Arb: a C library for ball arithmetic

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

1 Introduction

Arb [1] is a new open source C library for provably correct arbitrary-precision numerics, extending FLINT [3] (which provides fast arithmetic over various exact rings) to the real and complex numbers. Following the example of iRRAM [7] and Mathemagix [13], Arb performs automatic error propagation using ball arithmetic [12] (not to be confused with heuristic significance arithmetic as used e.g. in Mathematica [9]). This gives performance close to floating-point arithmetic such as provided by MPFR [2] while avoiding the cost at high precision of endpoint-based interval arithmetic as provided for instance by MPFI [8].

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 MPFRCX [1], 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 $a \leftarrow b \times c \bmod x^n$ with rounding to *prec* bits, where a, b, c are of type `fmprb_poly_t`, is written as:

```
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 `mpn` layer of GMP):

```
_fmprb_poly_mullow(a->coeffs, b->coeffs, b->length, c->coeffs, c->length, n, prec)
```

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

[1] `http://fredrikj.net/arb/` (Arb is licensed GNU GPL version 2 or later)

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 `fmpr_t` 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 `fmpr_t` 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.

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

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 `fmprb_t` consists of a midpoint and a radius, both of type `fmpr_t`. 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 `fmpr_t` 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.

Evaluation	Digits	MPFR 3.1.1	Pari/GP 2.5.3	Mathematica 8.0	Arb
A: γ (Euler's constant)	10^6	93 s	> 1 h	30 s	18 s
B: $\cos(\pi/31)$	10^5	6.1 s	42	12 s	0.48 s
C: ${}_3F_2(\frac{1}{2},\frac{1}{3};\frac{1}{4},\frac{1}{5},\frac{1}{6};\frac{1}{7})$	10^5	n/a	n/a	1396 s	0.45 s
D: $\Gamma(\sqrt{2})$	10^4	60 s	1.9 (233) s	13 s	0.21 (1.3) s
E: $\Gamma(\sqrt{2}+i\sqrt{3})$	10^4	n/a	2.9 (235) s	5.8 (44) s	0.67 (1.7) s
F: $\zeta(1/2+1000i)$	10^4	n/a	24 (1571) s	672 s	22 (25) s
G: $\zeta(1+2i,3+4i)$	10^3	n/a	n/a	2.4 s	0.38 s

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 *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 1×1 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 $\xi(s) = (s-1)\pi^{-s/2}\Gamma\left(1+\frac{1}{2}s\right)\zeta(s)$, Li's criterion [6] states that 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. 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.

	Sloppy	Classical	Blockwise
Working precision	100000 bits	13000 bits	13000 bits
Zeta	147180 s	1242 s	1272 s
Logarithm	56 s	2760 s	8.3 s
Gamma	781 s	3.4 s	3.4 s
Composition	1994 s	7971 s	185 s
Total	150011 s	11976 s	1469 s

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

References

[1] A. Enge. MPFRCX: a library for univariate polynomials over arbitrary precision real or complex numbers, 2012. `http://www.multiprecision.org/index.php?prog=mpfrcx`.

[2] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007. `http://mpfr.org`.

[3] W. B. Hart. Fast Library for Number Theory: An Introduction. In *Proceedings of the Third international congress conference on Mathematical software*, ICMS'10, pages 88–91, Berlin, Heidelberg, 2010. Springer-Verlag. `http://flintlib.org`.

[4] W. B. Hart and A. Novocin. Practical divide-and-conquer algorithms for polynomial arithmetic. In *Computer Algebra in Scientific Computing*, pages 200–214. Springer, 2011.

[5] F. Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic, version 0.17*, 2011. `http://mpmath.org`.

[6] Xian-Jin Li. The positivity of a sequence of numbers and the Riemann Hypothesis. *Journal of Number Theory*, 65(2):325–333, 1997.

[7] N. Müller. The iRRAM: Exact arithmetic in C++. In *Computability and Complexity in Analysis*, pages 222–252. Springer, 2001. `http://irram.uni-trier.de`.

[8] N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic library and the MPFI library. *Reliable Computing*, 11(4):275–290, 2005. `http://perso.ens-lyon.fr/nathalie.revol/software.html`.

[9] M. Sofroniou and G. Spaletta. Precise numerical computation. *Journal of Logic and Algebraic Programming*, 64(1):113–134, 2005.

[10] The PARI Group, Bordeaux. *PARI/GP, version 2.5.3*, 2012. `http://pari.math.u-bordeaux.fr`.

[11] J. van der Hoeven. Making fast multiplication of polynomials numerically stable. Technical Report 2008-02, Université Paris-Sud, Orsay, France, 2008.

[12] J. van der Hoeven. Ball arithmetic. Technical report, HAL, 2009. `http://hal.archives-ouvertes.fr/hal-00432152/fr/`.

[13] J. van der Hoeven, G. Lecerf, B. Mourrain, P. Trébuchet, J. Berthomieu, D. N. Diatta, and A. Mantzaflaris. Mathemagix: the quest of modularity and efficiency for symbolic and certified numeric computation? *ACM Communications in Computer Algebra*, 45(3/4):186–188, January 2012. `http://mathemagix.org`.