Arb: a C library for ball arithmetic

Fredrik Johansson *
RISC
Johannes Kepler University
4040 Linz, Austria
fjohanss@risc.jku.at

1 Introduction
Arb is a new open source C library for provably correct arbitrary-precision numerics, extending FLINT (which provides fast arithmetic over various exact rings) to the real and complex numbers. Following the example of iRRAM and Mathemagix, Arb performs automatic error propagation using ball arithmetic (not to be confused with heuristic significance arithmetic as used e.g. in Mathematica). This gives performance close to floating-point arithmetic such as provided by MPFR, while avoiding the cost at high precision of endpoint-based interval arithmetic as provided for instance by MPFI.

One of our motivations for developing a new library has been to provide a low-level, low-overhead interface, and our implementation differs from others in some technical aspects. Arb also provides fast polynomial arithmetic, to our knowledge only available in Mathemagix and without error control in MPRFCX, as well as matrix arithmetic. Finally, Arb implements some special functions that have been absent from arbitrary-precision interval software, with performance that compares favorably to available nonrigorous implementations. The presentation covers implementation details and shows some benchmarks.

2 Feature overview
Arb provides the following types:

- **fmpr_t**: floating-point real 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 \( \mathbb{R}_B, \mathbb{C}_B \)

Each type has a set of associated methods for memory management, conversions, arithmetic and special functions, with an interface resembling that of FLINT. For example, the power series multiplication \( a \leftarrow b \times c \mod x^n \) with rounding to \( \text{prec} \) bits, where \( a, b, c \) are of type \texttt{fmprb_poly_t}, is written as:

\[
\texttt{fmprb_poly_mullow(a, b, c, n, prec)}
\]

Polynomial methods have corresponding “underscore” versions that act directly on coefficient arrays, reducing overhead and giving more control over memory allocation and copying (like the \texttt{mpn} layer of GMP):

\[
\texttt{fmprb_poly_mullow(a->coeffs, b->coeffs, b->length, c->coeffs, c->length, n, prec)}
\]
3 Representation of numbers

Arb does not directly base its arithmetic on MPFR (but does call MPFR for a few operations, and the test suite extensively verifies correctness against MPFR). MPFR attaches a precision to each variable, and allocates memory for a full-precision number even if only a few bits are used. In Arb, the precision is always passed as an argument to each function; the components of an \texttt{fmpr} are FLINT integers, and can grow dynamically. A mantissa or exponent with at most 62 bits (30 bits on a 32-bit system) is particularly efficient, as it takes up a single word in the \texttt{fmpr} struct without allocating memory on the heap.

We have found this approach convenient for mixed-precision algorithms and particularly valuable for computations involving integer coefficients of variable size (such as binary splitting and various polynomial operations). The same type also works well for low-precision arithmetic such as error bound calculations. The drawback is some overhead at precisions up to a few hundred digits, although experiments suggest that this overhead could be reduced with further implementation effort.

<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>

Table 1: Time relative to MPFR of floating-point and ball multiplication. The difference below approximately 1000 bits results from implementation overhead, and the 10% – 20% difference around $10^3$ - $10^5$ bits is due to MPFR using the mulhigh algorithm.

An \texttt{fmprb} consists of a midpoint and a radius, both of type \texttt{fmpr}. Radius operations use a predefined precision (30 bits). Midpoint arithmetic is always carried out at the requested working precision. It would be more efficient to round midpoints to the accuracy indicated by the radius, though such a normalization naturally can be performed explicitly, and the present convention is sometimes useful for detecting when the computed error bound greatly overshoots the actual numerical error.

Complex numbers are represented as pairs of real balls. This seems preferable to a complex midpoint with a single radius, for reasons of convenience, and it is frequently useful to track whether either the real or imaginary part is exact. Similarly, polynomials and matrices are represented as arrays of coefficients to maximize flexibility. Where a different data order is required, temporary copies are relatively cheap since the base \texttt{fmpr} type takes up only two words and usually only needs to be copied shallowly.

4 Special functions

Except for some special cases, the elementary functions in Arb call the MPFR implementations of exp, log, sin, cos and atan (our future plan is to develop faster implementations for precisions up to a few thousand digits), using function derivatives to bound propagated errors. Care has been taken to ensure numerically satisfactory behavior on the whole complex plane, for example when evaluating $\tan(x + yi)$ for large $|y|$. Extremely large numbers are handled specially: we allow arbitrary-precision exponents, and we restrict the internal working precision allowed for argument reduction to a small multiple of the requested precision. For example, an attempt to evaluate $\cos(2^{10^{10}})$ quickly returns a crude bounding interval (e.g. $[-1, 1]$) unless the precision is set in the hundreds of millions of digits. This makes worst-case evaluation
time at a given precision predictable and avoids unnecessary stalls caused by tiny terms that might not
even contribute to the final result, particularly aiding “black-box” use in computer algebra settings.

Interval software has historically been limited to the elementary functions and some special functions
of a real variable, while software with good support for special functions (e.g. [10], [5]) has not guaranteed
correctness. We wish to improve this situation. As of the current version, Arb provides Bernoulli numbers,
the Hurwitz zeta function $\zeta(s, a)$ and its derivatives with respect to $s$ for complex $s$ and $a$, and the gamma
and digamma functions for real and complex arguments. The implementations are tuned for different sizes
and precisions, incorporating many optimizations. Arb also contains code for binary splitting evaluation of
generic rational hypergeometric series with automatic error bounding, used for evaluation of mathematical
constants, as well as code for rigorous polynomial root refinement, used for some algebraic numbers.

<table>
<thead>
<tr>
<th>Evaluation</th>
<th>Digits</th>
<th>MPFR 3.1.1</th>
<th>Pari/GP 2.5.3</th>
<th>Mathematica 8.0</th>
<th>Arb</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: $\gamma$ (Euler’s constant)</td>
<td>$10^6$</td>
<td>93 s</td>
<td>&gt; 1 h</td>
<td>30 s</td>
<td>18 s</td>
</tr>
<tr>
<td>B: $\cos(\pi/31)$</td>
<td>$10^5$</td>
<td>6.1 s</td>
<td>42</td>
<td>12 s</td>
<td>0.48 s</td>
</tr>
<tr>
<td>C: $3F_2\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{5}, \frac{1}{5}; \frac{1}{7}\right)$</td>
<td>$10^5$</td>
<td>n/a</td>
<td>n/a</td>
<td>1396 s</td>
<td>0.45 s</td>
</tr>
<tr>
<td>D: $\Gamma(\sqrt{2})$</td>
<td>$10^4$</td>
<td>60 s</td>
<td>1.9 (233) s</td>
<td>13 s</td>
<td>0.21 (1.3) s</td>
</tr>
<tr>
<td>E: $\Gamma(\sqrt{2} + i\sqrt{3})$</td>
<td>$10^4$</td>
<td>n/a</td>
<td>2.9 (235) s</td>
<td>5.8 (44) s</td>
<td>0.67 (1.7) s</td>
</tr>
<tr>
<td>F: $\zeta(1/2 + 1000i)$</td>
<td>$10^4$</td>
<td>n/a</td>
<td>24 (1571) s</td>
<td>672 s</td>
<td>22 (25) s</td>
</tr>
<tr>
<td>G: $\zeta(1 + 2i, 3 + 4i)$</td>
<td>$10^3$</td>
<td>n/a</td>
<td>n/a</td>
<td>2.4 s</td>
<td>0.38 s</td>
</tr>
</tbody>
</table>

Table 2: Special function timings, measuring repeated calls with the initial call inside
parentheses. Algorithms in Arb: A) binary splitting B) minimal polynomial root refinement
C) generic binary splitting D-E) Stirling’s series F-G) Euler-Maclaurin summation.

5 Polynomials and power series

Polynomial operations are implemented in an asymptotically fast way by reducing to multiplication using
standard techniques such as Newton iteration for division, series logarithm and series exponential, divide-
and-conquer for composition [4], rectangular splitting for power series composition, and product trees for
fast multipoint evaluation and interpolation. We have implemented three algorithms for multiplication
in $\mathbb{R}[x]$: classical, sloppy, and blockwise. The latter two translate to $\mathbb{Z}[x]$ and call FLINT (which uses
classical, Karatsuba, Kronecker substitution, and Schönhage-Strassen FFT multiplication).

The sloppy algorithm cuts off the coefficients of each input polynomial $\text{prec}$ bits below the top bit
of the polynomial as a whole, performs a single multiplication over $\mathbb{Z}[x]$, and bounds errors using max
norms. This is fast, and numerically satisfactory if all coefficients have the same magnitude, but not used
by default due to the poor numerical stability for polynomials with coefficients of varying magnitude.

The blockwise algorithm splits the input polynomials into blocks of similarly-sized coefficients and
multiplies each pair of blocks exactly in $\mathbb{Z}[x]$. In the worst case, this degenerates to multiplication of $1 \times 1$
blocks equivalent to classical multiplication. In the typical case, it only performs slightly worse than the
sloppy multiplication. Accurate per-coefficient error bounds are computed using an $O(n^2)$ loop running
over exponents. The algorithm could be improved further using scaling and by discarding parts of the
inputs that do not contribute to the result, as discussed in [11].

We illustrate the importance of polynomial arithmetic that is both fast and numerically stable. Letting
$\xi(s) = (s-1)\pi^{-s/2}\Gamma\left(1 + \frac{s}{2}\right)\zeta(s)$, Li’s criterion [6] states that the Riemann hypothesis is equivalent to the
positivity for all $n > 0$ of the coefficients $\lambda_n$ defined by $\log \xi(z/(z-1)) = \sum_{n=0}^{\infty} \lambda_n z^n$. We prove positivity
of the first 10,000 coefficients by evaluation. This requires derivatives of $\zeta(s)$, a series logarithm, derivatives
of $\log \Gamma(s)$, and a series composition with $z/(z-1)$. In this example, the final composition catastrophically
magnifies the error bounds if sloppy multiplication is used, making a precision of nearly $10n$ bits necessary. With classical multiplication, about $1.3n$ bits suffice, speeding up the the zeta function evaluation, but the subsequent power series operations now dominate. Blockwise multiplication allows using the same precision as with classical multiplication, and the power series operations only take a fraction of the time.

<table>
<thead>
<tr>
<th></th>
<th>Sloppy</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>Zeta</td>
<td>147180 s</td>
<td>1242 s</td>
<td>1272 s</td>
</tr>
<tr>
<td>Logarithm</td>
<td>56 s</td>
<td>2760 s</td>
<td>8.3 s</td>
</tr>
<tr>
<td>Gamma</td>
<td>781 s</td>
<td>3.4 s</td>
<td>3.4 s</td>
</tr>
<tr>
<td>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>

Table 3: Precisions and timings for computing the Li coefficients $\lambda_n$ up to $n = 10000$ with correctly determined signs, using three different multiplication algorithms.

References


