Evaluating Straight-Line Programs over Balls

Joris van der Hoeven, Grégoire Lecerf

Evaluating Straight-Line Programs over Balls

by Joris van der Hoeven[a], Grégoire Lecerf[b]

Laboratoire d'informatique de l'École polytechnique
LIX, UMR 7161 CNRS
Campus de l'École polytechnique
1, rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing, CS35003
91120 Palaiseau, France

a. *Email:* `vdhoeven@lix.polytechnique.fr`
b. *Email:* `lecerf@lix.polytechnique.fr`

Abstract

Interval arithmetic achieves numerical reliability for a wide range of applications, at the price of a performance penalty. For applications to homotopy continuation, one key ingredient is the efficient and reliable evaluation of complex polynomials represented by straight-line programs. This is best achieved using ball arithmetic, a variant of interval arithmetic. In this article, we describe strategies for reducing the performance penalty of basic operations on balls. We also show how to bound the effect of rounding errors at the global level of evaluating a straight-line program. This allows us to introduce a new and faster "transient" variant of ball arithmetic.

Keywords: ball arithmetic, polynomial evaluation, software implementation

1 Introduction

Interval arithmetic is a classical tool for making numerical computations reliable, by systematically computing interval enclosures for the desired results instead of numerical approximations [1, 10, 11, 13, 14, 17, 24]. Interval arithmetic has been applied with success in many areas, such as the resolution of systems of equations, homotopy continuations, reliable integration of dynamical systems, etc. There exist several variants of interval arithmetic such as ball arithmetic, interval slope arithmetic, Taylor models, etc. Depending on the context, these variants may be more efficient than standard interval arithmetic and/or provide tighter enclosures.

In this paper, we will mainly focus on *ball arithmetic* (also known as *midpoint-radius interval arithmetic*), which is particularly useful for reliable computations with complex numbers and multiple precision numbers [5]. One of our main motivations is the implementation of reliable numerical homotopy methods for polynomial system solving [2, 3, 6, 27, 28, 29]. One basic prerequisite for this project concerns the efficient and reliable evaluation of multivariate polynomials represented by so called *straight-line programs* (SLPs).

Two classical disadvantages of interval arithmetic and its variants are the additional computational overhead and possible overestimation of errors. There is a trade-off between these two evils: it is always possible to reduce the overestimation at the expense of a more costly variant of interval arithmetic (such as high order Taylor models). For a fixed variant, the computational overhead is usually finite, but it remains an important issue for high performance applications to reduce the involved constant factors as much as possible. In this paper, we will focus on this "overhead problem" in the case of basic ball arithmetic (and without knowledge about the derivatives of the functions being evaluated).

About ten years ago, we started the implementation of a basic C++ template library for ball arithmetic in the MATHEMAGIX system [9]. However, comparing speed of operations over complex numbers in double precision and over the corresponding balls was quite discouraging. The overhead comes from extra computations of radii, but also from changes of rounding mode, and the way the C++ compiler generates executable code from C99 portable sources. For our application to reliable homotopies, the critical part to be optimized concerns the evaluation of input polynomials using ball arithmetic. The goal of the present work is to minimize the overhead of such evaluations with respect to their numeric counterparts.

Our contributions

In this article we investigate various strategies to reduce the overhead involved when evaluating straight-line programs over balls. Our point of view is pragmatic and directed towards the development of more efficient implementations. We will not turn around the fact that the development of efficient ball arithmetic admits a quite technical side: the optimal answer greatly depends on available hardware features. We will consider on the following situations, encountered for modern processors:

- Without specific IEEE 754 compliant hardware (which is frequently the case for GPUs), we may only assume *faithful rounding*, specifying a bound on relative errors for each operation, and that errors can be thrown on numerical exceptions.
- For recent INTEL-compatible processors (integrating SSE or AVX technologies) it is recommended to perform numerical computations using SIMD (single instruction, multiple data) units. Programming must be done mostly at the assembly code level with specific builtin instructions called *intrinsics*. Dynamically switching the rounding mode involves a rather small overhead.
- Recent AVX-512 processors propose floating point instructions which directly incorporate a rounding mode, thereby eliminating all need to switch rounding modes *via* the status register and all resulting delays caused by broken pipelines.

As a general rule, modern processors have also become highly parallel. For this reason, it is important to focus on algorithms that can easily be vectorized. The first contribution of this paper concerns various strategies for ball arithmetic as a function of the available hardware. From the conceptual point of view, this will be done in Section 2, whereas additional implementation details will be given in Section 4.

Our second main contribution is the introduction of *transient* ball arithmetic in Section 2.5 and its application to the evaluation of SLPs in Section 3. The idea is to not bother about rounding errors occurring when computing centers and radii of individual balls, which simplifies the implementation of the basic ring operations. Provided that the radii of the input balls are not too small and that neither overflow nor underflow occur, we will show that the cumulated effect of the ignored rounding errors can be bounded for the evaluation of a complete SLP. More precisely, we will show that the relative errors due to rounding are essentially dominated by the depth of the computation. Although it is most convenient to apply our result to SLPs, much of it can be generalized to arbitrary programs, by regarding the trace of the execution as an SLP. For such generalizations, the main requirement is to have an *a priori* bound for the (parallel) depth of the computation.

Most of our strategies have been implemented inside the MULTIMIX and JUSTINLINE libraries of MATHEMAGIX [9], in C++ and in the MATHEMAGIX language. In Section 4, we compare costs for naive, statically compiled, and dynamically compiled polynomial evaluations. Using our new theoretical ideas and various implementation tricks, we managed to reduce the overhead of ball arithmetic to a small factor between two and four. We also propose SSE and AVX based functions for arithmetic operations on balls. Dynamic compilation, also known as *just in time* (JIT) compilation, is implemented from scratch in our RUNTIME library, and turns out to reach high performance for large polynomial functions.

Related Work

In the context of real *midpoint-radius interval arithmetic*, and for specific calculations, such as matrix products, several authors proposed dedicated solutions that mostly perform operations on centers and then rely on fast bounds on radii [18, 19, 20, 23, 25]. In this case, the real gain comes from the ability to exploit HPC (*high performance computing*) solutions to linear algebra, and similar tricks sometimes apply to classical interval methods [16, 21].

The use of SIMD instructions for intervals started in [30], and initially led to a rather modest speed-up, according to the author. Changing the rounding mode for almost each arithmetic operation seriously slows down computations, and might involve a serious stall of a hundred of cycles by breaking FPU pipelines of some hardware such as INTEL X87. To avoid switching rounding modes, the author of [26] uses different independent rounding modes on the X87 and SSE units. Specific solutions to diminish the swap of rounding modes also rely on the *opposite trick* in internal computations or directly in the representation [4, 12, 30]. Modern wide SIMD processors tend to handle roundings more efficiently and AVX-512 even features instructions that directly integrate a rounding mode.

2 Different types of ball arithmetic

2.1 Machine arithmetic

Throughout this paper, we denote by $\mathfrak{R}$ the set of machine floating point numbers. We let $p \geqslant 16$ be the machine precision (which corresponds to the number of fractional bits of the mantissa plus one), and let $E_{\min}$ and $E_{\max}$ be the minimal and maximal exponents (included). For IEEE 754 double precision numbers, this means that $p = 53$, $E_{\min} = -1022$ and $E_{\max} = 1023$. We enlarge $\mathfrak{R}$ with symbols $-\infty$, $+\infty$, and NaN, with their standard meaning.

For our basic arithmetic, we allow for various rounding modes written $\downarrow, \natural, \uparrow, \wr$. The first three rounding modes correspond to IEEE arithmetic with correct rounding (*downwards*, *to the nearest* and *upwards*). The rounding mode $\wr$ corresponds to faithful rounding without any well specified direction. For $x \in \mathbb{R}$, we write $x_\circ \in \mathfrak{R}$ for the approximation of x in $\mathfrak{R}$ with the specified rounding mode $\circ \in \{\downarrow, \natural, \uparrow, \wr\}$. The quantity $\varepsilon_\circ(x) := |x_\circ - x|$ stands for the corresponding rounding error, that may be $+\infty$. Given a single operation $* \in \{+, -, \times, \ldots\}$, it will be convenient to write $x *_\circ y$ for $(x * y)_\circ$. For compound expressions φ, we will also write $\circ[\varphi]$ for the full evaluation of φ using the rounding mode $\circ$. For instance, $\circ[x\,y + a^2\,b] = x_\circ \times_\circ y_\circ +_\circ (a_\circ \times_\circ a_\circ) \times_\circ b_\circ$.

Modern processors usually support *fused-multiply-add* (fma) and *fused-multiply-subtract* (fms) instructions, both for scalar and SIMD vector operands: $\mathrm{fma}_\circ(x, y, z)$ and $\mathrm{fms}_\circ(x, y, z)$ represent the roundings of $x y + z$ and $x y - z$ according to the mode $\circ$. For generalities about these instructions, we refer the reader to the book [15] for instance.

We will denote by $\bar{\varepsilon}_\circ$ any upper bound function for $\varepsilon_\circ$ that is easy to compute. In absence of underflows/overflows, we claim that we may take $\bar{\varepsilon}_\circ(x) = |x_\circ|\, \epsilon_\circ$, with $\epsilon_\circ = 2^{-p+1}$ for $\circ \neq \natural$ and $\epsilon_\natural = 2^{-p}$. Indeed, given $x, y \in \mathbb{R}$ and $* \in \{+, -, \times\}$, let e be the exponent of $x_\circ$, so that $2^e \leqslant |x_\circ| < 2^{e+1}$ and $E_{\min} \leqslant e \leqslant E_{\max}$. Then $\varepsilon_\circ(x) < 2^{e-p+1} \leqslant |x_\circ|\, 2^{-p+1}$ for all rounding modes, and $\varepsilon_\natural(x) \leqslant 2^{e-p} \leqslant |\natural(x)|\, 2^{-p}$.

If we want to allow for underflows during computations, we may safely take $\bar{\varepsilon}_\circ(x) = |x_\circ|\ \epsilon_\circ + 2^{E_{\min}-p+1}$, where $2^{E_{\min}-p+1}$ is the smallest positive subnormal number in $\mathfrak{R}$. Notice that if $x, y \in \mathfrak{R}$, then we may still take $\bar{\varepsilon}_\circ(x \pm y) = |x \pm_\circ y|\, \epsilon_\circ$ since no underflow occurs in that special case. Underflows and overflows will be further discussed in Section 3.3.

2.2 Complex arithmetic and generalizations

In order to describe our algorithms in a flexible context, $\mathbb{A}$ denotes a Banach algebra over $\mathbb{R}$, endowed with a norm written $\|\cdot\|$. The most common examples are $\mathbb{A} = \mathbb{R}$ and $\mathbb{A} = \mathbb{C}$ with $\|z\| = |z|$ for all $z \in \mathbb{A}$.

For actual machine computations, we will denote by $\mathfrak{A}$ the counterpart of $\mathfrak{R}$ when $\mathbb{R}$ is replaced by $\mathbb{A}$. For instance, if $\mathbb{A} = \mathbb{C}$, then we may take $\mathfrak{A} = \mathfrak{R}[\mathrm{i}]$. In other words, if $\mathfrak{R}$ is the C++ type `double`, then we may take `complex<double>` for $\mathfrak{A}$, where `complex` represents the template type available from the standard C++ library, or from the Numerix library of Mathemagix.

For any rounding mode $\circ \in \{\downarrow, \natural, \uparrow, \updownarrow\}$, the notations from the previous section naturally extend to $\mathfrak{A}$. For instance, $\varepsilon_\circ(x) := \|x_\circ - x\|$. Similarly, if $\mathbb{A} = \mathbb{C}$, then $(x + y\,\mathrm{i})_\circ = x_\circ + y_\circ\,\mathrm{i}$. For complex arithmetic we consider the following implementation:

$$
\begin{aligned}
(a + b\,\mathrm{i}) \pm_\circ (c + d\,\mathrm{i}) &:= \circ[a \pm c] + \circ[b \pm d]\,\mathrm{i} && (1)\\
(a + b\,\mathrm{i}) \times_\circ (c + d\,\mathrm{i}) &:= \circ[a\,c - b\,d] + \circ[a\,d + b\,c]\,\mathrm{i} && (2)\\
\|a + b\,\mathrm{i}\|_\circ &:= \circ\left[\sqrt{a^2 + b^2}\right]. && (3)
\end{aligned}
$$

As for $\bar{\varepsilon}_\circ$ we may clearly take $\bar{\varepsilon}_\circ(x) = \|x_\circ\|\ \epsilon_\circ$ in absence of underflows/overflows, and $\bar{\varepsilon}_\circ(x) = \|x_\circ\|\, \epsilon_\circ + \sqrt{2} \cdot 2^{E_{\min}-p+1}$ in general.

Remark 1. For applications to the case when $\mathbb{A} = \mathbb{C}$, it is sometimes interesting to replace computations of norms $\|x\|$ by computations of quick and rough upper bounds $\lceil\!\lceil x \rceil\!\rceil \geqslant \|x\|$. For instance, on architectures where square roots are expensive, one may use

$$\lceil\!\lceil a + b\,\mathrm{i} \rceil\!\rceil \;=\; \min\left(\sqrt{2}\max(|a|, |b|), |a| + |b|\right).$$

2.3 Exact ball arithmetic

Given $a \in \mathbb{A}$ and $r \in \mathbb{R}^{\geqslant} := \{x \in \mathbb{R} : x \geqslant 0\}$, we write $\mathcal{B}(a, r) = \{x \in \mathbb{A} : \|x - a\| \leqslant r\}$ for the *closed ball* with center a and radius r. The set of such balls is denoted by $\mathcal{B}(\mathbb{A}, \mathbb{R})$. One may lift the ring operations $+, -, \times$ in $\mathbb{A}$ to balls in $\mathcal{B}(\mathbb{A}, \mathbb{R})$, by setting:

$$
\begin{aligned}
\mathcal{B}(a, r) \pm \mathcal{B}(b, s) &:= \mathcal{B}(a \pm b, r + s)\\
\mathcal{B}(a, r) \times \mathcal{B}(b, s) &:= \mathcal{B}(a\,b, (\|a\| + r)\,s + \|b\|\,r).
\end{aligned}
$$

These formulas are simplest so as to satisfy the so called *inclusion principle*: given $* \in \{+, -, \times\}$, $x \in B(a, r)$ and $y \in B(b, s)$, we have $x * y \in B(a, r) * B(b, s)$. This arithmetic for computing with balls is called *exact ball arithmetic*. It extends to other operations that might be defined on $\mathbb{A}$, as long as the ball lifts of operations satisfy the inclusion principle. For instance, we may implement fused-multiply-add and fused-multiply-subtract using

$$\operatorname{fma}(B(a,r), B(b,s), B(c,t)) \;:=\; B(\operatorname{fma}(a,b,c), \operatorname{fma}(\|a\|+r, s, \operatorname{fma}(\|b\|, r, t)).$$

2.4 Certified machine ball arithmetic

We will denote by $B(\mathfrak{A}, \mathfrak{R})$ the set of balls with centers in $\mathfrak{A}$ and radii in $\mathfrak{R}$. In order to implement machine ball arithmetic for $B(\mathfrak{A}, \mathfrak{R})$, we need to adjust formulas from the previous section so as to take rounding errors into account. Indeed, the main constraint is the inclusion principle, which should still hold for each operation in such an implementation. The best way to achieve this depends heavily on rounding modes used for operations on centers and radii.

On processors that feature IEEE 754 style rounding, it is most natural to perform operations on centers using any rounding mode $\circ$ (and preferably rounding to the nearest), and operations on radii using upward rounding. This leads to the following formulas:

$$\begin{aligned}
B(a,r) \pm_{\circ} B(b,s) &:= B(a \pm_{\circ} b, \uparrow[r + s + \bar{\varepsilon}_{\circ}(a \pm b)]) \\
B(a,r) \times_{\circ} B(b,s) &:= B(a \times_{\circ} b, \uparrow[(\|a\| + r)\, s + \|b\|\, r + \bar{\varepsilon}_{\circ}(a\, b)]) \\
\operatorname{fma}_{\circ}(B(a,r), B(b,s), B(c,t)) &:= B(\operatorname{fma}_{\circ}(a,b,c), \uparrow[R]) \\
R &\equiv \operatorname{fma}(\|a\| + r, s, \operatorname{fma}(\|b\|, r, t + \bar{\varepsilon}_{\circ}(\operatorname{fma}_{\circ}(a,b,c)))).
\end{aligned}$$

For instance, when using $\bar{\varepsilon}_{\circ}(x)$ of the form $\|x_{\circ}\|\, \epsilon + \eta$ in the last case of fma/fms, this means that three additional instructions are needed with respect to the exact arithmetic from the previous subsection:

$$\begin{aligned}
C &:= \|\operatorname{fma}_{\circ}(a,b,c)\|_{\uparrow} \\
C' &:= \operatorname{fma}_{\uparrow}(C, \epsilon, \eta) \\
\text{radius}' &:= \text{radius} +_{\uparrow} C'.
\end{aligned}$$

In practical implementations, unless $\mathbb{A} = \mathbb{R}$, the radius computations involve many roundings. The correctness of the above formulas is justified by the fact that we systematically use upward rounding for all bound computations; the rounding error for the single operation on the centers is captured by $\bar{\varepsilon}_{\circ}$.

Remark 2. Unfortunately, dynamically switching rounding modes may be expensive on some processors. In that case, one approach is to reorganize computations in such a way that we first perform sufficiently many operations on centers using one rounding mode, and next perform all corresponding operations on radii using upward rounding.

Another approach is to use the same rounding mode for computations on centers and radii. If we take $\circ = \uparrow$ as the sole rounding mode, then we may directly apply the above formulas, but the quality of computations with centers degrades. If we take $\circ \neq \uparrow$, then we have to further adjust the radii so as to take into account the additional rounding errors that might occur during radius computations. For instance, if $\mathbb{A} = \mathbb{R}$, in the cases of addition and subtraction, and in absence of underflows/overflows, we may use

$$B(a,r) \pm_{\circ'} B(b,s) \;:=\; B(a \pm_{\circ} b, \circ[(r + s + \bar{\varepsilon}_{\circ}(a \pm b)) \times (1 + 8 \cdot 2^{-p+1})]),$$

since we have $(r +_\circ s +_\circ \bar{\varepsilon}_\circ(a \pm b)) \times_\circ (1 + 8 \cdot 2^{-p+1}) \geqslant r + s + \bar{\varepsilon}_\circ(a \pm b)$. This arithmetic, sometimes called *rough ball arithmetic*, fits one of the main recommendations of [22]: "Get free from the rounding mode by bounding, roughly but robustly, errors with formulas independent of the rounding mode if needed."

2.5 Transient ball arithmetic

The adjustments which where needed above in order to counter the problem of rounding errors induce a non trivial computational overhead, despite the fact that these adjustments are usually very small. It is interesting to consider an alternative *transient ball arithmetic* for which we simply ignore all rounding errors. In the next Section 3, we will see that it is often possible to treat these rounding errors at a more global level for a complete computation instead of a single operation.

For any rounding mode $\circ$, we will denote by $\tilde{\circ}$ the corresponding "rounding mode" for transient ball arithmetic; the basic operations are defined as follows:

$$\begin{aligned}
B(a,r) \pm_{\tilde{\circ}} B(b,s) &:= B(a \pm_\circ b, \circ[r+s]) \\
B(a,r) \times_{\tilde{\circ}} B(b,s) &:= B(a \times_\circ b, \circ[(\|a\|+r)\, s + \|b\|\, r]) \\
\operatorname{fma}_{\tilde{\circ}}(B(a,r), B(b,s), B(c,t)) &:= B(\operatorname{fma}_\circ(a,b,c), \circ[\operatorname{fma}((\|a\|+r), s, \operatorname{fma}(\|b\|, r, t)]).
\end{aligned}$$

Of course, these formulas do not satisfy the inclusion principle. On processors that allow for efficient switching of rounding modes, it is also possible to systematically use upward rounding for radius computations, with resulting simplifications in the error analysis below.

Let us fix a rounding mode $\circ \in \{\downarrow, \natural, \uparrow, \wr\}$. We assume that we are given a suitable floating point number ϵ in $\mathfrak{R} \cap (\mathbb{N}\, 2^{-p+1})$ such that $\epsilon \leqslant 1/16$ and

$$\begin{aligned}
\|a *_\circ b - a * b\| &\leqslant \|a *_\circ b\|\, \epsilon \\
|\|a\|_\circ - \|a\|| &\leqslant \|a\|_\circ\, \epsilon,
\end{aligned}$$

hold for all $a, b \in \mathfrak{A}$ and $* \in \{+, -, \times\}$, in absence of underflows and overflows. If $\mathbb{A} = \mathbb{R}$, then we may take $\epsilon = \epsilon_\circ$. If $\mathbb{A} = \mathbb{C}$, and assuming that complex arithmetic is implemented using (1–3), then it can be checked that we may take $\epsilon = 4\,\epsilon_\circ$ (see Appendix A). The following lemma provides an error estimate for the radius computation of a transient ball product.

Lemma 3. *For all $a, b \in \mathfrak{A}$ and $r, s \in \mathfrak{R}$ such that the computation of*

$$R = \circ[(\|a\| + r)\, s + \|b\|\, r]$$

involves no underflow or overflow, we have $(\|a\| + r)\, s + \|b\|\, r \leqslant R\, (1 + \epsilon)^4$.

Proof. We have

$$\begin{aligned}
(\|a\|_\circ +_\circ r) \times_\circ s +_\circ \|b\|_\circ \times_\circ r &\geqslant ((\|a\|_\circ +_\circ r) \times_\circ s + \|b\|_\circ \times_\circ t)\, (1+\epsilon)^{-1} \\
&\geqslant ((\|a\|_\circ +_\circ r)\, s + \|b\|_\circ r)\, (1+\epsilon)^{-2} \\
&\geqslant ((\|a\|_\circ + r)\, s + \|b\|_\circ r)\, (1+\epsilon)^{-3} \\
&\geqslant ((\|a\| + r)\, s + \|b\|\, r)\, (1+\epsilon)^{-4}.
\end{aligned}$$

□

3 Evaluation of SLPs

3.1 Straight-line programs

A *straight-line program* Γ over a ring $\mathbb{A}$ is a sequence $\Gamma_1, \ldots, \Gamma_l$ of instructions of the form

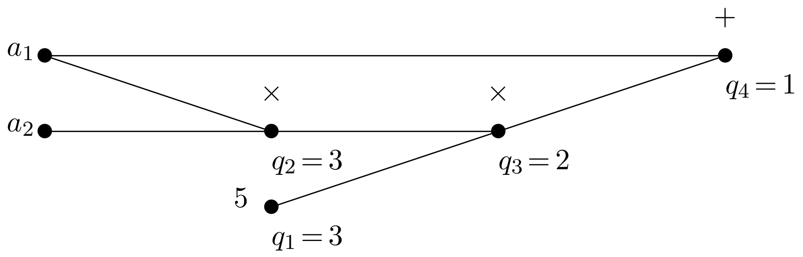
$$\begin{aligned} \Gamma_k &\equiv X_k := C_k \qquad\qquad \text{or} \\ \Gamma_k &\equiv X_k := Y_k * Z_k, \end{aligned}$$

where X_k, Y_k, Z_k are variables in a finite ordered set $\mathcal{V}$, C_k constants in $\mathbb{A}$, and $* \in \{+, -, \times\}$. Variables that appear for the first time in the sequence in the right-hand side of an instruction are called *input variables*. A distinguished subset of the set of all variables occurring at the left-hand side of instructions is called the set of *output variables*. Variables which are not used for input or output are called *temporary variables* and determine the amount of auxiliary memory needed to evaluate the program. The length $l_\Gamma = l$ of the sequence is called the *length* of Γ.

Let $I_1, \ldots, I_m$ be the input variables of Γ and $O_1, \ldots, O_n$ the output variables, listed in increasing order. Then we associate an *evaluation function* E_Γ: $\mathbb{A}^m \rightarrow \mathbb{A}^n$ to Γ as follows: given $(a_1, \ldots, a_m) \in \mathbb{A}^m$, we assign a_i to I_i for $i = 1, \ldots, m$, then evaluate the instructions of Γ in sequence, and finally read off the values of $O_1, \ldots, O_n$, which determine $E_\Gamma(a_1, \ldots, a_m)$.

To each instruction Γ_k, one may associate the *remaining path lengths* q_k as follows. Let $q_l = 1$, and assume that $q_{k+1}, \ldots, q_l$ have been defined for some $k \in \{1, \ldots, l\}$. Then we take $q_k = \max(q_{i_1}, \ldots, q_{i_n}) + 1$, where $i_1 > \cdots > i_n$ are those indices $i > k$ such that Γ_i is of the form $X_i := Y_i \operatorname{\mathsf{op}} Z_i$ with $X_k \in \{Y_i, Z_i\}$ and $X_k \notin \{X_{k+1}, \ldots, X_{i-1}\}$. If no such indices i exist, then we set $q_k = 1$. Similarly, for each input variable I_k we define $q_{I_k} = \max(q_{i_1}, \ldots, q_{i_n}) + 1$, where $i_1 > \cdots > i_n$ are those indices such that Γ_i is of the form $X_i := Y_i \operatorname{\mathsf{op}} Z_i$ with $I_k \in \{Y_i, Z_i\}$ and $I_k \notin \{X_1, \ldots, X_{i-1}\}$. We also define $q_\Gamma = \max(q_{I_1}, \ldots, q_{I_m}, q_{i_1}, \ldots, q_{i_n})$, where $I_1, \ldots, I_m$ are the input variables of Γ and $i_1, \ldots, i_n$ all indices i such that Γ_i is of the form $X_i := C_i$.

Example 4. Let us consider $\Gamma = (x_1 := 5, x_2 := a_1 \times a_2, x_1 := x_1 \times x_2, x_3 := x_1 + a_1)$, of length $l = 4$. The input variables are a_1 and a_2, and we distinguish x_3 as the sole output variable. This SLP thus computes the function $5\, a_1\, a_2 + a_1$. The associated computation graph, together with remaining path lengths are as pictured:



3.2 Transient evaluation

Consider the "semi-exact" ball arithmetic in which all computations on centers are done using a given rounding mode $\circ$ and all computations on radii are exact. More precisely, we take

$$\begin{aligned} \mathcal{B}(a, r^*) \pm^* \mathcal{B}(b, s^*) &= \mathcal{B}(a +_\circ b, r^* + s^* + \bar{\varepsilon}_\circ(a + b)) \\ \mathcal{B}(a, r^*) \times^* \mathcal{B}(b, s^*) &= \mathcal{B}(a \times_\circ b, (\|a\| + r^*)\, s^* + \|a\|\, s^* + \bar{\varepsilon}_\circ(a\, b)), \end{aligned}$$

for any $a, b \in \mathfrak{A}$ and $r^*, s^* \in \mathbb{R}^{\geqslant}$. We wish to investigate how far the transient arithmetic from Section 2.5 can deviate from this idealized arithmetic (which satisfies the inclusion principle). We will write $H_k = \frac{1}{1} + \cdots + \frac{1}{k}$ for the k-th harmonic number and $H_{k,l} = H_l - H_k$ for all $l \geqslant k$.

Theorem 5. *Let Γ be a SLP of length l as above and let $\alpha > 0$ be an arbitrary parameter such that $1 + \alpha > (1 + \epsilon)^{4 q_\Gamma}$. Consider two evaluations of Γ with two different ball arithmetics. For the first evaluation, we use the above semi-exact arithmetic with $\bar{\varepsilon}_\circ(x) = \|x_\circ\| \epsilon$. For the second evaluation, we use transient ball arithmetic with the same arithmetic for centers, and the additional property that any input or constant ball $B(a, r^*)$ is replaced by a larger ball $B(a, r)$ with*

$$r \geqslant \max (\|a\| ((1 + \epsilon)^{\beta q_\Gamma} - 1), (1 + \alpha) r^*),$$

where

$$\beta \geqslant \max \left(3, \frac{1 + \alpha}{\alpha} \gamma\right), \qquad \gamma \geqslant H_{q_\Gamma} (1 + \epsilon)^{4 q_\Gamma} \frac{\alpha}{1 + \alpha} \left(1 - \frac{(1 + \epsilon)^{4 q_\Gamma}}{1 + \alpha}\right)^{-1}.$$

If no underflow or overflow occurs during the second evaluation, then for all $B(c, t^*)$ in the output of the first evaluation with corresponding entry $B(c, t)$ for the second evaluation, we have $t^* \leqslant t$.

Proof. It will be convenient to systematically use the star superscript for the semi-exact radius evaluation and no superscript for the transient evaluation.

Let $B(c_k, t_k)$ be the ball value of the variable X_k just after evaluation of $\Gamma_1, \ldots, \Gamma_k$ using transient ball arithmetic. Let us show by induction over k that

$$t_k \geqslant \|c_k\| ((1 + \epsilon)^{\beta q_k} - 1).$$

So assume that the hypothesis holds for all strictly smaller values of k. If Γ_k is of the form $X_k := C_k$, then we are done by assumption since $q_k \leqslant q_\Gamma$. If Γ_k is of the form $X_k := Y_k * Z_k$, then we claim that Y_k contains a ball $B(a, r)$ with

$$r \geqslant \|a\| ((1 + \epsilon)^{\beta (q_k + 1)} - 1).$$

Indeed, this holds by assumption if Y_k is an input variable. Otherwise, let $i < k$ be largest with $X_i = Y_k$. Then $q_i \geqslant q_k + 1$ by the construction of q_i, whence our claim follows by the induction hypothesis. In a similar way, Z_k contains a ball $B(b, s)$ with $s \geqslant \|b\| ((1 + \epsilon)^{\beta (q_k + 1)} - 1)$.

Having shown our claim, let us first consider the case when $* \in \{+, -\}$. Then we get

$$\begin{aligned} r + s &\geqslant (\|a\| + \|b\|) ((1 + \epsilon)^{\beta (q_k + 1)} - 1) \\ &\geqslant \|a \pm b\| ((1 + \epsilon)^{\beta (q_k + 1)} - 1). \end{aligned}$$

Using the inequalities $((1 + \epsilon)^A - 1) (1 + \epsilon)^{-1} \geqslant (1 + \epsilon)^{A - 1} - 1$, $1 - \epsilon \geqslant (1 + \epsilon)^{-2}$, and the fact that $\beta \geqslant 3$, we obtain:

$$\begin{aligned} t_k &= r +_\circ s \\ &\geqslant (r + s) (1 + \epsilon)^{-1} && (4) \\ &\geqslant \|a \pm b\| ((1 + \epsilon)^{\beta (q_k + 1)} - 1) (1 + \epsilon)^{-1} \\ &\geqslant \|c_k\| ((1 + \epsilon)^{\beta (q_k + 1)} - 1) (1 - \epsilon) (1 + \epsilon)^{-1} && (5) \\ &\geqslant \|c_k\| ((1 + \epsilon)^{\beta q_k} - 1), \end{aligned}$$

as desired. In the same way, if $* = \times$, then

$$\begin{aligned}(\|a\| + r)\, s + \|b\|\, r &\geqslant \|a\|\, s + \|b\|\, r\\ &\geqslant 2\, \|a\, b\|\, ((1+\epsilon)^{\beta(q_k+1)} - 1),\end{aligned}$$

whence, using Lemma 3, and the fact that $2/(1+\epsilon)^{-4} > 1/(1+\epsilon)$,

$$\begin{aligned}t_k &= (\|a\|_\circ +_\circ r) \times_\circ s +_\circ \|b\|_\circ \times_\circ r\\ &\geqslant ((\|a\| + r)\, s + \|b\|\, r)\, (1+\epsilon)^{-4} \qquad (6)\\ &\geqslant 2\, \|a\, b\|\, ((1+\epsilon)^{\beta(q_k+1)} - 1)\, (1+\epsilon)^{-4}\\ &\geqslant \|c_k\|\, ((1+\epsilon)^{\beta(q_k+1)} - 1)\, (1-\epsilon)\, (1+\epsilon)^{-1}\\ &\geqslant \|c_k\|\, ((1+\epsilon)^{\beta q_k} - 1), \qquad (7)\end{aligned}$$

which completes our claim by induction.

For all $k \in \{1, \ldots, l\}$, we introduce

$$\gamma_k = \left(\frac{1}{1+\alpha} + \frac{\alpha}{\gamma\,(1+\alpha)}\, H_{q_k, q_\Gamma}\right) (1+\epsilon)^{4(q_\Gamma - q_k)},$$

so that $(1+\alpha)^{-1} \leqslant \gamma_k \leqslant 1$. Using a second induction over k, let us next prove that

$$t_k^* \leqslant \gamma_k\, t_k.$$

Assume that this inequality holds up until order $k-1$. If Γ_k is of the form $X_k := C_k$, then we are done by the fact that $\gamma_k \geqslant (1+\alpha)^{-1}$.

If Γ_k is of the form $X_k := Y_k \pm Z_k$, then let $i, j < k$ be largest with $X_i = Y_k$ and $X_j = Z_k$. Then $\min(q_i, q_j) \geqslant q_k + 1$, and

$$\max(\gamma_i, \gamma_j) \leqslant \left(\frac{1}{1+\alpha} + \frac{\alpha}{\gamma\,(1+\alpha)}\, H_{q_k+1, q_\Gamma}\right) (1+\epsilon)^{4(q_\Gamma - (q_k+1))},$$

$$\eta_k := \frac{\|c_k\|\, \epsilon}{t_k} \leqslant \frac{\epsilon}{((1+\epsilon)^{\beta q_k} - 1)} \leqslant \frac{1}{\beta\, q_k} \leqslant \frac{\alpha}{q_k\, \gamma\, (1+\alpha)}.$$

In particular, $(\max(\gamma_i, \gamma_j) + \eta_k)\, (1+\epsilon)^4 \leqslant \gamma_k$. With r and s as above, it follows that

$$\begin{aligned}t_k^* &= r^* + s^* + \|c_k\|\, \epsilon\\ &\leqslant ((r+s) \max(\gamma_i, \gamma_j) + \eta_k\, t_k)\\ &\leqslant (\max(\gamma_i, \gamma_j) + \eta_k)\, t_k\, (1+\epsilon) \qquad (8)\\ &\leqslant \gamma_k\, t_k.\end{aligned}$$

If Γ_k is of the form $X_k := Y_k \times Z_k$, then thanks to Lemma 3, a similar computation yields

$$\begin{aligned}t_k^* &= (\|a\| + r^*)\, s^* + \|b\|\, r^* + \|c_k\|\, \epsilon\\ &\leqslant ((\|a\| + r)\, s + \|b\|\, r) \max(\gamma_i, \gamma_j) + \eta_k\, t_k\\ &\leqslant (\max(\gamma_i, \gamma_j) + \eta_k)\, t_k\, (1+\epsilon)^4 \qquad (9)\\ &\leqslant \gamma_k\, t_k.\end{aligned}$$

This completes the second induction and the proof of this theorem. □

For fixed α and $q_\Gamma = o(1/\varepsilon)$, we observe that $(1+\epsilon)^{\beta q_\Gamma} - 1 = O(\epsilon\, q_\Gamma \log q_\Gamma)$. The value of the parameter α may be optimized for given SLPs and inputs. Without entering details, α should be taken large when inputs are accurate, and small when inputs are rough, as encountered for instance within subdivision algorithms.

3.3 Managing underflows and overflows

In Theorem 5, we have assumed the absence of underflows and overflows. There are several strategies for managing such exceptions, whose efficiencies heavily depend on the specific kind of hardware being used.

One strategy to manage exceptions is to use the status register of an IEEE 754 FPU. The idea is to simply clear the underflow and overflow flags of the status register, then perform the evaluation, and finally check both flags at the end. Whenever an overflow occurred during the evaluation, then we may set the radii of all results to plus infinity, thereby preserving the inclusion principle. If an underflow occurred (which is quite unlikely), then we simply reevaluate the entire SLP using a more expensive, fully certified ball arithmetic.

When performing all computations using the IEEE 754 rounding-to-nearest mode $\natural$, we also notice that consulting the status register can be avoided for managing overflows. Indeed, denoting by H the largest positive finite number in $\mathfrak{R}$, we have $x_\natural = -\infty$ for all $x < -H$ and $x_\natural = +\infty$ for all $x > H$. Consequently, whenever a computation on centers overflows, the corresponding radius will be set to infinity.

From now on, we assume the absence of overflows and we focus on underflows. In addition to the constant ϵ, we suppose given an other positive constant η in $\mathfrak{R}$ such that

$$\begin{aligned} \bar{\varepsilon}_\circ(x) &= \|x_\circ\|\,\epsilon + \eta && (10)\\ \|a \times_\circ b - a\,b\| &\leqslant \|a \times_\circ b\|\,\epsilon + \eta && (11)\\ |\|a\|_\circ - \|a\|| &\leqslant \|a\|_\circ \epsilon + \eta && (12) \end{aligned}$$

for all $x \in \mathbb{A}$ and $a, b \in \mathfrak{A}$. For instance, if $\mathbb{A} = \mathbb{R}$, then we may take $\eta = 2^{E_{\min} - p + 1}$. If $\mathbb{A} = \mathbb{C}$, and assuming the arithmetic from Section 2.2, then it is safe to take $\eta = 8 \cdot 2^{(E_{\min} - p + 1)/2}$ (see Appendix A). In addition to the method based on the status register, the following strategies can be used for managing underflows.

Using upward rounding for radii. If it is possible to use different rounding modes for computations on centers and radii, then we may round centers to the nearest and radii upwards. In other words, we replace the transient arithmetic from Section 2.5 by

$$\begin{aligned} \mathcal{B}(a, r) \pm_{\check{\circ}} \mathcal{B}(b, s) &:= \mathcal{B}(a \pm_\circ b, \uparrow[r + s])\\ \mathcal{B}(a, r) \times_{\check{\circ}} \mathcal{B}(b, s) &:= \mathcal{B}(a \times_\circ b, \uparrow[(\|a\| + r)\,s + \|b\|\,r]). \end{aligned}$$

This arithmetic makes Theorem 5 hold without the assumption on the absence of underflows, and provided that

$$3\,\eta\,((1+\epsilon)^{\beta q_\Gamma} - 1) \leqslant 2^{E_{\min} - p + 1}. \tag{13}$$

Indeed, inequalities (5), (8), and (9) immediately hold, even without extra factors $(1 + \epsilon)$. It remains to prove that (6) also holds. From $\|a \times_\circ b - a\,b\| \leqslant \|a \times_\circ b\|\,\epsilon + \eta$ we have $\|a \times_\circ b\| \leqslant (\|a\,b\| + \eta)\,(1 - \epsilon)^{-1}$. Therefore, if $\|a\,b\| \geqslant \eta\,(1 - 2\,\epsilon)^{-1}$, then $\|a \times_\circ b\| \leqslant 2\,\|a\,b\|$ holds, whence inequality (6). Otherwise $\|a \times_\circ b\| \leqslant \eta\,(1 + (1 - 2\,\epsilon)^{-1})\,(1 - \epsilon)^{-1} \leqslant 3\,\eta$, and the extra assumption (13) directly implies $t_k \geqslant \|c_k\|\,((1 + \epsilon)^{\beta q_k} - 1)$ because $t_k \geqslant 2^{E_{\min} - p + 1}$ always holds by construction.

Adding corrective terms to the radii. Another strategy is to manually counter underflows by adding corrective terms to the radii. This yields the following arithmetic which is half way between transient and certified:

$$\begin{aligned} \mathcal{B}(a, r) \pm_{\check{\circ}} \mathcal{B}(b, s) &:= \mathcal{B}(a \pm_\circ b, r +_\circ s)\\ \mathcal{B}(a, r) \times_{\check{\circ}} \mathcal{B}(b, s) &:= \mathcal{B}(a \times_\circ b, \circ[\max((\|a\| + r)\,s + \|b\|\,r, 2\,\eta/\epsilon)]) \end{aligned}$$

No corrections are necessary for additions and subtractions which never provoke underflows. As to multiplication, Lemma 3 admits Lemma 6 below as its analogue in the case when $\mathbb{A} = \mathbb{R}$. Using this, it may again be shown that Theorem 5 holds without the assumption on the absence of underflows, and provided that

$$3\,\epsilon\,((1+\epsilon)^{\beta q_{\mathrm{T}}} - 1) \leqslant 2. \tag{14}$$

Indeed, inequalities (5), (8), and (9) immediately hold. It remains to prove that (6) also holds. The case $\|a\,b\| \geqslant \eta\,(1-2\,\epsilon)^{-1}$ is handled as for the previous strategy. Otherwise, we have $\|a \times_\circ b\| \leqslant \eta\,(1+(1-2\,\epsilon)^{-1})\,(1-\epsilon)^{-1} \leqslant 3\,\eta$, and the extra assumption (14) directly implies $t_k \geqslant \|c_k\|\,((1+\epsilon)^{\beta q_k} - 1)$ because $t_k \geqslant 2\,\eta/\epsilon$ holds by construction.

Lemma 6. *For all $a, b \in \mathfrak{R}$ and $r, s \in \mathfrak{R}$, letting $\check{R} = \circ[\max((|a|+r)\,s + |b|\,r, 2\,\eta/\epsilon)]$ and $R = (|a|+r)\,s + |b|\,r$, we have $R \leqslant \check{R}\,(1+\epsilon_\circ)^4$.*

Proof. We have

$$\begin{array}{rcl} (|a| +_\circ r) \times_\circ s +_\circ |b| \times_\circ r & \geqslant & ((|a| +_\circ r) \times_\circ s + |b| \times_\circ t)\,(1+\epsilon_\circ)^{-1} \\ & \geqslant & ((|a| +_\circ r)\,s + |b|\,r - 2\,\eta)\,(1+\epsilon_\circ)^{-2} \\ & \geqslant & ((|a| + r)\,s + |b|\,r)\,(1+\epsilon_\circ)^{-3} - 2\,\eta. \end{array}$$

It follows that $\check{R} \geqslant R\,(1+\epsilon_\circ)^{-3} - \epsilon\,\check{R}$. □

Remark 7. If $\mathbb{A} \neq \mathbb{R}$, then the above method still applies under the condition that all norm computations are replaced by reliable upper bounds. In other words, assuming that $\llbracket x \rrbracket \in \mathfrak{R}$ satisfies $\llbracket x \rrbracket \geqslant \|x\|$ for all $x \in \mathbb{A}$, we may take

$$B(a, r) \times_{\check{\circ}} B(b, s) \;:=\; B(a \times_\circ b, \circ[\max((\llbracket a \rrbracket + r)\,s + \llbracket b \rrbracket\,r, 2\,\eta/\epsilon)]).$$

Subnormal numbers. On some processors, computations with subnormal numbers incur a significant performance penalty. Such processors often support a "fast math" mode in which subnormal numbers in input and output are set to zero, which does not comply with the IEEE 754 standard. For instance, SSE and AVX technologies include two flags in the MXCSR control register dedicated to this purpose, namely *denormal-are-zero* and *flush-to-zero*. When setting these two flags together, we must take $\bar{\varepsilon}_\circ(x) = |x_\circ|\ \epsilon_\circ + 2^{E_{\min}}$ over $\mathbb{R}$ as a protection against this additional error, $2^{E_{\min}}$ being the smallest positive normal number in $\mathfrak{R}$. Roughly speaking, the above strategies may be adapted by replacing $E_{\min}$ by $E_{\min} + p$ in the proofs.

4 Implementation

Our new algorithms have been implemented in Mathemagix. The source code, tests, and benchmark files are freely available from the SVN server of Mathemagix *via* http://gforge.inria.fr/projects/mmx/, from revision 10356. In this section, we briefly present our implementation. We first describe the implemented strategies for evaluations over the standard numeric types `double` and `complex<double>`. We next consider balls over these types, and finally say a few words about vectorized variants.

4.1 Software overview

Our implementation is divided into C++ and MATHEMAGIX libraries. Import/export mechanisms between these two languages are rather easy, as described in [7, 8]. The C++ library MULTIMIX contains several implementations of multivariate polynomials, including SLPs (type `slp_polynomial` defined in `slp_polynomial.hpp`), naive interpreted evaluation (`slp_polynomial_naive.hpp`), compilation into dynamic libraries loaded *via* `dlopen` (`slp_polynomial_compiled.hpp`), and fast JIT compilation.

JIT compilation, is a classical technique, traditionally used in scientific computing: when an expression such as a SLP needs to be evaluated at many points, it pays off to compile the expression and then perform the evaluations using fast executable code. When allowed by the operating system, it suffices to compile SLPs into executable memory regions, without temporary files. Since SLPs are very basic programs, there is no special need to appeal to general purpose compilers. In MULTIMIX, such a compilation is supported for `double` coefficients, for SSE2 enabled CPU, and System V AMD64 application binary interface (ABI), which covers most 64-bit UNIX-like platforms.

To develop a confortable JIT library dedicated to SLPs, we turned to the MATHEMAGIX language [8], from which we benefit of extensible union types and fast pattern matching. Our RUNTIME library provides basic facilities to produce JIT executable code from assembly language. On the top of it, our JUSTINLINE library defines a templated SLP data type with additional JIT facilities. This includes *common subexpression simplification*, *constant simplification*, *register allocation*, and *vectorization*. Recall that vectorization is the ability to transform a SLP over a given ring type `R` to a SLP over vectors of `R`. This is especially useful in order to exploit SIMD technologies.

4.2 Timings for numeric types

In order to estimate the concrete overhead of ball arithmetic, we first focus on timings for `double` and `complex<double>`. In the rest of this article, timings are measured on a platform equipped with an INTEL(R) CORE(TM) i7-4770 CPU at 3.40 GHz and 8 GB of 1600 MHz DDR3, which includes AVX2 and FMA technologies. The platform runs the JESSIE GNU DEBIAN operating system with a 64 bit LINUX kernel version 3.14. Care has been taken for avoiding *CPU throttling* and *Turbo Boost* issues while measuring timings. We compile with GCC [31] version 4.9.2 with options `-O3 -mavx2 -mfma -mfpmath=sse`.

In Table 1, column "`double`" displays timings for evaluating a multivariate polynomial over `double`, with 10 variables, made of 100 terms, built from random monomials of partial degrees at most 10. We build the SLP using a dedicated algorithm implemented in `multimix/dag_polynomial.hpp`. Support for this specific coefficient type may be found in `multimix/slp_polynomial_double.hpp`. This example, with the corresponding functions to reproduce these timings are available in `multimix/bench/slp_polynomial_bench.cpp`. The evaluation of this SLP takes 1169 products and 100 sums.

The first row corresponds to using the naive interpreted evaluation available by default from MULTIMIX. In the second row, the SLP is printed into a C++ file, which is then compiled into a dynamic library with options `-O3 -fPIC -mavx2 -mfma -mfpmath=sse`. The compilation and dynamic loading take 260ms, which corresponds to about 10^4 naive evaluations.

The third row concerns JIT compilation from MULTIMIX, which achieves compilation for SSE2 legacy scalar instructions in 50µs, with no optimization and no register allocation. Notice that this only corresponds to less than 30 naive evaluations. However the lack of optimization implies a loss of a factor more than two with respect to GCC.

The fourth row is for our JIT implementation in the JUSTINLINE library. Compilation performs register allocation as sole optimization, and takes about 8ms: this is much faster than with GCC, but still higher than in MULTIMIX. Nevertheless, the integration of more powerful optimization features in the MATHEMAGIX compiler should progressively reduce this gap. The implementation of additional optimizations in the JUSTINLINE library should also make it possible to get closer to the evaluation performance *via* GCC.

The second column of Table 1 shows similar timings for the `complex<double>` type from the NUMERIX library. We did not implement the JIT strategy in MULTIMIX. Of course, the performance ratio between compiled and interpreted strategies is much lower than for `double`.

	`double`	`complex<double>`
MULTIMIX, naive	2.1	3.4
MULTIMIX, compiled	0.29	1.3
MULTIMIX, JIT	0.84	N/A
JUSTINLINE, JIT	0.43	1.4

Table 1. Polynomial evaluation with 100 terms of degree 10 in 10 variables, in µs.

4.3 Timings for balls

We are now interested in measuring the speed-up of our evaluation strategies over balls. The first column of Table 2 shows timings for balls over `double` (i.e. $\mathbb{A} = \mathbb{R}$). Early versions of the MATHEMAGIX libraries already contained a C99 portable implementation of ball arithmetic in the NUMERIX library (see `numerix/ball.hpp` and related files). The first row of Table 2 is obtained with naive evaluation over this portable arithmetic when rounding centers to the nearest and radii upwards. We observe rather high timings involved by the compiler and the mathematical library. The second row is similar but concerns the rough ball arithmetic of Remark 2. The row "Naive, transient" is the interpreted transient ball arithmetic from MULTIMIX. The next three rows correspond to dynamic compilation *via* GCC and JUSTINLINE; they reveal the gain of the JIT strategy.

Notice that we carefully tuned the assembly code generated by our SLP compiler. For instance, if `ymm0` contains -0.0 and if `ymm1` contains a center a, then $-a$ is obtained as `vxorpd ymm0 ymm1 ymm2`, and $|a|$ as `vandnpd ymm0 ymm1 ymm2`. The latency and throughput of both these instructions are a single cycle and no branchings are required to compute $|a|$. In this way, each transient addition/subtraction takes 2 cycles, and each product 6 cycles. For rough arithmetic this increases to 5 and 9 cycles respectively. The gain of the transient arithmetic is thus well reflected in practice, since our example essentially performs products. Comparing to Table 1, we observe that transient arithmetic is just about 4 times slower than numeric arithmetic. This turns out to be competitive with interval arithmetic, where each interval product usually requires 8 machine multiplications and 6 min/max operations.

The second column of timings in Table 2 is similar to the first one, but with balls over `complex<double>`. The computation of norms is expensive in this case because the scalar square root instruction takes 14 CPU cycles. In order to reduce this cost, we rewrite SPLs so that norms of products are computed as products of norms. Taking care of using or simulating upwards rounding, this involves a slight loss in precision but does not invalidate the results. At the end, comparing to Table 1, we are glad to observe that our new transient arithmetic strategy is only about twice as expensive as standard numeric evaluations.

	Ball over `double`	Ball over `complex<double>`
Naive, rounded, C99	62	130
Naive, rough, C99	66	200
Naive, transient	14	18
Compiled, transient	2.0	4.0
JIT, rough	3.2	4.5
JIT, transient	1.8	3.1

Table 2. Polynomial evaluation with 100 terms of degree 10 in 10 variables, in µs.

4.4 Vectorization

In order to profit from SIMD technologies, we also implemented vectorization in our Justinline library, in the sense that the executable code runs over SSE or AVX hardware vectors. The expected speed-up of 2 or 4 is easily observed for `double`, `complex<double>`, and balls over `double`. For balls over `complex<double>`, a penalty occurs, because the vectorial square root instruction is about twice slower than its scalar counterpart. For instance, the evaluation of a univariate polynomial in degree 1000 with Hörner's method takes about 5000 CPU cycles over `double` (each fma instruction takes the expected latency), 13000 cycles over `complex<double>`, and 16000 cycles for transient balls of `double`, with both scalar and vectorial instructions. As for transient balls over `complex<double>`, scalar instructions amount to about 17000 cycles, while vectorial ones take about 28000 cycles.

5 Conclusion

In this paper, we have shown how to implement highly efficient ball arithmetic dedicated to polynomial evaluation. In the near future, we expect significant speed-ups in our numerical system solvers implemented within Mathemagix.

It is interesting to notice in Table 2 that the code generated by our rather straightforward JIT compiler for SLPs is more efficient than the code produced by GCC. This suggests that it might be worthwhile to put more efforts into the development of specific JIT compilers for SLPs dedicated to high performance computing.

We also plan to adapt the present results to standard intervals, that are useful for real solving. In that case, we replace input and constant intervals $[x, y]$ by intervals of the form $[x - \delta, y + \delta]$, where $\delta = \kappa \max(|x|, |y|, |y - x|)\,\epsilon$ for suitable κ. We next proceed as usual, but without any assumption on the rounding mode.

We finally notice that interval arithmetic benefits from hardware accelerations on many current processors, as soon as IEEE 754 style rounding is integrated in an efficient manner. An interesting question is whether ball arithmetic might benefit from similar hardware accelerations. In particular, is it possible to integrate rounding errors more efficiently into error bounds (the radii of balls)? This might for instance be achieved using an *accumulate-rounding-error* instruction for computing a guaranteed upper bound for $r + \kappa\, \bar{\varepsilon}_{\circ}(x)$ as a function of $r, \kappa, x \in \mathfrak{R}$.

Bibliography

[1] G. Alefeld and J. Herzberger. *Introduction to interval analysis*. Academic Press, New York, 1983.

[2] D. Bates, J. Hauenstein, A. Sommese, and C. Wampler. Bertini: software for numerical algebraic geometry. `http://www.nd.edu/~sommese/bertini/`, 2006.

[3] C. Beltrán and A. Leykin. Certified numerical homotopy tracking. *Experiment. Math.*, 21(1):69–83, 2012.

[4] F. Goualard. Fast and correct SIMD algorithms for interval arithmetic. Technical Report, Université de Nantes, 2008. `https://hal.archives-ouvertes.fr/hal-00288456`.

[5] J. van der Hoeven. Ball arithmetic. In A. Beckmann, Ch. Gaßner, and B. Löwe, editors, *Logical approaches to Barriers in Computing and Complexity*, number 6 in Preprint-Reihe Mathematik, pages 179–208. Ernst-Moritz-Arndt-Universität Greifswald, 2010. International Workshop.

[6] J. van der Hoeven. Reliable homotopy continuation. Technical Report, CNRS & École polytechnique, 2011. `http://hal.archives-ouvertes.fr/hal-00589948`.

[7] J. van der Hoeven and G. Lecerf. Interfacing Mathemagix with C++. In M. Monagan, G. Cooperman, and M. Giesbrecht, editors, *Proc. ISSAC '13*, pages 363–370. ACM, 2013.

[8] J. van der Hoeven and G. Lecerf. *Mathemagix User Guide*. CNRS & École polytechnique, France, 2013. `http://hal.archives-ouvertes.fr/hal-00785549`.

[9] J. van der Hoeven, G. Lecerf, B. Mourrain et al. Mathemagix. 2002. `http://www.mathemagix.org`.

[10] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied interval analysis*. Springer, London, 2001.

[11] U. W. Kulisch. *Computer Arithmetic and Validity. Theory, Implementation, and Applications*. Number 33 in Studies in Mathematics. De Gruyter, 2008.

[12] B. Lambov. Interval arithmetic using SSE-2. In P. Hertling, C. M. Hoffmann, W. Luther, and N. Revol, editors, *Reliable Implementation of Real Number Algorithms: Theory and Practice*, volume 5045 of *Lect. Notes Comput. Sci.*, pages 102–113. Springer Berlin Heidelberg, 2008.

[13] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, N.J., 1966.

[14] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM Press, 2009.

[15] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[16] Hong Diep Nguyen. *Efficient algorithms for verified scientific computing: numerical linear algebra using interval arithmetic*. PhD thesis, École Normale Supérieure de Lyon, Université de Lyon, 2011.

[17] A. Neumaier. *Interval methods for systems of equations*. Cambridge university press, Cambridge, 1990.

[18] T. Ogita and S. Oishi. Fast inclusion of interval matrix multiplication. *Reliab. Comput.*, 11(3):191–205, 2005.

[19] K. Ozaki, T. Ogita, and S. Oishi. Tight and efficient enclosure of matrix multiplication by using optimized BLAS. *Numer. Linear Algebra Appl.*, 18:237–248, 2011.

[20] K. Ozaki, T. Ogita, S. M. Rump, and S. Oishi. Fast algorithms for floating-point interval matrix multiplication. *J. Comput. Appl. Math.*, 236(7):1795–1814, 2012.

[21] N. Revol and Ph. Théveny. Parallel implementation of interval matrix multiplication. *Reliab. Comput.*, 19(1):91–106, 2013.

[22] N. Revol and Ph. Théveny. Numerical reproducibility and parallel computations: issues for interval algorithms. *IEEE Trans. Comput.*, 63(8):1915–1924, 2014.

[23] S. M. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3):534–554, 1999.

[24] S. M. Rump. Verification methods: rigorous results using floating-point arithmetic. *Acta Numer.*, 19:287–449, 2010.

[25] S. M. Rump. Fast interval matrix multiplication. *Numer. Algor.*, 61(1):1–34, 2012.

[26] N. Stolte. Arbitrary 3D resolution discrete ray tracing of implicit surfaces. In E. Andres, G. Damiand, and P. Lienhardt, editors, *Discrete Geometry for Computer Imagery*, volume 3429 of *Lect. Notes Comput. Sci.*, pages 414–426. Springer Berlin Heidelberg, 2005.

[27] A. J. Sommese and C. W. Wampler. *The Numerical Solution of Systems of Polynomials Arising in Engineering and Science*. World Scientific, 2005.

[28] J. Verschelde. *Homotopy continuation methods for solving polynomial systems*. PhD thesis, Katholieke Universiteit Leuven, 1996.

[29] J. Verschelde. PHCpack: a general-purpose solver for polynomial systems by homotopy continuation. *ACM Trans. Math. Software*, 25(2):251–276, 1999. Algorithm 795.

[30] J. W. Von Gudenberg. Interval arithmetic on multimedia architectures. *Reliab. Comput.*, 8(4):307–312, 2002.

[31] GCC, the GNU Compiler Collection. Software available at http://gcc.gnu.org, from 1987.

Appendix A Rounding errors for complex arithmetic

Let $x = a + b\,\mathrm{i}$ and $y = c + d\,\mathrm{i}$ be two complex numbers in $\mathfrak{R}[\mathrm{i}]$. We will show that

$$\|x \pm_\circ y - x \pm y\| \leqslant \|x \pm_\circ y\|\, \epsilon_\circ \tag{15}$$

$$\|x \times_\circ y - x\,y\| \leqslant \|x \times_\circ y\|\, 4\,\epsilon_\circ + 5 \cdot 2^{E_{\min} - p + 1} \tag{16}$$

$$|\|x\|_\circ - \|x\|| \leqslant \|x\|_\circ\, 3\,\epsilon_\circ + 3 \cdot 2^{(E_{\min} - p + 1)/2}. \tag{17}$$

In addition, in absence of underflows, the terms in $2^{E_{\min} - p + 1}$ may be discarded.

For short we set $\iota := 2^{E_{\min} - p + 1}$, and we shall freely use the assumption $\varepsilon_\circ \leqslant 2^{-16}$. The first bound (15) is immediate. As for the second one, it will be convenient to use the norm $\|a + b\,\mathrm{i}\|_1 = |a| + |b|$, that satisfies $\|a + b\,\mathrm{i}\| \leqslant \|a + b\,\mathrm{i}\|_1 \leqslant \sqrt{2}\, \|a + b\,\mathrm{i}\|$. We begin with combining

$$|a \times_\circ c -_\circ b \times_\circ d - (a\,c - b\,d)| \leqslant |a \times_\circ c -_\circ b \times_\circ d|\, \epsilon_\circ + (|a \times_\circ c| + |b \times_\circ d|)\, \epsilon_\circ + 2\,\iota$$

and

$$|a \times_\circ d +_\circ b \times_\circ c - (a\,d + b\,c)| \leqslant |a \times_\circ d -_\circ b \times_\circ c|\, \epsilon_\circ + (|a \times_\circ d| + |b \times_\circ c|)\, \epsilon_\circ + 2\,\iota$$

into

$$\|x \times_\circ y - x\,y\|_1 \leqslant \|x \times_\circ y\|_1\, \epsilon_\circ + \|x\|_1\, \|y\|_1\, \epsilon_\circ\, (1 - \epsilon_\circ)^{-1} + (4 + 8\,\epsilon_\circ)\,\iota.$$

We deduce that

$$\|x \times_\circ y - x\,y\| \leqslant \|x \times_\circ y\|\, \sqrt{2}\,\epsilon_\circ + \|x\|\, \|y\|\, 2\,\epsilon_\circ\, (1 - \epsilon_\circ)^{-1} + (4 + 8\,\epsilon_\circ)\,\iota,$$

whence

$$\|x\,y\| \leqslant \left(\|x \times_\circ y\|\, \left(1 + \sqrt{2}\,\epsilon_\circ\right) + (4 + 8\,\epsilon_\circ)\,\iota\right) / (1 - 2\,\epsilon_\circ\, (1 - \epsilon_\circ)^{-1}),$$

which simplifies into

$$\|x\,y\| \leqslant \|x \times_\circ y\|\, (1 + 4\,\epsilon_\circ) + (4 + 11\,\epsilon_\circ)\,\iota.$$

It follows that

$$\|x \times_\circ y - x\,y\| \leqslant \|x \times_\circ y\|\, \left(\sqrt{2}\,\epsilon_\circ + 2\,\epsilon_\circ\, (1 + 4\,\epsilon_\circ)\, (1 - \epsilon_\circ)^{-1}\right) + (4 + 17\,\epsilon_\circ)\,\iota,$$

which simplifies into (16).

As to (17), we begin with $|a \times_\circ a - a^2| \leqslant |a \times_\circ a|\, \epsilon_\circ + \iota$ and $|b \times_\circ b - b^2| \leqslant |b \times_\circ b|\, \epsilon_\circ + \iota$, that give

$$|a \times_\circ a + b \times_\circ b - (a^2 + b^2)| \;\leqslant\; |a \times_\circ a + b \times_\circ b|\, \epsilon_\circ + 2\,\iota.$$

It follows that

$$\begin{aligned} |\circ[a^2 + b^2] - (a^2 + b^2)| &\leqslant |\circ[a^2 + b^2] - (a \times_\circ a + b \times_\circ b)| + |a \times_\circ a + b \times_\circ b - (a^2 + b^2)| \\ &\leqslant \circ[a^2 + b^2]\, \epsilon_\circ + |a \times_\circ a + b \times_\circ b|\, \epsilon_\circ + 2\,\iota \\ &\leqslant \circ[a^2 + b^2]\, \epsilon_\circ + (a^2 + b^2 + 2\,\iota)\, \epsilon_\circ\, (1 - \epsilon_\circ)^{-1} + 2\,\iota \\ &\leqslant \circ[a^2 + b^2]\, \epsilon_\circ + (a^2 + b^2)\, 1.1\, \epsilon_\circ + 2.1\,\iota, \end{aligned}$$

from which we extract

$$\begin{aligned} a^2 + b^2 &\leqslant (\circ[a^2 + b^2]\, (1 + \epsilon_\circ) + 2.1\,\iota)\, (1 - 1.1\,\epsilon_\circ)^{-1} \\ &\leqslant \circ[a^2 + b^2]\, (1 + 2.2\,\epsilon_\circ) + 2.2\,\iota. \end{aligned}$$

We thus obtain that

$$\begin{aligned}|\circ[a^2+b^2]-(a^2+b^2)| &\leqslant \circ[a^2+b^2]\,(\epsilon_\circ+(1+2.2\,\epsilon_\circ)\,1.1\,\epsilon_\circ)+1.1\,\epsilon_\circ\,2.2\,\iota+2.1\,\iota\\ &\leqslant \circ[a^2+b^2]\,2.2\,\epsilon_\circ+2.2\,\iota.\end{aligned}$$

We distinguish the two following cases:

- If $\circ[a^2+b^2]\geqslant 2\cdot 2^{E_{\min}}$, then $2.2\,\iota\leqslant\circ[a^2+b^2]\,1.1\,\epsilon_\circ$. Using

  $$a^2+b^2\geqslant\circ[a^2+b^2]\,(1-2.2\,\epsilon_\circ)-2.2\,\iota\geqslant\circ[a^2+b^2]\,(1-3.3\,\epsilon_\circ),$$

  since the square root does not provoke underflows, we obtain

  $$\begin{aligned}\left|\circ\left[\sqrt{a^2+b^2}\right]-\sqrt{a^2+b^2}\right| &\leqslant \left|\circ\left[\sqrt{a^2+b^2}\right]-\sqrt{\circ[a^2+b^2]}\right|+\left|\sqrt{\circ[a^2+b^2]}-\sqrt{a^2+b^2}\right|\\ &\leqslant \circ\left[\sqrt{a^2+b^2}\right]\epsilon_0+\frac{\circ[a^2+b^2]\,3.3\,\epsilon_\circ}{2\sqrt{\circ[a^2+b^2]\,(1-3.3\,\epsilon_\circ)}}\\ &\leqslant \circ\left[\sqrt{a^2+b^2}\right]\epsilon_0+1.8\,\epsilon_\circ\sqrt{\circ[a^2+b^2]}\\ &\leqslant 3\,\epsilon_0\,\circ\left[\sqrt{a^2+b^2}\right].\end{aligned}$$

- Otherwise $\circ[a^2+b^2]<2\cdot 2^{E_{\min}}$, which means that both products $a\times_\circ a$ and $b\times_\circ b$ involve underflows. We restart the analysis from

  $$|\circ[a^2+b^2]-(a^2+b^2)|\;\leqslant\;2\,\iota,$$

  which implies $|\circ[a^2+b^2]-(a^2+b^2)|\leqslant\circ[a^2+b^2]\,\epsilon_\circ+2\,\iota$. If $\circ[a^2+b^2]\leqslant 4\,\iota$, then we are done, since $\sqrt{a^2+b^2}<\sqrt{7\,\iota}$. Otherwise $\circ[a^2+b^2]\geqslant 4\,\iota$, and $a^2+b^2\geqslant\circ[a^2+b^2]\,(1-\epsilon_\circ)-2\,\iota\geqslant\iota$, and we have

  $$\begin{aligned}\left|\circ\left[\sqrt{a^2+b^2}\right]-\sqrt{a^2+b^2}\right| &\leqslant \left|\circ\left[\sqrt{a^2+b^2}\right]-\sqrt{\circ[a^2+b^2]}\right|+\left|\sqrt{\circ[a^2+b^2]}-\sqrt{a^2+b^2}\right|\\ &\leqslant \circ\left[\sqrt{a^2+b^2}\right]\epsilon_0+2\,\iota/(2\sqrt{\iota}).\end{aligned}$$