Tight Interval Inclusions with Compensated Algorithms

Stef Graillat and Fabienne Jézéquel

Abstract—Compensated algorithms consist in computing the rounding errors of individual operations and then adding them later on to the computed result. This makes it possible to increase the accuracy of the computed result efficiently. Computing the rounding error of an individual operation is possible through the use of a so-called *error-free transformation*. In this article, we show that it is possible to use compensated algorithms for having tight interval inclusions. We study compensated algorithms for summation, dot product and polynomial evaluation. We prove that the use of directed rounding makes it possible to get narrow inclusions with compensated algorithms. This is due to the fact that error-free transformations are no more exact but still sufficiently accurate to improve the numerical quality of results.

Index Terms—Interval arithmetic, directed rounding, compensated algorithms, error-free transformations, floating-point arithmetic, numerical validation, rounding errors, summation algorithms, dot product, Horner scheme

1 Introduction

In June 2018, researchers at the US Department of Energys Oak Ridge National Laboratory broke the exascale barrier, achieving on the *Summit* supercomputer[1] a peak throughput of 1.88 exaops (i.e., $1.88\ 10^{18}$ arithmetic operations per second). Unfortunately, with exascale computing, or more generally with high performance computing, a large number of rounding errors may be generated. Indeed, nearly all floating-point operations imply a small rounding which can accumulate along the computation and finally an incorrect result may be produced. As a consequence, it is crucial to propose methods and tools for numerical validation and accurate computation.

To improve the numerical quality of results, one can increase the working precision. In addition to the widely used *binary32* and *binary64* formats, the IEEE 754-2008 standard [1] defines the *binary128* format, also called quadruple precision, that is implemented in compilers such as the GNU compiler *gcc* and the Intel compiler *icc*. Moreover arbitrary precision libraries exist: one can cite ARPREC [2] and MPFR [3]. The computing precision can also be extended thanks to expansions, unevaluated sums of standard floating-point numbers. The QD package [4] provides the *double-double* and the *quad-double* data types, that consist of respectively two and four *binary64* floating-point numbers. One can also use arbitrary length expansions [5], [6], [7]. If a simple enough computation is performed, its accuracy can be improved thanks to compensated algorithms [8], [9], [10]. These algorithms are based on error-free transformations (EFTs) that make it possible to compute the rounding errors of some elementary operations like addition and multiplication exactly.

Interval arithmetic [11], [12] is a well known approach to control the validity of numerical results. It briefly consists in performing floating-point operations on intervals instead of scalars. These operations give a 100 percent-certain result, represented as an interval containing the exact result. The main advantage of this approach lies in the guaranteed error bounds it provides.

In this paper we show how to compute tight interval inclusions with compensated algorithms. To obtain guaranteed interval bounds, directed rounding should be used. However EFTs are intended to be used with rounding to nearest. Therefore we study the behaviour of EFTs with directed rounding. In this paper we show that EFTs executed with directed rounding provide guaranteed bounds on the results of additions and multiplications. We complete results established in [13], [14] on the behaviour with directed rounding of compensated algorithms based on these EFTs. Then we show that, thanks to compensated algorithms executed with directed rounding, tight interval inclusions can be computed for summation, dot product, and polynomial evaluation with Horner scheme.

The outline of this article is as follows. In Section 2 we give some definitions and notations used in the sequel. In Section 3 we show the impact of a directed rounding mode on EFTs and prove that guaranteed interval bounds can be obtained thanks to EFTs executed with directed rounding. In Sections 4, 5, and 6 we study the behaviour with directed rounding of compensated algorithms for respectively summation, dot product, and polynomial evaluation and show

1. URL address: http://www.olcf.ornl.gov/summit

• S. Graillat is with Sorbonne Université, CNRS, LIP6, Paris F-75005, France. E-mail: Stef.Graillat@lip6.fr.
• F. Jézéquel is with Sorbonne Université, CNRS, LIP6, Paris F-75005, France, and with Université Panthéon-Assas, 12 place du Panthéon, Paris, CEDEX 05 75231, France. E-mail: Fabienne.Jezequel@lip6.fr.

*Manuscript received 15 Oct. 2018; revised 31 May 2019; accepted 16 June 2019. Date of publication 20 June 2019; date of current version 6 Nov. 2020. (Corresponding author: Fabienne Jézéquel.)
Recommended for acceptance by A. Tenca and N. Takagi.
Digital Object Identifier no. 10.1109/TC.2019.2924005*

how they can provide narrow inclusions. Numerical experiments carried out using INTLAB [15] are presented in Section 7. Finally, conclusions and perspectives on this work are given in Section 8.

2 Definitions and Notations

In this paper, we assume to work with a binary floating-point arithmetic adhering to IEEE 754-2008 floating-point standard [1] and we suppose that no overflow occurs. The error bounds for the compensated summation that are presented in Section 4 remain valid in the presence of underflow. For the other compensated algorithms considered in this article (dot product and Horner scheme) we assume that no underflow occurs so as to present simpler error bounds.

The set of floating-point numbers is denoted by $\mathbb{F}$, the bound on relative error for round to nearest by $\mathbf{u}$. With the IEEE 754 *binary64* format (double precision), we have $\mathbf{u} = 2^{-53}$ and with the *binary32* format (single precision), $\mathbf{u} = 2^{-24}$.

We denote by $fl_*(\cdot)$ the result of a floating-point computation, where all operations inside parentheses are done in floating-point working precision with a directed rounding (that is to say toward $-\infty$ or $+\infty$). Floating-point operations in IEEE 754 satisfy [16]

For $\circ = \{+, -\}$, $\exists \varepsilon_1 \in \mathbb{R}$, $\varepsilon_2 \in \mathbb{R}$ such that

$$\mathrm{fl}_*(a \circ b) = (a \circ b)(1 + \varepsilon_1) = (a \circ b)/(1 + \varepsilon_2) \text{ with } |\varepsilon_\nu| \le 2\mathbf{u}.$$

As a consequence, for $\circ = \{+, -\}$,

$$|a \circ b - \mathrm{fl}_*(a \circ b)| \le 2\mathbf{u}|a \circ b| \text{ and } |a \circ b - \mathrm{fl}_*(a \circ b)| \le 2\mathbf{u}|\mathrm{fl}_*(a \circ b)|. \quad (1)$$

We use standard notations for error estimations. The quantities γ_n are defined as usual [16] by

$$\gamma_n(\mathbf{u}) := \frac{n\mathbf{u}}{1 - n\mathbf{u}} \quad \text{for } n \in \mathbb{N},$$

where it is implicitly assumed that $n\mathbf{u} < 1$.

Remark 1. We give the following relations on γ_n, that will be frequently used in the sequel of the paper. For any nonnegative integer n, $n\mathbf{u} \le \gamma_n(\mathbf{u})$, $\gamma_n(\mathbf{u}) \le \gamma_{n+1}(\mathbf{u})$, $(1+\mathbf{u})\gamma_n(\mathbf{u}) \le \gamma_{n+1}(\mathbf{u})$, $2(n+1)\mathbf{u}(1 + \gamma_{2n}(\mathbf{u})) \le \gamma_{2(n+1)}(\mathbf{u})$.

Remark 2. Recently, it has been shown that classic Wilkinson-type error bounds for summation, dot product and polynomial evaluation [17], [18], [19] can be slightly improved by replacing the factor $\gamma_n(\mathbf{u})$ by $n\mathbf{u}$ with no condition on n (for summation, dot product and Horner scheme). It is likely that the error bounds given in this paper could also be slightly improved by replacing all the $\gamma_n(\mathbf{u})$ by $n\mathbf{u}$. However the proofs for improving the bounds would be more complicated and tricky, and would not be useful for this paper. We just aim at showing that the relative accuracy is in $\mathcal{O}(\mathbf{u})$ for classic algorithms and in $\mathcal{O}(\mathbf{u}^2)$ for compensated algorithms with directed roundings.

3 Error-Free Transformations with Directed Rounding

3.1 Error-Free Transformations for Addition

EFTs exist for the sum of two floating-point numbers with rounding to nearest: `FastTwoSum` [20], given as Algorithm 1, which requires a test and 3 floating-point operations, and `TwoSum` [21], given as Algorithm 2, which requires 6 floating-point operations. These algorithms compute both the floating-point sum x of two numbers a and b and the associated rounding error y such that $x + y = a + b$ when using rounding to the nearest. This is no longer true with directed rounding. Indeed, with directed rounding, the rounding error may not be exactly representable (see [22] page 125).

We will study the behaviour of `FastTwoSum` and `TwoSum` with directed rounding. In the rest of this section, any arithmetic operation is rounded using the fl_* function defined in Section 2. In the Propositions presented in this section, and also in Section 4.2, we assume underflow may occur because, in this case, additions or subtractions generate no rounding error if subnormal numbers are available [23].

3.1.1 `FastTwoSum` *with Directed Rounding*

With rounding to nearest, the `FastTwoSum` EFT, given in Algorithm 1, computes the floating-point sum x of two numbers a and b and its associated rounding error y.

Algorithm 1. Error-Free Transformation for the Sum of two Floating-Point Numbers with Rounding to Nearest

```
   function $[x, y]$ = FastTwoSum($a, b$)
1: if $|b| > |a|$ then
2:     exchange $a$ and $b$
3: end if
4: $x \leftarrow a + b$
5: $z \leftarrow x - a$
6: $y \leftarrow b - z$
```

In [24], it is shown that the floating-point number z in Algorithm 1 is computed exactly with directed rounding. This property is recalled as Proposition 3.1.

Proposition 3.1. *The floating-point number z provided by Algorithm 1 using directed rounding is computed exactly, i.e.,* $z = x - a$.

In general the correction y computed by Algorithm 1 using directed rounding is different from the rounding error e on the sum of a and b. In Proposition 3.2, we bound the difference between e and y.

Proposition 3.2. *Let x and y be the floating-point addition of a and b and the correction both computed by Algorithm 1 using directed rounding. Let e be the error on x: $a + b = x + e$. Then*

$$|e - y| \le 4\mathbf{u}^2|a + b| \quad \text{and} \quad |e - y| \le 4\mathbf{u}^2|x|.$$

Proof. As $e = a + b - x = a + b - \mathrm{fl}_*(a + b)$, from the inequalities (1), we have

$$|e| \le 2\mathbf{u}|a + b| \quad \text{and} \quad |e| \le 2\mathbf{u}|x|. \quad (2)$$

Now let δ denote the rounding error in y, so that

$$b - z = y + \delta. \tag{3}$$

Using the first of the inequalities (1), we have

$$|\delta| \le 2\mathbf{u}|b - z|. \tag{4}$$

From Proposition 3.1, we know that $z = x - a$ exactly, and thus $b - z = b - (x - a) = a + b - x = e$. Therefore

$$\begin{aligned} |e - y| &= |(b - z) - y| && \text{as } e = b - z,\\ &= |\delta| && \text{from (3)},\\ &\le 2\mathbf{u}|e| && \text{from (4) and } e = b - z. \end{aligned}$$

Combining this inequality with those in (2) gives the claimed result. □

In Proposition 3.3 we establish a relation between the error e and the correction y if Algorithm 1 is executed with directed rounding.

Proposition 3.3. *Let x and y be the floating-point addition of a and b and the correction both computed by Algorithm 1 using directed rounding. Let e be the error on x: $a + b = x + e$.*

- *If computations are performed with rounding toward $+\infty$ then $e \le y$.*
- *If computations are performed with rounding toward $-\infty$ then $y \le e$.*

Proof. We always have by definition $a + b = x + e$. From Proposition 3.2, it follows that $y = \mathrm{fl}_*(e)$. So if we use rounding toward $+\infty$ then $e \le y$ and if we use rounding toward $-\infty$ then $y \le e$. □

3.1.2 `TwoSum` *with Directed Rounding*

With rounding to nearest, the `TwoSum` EFT, given in Algorithm 2, computes the floating-point sum x of two numbers a and b and its associated rounding error y.

Algorithm 2. Error-Free Transformation for the Sum of two Floating-Point Numbers with Rounding to Nearest

```
function $[x, y]$ = TwoSum$(a, b)$
1: $x \leftarrow a + b$
2: $d \leftarrow x - a$
3: $f \leftarrow b - d$
4: $g \leftarrow x - d$
5: $h \leftarrow a - g$
6: $y \leftarrow f + h$
```

We recall here a result of [25].

Theorem 3.4 ([25, Thm. 4.1]). *Let x and y be the floating-point addition of a and b and the correction both computed by Algorithm 2 using directed rounding. Let e be the error on x: $a + b = x + e$. Then*

$$|e - y| \le 4\mathbf{u}^2|a + b| \qquad \text{and} \qquad |e - y| \le 4\mathbf{u}^2|x|.$$

Proposition 3.6 has been established using Sterbenz's lemma [26] which is recalled as Lemma 3.5.

Lemma 3.5 (Sterbenz). *In a floating-point system with subnormal numbers available, if c and d are finite floating-point numbers such that $d/2 \le c \le 2d$, then $c - d$ is exactly representable.*

In Proposition 3.6 we establish a relation between the error e and the correction y if Algorithm 2 is executed with directed rounding.

Proposition 3.6. *Let x and y be the floating-point addition of a and b and the correction both computed by Algorithm 2 using directed rounding. Let e be the error on x: $a + b = x + e$.*

- *If computations are performed with rounding toward $+\infty$ then $e \le y$.*
- *If computations are performed with rounding toward $-\infty$ then $y \le e$.*

Proof. Without loss of generality, we can assume that $b \ge 0$. We will separate the proof into three different cases: $b \le |a|$, $-b < a \le -b/2$ and $-b/2 < a < b$.

- case 1: $b \le |a|$
  In this case, the lines 1, 2 and 3 correspond exactly to `FastTwoSum` (Algorithm 1). It follows that $d = x - a$ and so $f = \mathrm{fl}_*(a + b - x)$, $g = a$, $h = 0$ and $y = f$. As a consequence, $y = \mathrm{fl}_*(e)$. So if we use rounding toward $+\infty$ then $e \le y$ and if we use rounding toward $-\infty$ then $y \le e$.
- case 2: $-b < a \le -b/2$
  Using Sterbenz's lemma, it follows that $x = a + b$ and so $d = b$, $f = 0$, $g = a$, $h = 0$ and $y = 0$. So in this case, we have $e = y = 0$.
- case 3: $-b/2 < a < b$
  It follows from [25, Thm 4.1] that computations in lines 3 and 4 are exact due to Sterbenz's lemma. As a consequence, $f = b - d$ and $g = x - d$. Let us now assume we use rounding toward $+\infty$. As a consequence, $f + h \le y$ and $a - g \le h$ so $f + a - g \le y$. Using the fact that $f = b - d$ and $g = x - d$, we obtain that $e = a + b - x \le y$.
  Let us now assume we use rounding toward $-\infty$. We have $y \le f + h$ and $h \le a - g$ so $y \le f + a - g$. Using the fact that $f = b - d$ and $g = x - d$, we obtain that $y \le a + b - x = e$.

This concludes the proof. □

3.2 Error-Free Transformation for Multiplication Based on FMA

The *Fused-Multiply-and-Add* (`FMA`) is an operator that enables a floating-point multiplication followed by an addition to be performed as a single floating-point operation. For $a, b, c \in \mathbb{F}$, $\mathtt{FMA}(a, b, c)$ is an approximation of $a \times b + c \in \mathbb{R}$ that satisfies, if no underflow occurs:

$$\mathtt{FMA}(a, b, c) = (a \times b + c)(1 + \varepsilon_1) = (a \times b + c)/(1 + \varepsilon_2),$$

where $|\varepsilon_\nu| \le \mathbf{u}$ with rounding to nearest and $|\varepsilon_\nu| \le 2\mathbf{u}$ with directed rounding.

The `FMA` operation is supported by numerous processors such as AMD or Intel processors starting with respectively the Bulldozer or the Haswell architecture and by the Intel Xeon Phi coprocessor. It is also supported by AMD and NVidia GPUs (Graphics Processing Units) since 2010.

With any rounding mode, the `TwoProdFMA` EFT, given in Algorithm 3, computes both the floating-point product x of two numbers a and b and the associated rounding error y, provided that no underflow occurs. If this property holds, the floating-point numbers x and y computed by the `TwoProdFMA` algorithm satisfy: $x + y = a \times b$.

Algorithm 3. Error-Free Transformation for the Product of two Floating-Point Numbers using an `FMA`

```
function $[x, y]$ = TwoProdFMA$(a, b)$
1: $x \leftarrow a \times b$
2: $y \leftarrow$ FMA$(a, b, -x)$
```

4 Accurate Summation

In this section we recall how to obtain interval inclusions for summation using the classical iterative algorithm. Then we present how to compute narrow inclusions thanks to compensated algorithms.

4.1 Classic Summation

The classic algorithm for summation is the iterative Algorithm 4.

Algorithm 4. Summation of n Floating-Point Numbers $p = \{p_i\}$

```
function res = Sum$(p)$
1: $s_1 \leftarrow p_1$
2: for $i = 2$ to $n$ do
3:     $s_i \leftarrow s_{i-1} + p_i$
4: end for
5: res $\leftarrow s_n$
```

The error generated by Algorithm 4 with directed rounding is given in [16] and is recalled in Proposition 4.1.

Proposition 4.1. *Let us suppose Algorithm 4 is applied to floating-point numbers $p_i \in \mathbb{F}$, $1 \le i \le n$. Let $s := \sum p_i$ and $S := \sum |p_i|$.*

With directed rounding, if $n\mathbf{u} < \frac{1}{2}$, then

$$|\mathtt{res} - s| \le \gamma_{n-1}(2\mathbf{u})S. \tag{5}$$

In Corollary 4.2 Equation (5) is rewritten in terms of the condition number on $\sum p_i$:

$$\mathrm{cond}\Big(\sum p_i\Big) = \frac{S}{|s|}.$$

Corollary 4.2. *With directed rounding, if $n\mathbf{u} < \frac{1}{2}$, the result* `res` *of Algorithm 4 satisfies*

$$\frac{|\mathtt{res} - s|}{|s|} \le \gamma_{n-1}(2\mathbf{u})\mathrm{cond}\Big(\sum p_i\Big).$$

Because $\gamma_{n-1}(2\mathbf{u}) \approx 2(n-1)\mathbf{u}$ as $n\mathbf{u} < 1/2$, the bound for the relative error is essentially $2n\mathbf{u}$ times the condition number. If the condition number is large (greater than $1/\mathbf{u}$) then the result of Algorithm 4 has no more correct digits. Compensated algorithms, that evaluate more accurately the sum of floating-point numbers, are presented in Section 4.2.

Algorithm 5 shows how to compute an enclosure of $\sum_{i=1}^{n} p_i$. It is given with the MATLAB syntax. With the argument -1 (resp. 1), the `setround` function enables one to perform the next instructions with rounding to $-\infty$ (resp. $+\infty$). The same algorithm could also be written in a programming language sush as C++ using the `fesetround` function to change the rounding mode.

Algorithm 5. Computation of Interval Bounds `Sinf` and `Ssup` with the Classic Summation Algorithm `Sum`

```
setround(-1)
Sinf = Sum(p)
setround(1)
Ssup = Sum(p)
```

As shown for example in [27], we have the following enclosure.

Proposition 4.3. *Let $p = \{p_i\}$ be a vector of n floating-point numbers. If* `Sinf` *and* `Ssup` *are computed using Algorithm 5, then we have*

$$\mathtt{Sinf} \le \sum_{i=1}^{n} p_i \le \mathtt{Ssup}.$$

4.2 Compensated Summation with Directed Rounding

A compensated algorithm to evaluate accurately the sum of n floating-point numbers is presented as Algorithm 6 (`FastCompSum`) [28], [29]. This sum is corrected thanks to an error-free transformation used for each individual summation. Although `FastTwoSum` is called in Algorithm 6, with rounding to nearest the same result can be obtained using another error-free transformation (`TwoSum`).

Algorithm 6. Compensated Summation of n Floating-Point Numbers $p = \{p_i\}$ using `FastTwoSum`

```
function res = FastCompSum$(p)$
1: $\pi_1 \leftarrow p_1$
2: $\sigma_1 \leftarrow 0$
3: for $i = 2$ to $n$ do
4:     $[\pi_i, q_i] \leftarrow$ FastTwoSum$(\pi_{i-1}, p_i)$
5: $\sigma_i \leftarrow \sigma_{i-1} + q_i$
6: end for
7: res $\leftarrow \pi_n + \sigma_n$
```

With directed rounding, Algorithm 1 (`FastTwoSum`) is not an error-free transformation. The error generated by Algorithm 6 with directed rounding is given in [13] and is recalled in Proposition 4.4.

Proposition 4.4. *Let us suppose Algorithm* `FastCompSum` *is applied, with directed rounding, to floating-point numbers $p_i \in F$, $1 \le i \le n$. Let $s := \sum p_i$ and $S := \sum |p_i|$. If $n\mathbf{u} < \frac{1}{2}$, then*

$$|\mathtt{res} - s| \le 2\mathbf{u}|s| + 2(1 + 2\mathbf{u})\gamma_n^2(2\mathbf{u})S. \tag{6}$$

From Proposition 4.4, a bound for the relative error on the result of Algorithm 6 (`FastCompSum`) obtained with directed rounding is deduced in Corollary 4.5.

Corollary 4.5. *With directed rounding, if $n\mathbf{u} < \frac{1}{2}$, then, the result* `res` *of Algorithm 6 (*`FastCompSum`*) satisfies*

$$\frac{|\texttt{res} - s|}{|s|} \leq 2\mathbf{u} + 2(1+2\mathbf{u})\gamma_n^2(2\mathbf{u})\mathrm{cond}\Big(\sum p_i\Big).$$

From Corollary 4.5, because $\gamma_n(2\mathbf{u}) \approx 2n\mathbf{u}$, the relative error bound is essentially $(n\mathbf{u})^2$ times the condition number plus the unavoidable rounding $2\mathbf{u}$ as we round to the working precision. As a remark, we could avoid this "unavoidable" rounding by keeping the result in two working precision numbers until a later stage of the computation. The computation is carried out almost as with twice the working precision ($\mathbf{u}^2$).

Algorithm 7 shows how to compute with MATLAB the `FastCompSum` algorithm with rounding to $-\infty$, and then with rounding to $+\infty$.

Algorithm 7. Computation of Interval Bounds `Sinf` and `Ssup` with the Compensated Summation Algorithm `FastCompSum`

```
setround(-1)
Sinf = FastCompSum(p)
setround(1)
Ssup = FastCompSum(p)
```

In Proposition 4.6 we show that Algorithm 7 provides an enclosure of $\sum_{i=1}^n p_i$. Thanks to the `FastCompSum` algorithm, the results provided by Algorithm 7 are almost as accurate as if the classical summation was computed in twice the working precision.

Proposition 4.6. *Let $p = \{p_i\}$ be a vector of n floating-point numbers. If* `Sinf` *and* `Ssup` *are computed using Algorithm 7, then we have*

$$\texttt{Sinf} \leq \sum_{i=1}^n p_i \leq \texttt{Ssup}.$$

Proof. Let e_i be the error on the floating-point addition of π_{i-1} and p_i ($i = 2, \ldots, n$). We know that $s = \sum_{i=1}^n p_i = \pi_n + \sum_{i=1}^n e_i$ where $\pi_i + e_i = \pi_{i-1} + p_i$.

- Let us assume computations are performed with rounding toward $+\infty$.
  From Proposition 3.2, it follows that $e_i \leq q_i$. As a consequence, we have $s \leq \pi_n + \sum_{i=1}^n q_i$. As we use rounding toward $+\infty$, we have $\sum_{i=1}^n q_i \leq \sigma_n$ so $s \leq \pi_n + \sigma_n$. As we always use rounding toward $+\infty$, we also have $s \leq \texttt{res} := \texttt{Ssup}$.
- Let us assume computations are performed with rounding toward $-\infty$.
  From Proposition 3.2, it follows that $q_i \leq e_i$. As a consequence, we have $\pi_n + \sum_{i=1}^n q_i \leq s$. As we use rounding toward $-\infty$, we have $\sigma_n \leq \sum_{i=1}^n q_i$ so $\pi_n + \sigma_n \leq s$. As we always use rounding toward $-\infty$, we also have $\texttt{Sinf} := \texttt{res} \leq s$. □

A compensated summation algorithm based on `TwoSum` instead of `FastTwoSum` will give similar results that is to say the same error bounds, propositions, and proofs.

5 Accurate dot Product

In this section we recall how to obtain inclusions of dot products using the classic dot product algorithm. Then we show that tighter inclusions can be computed using compensated dot product algorithms executed with directed rounding. In this section, we assume that no underflow occurs.

5.1 Classic dot Product

The classic algorithm for computing a dot product is Algorithm 8.

Algorithm 8. Classic dot Product of $x = \{x_i\}$ and $y = \{y_i\}$, $1 \leq i \leq n$

```
function res = Dot(x, y)
1: $s_1 \leftarrow x_1 \times y_1$
2: for $i = 2$ to $n$ do
3:     $s_i \leftarrow x_i \times y_i + s_{i-1}$
4: end for
5: res $\leftarrow s_n$
```

The error generated by Algorithm 8 with directed rounding is recalled in Proposition 5.1.

Proposition 5.1. *Let floating point numbers $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$, be given and denote by* `res` *$\in \mathbb{F}$ the result computed by Algorithm 8 (*`Dot`*). With directed rounding, if $n\mathbf{u} < \frac{1}{2}$, we have*

$$|\texttt{res} - x^Ty| \leq \gamma_n(2\mathbf{u})|x^T||y|. \quad (7)$$

Proof. The proof can be found in Higham [16, p.63]. □

We can rewrite the previous inequality in terms of the condition number of the dot product defined by

$$\mathrm{cond}(x^Ty) = 2\frac{|x|^T|y|}{|x^Ty|}.$$

Corollary 5.2. *With directed rounding, if $n\mathbf{u} < \frac{1}{2}$, the result* `res` *of Algorithm 8 satisfies*

$$\frac{|\texttt{res} - x^Ty|}{|x^Ty|} \leq \frac{1}{2}\gamma_n(2\mathbf{u})\mathrm{cond}(x^Ty).$$

Because $\gamma_n(2\mathbf{u}) \approx 2n\mathbf{u}$ as $n\mathbf{u} < 1/2$, the bound for the relative error is essentially $n\mathbf{u}$ times the condition number.

Algorithm 9 shows how to compute the `Dot` algorithm with rounding to $-\infty$, and then with rounding to $+\infty$.

Algorithm 9. Computation of Interval Bounds `Dinf` and `Dsup` with the Classic dot Product Algorithm `Dot`

```
setround(-1)
Dinf = Dot(x,y)
setround(1)
Dsup = Dot(x,y)
```

As shown for example in [27], we have the following enclosure.

Proposition 5.3. *Let floating-point numbers $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$, be given. If* `Dinf` *and* `Dsup` *are computed using Algorithm 9, then we have*

$$\texttt{Dinf} \leq x^Ty \leq \texttt{Dsup}.$$

5.2 Compensated dot Product with Directed Rounding

A compensated dot product algorithm [9] that uses the `TwoProdFMA` EFT is recalled as Algorithm 10 (`CompDot`).

Algorithm 10. Compensated dot Product of $x = \{x_i\}$ and $y = \{y_i\}$, $1 \leq i \leq n$

```
function res=CompDot($x, y$)
1: $[p_1, s_1] \leftarrow$ TwoProdFMA($x_1, y_1$)
2: for $i = 2$ to $n$ do
3:     $[h_i, r_i] \leftarrow$ TwoProdFMA($x_i, y_i$)
4:     $[p_i, q_i] \leftarrow$ TwoSum($p_{i-1}, h_i$)
5: $s_i \leftarrow s_{i-1} + (q_i + r_i)$
6: end for
7: res $\leftarrow p_n + s_n$
```

A bound for the absolute error on the result `res` of Algorithm 10 with directed rounding is given in Proposition 5.4.

Proposition 5.4. *Let floating-point numbers $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$, be given and denote by* `res` $\in \mathbb{F}$ *the result computed by Algorithm 10 with directed rounding. If $(n+1)\mathbf{u} < \frac{1}{2}$, then,*

$$|\texttt{res} - x^T y| \leq 2\mathbf{u}|x^T y| + 2\gamma_{n+1}^2(2\mathbf{u})|x^T||y|.$$

Proof. In [14], a similar algorithm has been analyzed with directed rounding, except `FastTwoSum` was used instead of `TwoSum` here. Because the error bounds are the same in Proposition 3.2 and Theorem 3.4, the error bound in Proposition 5.4 is the same as in [14]. □

From Proposition 5.4, a bound for the relative error on the result of Algorithm 10 obtained with directed rounding is deduced in Corollary 5.5.

Corollary 5.5. *With directed rounding, if $(n+1)\mathbf{u} < \frac{1}{2}$, then, the result* `res` *of Algorithm 10 satisfies*

$$\frac{|\texttt{res} - x^T y|}{|x^T y|} \leq 2\mathbf{u} + \gamma_{n+1}^2(2\mathbf{u})\mathrm{cond}\left(x^T y\right).$$

From Corollary 5.5, the relative error bound on the result of Algorithm 10 computed with directed rounding is essentially $(n\mathbf{u})^2$ times the condition number plus the rounding $2\mathbf{u}$ due to the working precision. The result obtained with Algorithm 10 is almost as accurate as if the classic dot product was computed in twice the working precision.

Algorithm 11 shows how to compute with MATLAB the `CompDot` algorithm with rounding to $-\infty$, and then with rounding to $+\infty$.

Algorithm 11. Computation of Interval Bounds `Dinf` and `Dsup` with the Compensated dot Product Algorithm `CompDot`

```
setround(-1)
Dinf = CompDot(x,y)
setround(1)
Dsup = CompDot(x,y)
```

In Proposition 5.6 we show that Algorithm 11 provides an enclosure of the dot product.

Proposition 5.6. *Let floating-point numbers $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$, be given. If* `Dinf` *and* `Dsup` *are computed using Algorithm 11, then we have*

$$\texttt{Dinf} \leq x^T y \leq \texttt{Dsup}.$$

Proof. Let e_i be the error on the floating-point addition of p_{i-1} and h_i $(i = 2, ..., n)$. We know that $x^T y = p_n + s_1 + \sum_{i=2}^n (e_i + r_i)$ where $p_i + e_i = p_{i-1} + h_i$ (see Proposition 4.5 in [14]).

- Let us assume computations are performed with rounding toward $+\infty$.
  From Proposition 3.6, it follows that $e_i \leq q_i$. As a consequence, we have $x^T y \leq p_n + s_1 + \sum_{i=2}^n (q_i + r_i)$. As we use rounding toward $+\infty$, we have $s_1 + \sum_{i=2}^n (q_i + r_i) \leq s_n$ so $x^T y \leq p_n + s_n$. As we always use rounding toward $+\infty$, we also have $x^T y \leq \texttt{res} := \texttt{Dsup}$.
- Let us assume computations are performed with rounding toward $-\infty$.
  From Proposition 3.6, it follows that $q_i \leq e_i$. As a consequence, we have $p_n + s_1 + \sum_{i=2}^n (q_i + r_i) \leq x^T y$. As we use rounding toward $-\infty$, we have $s_n \leq s_1 + \sum_{i=2}^n (q_i + r_i)$ so $p_n + s_n \leq x^T y$. As we always use rounding toward $-\infty$, we also have $\texttt{Dinf} := \texttt{res} \leq x^T y$. □

6 Accurate Horner Scheme

In this section we recall how to obtain inclusions of a polynomial evaluation using the classic Horner scheme. Then we show that tighter inclusions can be computed using a compensated Horner scheme executed with directed rounding. In this section, we assume that no underflow occurs.

6.1 Classic Horner Scheme

The classical method for evaluating a polynomial

$$p(x) = \sum_{i=0}^n a_i x^i,$$

is the Horner scheme which consists of Algorithm 12.

Algorithm 12. Polynomial Evaluation with Horner's Scheme

```
function res = Horner($p, x$)
1: $s_n \leftarrow a_n$
2: for $i = n - 1$ downto 0 do
3:     $s_i \leftarrow s_{i+1} \times x + a_i$
4: end for
5: res $\leftarrow s_0$
```

Whatever the rounding mode, a forward error bound on the result of Algorithm 12 is (see [16, p. 95]):

$$|p(x) - \texttt{res}| \leq \gamma_{2n}(2\mathbf{u}) \sum_{i=0}^n |a_i||x|^i = \gamma_{2n}(2\mathbf{u})\widetilde{p}(|x|),$$

where $\widetilde{p}(x) = \sum_{i=0}^n |a_i|x^i$. The relative error on the result can be expressed in terms of the condition number of the polynomial evaluation defined by

$$\mathrm{cond}(p,x) = \frac{\sum_{i=0}^{n} |a_i||x|^i}{|p(x)|} = \frac{\widetilde{p}(|x|)}{|p(x)|}. \tag{8}$$

Thus we have

$$\frac{|p(x) - \texttt{res}|}{|p(x)|} \leq \gamma_{2n}(2\mathbf{u})\mathrm{cond}(p,x).$$

If an FMA instruction is available, then the statement $s_i \leftarrow s_{i+1} \times x + a_i$ in Algorithm 12 can be rewritten as $s_i \leftarrow \texttt{FMA}(s_{i+1}, x, a_i)$ which slightly improves the error bound (see [16]).

Algorithm 13 presents how to compute an enclosure of $p(x)$ if $x \geq 0$. If $x \leq 0$, $\texttt{Horner}(\bar{p}, -x)$ is computed with $\bar{p}(x) = \sum_{i=0}^{n} a_i(-1)^i x^i$.

Algorithm 13. Computation of Interval Bounds Einf and Esup with the Classic Horner Scheme for $x \geq 0$

```
setround(-1)
Einf = Horner(p,x)
setround(1)
Esup = Horner(p,x)
```

As for dot product and summation with directed rounding ([27]), the following enclosure holds.

Proposition 6.1. *Consider a polynomial p of degree n with floating-point coefficients, and a floating-point value $x \geq 0$. If* Einf *and* Esup *are computed using Algorithm 13, then*

$$\texttt{Einf} \leq p(x) \leq \texttt{Esup}.$$

6.2 Compensated Horner Scheme with Directed Rounding

A compensated Horner scheme [10], [30] is recalled as Algorithm 14 (CompHorner).

Algorithm 14. Polynomial Evaluation with a Compensated Horner Scheme

```
function res = CompHorner(p, x)
1: $s_n \leftarrow a_n$
2: $r_n \leftarrow 0$
3: for $i = n-1$ down to 0 do
4:     $[p_i, \pi_i] \leftarrow \texttt{TwoProdFMA}(s_{i+1}, x)$
5:     $[s_i, \sigma_i] \leftarrow \texttt{FastTwoSum}(p_i, a_i)$
6:     $r_i \leftarrow r_{i+1} \times x + (\pi_i + \sigma_i)$
7: end for
8: res $\leftarrow s_0 + r_0$
```

The error generated by Algorithm 14 with directed rounding is given in [14] and is recalled in Proposition 6.2.

Proposition 6.2. *Consider a polynomial p of degree n with floating-point coefficients, and a floating-point value x. With directed rounding, the forward error in the compensated Horner algorithm is such that*

$$|\texttt{CompHorner}(p,x) - p(x)| \leq 2\mathbf{u}|p(x)| + 2\gamma_{2n+1}(2\mathbf{u})^2\widetilde{p}(|x|).$$

Combining this error bound with the condition number (8) for the polynomial evaluation gives

$$\frac{|\texttt{CompHorner}(p,x) - p(x)|}{|p(x)|} \leq 2\mathbf{u} + 2\gamma_{2n+1}(2\mathbf{u})^2\mathrm{cond}(p,x).$$

Because $\gamma_{2n+1}(2\mathbf{u}) \approx 4n\mathbf{u}$ as $n\mathbf{u} < 1/2$, the bound for the relative error of the computed result is essentially $(n\mathbf{u})^2$ times the condition number of the polynomial evaluation, plus the unavoidable term $2\mathbf{u}$ for rounding the result to the working precision. The computed result is almost as accurate as if it was computed by the classic Horner algorithm with twice the working precision, and then rounded to the working precision.

Algorithm 15 presents how to compute an enclosure of $p(x)$ if $x \geq 0$. Like with Algorithm 13, if $x \leq 0$, $\texttt{CompHorner}(\bar{p}, -x)$ is computed with $\bar{p}(x) = \sum_{i=0}^{n} a_i(-1)^i x^i$.

Algorithm 15. Computation of Interval Bounds Einf and Esup with the Compensated Horner Scheme CompHorner for $x \geq 0$

```
setround(-1)
Einf = CompHorner(p,x)
setround(1)
Esup = CompHorner(p,x)
```

In Proposition 6.3 we show that Algorithm 15 provides an enclosure of $p(x)$. The results of Algorithm 15 are almost as accurate as if the classical Horner scheme was computed in twice the working precision.

Proposition 6.3. *Consider a polynomial p of degree n with floating-point coefficients, and a floating-point value $x \geq 0$. If* Einf *and* Esup *are computed using Algorithm 15, then*

$$\texttt{Einf} \leq p(x) \leq \texttt{Esup}.$$

Proof. We analyze the impact of a directed rounding mode on Algorithm 14 (CompHorner).

Let τ_i be the rounding error in the floating-point addition of p_i and a_i (τ_i is not necessarily a floating-point number):

$$s_i + \tau_i = p_i + a_i.$$

It follows that $s_{i+1} \times x = p_i + \pi_i$ and $p_i + a_i = s_i + \tau_i$ with $|\tau_i - \sigma_i| \leq 2\mathbf{u}|\tau_i|$. As a consequence, we have

$$s_i = s_{i+1} \times x + a_i - \pi_i - \tau_i \quad \text{for } i = 0, \ldots, n-1.$$

By induction, we deduce that

$$p(x) = s_0 + p_\pi(x) + p_\tau(x),$$

with

$$s_0 = \mathrm{fl}_*(p(x)), \quad p_\pi(x) = \sum_{i=0}^{n-1} \pi_i x^i, \quad \text{and} \quad p_\tau(x) = \sum_{i=0}^{n-1} \tau_i x^i.$$

- Let us assume computations are performed with rounding toward $+\infty$.

  From Proposition 3.2, it follows that $\tau_i \leq \sigma_i$. As a consequence, we have

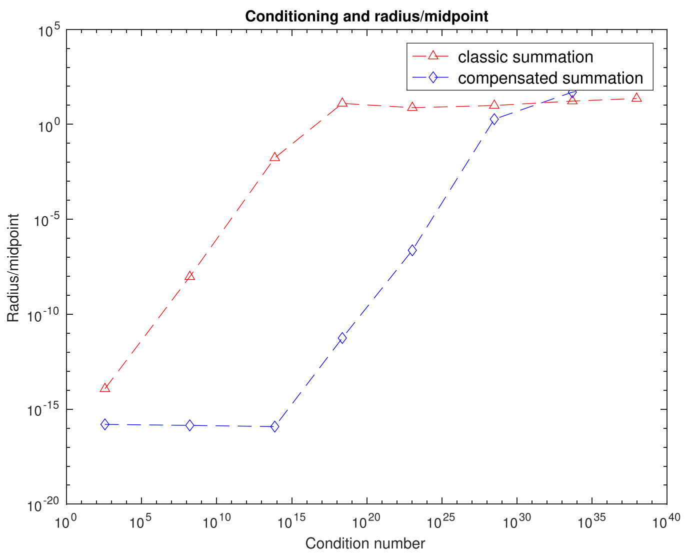

Fig. 1. Classic and compensated summation computed with interval arithmetic.

$$p(x) \leq s_0 + \sum_{i=0}^{n-1} \pi_i x^i + \sum_{i=0}^{n-1} \sigma_i x^i.$$

As we use rounding toward $+\infty$, we have $p(x) \leq s_0 + r_0 = \texttt{res} := \texttt{Esup}$.

- Let us assume computations are performed with rounding toward $-\infty$.

  From Proposition 3.2, it follows that $\sigma_i \leq \tau_i$. As a consequence, we have

$$s_0 + \sum_{i=0}^{n-1} \pi_i x^i + \sum_{i=0}^{n-1} \sigma_i x^i \leq p(x).$$

As we use rounding toward $-\infty$, we have $\texttt{Einf} := \texttt{res} = s_0 + r_0 \leq p(x)$. □

As a remark, the same result will be obtained if `FastTwoSum` is replaced by `TwoSum` in Algorithm 14.

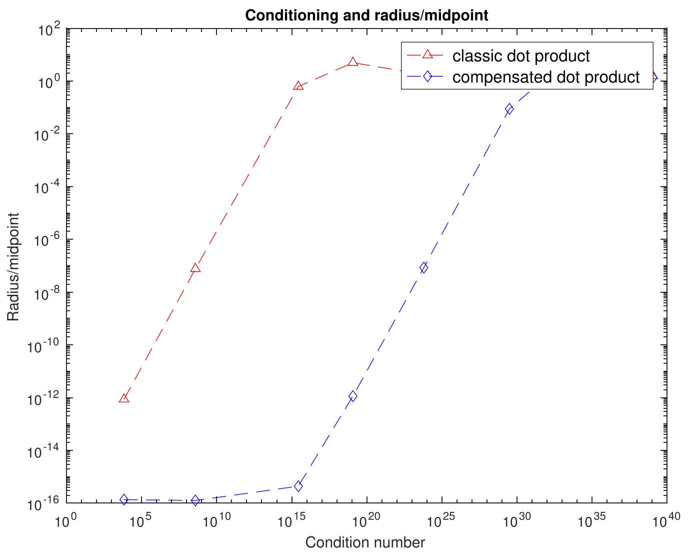

Fig. 2. Classic and compensated dot product computed with interval arithmetic.

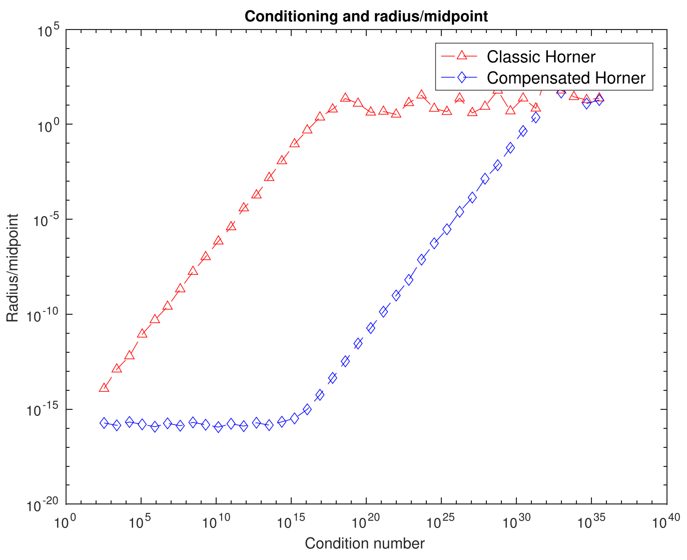

Fig. 3. Classic and compensated Horner scheme computed with interval arithmetic.

7 Numerical Results

In this section, we present results computed with interval arithmetic using the classic and the compensated algorithms for summation, dot product and Horner scheme. With the compensated algorithms, the interval bounds have been computed as described in the previous sections. The numerical experiments have been carried out on a laptop with an Intel Core i5 processor at 2.9 GHz with 16 Gb of RAM. We used MATLAB R2016b with INTLAB v10 [15]. The computation has been performed with the *binary64* (double precision) format of the IEEE 754-2008 standard [1]. Figs. 1, 2, and 3 display the radius over the midpoint of interval results obtained for various condition numbers.

From Figs. 1, 2, and 3, with the classic algorithms, if the condition number increases, the radius over the midpoint of the computed interval also increases, which means that the accuracy of the result decreases. If the condition number reaches about 10^{15}, the computed result has no more correct digits. With the compensated algorithms, if the condition number remains less than about 10^{15}, the numerical quality of the computed result is very satisfactory. If the condition number increases from about 10^{15} to 10^{30}, the numerical quality of the result decreases. If the condition number reaches about 10^{30}, the result has no more correct digits. As expected, the interval results obtained with the compensated algorithms are almost as accurate as if they were computed in twice the working precision. Tight interval inclusions have been computed thanks to compensated algorithms.

8 Conclusion and Perspectives

In this paper we have shown that tight inclusions can be computed for summation, dot product, and polynomial evaluation thanks to compensated algorithms executed with directed rounding. The results obtained are almost as accurate as if they were computed using twice the working precision. The approach chosen in this paper consists in executing the compensated algorithms entirely with rounding toward $-\infty$, and then with rounding toward $+\infty$. An advantage of this approach lies in the fact that the original compensated algorithms can be used, possibly from a library usually executed with rounding to nearest.

Another approach would consist in computing the results once with rounding to nearest and the corrections with rounding toward $-\infty$, and then with rounding toward $+\infty$. This approach would be more memory consuming than the approach presented in this paper. However it would perform better in terms of execution time. It would be interesting to compare the two approaches.

K-fold compensated algorithms enable one to compute summation and dot product as in K-fold precision [9]. Priest's EFT [8] for the addition and `TwoProdFMA` both compute the generated rounding error whatever the rounding mode. The impact of a directed rounding mode on K-fold compensated algorithms based on these EFTs has been shown in [14]. Another perspective would consist in studying K-fold compensated algorithms to see if they can provide narrow inclusions for summation and dot product, as in K-fold precision.

As a future work, we could also determine if it would be possible to obtain tight inclusions using other compensated algorithms, such as compensated exponentiation [31], compensated Newton's scheme [32], [33], the compensated evaluation of elementary symmetric functions [34], or the compensated algorithm for solving triangular systems [35].

Acknowledgments

This work was partly supported by the project FastRelax ANR-14-CE25-0018-01. The authors wish to thank the reviewers for their helpful comments.

References

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754–2008, Aug. 2008.

[2] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson, "ARPREC: An arbitrary precision computation package," Tech. Rep. LBNL-53651, Lawrence Berkeley National Lab., Berkeley, CA, 2002.

[3] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[4] Y. Hida, X. Li, and D. Bailey, "Library for double-double and quad-double arithmetic," NERSC Division, Lawrence Berkeley National Laboratory, Berkeley, CA, 2008. [Online]. Available: http://www.davidhbailey.com/dhbpapers/qd.pdf

[5] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *Proc. 10th IEEE Symp. Comput.*, Jun. 1991, pp. 132–144.

[6] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete Comput. Geometry*, vol. 18, pp. 305–368, 1997.

[7] M. Joldes, J.-M. Muller, V. Popescu, and W. Tucker, "CAMPARY: Cuda multiple precision arithmetic library and applications," in *Proc. 5th Int. Congr. Math. Softw.*, Jul. 2016, pp. 232–240. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01312858

[8] D. Priest, "On properties of floating point arithmetics: Numerical stability and the cost of accurate computations," Ph.D. dissertation, Mathematics Dept., Univ. California, Berkeley, CA, Nov. 1992. [Online]. Available: ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z

[9] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM J. Sci. Comput.*, vol. 26, no. 6, pp. 1955–1988, 2005.

[10] S. Graillat, P. Langlois, and N. Louvet, "Algorithms for accurate, validated and fast polynomial evaluation," *Japan J. Indust. Appl. Math.*, vol. 2-3, no. 26, pp. 191–214, 2009.

[11] G. Alefeld and J. Herzberger, *Introduction to Interval Analysis*. Cambridge, MA, USA: Academic Press, 1983.

[12] U. Kulisch, *Advanced Arithmetic for the Digital Computer*, Wien: Springer, 2002.

[13] S. Graillat, F. Jézéquel, and R. Picot, "Numerical validation of compensated summation algorithms with stochastic arithmetic," *Electron. Notes Theoretical Comput. Sci.*, vol. 317, pp. 55–69, 2015.

[14] S. Graillat, F. Jézéquel, and R. Picot, "Numerical validation of compensated algorithms with stochastic arithmetic," *Appl. Mathematics and Comput.*, vol. 329, pp. 339–363, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0096300318300985

[15] S. Rump, "INTLAB - INTerval LABoratory," in *Developments in Reliable Computing*, T. Csendes, Ed. Kluwer, Dordrecht, the Netherlands: Academic Publishers, 1999, pp. 77–104, [Online]. Available: http://www.ti3.tu-harburg.de/rump/intlab

[16] N. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics (SIAM), 2002.

[17] C.-P. Jeannerod and S. M. Rump, "Improved error bounds for inner products in floating-point arithmetic," *SIAM J. Matrix Anal. Appl.*, vol. 34, no. 2, pp. 338–344, 2013. [Online]. Available: http://dx.doi.org/10.1137/120894488

[18] S. M. Rump, "Error estimation of floating-point summation and dot product," *BIT. Numerical Math.*, vol. 52, no. 1, pp. 201–220, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10543-011-0342-4

[19] S. M. Rump, F. Bünger, and C.-P. Jeannerod, "Improved error bounds for floating-point products and Horner's scheme," *BIT Numerical Math.*, vol. 56, no. 1, pp. 293–307, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10543-015-0555-z

[20] T. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971. [Online]. Available: http://dx.doi.org/10.1007/BF01397083

[21] D. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Boston, MA, USA: Addison-Wesley, 1997.

[22] 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*, Boston, MA, USA: Birkhäuser, 2010.

[23] J. Hauser, "Handling floating-point exceptions in numeric programs," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 2, pp. 139–174, 1996.

[24] J. Demmel and H. D. Nguyen, "Fast reproducible floating-point summation," in *Proc. 21st IEEE Symp. Comput. Arithmetic*, 2013, pp. 163–172.

[25] S. Boldo, S. Graillat, and J.-M. Muller, "On the robustness of the 2Sum and Fast2Sum algorithms," *ACM Trans. Math. Softw.*, vol. 44, no. 1, pp. 4:1–4:14, Jul. 2017.

[26] P. Sterbenz, *Floating-Point Computation*, Englewood Cliffs, NJ, USA: Prentice-Hall, 1973. [Online]. Available: http://books.google.fr/books?id=MKpQAAAAMAAJ

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

[28] A. Neumaier, "Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen," *ZAMM (Zeitschrift für Angewandte Mathematik und Mechanik)*, vol. 54, pp. 39–51, 1974.

[29] M. Pichat, "Correction d'une somme en arithmétique à virgule flottante," *Numerische Mathematik*, vol. 19, pp. 400–406, 1972.

[30] S. Graillat, N. Louvet, and P. Langlois, "Compensated Horner scheme," Research Report 4, DALI Research Team, Laboratoire LP2A, Université de Perpignan Via Domitia, Perpignan, France, July 2005.

[31] S. Graillat, "Accurate floating point product and exponentiation," *IEEE Trans. Comput.*, vol. 58, no. 7, pp. 994–1000, Jul. 2009.

[32] S. Graillat, "Accurate simple zeros of polynomials in floating point arithmetic," *Comput. Math. Appl.*, vol. 56, no. 4, pp. 1114–1120, 2008.

[33] H. Jiang, S. Graillat, C. Hu, S. Lia, X. Liao, L. Cheng, and F. Su, "Accurate evaluation of the k-th derivative of a polynomial," *J. Comput. Appl. Math.*, vol. 191, pp. 28–47, 2013.

[34] H. Jiang, S. Graillat, and R. Barrio, "Accurate and fast evaluation of elementary symmetric functions," in *Proc. 21st IEEE Symp. Comput. Arithmetic*, 2013, pp. 183–190.

[35] N. Louvet, "Algorithmes compensés en arithmétique flottante : Précision, validation, performances," PhD thesis, Laboratoire ELIAUS : Électronique, Informatique, Automatique et Systèmes, Université de Perpignan Via Domitia, Perpignan, France, Nov. 2007.

Stef Graillat received the PhD degree from Université de Perpignan, France, in 2005. He is a professor of computer science at Sorbonne Université and co-head of the PEQUAN team at LIP6 laboratory. His research interests include the computer arithmetic, floating-point arithmetic, and validated computing.

Fabienne Jézéquel received the PhD degree from Université Pierre & Marie Curie, France, in 1996. She is an associate professor of computer science at Université Panthéon-Assas and co-head of the PEQUAN team at LIP6 laboratory in Sorbonne Université. Her research interests include floating-point arithmetic, numerical validation, and high performance computing.