mathemagix (J. van der Hoeven)
  - $\mathbb{R}[x] \to \mathbb{Z}[x] \to \mathbb{Z}$ (Kronecker substitution)
  - Truncating multiplication, scaling to improve accuracy, error bounds using Newton polygons
- Arb / FLINT
  - $\mathbb{R}[x] \to \mathbb{Z}[x]$ (Schönhage-Strassen) or $\to \mathbb{Z}$ (Kronecker)
  - Default: accurate multiplication (blockwise, error bounds using $O(n^2)$ classical multiplication)
  - Optional: truncating multiplication (naive error bounds)

Polynomial multiplication cost: Time (nanoseconds) / (Length · Bits)
All coefficients have similar magnitude.

Length	Bits	MPFRCX	Mathemagix	Arb (block)	Arb (trunc.)
10	100	17.2	173.0	**13.3**	11.5
10	1000	**3.9**	34.3	5.0	4.3
10	10000	**5.5**	56.9	9.4	9.2
10	100000	13.7	201.0	**12.4**	12.4
100	100	65.6	171.0	**18.4**	10.5
100	1000	14.0	31.9	**6.0**	5.2
100	10000	18.3	30.0	**9.8**	9.7
100	100000	43.3	70.2	**13.8**	13.5
1000	100	155.0	182.0	**50.2**	17.4
1000	1000	34.1	35.8	**9.4**	6.2
1000	10000	50.8	37.9	**11.3**	10.6
1000	100000	132.7	60.1	**14.4**	14.4
10000	100	324.0	**204.0**	307.0	20.0
10000	1000	74.5	**49.2**	50.5	21.7
10000	10000	109.6	40.2	**18.4**	15.5
10000	100000	282.5	63.0	**16.3**	16.0

Checking Li's criterion

Li's criterion

Let $\xi(s) = (s-1)\pi^{-s/2}\Gamma\left(1+\frac{1}{2}s\right)\zeta(s)$. The Riemann hypothesis is equivalent to the positivity for all $n > 0$ of the coefficients λ_n defined by $\log \xi\left(z/(z-1)\right) = \sum_{n=0}^{\infty} \lambda_n z^n$.

We prove positivity of the first 10,000 coefficients by evaluation.
The polynomial multiplication algorithm has a huge influence.

Multiplication algorithm:	Truncating	Classical	Blockwise
Working precision:	100000 bits	13000 bits	13000 bits
Derivatives of $\zeta(s)$	147180 s	1242 s	1272 s
Series logarithm	56 s	2760 s	8.3 s
Derivatives of $\log \Gamma(s)$	781 s	3.4 s	3.4 s
Series composition	1994 s	7971 s	185 s
Total	150011 s	11976 s	1469 s

Future development goals

- Optimize low precision (expecting $\sim$ 2x speedup)
- Extend support for hypergeometric (maybe also holonomic) functions
- Add other special functions
- Generally improve polynomials and matrices
- Parallel algorithms
- Interfaces (C++, Python, Sage, etc.)