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

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

Floating-point: 3.14159265358979323

Interval: [3.14159265358979323, 3.14159265358979324]

Ball: 3.14159265358979323 ± 8.5 × 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 (≈ 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 \( \mathbb{R}_D = \mathbb{Z} \times 2^\mathbb{Z} \cup \{-\infty, +\infty, \text{NaN}\} \)

`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: $\text{man} \times 2^{\text{exp}}$

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 $\sim 1000$ bits
✗ Conversions required for calling MPFR functions
Cost of arithmetic

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>mpfr_mul*</th>
<th>fmpr_mul*</th>
<th>fmprb_mul**</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1.0</td>
<td>0.6</td>
<td>2.3</td>
</tr>
<tr>
<td>128</td>
<td>1.0</td>
<td>1.4</td>
<td>2.7</td>
</tr>
<tr>
<td>512</td>
<td>1.0</td>
<td>0.9</td>
<td>1.4</td>
</tr>
<tr>
<td>2048</td>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>8192</td>
<td>1.0</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>32768</td>
<td>1.0</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>131072</td>
<td>1.0</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>524288</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
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. $\cos(2^{10^{10}}) \rightarrow [-1, 1]$)

- 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

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

Time in seconds
### Timings for high-precision special functions

<table>
<thead>
<tr>
<th>MPFR 3.1.1</th>
<th>Pari 2.5.3</th>
<th>Mathematica 8.0</th>
<th>Arb</th>
</tr>
</thead>
</table>

10⁴ digits of $\Gamma(\sqrt{2})$ – Stirling’s series
60         1.9 (233)       13   **0.21 (1.3)**

10⁴ digits of $\Gamma(\sqrt{2} + i\sqrt{3})$ – Stirling’s series
-          2.9 (235)       5.8 (44)  **0.67 (1.7)**

10⁴ digits of $\zeta(1/2 + 1000i)$ – Euler-Maclaurin summation
-          24 (1571)       672   **22 (25)**

10³ 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 $\Gamma(x)$, $x = \sqrt{2}$.

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

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

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

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>Mathematica 8.0</th>
<th>FLINT 2.3*</th>
<th>Arb**</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^6 )</td>
<td>0.328 s</td>
<td>0.00147 s</td>
<td>0.00478 s</td>
</tr>
<tr>
<td>( 10^9 )</td>
<td>23.7 s</td>
<td>0.142 s</td>
<td>0.181 s</td>
</tr>
<tr>
<td>( 10^{12} )</td>
<td>2458 s</td>
<td>11.32 s</td>
<td>11.50 s</td>
</tr>
<tr>
<td>( 10^{15} )</td>
<td>307810 s</td>
<td>1109 s</td>
<td><strong>1097 s</strong></td>
</tr>
<tr>
<td>( 10^{18} )</td>
<td>66738 s</td>
<td>57102 s</td>
<td></td>
</tr>
</tbody>
</table>

* 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)
  - \( \mathbb{R}[x] \rightarrow \mathbb{Z}[x] \rightarrow \mathbb{Z} \) (Kronecker substitution)
  - Truncating multiplication, scaling to improve accuracy, error bounds using Newton polygons

- **Arb / FLINT**
  - \( \mathbb{R}[x] \rightarrow \mathbb{Z}[x] \) (Schönhage-Strassen) or \( \rightarrow \mathbb{Z} \) (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 \cdot Bits)

All coefficients have similar magnitude.

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

\[
\log \xi \left(\frac{z}{z - 1}\right) = \sum_{n=0}^{\infty} \lambda_n z^n.
\]

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

<table>
<thead>
<tr>
<th>Multiplication algorithm:</th>
<th>Truncating</th>
<th>Classical</th>
<th>Blockwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working precision:</td>
<td>100000 bits</td>
<td>13000 bits</td>
<td>13000 bits</td>
</tr>
<tr>
<td>Derivatives of ( \zeta(s) )</td>
<td>147180 s</td>
<td>1242 s</td>
<td>1272 s</td>
</tr>
<tr>
<td>Series logarithm</td>
<td>56 s</td>
<td>2760 s</td>
<td>8.3 s</td>
</tr>
<tr>
<td>Derivatives of ( \log \Gamma(s) )</td>
<td>781 s</td>
<td>3.4 s</td>
<td>3.4 s</td>
</tr>
<tr>
<td>Series composition</td>
<td>1994 s</td>
<td>7971 s</td>
<td>185 s</td>
</tr>
<tr>
<td>Total</td>
<td>150011 s</td>
<td>11976 s</td>
<td>1469 s</td>
</tr>
</tbody>
</table>
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.)