Faster arbitrary-precision dot product and matrix multiplication

Fredrik Johansson
LFANT – Inria Bordeaux
Talence, France
fredrik.johansson@gmail.com

Abstract—We present algorithms for real and complex dot product and matrix multiplication in arbitrary-precision floating-point and ball arithmetic. A low-overhead dot product is implemented on the level of GMP limb arrays; it is about twice as fast as previous code in MPFR and Arb at precision up to several hundred bits. Up to 128 bits, it is 3-4 times as fast, costing 20-30 cycles per term for floating-point evaluation and 40-50 cycles per term for balls. We handle large matrix multiplications even more efficiently via blocks of scaled integer matrices. The new methods are implemented in Arb and significantly speed up polynomial operations and linear algebra.

Index Terms—arbitrary-precision arithmetic, ball arithmetic, dot product, matrix multiplication

I. INTRODUCTION

The dot product and matrix multiplication are core building blocks for many numerical algorithms. Our goal is to optimize these operations in real and complex arbitrary-precision arithmetic. We treat both floating-point arithmetic and ball arithmetic [21] in which errors are tracked rigorously using midpoint-radius [m±r] intervals. Our implementations are part of the open source (LGPL) Arb library [14] (http://arblib.org/) as of version 2.16.

In this work, we only consider the traditional CPU setting, relying on GMP [7] for low-level arithmetic on mantissas represented by arrays of 32-bit or 64-bit words (limbs). This is the format used in MPFR [5] as well as Arb. The benefit is flexibility (we handle mixed precision from few bits to billions of bits); the drawback is high bookkeeping overhead and limited vectorization opportunities. In “medium” precision (up to several hundred bits), floating-point expansions such as double-double and quad-double numbers offer higher performance on modern hardware with wide SIMD floating-point units (including GPUs) [11], [15], [22]. However, such formats typically give up some flexibility (having a limited exponent range, usually assuming a fixed precision for all data).

The MPFR developers recently optimized arithmetic for same-precision operands up to 191 bits [17]. In this work, we reach even higher speed without restrictions on the operands by treating a whole dot product as an atomic operation. This directly speeds up many “basecase” algorithms expressible using dot products, such as classical O(N^2) polynomial multiplication and division and O(N^3) matrix multiplication. Section [1] describes the new dot product algorithm in detail.

For large polynomials and matrices (say, of size N > 50), reductions to fast polynomial and matrix multiplication are ultimately more efficient than iterated dot products. Section [III] looks at fast and accurate matrix multiplication via scaled integer matrices. Section [IV] presents benchmark results and discusses the combination of small-N and large-N algorithms for polynomial operations and linear algebra.

A. Precision and accuracy goals

Throughout this text, p ≥ 2 denotes a precision in bits. For a dot product s = ∑_{i=0}^{N-1} x_i y_i where x_i, y_i are floating-point numbers (not required to have the same precision), we aim to approximate s with error of order \varepsilon = 2^{-p} ∑_{i=0}^{N-1} |x_i y_i| . In a practical sense, this accuracy goal is nearly optimal in p-bit arithmetic; up to cancellations that are unlikely for generic data, uncertainty in the input will typically exceed \varepsilon . Since p is arbitrary, we can set p to (for instance) twice the input precision for specific tasks such as residual calculations. To guarantee an error of 2^{-p} s or even p-bit correct rounding of s, we may do a fast calculation as above with (say) p + 20 bits of precision and fall back to a slower correctly rounded summation [18] only when the fast dot product fails.

A dot product in ball arithmetic becomes

\[ ∑_{i=0}^{N-1} m_i \pm r_i [m_i' \pm r_i'] = [m \pm r], \]

\[ m = ∑_{i=0}^{N-1} m_i m_i', \quad r = ∑_{i=0}^{N-1} |m_i| r_i + |m_i'| r_i + r_i', \] (2)

We compute m with p-bit precision as above, and we compute a low-precision upper bound for r that is tight up to rounding errors on r itself. We must also add a bound for the rounding error on m to the radius r. (If r is zero and the computation of m is exact, then the output radius will be zero.) If r is large, we can sometimes automatically reduce the precision without affecting the accuracy of the output ball.

We require that matrix multiplication gives each output entry with optimal (up to cancellation) accuracy, like the classical algorithm of evaluating N^2 separate dot products. In particular, for a structured or badly scaled ball matrix like

\[
\begin{pmatrix}
1.23 \cdot 10^{100} & 10^{60} \\
1 & 2.34 \cdot 10^{-20} \\
0 & 2
\end{pmatrix}
\begin{pmatrix}
-1.5 & 0 \\
[3.45 \cdot 10^{-50}] & [4.56 \cdot 10^{100}] \cdot 10^{130}
\end{pmatrix},
\]

we preserve small entries and the individual error magnitudes. Many techniques for fast multiplication sacrifice such information. Losing information is sometimes the right tradeoff, but can lead to disaster (for example, 100-fold slowdown [12]). Performance concerns aside, preserving entrywise information avoids surprises for users.
II. ARBITRARY-PRECISION DOT PRODUCT

The obvious algorithm to evaluate a dot product \( \sum_{i=0}^{N-1} x_i y_i \) performs one multiplication followed by \( N-1 \) multiplications and additions (or fused multiply-add operations) in a loop. The functions \texttt{arb\_dot}, \texttt{arb\_approx\_dot}, \texttt{acb\_dot} and \texttt{acb\_approx\_dot} were introduced in Arb 2.15 to replace most such loops. The function

```c
void arb_dot(arb_t res, const arb_t initial, int sub, arb_srcptr x, long xstep, arb_srcptr y, long ystep, long N, long p)
```

sets \( \text{res} \) to a ball containing \( \text{initial} + (-1)^\text{sub} \sum_{i=0}^{N-1} x_i y_i \) where \( x_i = [m_i \pm r_i], y_i = [m'_i \pm r'_i] \) and \( \text{initial} \) are balls of type \texttt{arb\_t} given in arrays with strides of \( x \) and \( y \) step. The optional \textit{initial} term (which may be NULL), \texttt{sub} flag and pointer stride lengths (which may be negative) permit expressing common operations in terms of \texttt{arb\_dot} without extra arithmetic operations or data rearrangement.

The \textit{approx} version is similar but ignores the radii and computes an ordinary floating-point dot product over the midpoints, omitting error bound calculations. The \texttt{acb} versions are the counterparts for complex numbers. All four functions are based on Algorithm 1 which is explained in detail below.

A. Representation of floating-point numbers

The dot product algorithm is designed around the representation of arbitrary-precision floating-point numbers and midpoint-radius intervals (balls) used in Arb. In the following, \( \beta \in \{32, 64\} \) is the word (limb) size, and \( p_{rad} = 30 \) is the radius precision which is a constant.

An \texttt{arb\_t} contains a midpoint of type \texttt{arf\_t} and a radius of type \texttt{mag\_t}. An \texttt{arf\_t} holds one of the special values \( 0, \pm \infty, \text{NaN} \), or a regular floating-point value

\[
m = (-1)^{\text{sign}} \cdot 2^e \cdot \sum_{k=0}^{n-1} b_k \beta^{k-n}
\]

where \( b_k \) are \( \beta \)-bit mantissa limbs normalized so that \( 2^{\beta - 1} \leq b_{n-1} \leq 2^{\beta} - 1 \) and \( b_0 \neq 0 \). Thus \( n \) is always the minimal number of limbs needed to represent \( x \), and we have \( 2^{e-n} \leq |m| < 2^e \). The limbs are stored inline in the \texttt{arf\_t} structure when \( n \leq 2 \) and otherwise in heap-allocated memory. The exponent \( e \) can be arbitrarily large: a single word stores \( |e| < 2^{32-2} \) inline and larger \( e \) as a pointer to a GMP integer.

A \texttt{mag\_t} holds an unsigned floating-point value \( 0, +\infty, \) or \( r = (b/2^{p_{rad}})2^e \) where \( 2^{p_{rad}} \leq b < 2^{p_{rad}+1} \) occupies the low \( p_{rad} \) bits of one word. We have \( 2^{e-n} \leq |r| < 2^e \), and as for \texttt{arf\_t}, the exponent \( e \) can be arbitrarily large.

The methods below can be adapted for MPFR with minimal changes. MPFR variables (\texttt{mpfr\_t}) use the same representation as (3) except that a precision \( p \) is stored in the variable, the number of limbs is always \( n = \lceil p/\beta \rceil \) even if \( b_0 = 0 \), there is no \( n \leq 2 \) allocation optimization, the exponent \( e \) cannot be arbitrarily large, and \( -0 \) is distinct from \( +0 \).

B. Outline of the dot product

We describe the algorithm for \textit{initial} = 0 and \textit{sub} = 0. The general case can be viewed as extending the dot product to length \( N + 1 \), with trivial sign adjustments.

The main observation is that each arithmetic operation on floating-point numbers of the form (3) has a lot of overhead for limb manipulations (case distinctions, shifts, masks), particularly during additions and subtractions. The remedy is to use a fixed-point accumulator for the whole dot product and only convert to a rounded and normalized floating-point number at the end. The case distinctions for subtractions are simplified by using two’s complement arithmetic. Similarly, we use a fixed-point accumulator for the radius dot product.

We make two passes over the data: the first pass inspects all terms, looks for exceptional cases, and determines an appropriate working precision and exponents to use for the accumulators. The second pass evaluates the dot product.

There are three sources of error: arithmetic error on the accumulator (tracked with one limb counting ulps), the final rounding error, and the propagated error from the input balls. At the end, the three contributions are added to a single \( p_{rad} \)-bit floating-point number. The \textit{approx} version of the dot product simply omits all these steps.

Except where otherwise noted, all quantities describing exponents, shift counts (etc.) are single-word (\( \beta \)-bit) signed integers between \( \text{MIN} = -2^{\beta - 1} \) and \( \text{MAX} = 2^{\beta - 1} - 1 \), and all limbs are \( \beta \)-bit unsigned words.

Algorithm 1 Dot product in arbitrary-precision ball arithmetic: given \([m_i \pm r_i], [m'_i \pm r'_i], 0 \leq i < N \) and a precision \( p \geq 2 \), compute \([m \pm r] \) containing \( \sum_{i=0}^{N-1} [m_i \pm r_i][m'_i \pm r'_i] \)

1. \textbf{Setup:} check unlikely cases (infinities, overflow and underflow); determine exponent \( e_s \) and number of limbs \( n_s \) for the midpoint accumulator; determine exponent \( e_{rad} \) for the radius accumulator; reduce \( p \) if possible.
2. \textbf{Initialization:} allocate temporary space; initialize accumulator limbs \( s: s_{n_s-1}, \ldots, s_0 \leftarrow 0, \ldots, 0 \), one limb \( \text{err} \leftarrow 0 \) for the ulp error on \( s_0 \), and a 64-bit integer \( \text{srad} \leftarrow 0 \) as radius accumulator.
3. \textbf{Evaluation:} for each term \( t = m_i m'_i \), compute the limbs of \( t \) that overlap with \( s \), shift and add to \( s \) (or two’s complement subtract if \( t < 0 \)), incrementing \( \text{err} \) if inexact. Add upper bound for \( |m_i|r'_i + |m'_i|r_i + r_i r'_i \) to \( \text{srad} \).
4. \textbf{Finalization:}
   1) If \( s_{n_s-1} \geq 2^{\beta-1} \), negate \( s \) (one call to GMP’s \texttt{mpn\_neg}) and set \( \text{sign} \leftarrow 1 \), else set \( \text{sign} \leftarrow 0 \)
   2) \( m \leftarrow (-1)^{\text{sign}} \cdot 2^{e_s} \cdot (\sum_{i=0}^{N-1} s_k 2^{(k+n_i)}) \) rounded to \( p \) bits, giving a possible rounding error \( \varepsilon_{\text{round}} \)
   3) \( r \leftarrow \varepsilon_{\text{round}} + \text{err} \cdot 2^{e_s-n_s} + \text{srad} \cdot 2^{e_{\text{rad}}-p_{\text{rad}}} \) as a floating-point number with \( p_{\text{rad}} \) bits (rounded up)
   4) Free temporary space and output \([m \pm r] \)

C. Setup pass

The setup pass in Algorithm 1 uses the following steps:

1) \( N_{\text{nonzero}} \leftarrow 0 \) (number of nonzero terms)
2) \( e_{\text{max}} \leftarrow \text{MIN} \) (bound for term exponents)
3) \( e_{\text{min}} \leftarrow \text{MAX} \) (lower bound for content)
4) \( e_{\text{rad}} \leftarrow \text{MIN} \) (bound for radius exponents)
For $0 \leq i < N$:

a) If any of $m_i, m'_i, r_i, r'_i$ is non-finite or has an exponent outside of $\pm 2^{3\beta-4}$ (unlikely), quit Algorithm 1 and use a fallback method.

b) If $m_i$ and $m'_i$ are both nonzero, with respective exponents $e, e'$ and limb counts $n, n'$:
   - Set $N[^\text{nonzero}] \leftarrow N[^\text{nonzero}] + 1$
   - Set $e_{\text{max}} \leftarrow \max(e_{\text{max}}, e + e')$
   - If $p > 2\beta$, $e_{\text{min}} \leftarrow \min(e_{\text{min}}, e + e' - \beta(n + n'))$

b) For each product $|m_i| r'_i$, $|m'_i| r_i, r_i r'_i$ that is nonzero, denote the exponents of the respective factors by $e, e'$ and set $e_{\text{rad}} \leftarrow \max(e_{\text{rad}}, e + e')$

6) If $e_{\text{max}} = e_{\text{rad}} = \text{MIN}$, quit Algorithm 1 and output $[0 \pm 0]$

7) (Optimize $p$.) If $e_{\text{max}} = \text{MIN}$, set $p \leftarrow 2$. Otherwise:
   a) If $e_{\text{rad}} \neq \text{MIN}$, set $p \leftarrow \min(p, e_{\text{max}} - e_{\text{rad}} + p_{\text{rad}})$ (if the final radius $r$ will be larger than the expected arithmetic error, we can reduce the precision used to compute $m$ without affecting the accuracy of the ball $[m \pm r]$).
   b) If $e_{\text{min}} \neq \text{MAX}$, set $p \leftarrow \min(p, e_{\text{max}} - e_{\text{min}} + p_{\text{rad}})$ (if all terms fit in a window smaller than $p$ bits, reducing the precision does not change the result).

C) Set $p \leftarrow \max(p, 2)$

8) Set $\text{padding} \leftarrow 4 + bc(N)$, where $bc(\nu) = \lceil \log_2(\nu + 1) \rceil$ denotes the binary length of $\nu$.

9) Set $\text{extend} \leftarrow bc(N[^\text{nonzero}] + 1)$

10) Set $n_s \leftarrow \max(2, \lceil(p + \text{extend} + \text{padding})/\beta \rceil)$

11) Set $e_s \leftarrow e_{\text{max}}$

All terms $|m_i|m'_i$ are bounded by $2^{2e_{\text{max}}}$ and similarly all radius terms are bounded by $2^e_{\text{rad}}$. The width of the accumulator is $p$ bits plus $\text{extend}$ leading bits and $\text{padding}$ trailing bits, rounded up to a whole number of limbs $n_s$. The quantity $\text{extend}$ guarantees that carries never overflow the leading limb $s_{n_s-1}$, including one bit for two’s complement negation; it is required to guarantee correctness. The quantity $\text{padding}$ adds a few guard bits to enhance the accuracy of the dot product; this is an entirely optional tuning parameter.

D. Evaluation

For a midpoint term $m_i m'_i \neq 0$, denote the exponents of $m_i, m'_i$ by $e, e'$ and the limb counts by $n, n'$. The multiply-add operation uses the following steps:

1) Set $\text{shift} \leftarrow e_s - (e + e')$, $\text{shift_bits} \leftarrow \text{shift mod } \beta$, $\text{shift_limbs} \leftarrow \text{shift}/\beta$

2) If $\text{shift} \geq \beta n_s$, set $\text{err} \leftarrow \text{err} + 1$ and go on to the next term (this term does not overlap with the limbs in $s$).

3) Set $p_t \leftarrow \beta n_s - \text{shift}$ (effective bit precision needed for this term), and set $n'' \leftarrow \lceil p_t/\beta \rceil + 1$. If $n > n''$ or $n' > n''$, set $\text{err} \leftarrow \text{err} + 1$. (We read at most $n''$ leading limbs from $m$ and $m'$ since the smaller limbs have a negligible contribution to the dot product; in case of truncation, we increment the error bound by 1 ulp.)

4) Set $n_t \leftarrow \min(n, n'') + \min(n', n'')$. The term will be stored in up to $n_t + 1$ temporary limbs $t_{n_t}, \ldots, t_0$ pre-allocated in the initialization of Algorithm 1.

5) Set $t_{n_t-1}, \ldots, t_0$ to the product of the top $\min(n, n'')$ limbs of $m$ and the top $\min(n', n'')$ limbs of $m'$ (this is one call to GMP’s $\text{mpn}_\text{mul}$). We now have the situation depicted in Figure 1.

6) (Bit-align the limbs.) If $\text{shift_bits} \neq 0$, set $t_{n_r-1}, \ldots, t_0$ right-shifted by $\text{shift_bits}$ bits (this is a pointer adjustment and one call to $\text{mpn}_\text{r_shift}$) and then set $n_t \leftarrow n_t + 1$.

7) (Strip trailing zero limbs.) While $t_0 = 0$, increment the pointer to $t$ and set $n_t \leftarrow n_t - 1$.

It remains to add the aligned limbs of $t$ to the accumulator $s$.

We have two cases, with $v$ denoting the number of overlapping limbs between $s$ and $t$ and $d_s$ and $d_t$ denoting the offsets from $s_0$ and $t_0$ to the overlapping segment. If $\text{shift_limbs} + n_t \leq n_s$ (no discarded limbs), set $d_s \leftarrow n_s - \text{shift_limbs} - n_t$, $d_t \leftarrow 0$ and $v \leftarrow n_t$. Otherwise, set $d_s \leftarrow 0$, $d_t \leftarrow n_s - \text{shift_limbs}$, $v \leftarrow n_s - \text{shift_limbs}$ and $\text{err} \leftarrow \text{err} + 1$. The addition is now done using the GMP code

$$
cy = \text{mpn}_\text{add}_n(s + d_s, s + d_s, t + dt, v);
$$

$$
\text{mpn}_\text{add}_1(s + d_s + v, s + d_s + v, \text{shift_limbs}, cy);
$$

if $m_i m'_i > 0$, or in case $m_i m'_i < 0$ using $\text{mpn}_\text{sub}_n$ and $\text{mpn}_\text{sub}_1$ to perform a two’s complement subtraction.

Our code has two more optimizations. If $n \leq 2, n' \leq 2, n_s \leq 3$, the limb operations are done using inline code instead of calling GMP functions, speeding up precision $p \leq 128$ (64-bit machines). When $p_t \geq 25\beta$ and $\min(n, n'')\beta > 0.9p_t$, we compute $n''$ leading limbs of the product using the MPFR-internal function $\text{mpfr}_\text{mulhigh}_n$ instead of $\text{mpn}_\text{mul}$. This is done with up to 1 ulp error on $s_0$ and is therefore accompanied by an extra increment of $\text{err}$.

E. Radius operations

For the radius dot product $\sum_{i=0}^{N-1} |m_i| r'_i + |m'_i| r_i + r_i r'_i$, we convert the midpoints $|m_i|, |m'_i|$ to upper bounds in the radius format $r = (b/2^{p_{rad}})2^e$ by taking the top $p_{rad}$ bits of the top limb and incrementing; this results in the weakly normalized mantissa $2^{p_{rad}-1} \leq b \leq 2^{p_{rad}}$. The summation is done with an accumulator ($\text{rad}/2^{p_{rad}}$)$2^e$ where $\text{rad}$ is one unsigned 64-bit integer (1 or 2 limbs). The step to add an upper bound for $(a/2^{p_{rad}})(b/2^{p_{rad}})2^e$ is $\text{rad} = \text{rad} + [(ab)/2^{p_{rad}+e_{\text{rad}}-e} + 1]$ if $e_{\text{rad}} - e < p_{\text{rad}}$ and $\text{rad} = \text{rad} + 1$ otherwise.

By construction, $e_{\text{rad}} \geq e$, and due to the 34 free bits for carry accumulation, $\text{rad}$ cannot overflow if $N < 2^{32}$. 

![Fig. 1. The accumulator $s_{n_s-1}, \ldots, s_0$ and the term $t_{n_t-1}, \ldots, t_0$, prior to limb alignment. More significant limbs are shown towards the left.](image-url)
(Larger \( N \) could be supported by increasing \( e_{\text{rad}} \), at the cost of some loss of accuracy.) We use conditionals to skip zero terms; the radius dot product is therefore evaluated as zero whenever possible, and if the input balls are exact, no radius computations are done apart from inspecting the terms.

F. Complex numbers

Arb uses rectangular “balls” \([a \pm r] + [b \pm s]i\) to represent complex numbers. A complex dot product is essentially performed as two length-\(2N\) real dot products. We note that this preserves information about whether real or imaginary parts are exact or zero, and both parts can be computed with high relative accuracy when they have different scales of magnitude. The algorithm could be adapted in the obvious way to work with true complex balls (disks).

For terms where both real and imaginary parts have similar magnitude and high precision, we use the additional optimization of avoiding one real multiplication via the formula

\[
(a + bi)(c + di) = ac - bd + i[(a + b)(c + d) - ac - bd].
\]

Since this formula is applied exactly and only for the midpoints, accuracy is not compromised. The cutoff occurs at the rather high 128 limbs (8192 bits) since \((4)\) is implemented using exact products and therefore competes against multimodular techniques employing several multiplications modulo word-size primes followed by reconstruction via the Chinese remainder theorem. This saves time since computations done over a fixed word size have less overhead than arbitrary-precision computations. Moreover, for modest \(p\), the running time essentially scales as \(O(p)\) compared to the \(O(p^2)\) with dot products, as long as the cost of the multimodular reductions and reconstructions does not dominate. The downside of converting floating-point numbers to integers is that we either must truncate entries (losing accuracy) or zero-pad (losing speed).

Algorithms for fast and accurate polynomial multiplication have been discussed in previous work \cite{20, 14}. Arb scales the inputs and converts the coefficients to integers, adaptively splitting the polynomials into smaller blocks to keep the height of the integers small. The integer polynomials are then multiplied using FLINT \cite{10}, which selects between classical, Karatsuba and Kroncker algorithms and an asymptotically fast Schönhage-Strassen FFT. Arb implements other operations (such as division) via methods such as Newton iteration that asymptotically reduce to polynomial multiplication.

III. MATRIX MULTIPLICATION

We consider the problem of multiplying an \(M \times N\) ball matrix \([A \pm R_A]\) by an \(N \times K\) ball matrix \([B \pm R_B]\) (where \(R_A, R_B\) are nonnegative matrices and \([\pm]\) is interpreted entrywise). The classical algorithm can be viewed as computing \(MP\) dot products of length \(N\). For large matrices, it is advantageous to convert from arbitrary-precision floating-point numbers to integers \cite{21}. Integer matrices can be multiplied efficiently using multimodular techniques employing several multiplications modulo word-size primes followed by reconstruction via the Chinese remainder theorem. This saves time since computations done over a fixed word size have less overhead than arbitrary-precision computations. Moreover, for modest \(p\), the running time essentially scales as \(O(p)\) compared to the \(O(p^2)\) with dot products, as long as the cost of the multimodular reductions and reconstructions does not dominate. The downside of converting floating-point numbers to integers is that we either must truncate entries (losing accuracy) or zero-pad (losing speed).

Algorithms for fast and accurate polynomial multiplication have been discussed in previous work \cite{20, 14}. Arb scales the inputs and converts the coefficients to integers, adaptively splitting the polynomials into smaller blocks to keep the height of the integers small. The integer polynomials are then multiplied using FLINT \cite{10}, which selects between classical, Karatsuba and Kroncker algorithms and an asymptotically fast Schönhage-Strassen FFT. Arb implements other operations (such as division) via methods such as Newton iteration that asymptotically reduce to polynomial multiplication.

In this section, we describe an approach to multiply matrices in Arb following similar principles. We compute \([A \pm R_A][B \pm R_B]\) using three products \(AB, |A|R_B, R_A|B| + R_B\) where we use FLINT integer matrices for the high-precision midpoint product \(AB\). FLINT in turn uses classical multiplication, the Strassen algorithm, a multimodular algorithm employing 60-bit primes, and combinations of these methods. An important observation for both polynomials and matrices is that fast algorithms such as Karatsuba, FFT and Strassen multiplication do not affect accuracy when used on the integer level.

A. Splitting and scaling

The earlier work by van der Hoeven \cite{21} proposed multiplying arbitrary-precision matrices via integers truncated to \(p\)-bit height, splitting size-\(N\) matrices into \(m^2\) blocks of size \(N/m\), where the user selects \(m\) to balance speed and accuracy. Algorithm \cite{21} improves on this idea by using a fully automatic and adaptive splitting strategy that guarantees near-optimal entrywise accuracy (like the classical algorithm).

We split \(A\) into column submatrices \(A_i\) and \(B\) into row submatrices \(B_s\). For any such \(A_i\) and for each row index \(i\), let \(e_{i,s}\) denote the unique scaling exponent such that row \(i\) of \(2^{e_i}A_i\) consists of integers of minimal height \(e_{i,s}\) is uniquely determined unless row \(i\) of \(A_i\) is identically zero, in which case we may take \(e_{i,s} = 0\). Similarly let \(f_{j,s}\) be the optimal scaling exponent for column \(j\) of \(B_s\). Then the contribution of \(A_i\) and \(B_s\) to \(C = AB\) consists of \(E_s^{-1}((E_sA_i)(B_sF_s))F_s^{-1}\) where \(E_s = \text{diag}(2^{e_{i,s}})\) scales the rows of \(A_i\) and \(F_s = \text{diag}(2^{f_{j,s}})\) scales the columns of \(B_s\) (see Figure \ref{fig:mult}), and where we may multiply \((E_sA_i)(B_sF_s)\) over the integers.

Crucially, only magnitude variations within rows of \(A_s\) (columns of \(B_s\)) affect the height; the rows of \(A_s\) can have different magnitude from each other (and similarly for \(B_s\)).

We extract indices \(s\) by performing a greedy search in increasing order, appending columns to \(A_s\) and rows to \(B_s\) as long as a height bound is satisfied. The tuning parameter \(h\) balances the advantage of using larger blocks against the disadvantage of using larger zero-padded integers. In the common case where both \(A\) and \(B\) are uniformly scaled and have the same (or smaller) precision as the output, one
Algorithm 2: Matrix multiplication using blocks: given ball matrices \( [A \pm R_A], [B \pm R_B] \) and a precision \( p \geq 2 \), compute \( [C \pm R_C] \) containing \( [A \pm R_A][B \pm R_B] \)

1: \[ [C \pm R_C] \leftarrow [0 \pm 0] \quad \triangleright \text{Initialize the zero matrix} \\
2: \[ h \leftarrow 1.25 \min(p, \max(p_A, p_B)) + 192, \text{ where } p_M \text{ is the} \]
\[ \quad \text{minimum floating-point precision needed to represent all} \]
\[ \quad \text{entries of } M \text{ exactly} \quad \triangleright \text{Tuning parameter} \\
3: \[ S \leftarrow \{0, \ldots, N - 1\} \text{ where } N \text{ is the inner dimension} \\
4: \quad \text{while } S \neq \{\} \text{ do} \\
5: \quad \quad \text{Extract } s \subseteq S \text{ such that } |E_s A_s| < 2^h \text{ and } |B_s F_s| < 2^h \\
6: \quad \quad \text{if } \text{size}(s) < 30 \text{ then} \quad \triangleright \text{Basecase for short blocks} \\
7: \quad \quad \quad \text{Extend } s \text{ to min}(30, \text{size}(S)) \text{ indices} \\
8: \quad \quad \text{else} \\
9: \quad \quad \quad T \leftarrow (E_s A_s)(B_s F_s) \quad \triangleright \text{Matrix product over } \mathbb{Z} \\
10: \quad \quad \quad [C \pm R_C] \leftarrow [C \pm R_C] + E_s^{-1} T F_s^{-1} \quad \triangleright \text{Ball} \\
\[ \quad \quad \text{addition, with possible rounding error on } C \text{ added to } R_C \]
11: \quad \quad \text{end if} \\
12: \quad S \leftarrow S \setminus s \\
13: \quad \text{end while} \\
14: \quad \text{Compute } R_1 = |A|R_B \text{ and } R_2 = R_A(|B| + R_B) \text{ by} \\
\[ \quad \text{splitting and scaling into blocks of double as above (but} \]
\[ \quad \text{using floating-point arithmetic with upper bounds instead} \]
\[ \quad \text{of ball arithmetic), and set } R_C \leftarrow R_C + R_1 + R_2 \]
15: \quad \text{Output } [C \pm R_C].

Algorithm 2 turns out to perform reasonably well in practice when many blocks are used, but it could certainly be improved. The bound \( h \) could be tuned, and the greedy strategy to select blocks is clearly not always optimal. Going even further, we could extract non-contiguous submatrices, add an extra inner scaling matrix \( G_s G_s^{-1} \) for the columns of \( A_s \) and rows of \( B_s \), and combine scaling with permutations. Finding the best strategy for badly scaled or structured matrices appears to be a difficult problem. There is some resemblance to the balancing problem for eigenvalue algorithms [16].

Both Algorithm 2 and the analogous algorithm used for polynomial multiplication in Arb have the disadvantage that all input bits are used, unlike classical multiplication based on Algorithm 1 which omits negligible limbs. This important optimization for non-uniform polynomials (compare [20]) and matrices should be considered in future work.

C. Complex matrices

We multiply complex matrices using four real matrix multiplications \((A + Bi)(C + Di) = (AC − BD) + (AD + BC)i\) outside of the basecase range for using complex dot products. An improvement would be to use \([1]\) to multiply the midpoint matrices when all entries are uniformly scaled; \([4]\) could also be used for blocks with a splitting and scaling strategy that considers the real and imaginary parts simultaneously.

IV. BENCHMARKS

Except where noted, the following results were obtained on an Intel i5-4300U CPU using GMP 6.1, MPFR 4.0, MPC 1.1 [4] (the complex extension of MPFR), QD 2.3.22 [11] (106-bit double-double and 212-bit quad-double arithmetic), Arb 2.16, and the December 2018 git version of FLINT.

A. Single dot products

Figures 3 and Table I show timings measured in CPU cycles per term for a dot product of length \( N = 100 \) with uniform \( p \)-bit entries. We compare a simple loop using QD arithmetic, three MPFR versions, a simple Arb loop (addmul denoting repeated multiply-adds with \texttt{arb_addmul}), and Algorithm 1 in Arb, both for balls (\texttt{dot} denoting \texttt{arb_dot}) and floating-point numbers (\texttt{approx} denoting \texttt{arb_approx_dot}). Similarly, we include results for complex dot products, comparing MPC and three Arb methods. The \texttt{muladd} MPFR version uses \texttt{mpfr_mul} and \texttt{mpfr_add}, with a preallocated temporary variable; \texttt{fma} denotes multiply-adds with \texttt{mpfr_fma}; our \texttt{sum} code writes exact products to an array and calls \texttt{mpfr_sum} [18] to compute the sum (and hence the dot product) with correct rounding. We make several observations:

- The biggest improvement is seen for \( p \leq 128 \) (up to two limbs). The ball dot product is up to 4.2 times faster than the simple Arb loop (and 2.0 times faster than MPFR); the \texttt{approx} version is up to 3.7 times faster than MPFR.
A factor 1.5 to 2.0 speedup persists up to several hundred bits, and the speed for very large $p$ is close to optimal. Ball arithmetic error propagation adds 20 cycles/term overhead, equivalent to a factor 2.0 when $p \leq 128$ and a negligible factor at higher precision.

- At $p = 106$, the approx dot product is about as fast as QD double-double arithmetic, while the ball version is half as fast; at $p = 212$, either version is twice as fast as QD quad-double arithmetic.
- Complex arithmetic costs quite precisely four times more than real arithmetic. The speedup of our code compared to MPC is even greater than compared to MPFR.
- A future implementation of a correctly rounded dot product for MPFR and MPC using Algorithm 1 with mpfr_sum as a fallback should be able to achieve nearly the same average speed as the approx Arb version.

For small $N$, the initialization and finalization overhead in Algorithm 1 is significant. Table II shows that it nevertheless performs better than a simple loop already for $N = 2$ and quickly converges to the speed measured at $N = 100$.

Algorithm 1 does even better with structured data, for example when the balls are exact, with small-integer coefficients, or with varying magnitudes. As an example of the latter, with $N = 1000, p = 1024$, and terms $(1/i!) \cdot (\pi^{-i})$, Algorithm 1 takes 0.035 ms while an arb_addmul loop takes 0.33 ms.

### Table I
Cycles/term to evaluate a dot product ($N = 100$).

| $p$ | QD | MPFR (real) | Arb (real) |
|-----|----|-------------|------------|
|     | QD | mul/addr    | fma | sum | addmul | dot | approx |
| 53  | 74 | 99 | 108 | 169 | 40 | 20 |
| 106 | 26 | 97 | 156 | 124 | 203 | 49 | 27 |
| 159 | 140 | 183 | 169 | 257 | 123 | 105 |
| 212 | 265 | 237 | 208 | 188 | 277 | 133 | 117 |
| 424 | 350 | 288 | 288 | 374 | 215 | 201 |
| 848 | 670 | 619 | 597 | 705 | 522 | 499 |
| 1696 | 1435 | 1675 | 1667 | 1823 | 1471 | 1451 |
| 3392 | 4059 | 4800 | 4741 | 4875 | 3906 | 3880 |
| 13568 | 33529 | 39546 | 39401 | 39275 | 32476 | 32467 |

### Table II
Cycles/term to evaluate a dot product, variable $N$.

| $p$ | Arb (real), dot $N = 2$ | Arb (real), approx $N = 2$ |
|-----|---------------------|---------------------|
|     | $N = 4$ | $N = 8$ | $N = 16$ | $N = 4$ | $N = 8$ | $N = 16$ |
| 53  | 89 | 65 | 53 | 44 | 64 | 41 | 31 | 23 |
| 106 | 98 | 75 | 61 | 53 | 71 | 47 | 37 | 31 |
| 212 | 215 | 175 | 159 | 143 | 177 | 147 | 136 | 125 |
| 848 | 614 | 571 | 552 | 535 | 567 | 547 | 522 | 507 |

B. Basecase polynomial and matrix operations

The new dot product code was added in Arb 2.15 along with re-tuned cutoffs between small-$N$ and large-$N$ algorithms. Figure 4 shows the speedup of Arb 2.15 over 2.14 for operations on polynomials and power series of length $N$ and matrices of size $N$, here for $p = 64$ and complex coefficients.

This shows the benefits of Algorithm 1 even in the presence of a fast large-$N$ algorithm (the block algorithm for matrix multiplication was added in Arb 2.14). The speedup typically grows with $N$ as the dot product gains an increasing advantage over a simple multiply-add loop, up to the old cutoff point for switching to a large-$N$ algorithm. To the right of this point, the dot product then gives a diminishing speedup over the large-$N$ algorithm up to the new cutoff. Jumps are visible where the old cutoff was suboptimal. We make some more observations:

- The speedup around $N \approx 10$ to 30 is notable since this certainly is a common size for real-world use.
- Some large-$N$ algorithms like Newton iteration series inversion and block recursive linear solving use recursive operations of smaller size, so the improved basecase gives an extended “tail” speedup into the large-$N$ regime.
- The power series exponential and sine/cosine improve dramatically. The large-$N$ method uses Newton iteration which costs several polynomial multiplications, while the $O(N^2)$ basecase method uses the dot product-friendly recurrence $\exp(a_1 x + a_2 x^2 + \ldots) = b_0 + b_1 x + b_2 x^2 + \ldots$, $b_0 = 1, b_k = (\sum_{j=1}^{k} (a_j b_{k-j})/k$. The cutoffs have been increased to $N = 750$ and $N = 1400$ (for this $p$).
- The characteristic polynomial (charpoly) does not currently use matrix multiplication in Arb, so we get the pure dot product speedup for large $N$.
- Series composition and reversion use baby-step giant-steps methods [2], [13] where dot products enter in both length-$N$ polynomial and size-$\sqrt{N}$ matrix multiplications.
of basecase triangular solving and matrix multiplication) for small $N$. Iterative Gaussian elimination is used for $N \leq 7$.

In ball (or interval) arithmetic, LU factorization is unstable and generically loses $O(N)$ digits even for a well-conditioned matrix. This problem can be fixed with preconditioning [19]. The classical Hansen-Smith algorithm [9] solves $AX = B$ by first computing an approximate inverse $R \approx A^{-1}$ in floating-point arithmetic and then solving $(RA)X = RB$ in interval or ball arithmetic. Direct LU-based solving in ball arithmetic behaves nicely for the preconditioned matrix $RA \approx I$.

Arb provides three methods for linear solving in ball arithmetic: the LU algorithm, the Hansen-Smith algorithm, and a default method using LU when $N \leq 4$ or $p > 10N$ and Hansen-Smith otherwise. In practice, Hansen-Smith is typically 3-6 times slower than the LU algorithm. The default method thus attempts to give good performance both for well-conditioned problems (where low precision should be sufficient) and for ill-conditioned problems (where high precision is required). Similarly, Arb computes determinants using ball LU factorization for $N \leq 10$ or $p > 10N$ and otherwise via preconditioning using approximate LU factors [19].

Table IV compares speed for solving $AX = B$ with a uniform well-conditioned $A$ and a vector $B$. Due to the new dot product and matrix multiplication, the LU-based approximate solving in Arb is significantly faster than LU-based solving with MPFR entries in both the Eigen 3.3.7 C++ library [8] and Julia 1.0 [1]. The verified ball solving in Arb is also competitive. Julia is extra slow for large $N$ due to garbage collection, which incidentally makes an even bigger case for an atomic dot product avoiding temporary operands.

### E. Eigenvalues and eigenvectors

Table V shows timings for computing the eigendecomposition of the matrix with entries $e^{i(jN+k)}$, $0 \leq j, k < N$. Three methods available in Arb 2.16 are compared. The approx method is the standard QR algorithm [16] (without error bounds), with $O(N^3)$ complexity. We include as a point

| Table III | Time (s) to multiply size-$N$ matrices. (#) is the number of blocks $A_s B_s$, used by the block algorithm, where greater than one. |
|---|---|
| Uniform real | Pascal | Uniform complex |
| $p$ | QD | MPFR | Arb | Arb | Arb | Arb | Arb | Arb | MPC | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arb | Arbor
of reference timings for the QR implementation in the Julia
tpackage GenericLinearAlgebra.jl using MPFR arithmetic. The
ter two Arb methods compute rigorous enclosures in ball
ithmetic by first finding an approximate eigendecomposition
using the QR algorithm and then performing a verification
using ball matrix multiplications and linear solving. The Rump
method \cite{Rump2015} verifies one eigenpair at a time requiring
\(O(N^3)\) total operations, and the vdHM method \cite{vdHM}
verifies all eigenpairs simultaneously in \(O(N^3)\) operations.

The kernel operations in the QR algorithm are rotations
\((x, y) \leftarrow (cx+sy, \tau y-\pi x)\), i.e. dot products of length 2, which
we have only improved slightly in this work. A useful future
project would be an arbitrary-precision QR implementation
with block updates to exploit matrix multiplication. Our work
does already speed up the initial reduction to Hessenberg
form in the QR algorithm, and it speeds up both verification
algorithms; we see that the vdHM method only costs a fraction
more than the unverified approx method. The Rump method is
more expensive but gives more precise balls than vdHM; this
can be a good tradeoff in some applications.

V. CONCLUSION AND PERSPECTIVES

We have demonstrated that optimizing the dot product as an
atomic operation leads to a significant reduction in overhead
for arbitrary-precision arithmetic, immediately speeding up
polynomial and matrix algorithms. The performance is com-
petitive with non-vectorized double-double and quad-double
arithmetic, without the drawbacks of these types. For accurate
large-\(N\) matrix multiplication, using scaled integer blocks (in
similar fashion to previous work for polynomial multiplica-
tion) achieves even better performance.

It should be possible to treat the Horner scheme for
polynomial evaluation in similar fashion to the dot product,
with similar speedup. (The dot product is itself useful for
polynomial evaluation, in situations where powers of the
argument can be recycled.) More modest improvements should
be possible for single arithmetic operations in Arb.

In addition to the ideas for algorithmic improvements al-
ready noted in this paper, we point out that Arb would benefit
from faster integer matrix multiplication in FLINT. More than
a factor two can be gained with better residue conversion code
and use of BLAS \cite{BLAS, OpenBLAS}. BLAS could also be used for
the radius matrix multiplications in Arb (we currently use simple
C code since the FLINT multiplications are the bottleneck).

The FLINT matrix code is currently single-threaded, and
because of this, we only benchmark single-core performance.

Arb does have a multithreaded version of classical matrix
multiplication performing dot products in parallel, but this
code is typically not useful due to the superior single-core
efficiency of the block algorithm. Parallelizing the block
algorithm optimally is of course the more interesting problem.

REFERENCES

\begin{enumerate}
\item J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh
approach to numerical computing. SIAM review, 59(1):65–98, 2017.
\item R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal
power series. Journal of the ACM, 25(4):581–595, 1978.
\item J. Doliskani, P. Giorgi, R. Lebreton, and E. Schost. Simultaneous
conversions with the residue number system using linear algebra. ACM
Transactions on Mathematical Software (TOMS), 44(3):27, 2018.
\item A. Enge, M. Gastineau, P. Théveny, and P. Zimmermann. MPC: a library
for multiprecision complex arithmetic with exact rounding. http://www.
multiprecision.org/mpc/. 2018.
\item 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, 2007.
\item F. Giorgi. Toward high performance matrix multiplication for exact
computation. https://www.lirmm.fr/~giorgi/seminaire-hk-14.pdf. 2014.
\item T. Granlund and the GMP development team. GNU MP: The GNU
Multiple Precision Arithmetic Library, 6.1.2 edition, 2017.
\item G. Guennebaud and B. Jacob. Eigen. http://eigen.tuxfamily.org/. 2018.
\item E. Hansen and R. Smith. Interval arithmetic in matrix computations,
Part II. SIAM Journal on Numerical Analysis, 4(1):1–9, 1967.
\item W. B. Hart. Fast library for number theory: an introduction. In Inter-
national Congress on Mathematical Software, pages 88–91. Springer,
2010.
\item Y. Hida, X. S. Li, and D. H. Bailey. Library for double-double and
quad-double arithmetic. NERSC Division, Lawrence Berkeley National
Laboratory, 2007.
\item F. Johansson. Arb: a C library for ball arithmetic. ACM
Communications in Computer Algebra, 47(4):166–169, 2013.
\item F. Johansson. A fast algorithm for reversion of power series. Mathe-
matics of Computation, 84:475–484, 2015.
\item F. Johansson. Arb: efficient arbitrary-precision midpoint-radius interval
arithmetic. IEEE Transactions on Computers, 66:1281–1292, 2017.
\item M. Joldes, O. Marty, J.-M. Muller, and V. Popescu. Arithmetic
algorithms for extended precision using floating-point expansions. IEEE
Transactions on Computers, 65(4):1197–1210, 2016.
\item D. Kressner. Numerical Methods for General and Structured
Eigenvector Problems. Springer-Verlag, 2005.
\item V. Lefèvre and P. Zimmermann. Optimized Binary64 and Binary128
arithmetic with GNU MPFR. In 2017 IEEE 24th Symposium on
Computer Arithmetic (ARITH), pages 18–26. IEEE, 2017.
\item Vincent Lefèvre. Correctly rounded arbitrary-precision floating-point
summation. IEEE Transactions on Computers, 66(12):2111–2124, 2017.
\item S. M. Rump. Verification methods: Rigorous results using floating-point
arithmetic. Acta Numerica, 19:287–449, 2010.
\item J. van der Hoeven. Making fast multiplication of polynomials numer-
ically stable. Technical Report 2008-02, Université Paris-Sud, Orsay,
France, 2008.
\item J. van der Hoeven. Ball arithmetic. Technical report, HAL, 2009.
\item J. van der Hoeven and G. Lecerf. Faster FFTs in medium precision. In
2015 IEEE 22nd Symposium on Computer Arithmetic (ARITH), pages
75–82. IEEE, 2015.
\item J. van der Hoeven and B. Mourrain. Efficient certification of numeric sol-
sitions to eigenproblems. In International Conference on Mathematical
Aspects of Computer and Information Sciences, pages 81–94. Springer,
2017.
\end{enumerate}