Numerische Mathematik (2025) 157:1615–1641
https://doi.org/10.1007/s00211-025-01487-2

Numerische
Mathematik

Emulation of the FMA and the correctly-rounded sum of three numbers in rounding-to-nearest floating-point arithmetic

Stef Graillat[1] · Jean-Michel Muller[2]

Received: 14 May 2024 / Revised: 16 March 2025 / Accepted: 22 July 2025 /
Published online: 9 September 2025

Abstract
We present algorithms that allow one to emulate the fused multiply-add (FMA) instruction and the correctly-rounded sum of three numbers (ADD3) in binary floating-point arithmetic, using only rounding-to-nearest floating-point additions, multiplications, and comparisons. We then introduce variants of these algorithms that make it possible to compute the error of an ADD3 or FMA operation.

Mathematics Subject Classification 65G50 · 65Y04 · 65Y10

1 Introduction

1.1 Motivation

The *fused multiply-add* (FMA) instruction evaluates an expression of the form $ab + c$, where a, b, and c are floating-point numbers, with one final rounding only. It appeared in 1990 in the IBM POWER instruction set [8], and its specification was incorporated in the 2008 version of the IEEE-754 Standard for Floating-Point Arithmetic [1]. It facilitates the software implementation of correctly rounded division and square root [9, 21], and, in general, allows for faster and more accurate evaluation of dot products and polynomials.

A typical application where the FMA instruction is very useful is the accurate implementation of transcendental functions [24]. Typically, after an initial *range*

✉ Stef Graillat
stef.graillat@lip6.fr

1 Sorbonne Université, CNRS, LIP6, Paris, France

2 CNRS, Laboratoire LIP, Université de Lyon, Lyon, France

reduction, the problem of evaluating the function is reduced to the problem of evaluating a polynomial $a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$, where $|x|$ is small (typically much less than 1). If the polynomial is evaluated using Horner's scheme, the last step is the calculation of $a_0 + x\rho$, where $\rho = a_1 + x(a_2 + x(a_3 + \cdots))$. Since the reduced argument x is small, a small error in ρ will not change the result much, so most of the final error is in this last step. Performing this step with only one rounding error instead of two (one for the multiplication and one for the addition) makes a significant difference.

The FMA instruction is implemented in most general-purpose computing environments. However, there are a few notable exceptions: the Java Virtual Machine [19] and WebAssembly[1]. In special-purpose environments such as the microcontroller units used for instance in automotive applications, the situation is more varied and the FMA instruction is often absent. An example is the Bosch BHI260AB microcontroller based on the ARC EM4 CPU. If one wishes to use on such systems an algorithm that requires the use of an FMA, one needs to emulate that instruction.

Another instruction that would simplify many calculations is the correctly-rounded sum of three floating-point numbers, let us call it *ADD3*. As mentioned by Lauter [18], it would help the final rounding step in correctly-rounded elementary functions. It would also provide a way to "normalize" triple-word numbers, which is a key feature for implementing high-precision arithmetic [6, 11]. A *fast, hardware,* ADD3 would also allow the replacement of the 2Sum algorithm (Algorithm 2 below) by a much simpler algorithm (but we will not fulfill that purpose here, as we are going to use 2Sum to emulate ADD3!). The main difference with the FMA is that ADD3 is not required by IEEE-754, and is therefore not offered by the current mainstream processors. In low-precision arithmetics (typically, 16-bit arithmetic), the recent Tensor cores could be used to directly compute (i.e., without decomposing it into two consecutive additions) the sum of 3 numbers. Whether this would always be with correct rounding is still unclear [12].

The ability to compute the *error* of the FMA and ADD3 is also interesting. The error of an FMA operation can be "reinjected" later on in a calculation to compensate for it. This is the key to *compensated* algorithms (the best-known compensated algorithm is Kahan's compensated summation algorithm [15]). For example, an algorithm developed by Boldo and Muller to compute the error of the FMA has been used by Louvet to construct a compensated polynomial evaluation algorithm [13, 20]. The error of ADD3 and the FMA can be used in double-word arithmetic, to improve or simplify the algorithms presented in [14].

Our goal in the paper is to present algorithms to emulate the FMA and ADD3 operations and compute the error of these operations. We assume *rounding-to-nearest* arithmetic. We aim to provide *high-level* algorithms, in the sense that they don't use the specific internal binary representations of the floating-point numbers: they use only the floating-point operations and comparisons. In particular, they do not perform integer or logical operations on the bit strings representing the floating-point operands. We believe that such an approach results in more general, portable and "robust" programs, which will work even when the floating-point format of the oper-

[1] https://developer.mozilla.org/en-US/docs/WebAssembly/Reference/Numeric.

ands is not one of the formats specified by IEEE-754 (incidentally, even within the IEEE-754 standard, problems of endianness[2] can in rare cases affect the correctness of algorithms that use integer arithmetic).

Various authors have suggested algorithms for implementing these functions. Boldo and Melquiond [6] showed that ADD3 and the FMA can be easily emulated provided by using a *round-to-odd* rounding function. Unfortunately, this rounding function is not yet available on current processors and is not specified in the current version of the IEEE-754 Standard for Floating-Point Arithmetic. Boldo and Melquiond give a solution for emulating round-to-odd (this is program OddRoundSum, included in the appendix of this paper). As we will see in Sect. 6, it is quite efficient, but it requires using the binary representation of the floating-point numbers. Lauter [18] provides a fast software implementation of ADD3 that uses integer arithmetic. Boldo and Muller [3, 7] give a high-level algorithm that computes the error of the FMA (assuming that a FMA instruction is available).

1.2 Notation and definitions

Throughout the paper, we assume a binary, precision-*p* floating-point (FP) arithmetic. Unless otherwise stated, the exponent range is assumed to be unbounded. This implies that the results presented here apply to conventional binary floating-point arithmetic provided that underflow and overflow do not occur. A floating-point number in such an arithmetic is a number of the form

$$x = M_x \cdot 2^{e_x - p + 1},$$

where M_x and e_x (called respectively *integral significand* and *floating-point exponent* of *x*) are integers, and either $M_x = 0$, or $2^{p-1} \leq |M_x| \leq 2^p - 1$. We note $\mathbb{F}$ the set of the FP numbers. If *t* is a real number, we call floating-point predecessor (resp. floating-point successor) of *t* the largest FP number less than *t* (resp. the smallest FP number larger than *t*).

We assume that the rounding function is *round-to-nearest, ties-to-even*, noted RN, which is the default in IEEE 754 arithmetic. RN is a piecewise-constant, increasing function. We call *midpoints* the real numbers where its value changes. A midpoint is exactly halfway between two consecutive FP numbers. The *unit round-off* is $u = 2^{-p}$. It is an upper bound on the relative error due to rounding. This implies that when an arithmetic operation $x \top y$ is performed (with $\top \in \{+, -, \times, \div\}$), the computed result $z = \mathrm{RN}(x \top y)$ satisfies

$$(1 - u) \cdot |(x \top y)| \leq |z| \leq (1 + u) \cdot |(x \top y)|. \tag{1}$$

If *t* is a real number, we define $\mathrm{ulp}(t)$ ("ulp" stands for *unit in the last place*) as

$$\begin{cases} 0 & \text{if } t = 0, \\ 2^{\lfloor \log_2 |t| \rfloor - p + 1} & \text{otherwise.} \end{cases}$$

[2] *Endianness* is the order in which bytes within a FP number are addressed in memory.

If $t \notin \mathbb{F}$, $\mathrm{ulp}(t)$ is the distance between the two consecutive FP numbers that surround t.

We will say that x is a *double-word*[3] (DW) number if it is the unevaluated[4] sum $x_h + x_\ell$ of two floating-point numbers x_h and x_ℓ such that $x_h = \mathrm{RN}(x)$ (so that $|x_\ell| \leq \frac{1}{2}\mathrm{ulp}(x_h + x_\ell)$). Some algorithms for manipulating double-word numbers are presented and analyzed in [14].

1.3 Structure of the article

In Sect. 2, we briefly present some classical results on floating-point arithmetic that are needed in the rest of the paper. More detailed presentations and proofs can be found in [5, 25]. Section 3 presents the first part of our contribution: preliminary results (test that determines whether a FP number is a power of 2 or 3 times a power of 2, sum of a DW number and a FP number) that will be needed to emulate the FMA and ADD3. Section 4 presents our algorithms that emulate ADD3 and the FMA, and compute the error of these operations. Section 5 is devoted to the special case of computing the error of an FMA operation on systems where a fast FMA is available in hardware. In Sect. 6, we discuss our results and compare them to the state of the art.

2 Some classical results on floating-point arithmetic

To emulate an FMA instruction using FP multiplications and additions, it is necessary to perform an analysis of the errors associated with these operations. Although very useful, the relative error bound (1) is not the last word:

- First, some operations are *exact*. A straightforward example is the case of multiplications and divisions by powers of 2. Another, less intuitive, example is the case of the subtraction of two numbers that are close enough to each other, as presented in Sect. 2.1;
- Second, a simple analysis shows that the error of an FP addition or multiplication is an FP number.[5] See for instance [2, 4]. Furthermore, these errors can be computed, using relatively simple algorithms, called *Error-Free Transforms* in the literature [26], presented in Sect. 2.2 (for addition) and Sect. 2.3 (for multiplication).

[3] We frequently see the name "double-double" in the literature. We prefer "double-word" because there is no reason to systematically assume that the underlying format is double precision/binary64.

[4] By "unevaluated sum" we mean that we keep the two values x_h and x_ℓ: they represent the number $x_h + x_\ell$ but no addition is performed.

[5] Concerning addition, this is true only when the rounding function is *round-to-nearest*, which we have assumed here.

2.1 Sterbenz's theorem

Sterbenz's theorem is extremely useful in error analysis. For instance, the proof of the double-word algorithms presented in [14] heavily relies on Sterbenz's theorem.

Theorem 2.1 *(Sterbenz Theorem [28]) Let $a, b \in \mathbb{F}$. If $\frac{a}{2} \le b \le 2a$ then $a - b \in \mathbb{F}$. This implies that the subtraction $a - b$ will be performed exactly in FP arithmetic.*

Theorem 2.1 seems to contradict the well-known (and wise!) rule that it is dangerous to subtract two close numbers because the result may be very inaccurate due to ill-conditioning of subtraction. There is no contradiction here: the computed subtraction s is exactly equal to $a - b$, but if a is an approximation to some real number $\hat{a}$ and b is an approximation to another number $\hat{b}$, even if these approximations are excellent, s can be quite far (in terms of relative error) from $\hat{a} - \hat{b}$. The subtraction itself is errorless but it somehow exposes the error on the input values.

2.2 The Fast2Sum and 2Sum algorithms

The error of a floating-point addition can be calculated using algorithms 1 and 2 below.

```
$x_h \leftarrow \mathrm{RN}(a+b)$
$z \leftarrow \mathrm{RN}(x_h - a)$
$x_\ell \leftarrow \mathrm{RN}(b - z)$
return $(x_h, x_\ell)$
```

Algorithm 1 Fast2Sum(a, b). The Fast2Sum algorithm [10]. Returns a pair $(x_h, x_\ell) \in \mathbb{F}^2$ such that x_h is the FP number nearest $a + b$ (i.e., the result of the FP addition of a and b), and, if $|a| \ge |b|$, $x_\ell = (a + b) - x_h$.

If the floating-point exponents e_a and e_b of a and b are such that $e_a \ge e_b$ then the number x_ℓ returned by Algorithm 1 is the error of the floating-point addition $\mathrm{RN}(a + b)$ (i.e., the double-word (x_h, x_ℓ) is exactly equal to $a + b$). The condition on the exponents may be difficult to check, but it is satisfied if $|a| \ge |b|$. Note that when $a = 0$ Algorithm 1 also returns the correct result ($x_h = b$ and $x_\ell = 0$).

```
$x_h \leftarrow \mathrm{RN}(a+b)$
$a' \leftarrow \mathrm{RN}(x_h - b)$
$b' \leftarrow \mathrm{RN}(x_h - a')$
$\delta_a \leftarrow \mathrm{RN}(a - a')$
$\delta_b \leftarrow \mathrm{RN}(b - b')$
$x_\ell \leftarrow \mathrm{RN}(\delta_a + \delta_b)$
return $(x_h, x_\ell)$
```

Algorithm 2 2Sum(a, b). The 2Sum algorithm [16, 23]. Returns a pair $(x_h, x_\ell) \in \mathbb{F}^2$ such that x_h is the FP number nearest $a + b$ (i.e., the result of the FP addition of a and b), and $x_\ell = (a + b) - x_h$.

For all FP numbers a and b, the number x_ℓ returned by Algorithm 2 is the error of the floating-point addition $\mathrm{RN}(a+b)$, i.e., $x_\ell = (a+b) - x_h$. Roughly speaking, the algorithm builds two FP numbers a' and b' such that $a' + b' = x_h$ exactly, and such that the FP subtractions $\delta_a = \mathrm{RN}(a - a')$ and $\delta_b = \mathrm{RN}(b - b')$ are errorless (i.e., $\delta_a = a - a'$ and $\delta_b = b - b'$).

2.3 The Dekker-Veltkamp multiplication algorithm

If an FMA instruction is available, then the error of an FP multiplication is very easy and fast to compute: the error of the multiplication $\pi_h = \mathrm{RN}(ab)$ is $\pi_\ell = \mathrm{RN}(ab - \pi_h)$. We will use this property in Sect. 5. However, if our goal is to emulate an FMA instruction, we obviously cannot assume that such an instruction is already available, so we must use a more complex algorithm, Algorithm 4 below, by Dekker and Veltkamp [10]. In order to compute the product *ab* "exactly", Algorithm 4 must first "split" the input operands *a* and *b* into sub-operands of precision around *p*/2, so that the product of two such sub-operands can be represented exactly in precision-*p* floating-point arithmetic (and is therefore obtained by a simple floating-point multiplication). This preliminary splitting is done by Algorithm 3. For a proof of these algorithms, see [25].

```
Require: $K = 2^s + 1$
Require: $2 \le s \le p - 2$
  $\gamma \leftarrow \mathrm{RN}(K \cdot x)$
  $\delta \leftarrow \mathrm{RN}(x - \gamma)$
  $x_h \leftarrow \mathrm{RN}(\gamma + \delta)$
  $x_\ell \leftarrow \mathrm{RN}(x - x_h)$
  return $(x_h, x_\ell)$
```

Algorithm 3 Split(*x*, *s*). Veltkamp's splitting algorithm. Returns a pair $(x_h, x_\ell) \in \mathbb{F}^2$ such that the significand of x_h fits in $p - s$ bits, the significand of x_ℓ fits in $s - 1$ bits, and $x_h + x_\ell = x$.

```
Require: $s = \lceil p/2 \rceil$
  $(a_h, a_\ell) \leftarrow \mathrm{Split}(a, s)$
  $(b_h, b_\ell) \leftarrow \mathrm{Split}(b, s)$
  $\pi_h \leftarrow \mathrm{RN}(a \cdot b)$
  $t_1 \leftarrow \mathrm{RN}(-\pi_h + \mathrm{RN}(a_h \cdot b_h))$
  $t_2 \leftarrow \mathrm{RN}(t_1 + \mathrm{RN}(a_h \cdot b_\ell))$
  $t_3 \leftarrow \mathrm{RN}(t_2 + \mathrm{RN}(a_\ell \cdot b_h))$
  $\pi_\ell \leftarrow \mathrm{RN}(t_3 + \mathrm{RN}(a_\ell \cdot b_\ell))$
  return $(\pi_h, \pi_\ell)$
```

Algorithm 4 DekkerProd(*a*, *b*). Dekker's product. Returns a pair $(\pi_h, \pi_\ell) \in \mathbb{F}^2$ such that $\pi_h = \mathrm{RN}(ab)$ and $\pi_h + \pi_\ell = ab$.

3 Preliminary results

Let a, b, and $c \in \mathbb{F}$. The problem of computing $\mathrm{RN}(a+b+c)$ (ADD3) and the problem of computing $\mathrm{RN}(ab+c)$ (FMA) can be reduced to a unique problem: computing $\mathrm{RN}(x_h + x_\ell + c)$, where (x_h, x_ℓ) is a DW number (which implies in particular that $|x_\ell| \le \frac{1}{2}\mathrm{ulp}(x_h)$. To show this, it is sufficient to choose

- $(x_h, x_\ell) = 2\mathrm{Sum}(a,b)$ (for ADD3), or
- $(x_h, x_\ell) = \mathrm{DekkerProd}(a,b)$ (for the FMA).

Therefore, we focus first on computing the sum of a DW number and a FP number. The algorithm we introduce for this purpose (Algorithm 6 below) requires at some point the ability to determine whether a FP number is of the form 2^k or $3 \cdot 2^k$, with $k \in \mathbb{Z}$ (or, equivalently, whether its integral significand is 2^{p-1} or $2^{p-1} + 2^{p-2}$). We address this problem first.

3.1 Determining if the absolute value of a FP number is a power of 2 or three times a power of 2

We have:

Theorem 3.1 *In binary, precision-p (with $p \ge 4$), floating-point arithmetic, assuming no overflow occurs, the absolute value of the nonzero FP number x is of the form 2^k or $3 \cdot 2^k$, with $k \in \mathbb{Z}$, if and only if*

$$RN\left[RN\left(\left(2^{p-2}+1\right)\cdot x\right) - 2^{p-2}x\right] = x. \qquad (2)$$

- Proof If $|x|$ is a power of 2, then multiplying by x is an exact operation and therefore (2) boils down to $\mathrm{RN}(x) = x$, which obviously holds since $x \in \mathbb{F}$.
- If $|x| = 3 \cdot 2^k$, with $k \in \mathbb{Z}$, then $\left(2^{p-2}+1\right)\cdot|x| = M \cdot 2^k$, where $M = 3 \cdot 2^{p-2} + 3$. As $p \ge 4$ implies $M \le 2^p - 1$, the number $\left(2^{p-2}+1\right)\cdot|x|$ is a FP number. Hence,

$$\mathrm{RN}\left(\left(2^{p-2}+1\right)\cdot x\right) = \left(2^{p-2}+1\right)\cdot x,$$

and, again, (2) boils down to $\mathrm{RN}(x) = x$;

- If $|x|$ is not of the form 2^k or $3 \cdot 2^k$ then there exist integers N and e such that N is odd, $N \ge 5$, and $|x| = N \cdot 2^e$. Let $P = 2^{p-2} + 1$. The number $P \cdot N$ is an odd integer of absolute value strictly larger than 2^p. Therefore,

$$P \cdot x = P \cdot N \cdot 2^e \notin \mathbb{F}.$$

Hence $\mathrm{RN}\,(P\cdot x)\neq P\cdot x$. From (1), we know that

$$x(2^{p-2}+1)(1-u)\leq \mathrm{RN}(P\cdot x)\leq x(2^{p-2}+1)(1+u).$$

This gives (remember: $u=2^{-p}$):

$$(1+4u)(1-u)\leq \frac{\mathrm{RN}(P\cdot x)}{2^{p-2}x}\leq (1+4u)(1+u),$$

so that (as $p\geq 4$ implies $u\leq 1/16$)

$$1\leq 1+3u-4u^2\leq \frac{\mathrm{RN}(P\cdot x)}{2^{p-2}x}\leq 1+5u+4u^2<2.$$

Therefore, we can apply Sterbenz Theorem (Theorem 2.1) to the subtraction $\mathrm{RN}\,(P\cdot x)-2^{p-2}x$, and deduce that that subtraction is exact. We therefore obtain that the left-hand part of (2) is equal to

$$\mathrm{RN}\,(P\cdot x)-2^{p-2}x,$$

which differs from $P\cdot x-2^{p-2}x=x.\square$

This gives the following algorithm

```
Require: $P = 2^{p-2}+1$
Require: $Q = 2^{p-2}$
  $L \leftarrow \mathrm{RN}(P\cdot x)$
  $R \leftarrow \mathrm{RN}(Q\cdot x)$
  $\Delta \leftarrow \mathrm{RN}(L-R)$
  return $(\Delta \neq x)$
```

Algorithm 5 IsNot1or3TimesPowerOf2(x). Returns **true** if and only if $|x|$ is not of the form 2^k or $3\cdot 2^k$.

Note that the condition $p\geq 4$ in Theorem 3.1 is necessary: if $p=3$, for $x=6$, we have $\mathrm{RN}\left[\mathrm{RN}\left(\left(2^{p-2}+1\right)\cdot x\right)-2^{p-2}x\right]=4\neq x$.

Algorithm 5 is related to other algorithms in the literature. Algorithm 3.6 (NextPowerTwo) in [27] computes the power of 2 immediately above $|x|$, where x is a FP number. It is based on the following property: if $x\in\mathbb{F}$ then

$$|\mathrm{RN}\,(\mathrm{RN}(2^p x+x)-2^p x)|=\begin{cases}2^{\lceil\log_2 x\rceil} & \text{if } x \text{ is not a power of 2,}\\ 0 & \text{otherwise.}\end{cases}$$

This can easily be used to check if x is a power of two. Our algorithm is slightly different, as the operand x is not multiplied by the same constants. Another example of an algorithm in the same "family" is Veltkamp's splitting algorithm (Algorithm 3) presented above: one could use this algorithm (with the parameter s set to 2) to "split" x into a 2-bit number x_h and a $(p-3)$-bit number x_ℓ, and x would be of the form 2^k or $3 \cdot 2^k$ if and only if $x_\ell = 0$. However, this would require more operations than Algorithm 5.

3.2 Correctly-rounded addition of a DW number and a FP number

Let x_h, x_ℓ, and c be FP numbers satisfying

$$|x_\ell| \le \frac{1}{2}\mathrm{ulp}(x_h),$$

and consider the following algorithm.

```
1: $(s_h, s_\ell) \leftarrow \mathrm{2Sum}(x_h, c)$
2: $(v_h, v_\ell) \leftarrow \mathrm{2Sum}(x_\ell, s_\ell)$
3: if IsNot1or3TimesPowerOf2($v_h$) or $v_\ell = 0$ then
4:    $z \leftarrow \mathrm{RN}(s_h + v_h)$
5: else
6:    if $v_\ell$ and $v_h$ have the same sign then
7:       $z \leftarrow \mathrm{RN}\left(s_h + \mathrm{RN}\left(\frac{9}{8}v_h\right)\right)$
8:    else
9:       $z \leftarrow \mathrm{RN}\left(s_h + \mathrm{RN}\left(\frac{7}{8}v_h\right)\right)$
10:   end if
11: end if
12: return $z$
```

Algorithm 6 **CR-DWPlusFP**(x_h, x_ℓ, c). Computes $\mathrm{RN}(x_h + x_\ell + c)$.

The constants 9/8 and 7/8 that appear in Algorithm 6 are exactly representable as soon as $p \ge 4$. We want to show that:

Theorem 3.2 *If $p \ge 5$ and $|x_\ell| \le \frac{1}{2}ulp(x_h)$, the number z returned by Algorithm CR-DWPlusFP (Algorithm 6) satisfies*

$$z = RN(x_h + x_\ell + c).$$

Let us first raise some remarks. In the following, we call Σ the number $\mathrm{RN}(x_h + x_\ell + c)$.

Remark 3.3 The 2Sum algorithm guarantees that the variables s_h, v_h and v_ℓ in Algorithm 6 satisfy

$$s_h + v_h + v_\ell = x_h + x_\ell + c, \text{ so that } \Sigma = \mathrm{RN}(s_h + v_h + v_\ell),$$
$$|v_\ell| \le \tfrac{1}{2}\mathrm{ulp}(v_h).$$

- Remark 3.4 If $v_\ell = 0$ then (from Remark 3.3) $s_h + v_h = x_h + x_\ell + c$ and therefore $z = \Sigma$;
- If $x_h = 0$ (which implies $x_\ell = 0$), then $\Sigma = c$ and one easily checks that $z = c$.

Remark 3.5 If $p \geq 5$, when $|v_h|$ is either a power of 2 or 3 times a power of 2, the terms $(7/8)v_h$ and $(9/8)v_h$ that appear in Algorithm 6 are FP numbers, and are therefore exactly computed: $\mathrm{RN}((7/8)v_h) = (7/8)v_h$ and $\mathrm{RN}((9/8)v_h) = (9/8)v_h$.

The following two remarks allow us to reduce the problem of proving Theorem 3.2 for all possible inputs x_h, x_ℓ, and c, to the problem of proving it for input values lying in a smaller domain.

Remark 3.6 If the input variables x_h, x_ℓ, and c of Algorithm 6 are multiplied by $s \cdot 2^k$, where $s = \pm 1$ and $k \in \mathbb{Z}$, then s_h, s_ℓ, v_h, v_ℓ, z and Σ are multiplied by the same factor $s \cdot 2^k$.

Remark 3.7 If we interchange x_h and c in Algorithm 6, the result remains unchanged (because $2\mathrm{Sum}(x_h, c) = 2\mathrm{Sum}(c, x_h)$). Furthermore, if $|c| > |x_h|$, the requirement $|x_\ell| \leq \frac{1}{2}\mathrm{ulp}(x_h)$ that appears in Theorem 3.2 still holds after having interchanged x_h and c. It therefore suffices to prove the theorem in the case $|x_h| \geq |c|$.

Let us now prove Theorem 3.2.
Without loss of generality, we can assume that:

- $x_h \neq 0$ (from Remark 3.4),
- $1 \leq x_h \leq 2 - 2u$ (from Remark 3.6) and therefore $|x_\ell| \leq \frac{1}{2}\mathrm{ulp}(1) = u$, and
- $|c| \leq x_h$ (from Remark 3.7).

We can immediately get rid of the case where

$$-x_h \leq c \leq -\frac{x_h}{2}.$$

In that case, Sterbenz's theorem (Theorem 2.1) implies that $x_h + c \in \mathbb{F}$. This in turn implies $s_\ell = 0$, so that $v_h = x_\ell$ and $v_\ell = 0$. As a consequence (from Remark 3.4), $z = \Sigma$.
We therefore only need to focus on the case

$$-\frac{x_h}{2} < c \leq x_h,$$

which implies $\frac{x_h}{2} < x_h + c \leq 2x_h$, and therefore $\frac{x_h}{2} \leq s_h \leq 2x_h$, so that

$$\frac{1}{2} \leq s_h \leq 4 - 4u.$$

Let us divide that case into three subcases:

- **Case A:** if $\frac{1}{2} \leq s_h \leq 1-u$ (so that s_h is a multiple of u) then $|s_\ell| \leq \frac{u}{2}$, which implies $|v_h| \leq \frac{3u}{2}$ and $|v_\ell| \leq u^2$;
- **Case B:** if $1 \leq s_h \leq 2-2u$ (so that s_h is a multiple of $2u$) then $|s_\ell| \leq u$, which implies $|v_h| \leq 2u$ and $|v_\ell| \leq u^2$;
- **Case C:** if $2 \leq s_h \leq 4-4u$ (so that s_h is a multiple of $4u$) then $|s_\ell| \leq 2u$, which implies $|v_h| \leq 3u$ and $|v_\ell| \leq 2u^2$.

Remark 3.8 In Case A, as $|v_h + v_\ell| = |x_\ell + s_\ell| \leq \frac{3u}{2}$, if $|v_h| = \frac{3u}{2}$ then either $v_\ell = 0$ or the signs of v_h and v_ℓ differ. Similarly, in Case C, as $|v_h + v_\ell| = |x_\ell + s_\ell| \leq 3u$, if $|v_h| = 3u$ then either $v_\ell = 0$ or the signs of v_h and v_ℓ differ.

As we already know, from Remark 3.4, that when $v_\ell = 0$ Algorithm 6 returns Σ, we assume in the following that $v_\ell \neq 0$.

If there is no midpoint between $s_h + v_h$ and $s_h + v_h + v_\ell$, then

$$\mathrm{RN}(s_h + v_h + v_\ell) = \mathrm{RN}(s_h + v_h).$$

The FP number v_h is a multiple of $\mathrm{ulp}(v_h)$ (which is a power of 2 less than or equal to $2u^2$), and in the three subcases (A, B, and C), s_h is a multiple of u. Therefore, $s_h + v_h$ is a multiple of $\mathrm{ulp}(v_h)$.

Also, as $s_h + v_h \geq \frac{1}{2} - \frac{3u}{2}$, the midpoints near $s_h + v_h$ are multiple of $\frac{u}{4}$. As $p \geq 5$ implies $u \leq \frac{1}{32}$ and therefore $\frac{u}{4} \geq 2u^2$, the midpoints near $s_h + v_h$ are multiple of $\mathrm{ulp}(v_h)$ too.

Hence, the distance between $s_h + v_h$ and a midpoint is a multiple of $\mathrm{ulp}(v_h)$. As $|v_\ell| \leq \frac{1}{2}\mathrm{ulp}(v_h)$, if $s_h + v_h$ is not itself a midpoint, then there is no midpoint between $s_h + v_h$ and $s_h + v_h + v_\ell$. This leads us to:

Remark 3.9 If $s_h + v_h$ is not a midpoint then $\mathrm{RN}(s_h + v_h + v_\ell) = \mathrm{RN}(s_h + v_h)$.

Now, as $s_h \in \mathbb{F}$, there are not many possibilities for $s_h + v_h$ to be a midpoint given the possible range of $|v_h|$. In the domain where $s_h + v_h$ can lie, the midpoints are the odd multiples of $\frac{u}{4}$ in $[\frac{1}{4}, \frac{1}{2})$, the odd multiples of $\frac{u}{2}$ in $[\frac{1}{2}, 1)$, and the odd multiples of u in $[1, 2)$. Therefore, if $s_h + v_h$ is a midpoint:

- **In Case A:** if $s_h = \frac{1}{2}$ then $v_h \in \{-\frac{3u}{4}, -\frac{u}{4}, \frac{u}{2}, \frac{3u}{2}\}$, and if $\frac{1}{2} < s_h \leq 1-u$ then $v_h \in \{-\frac{3u}{2}, -\frac{u}{2}, \frac{u}{2}, \frac{3u}{2}\}$;
- **In Case B:** if $s_h = 1$ then $v_h \in \{-\frac{3u}{2}, -\frac{u}{2}, u\}$, and if $1 < s_h \leq 2-2u$ then $v_h \in \{-u, u\}$;
- **In Case C:** if $s_h = 2$ then $v_h \in \{-3u, -u, 2u\}$ and if $2 < s_h \leq 4-4u$ then $v_h \in \{-2u, 2u\}$.

In all these cases, we observe that the possible values of v_h are either a power of 2, or 3 times a power of 2. Therefore:

Remark 3.10 If v_h is not a power of 2, or 3 times a power of 2 then $s_h + v_h$ is not a midpoint and therefore (from Remark 3.9), $\mathrm{RN}(s_h + v_h + v_\ell) = \mathrm{RN}(s_h + v_h)$. As a consequence, in this case, Algorithm 6 returns Σ.

Now, it remains to focus on the cases where v_h is a power of 2, or 3 times a power of 2. Let us assume that this is the case and first, consider Case A. Note that when $|v_h|$ is a power of two, the bound $|v_h| \le \frac{3u}{2}$ does not allow v_h to be larger than u. We have:

1. If ($s_h = \frac{1}{2}$ and $v_h \in \{-\frac{u}{4}, \frac{u}{2}\}$) or ($s_h > \frac{1}{2}$ and $v_h \in \{-\frac{u}{2}, \frac{u}{2}\}$) then $s_h + v_h$ is a midpoint. We must therefore return:
   - s_h if v_h and v_ℓ have different signs, and
   - the FP predecessor (if $v_h < 0$) or successor (if $v_h > 0$) of s_h otherwise.

 This is what Algorithm 6 does, as $\frac{7}{8}v_h$ and $\frac{9}{8}v_h$ are exactly computed. This is illustrated in Fig. 1;

2. If $|v_h|$ is a power of two less than γ, with $\gamma = \frac{u}{4}$ (when $s_h = \frac{1}{2}$ and $v_h < 0$) or $\gamma = \frac{u}{2}$ (other cases) then $s_h + v_h$ is not a midpoint (hence, from Remark 3.9, $\Sigma = \mathrm{RN}(s_h + v_h)$). One notices that, as v_h is a power of 2, $(9/8)v_h$ too is less than γ. Hence,
$$\mathrm{RN}\left(s_h + \frac{7}{8}v_h\right) = \mathrm{RN}\left(s_h + \frac{9}{8}v_h\right) = \mathrm{RN}\left(s_h + v_h\right) = s_h,$$
 so that the value z returned by Algorithm 6 is equal to Σ;
3. if ($s_h = \frac{1}{2}$ and $v_h \in \{-\frac{u}{2}, u\}$) or ($s_h \in \{\frac{1}{2}, \frac{1}{2} + u\}$ and $v_h = -\frac{3u}{2}$) or ($s_h > \frac{1}{2}$ and $|v_h| = u$) then $s_h + v_h \in \mathbb{F}$, hence it is not a midpoint and one easily checks that
$$\mathrm{RN}\left(s_h + \frac{7}{8}v_h\right) = \mathrm{RN}\left(s_h + \frac{9}{8}v_h\right) = \mathrm{RN}\left(s_h + v_h\right) = \Sigma;$$
 This is illustrated in Fig. 2;
4. if ($s_h \in \{\frac{1}{2}, \frac{1}{2} + u\}$ and $v_h = \frac{3u}{2}$) or ($s_h \ge \frac{1}{2} + 2u$ and $v_h = \pm\frac{3u}{2}$) then, as v_h and v_ℓ are necessarily of opposite signs (from Remark 3.8), Σ is equal to the FP

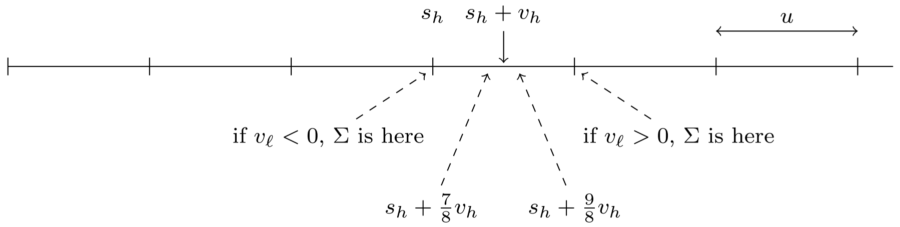
s_h
$s_h + v_h$
u
if $v_\ell < 0$, Σ is here
if $v_\ell > 0$, Σ is here
$s_h + \frac{7}{8}v_h$
$s_h + \frac{9}{8}v_h$

Fig. 1 The subcase $\frac{1}{2} < s_h < 1 - u$ and $v_h = +\frac{u}{2}$

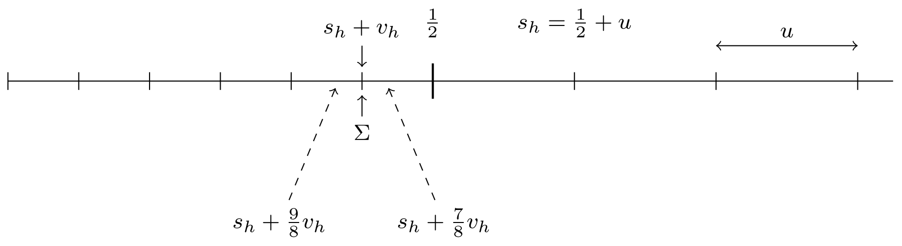

Fig. 2 The subcase $s_h = \frac{1}{2} + u$ and $v_h = -\frac{3u}{2}$

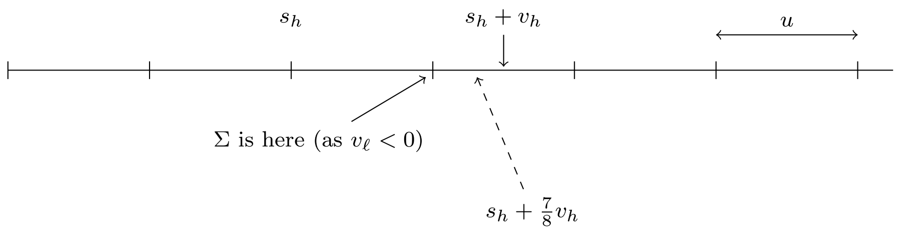

Fig. 3 The subcase $\frac{1}{2} \le s_h < 1 - u$ and $v_h = +\frac{3u}{2}$

predecessor (if $v_h < 0$) or the FP successor (if $v_h > 0$) of s_h. This is precisely what Algorithm 6 returns, as z is equal to

$$\mathrm{RN}\left(s_h + \frac{7}{8}v_h\right) = \mathrm{RN}\left(s_h + \mathrm{sign}(v_h) \cdot \frac{21u}{16}\right),$$

which equals $\mathrm{RN}(s_h + \mathrm{sign}(v_h) \cdot u)$, as there is no midpoint between $s_h + \mathrm{sign}(v_h) \cdot u$ and $s_h + \mathrm{sign}(v_h) \cdot \frac{21u}{16}$. This is illustrated in Fig. 3;

5. if $s_h = \frac{1}{2}$ and $v_h = -\frac{3u}{4}$ (so that $s_h + v_h$ is a midpoint), one must return

   - $\frac{1}{2} - u$ if $v_\ell < 0$, and
   - $\frac{1}{2} - \frac{u}{2}$ otherwise.

This is what the algorithm does, as

$$\left.\begin{array}{l} s_h + \frac{7}{8}v_h = \frac{1}{2} - \frac{21u}{32} \text{ is slightly above} \\ s_h + \frac{9}{8}v_h = \frac{1}{2} - \frac{27u}{32} \text{ is slightly below} \end{array}\right\} \text{ the midpoint } \frac{1}{2} - \frac{3u}{4}.$$

This case is illustrated in Fig. 4.

6. if $(s_h = \frac{1}{2}$ and $v_h = \frac{3u}{4})$ or $(s_h > \frac{1}{2}$ and $v_h = \pm\frac{3u}{4})$ then, as $s_h + v_h$ is not a midpoint, Σ is equal to $\mathrm{RN}(s_h + v_h)$. As there is no midpoint between

$$s_h + \frac{7}{8}v_h = s_h + \mathrm{sign}(v_h) \cdot \frac{21u}{32}$$

and

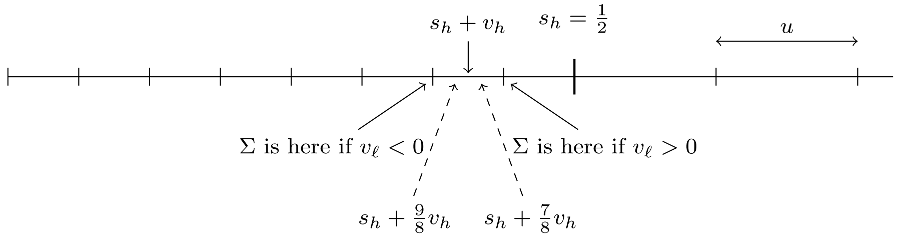

Fig. 4 The subcase $s_h = \frac{1}{2}$ and $v_h = -\frac{3u}{4}$

$$s_h + \frac{9}{8}v_h = s_h + \mathrm{sign}(v_h) \cdot \frac{27u}{32},$$

Algorithm 6 returns $\mathrm{RN}(s_h + v_h)$ whatever the sign of v_ℓ;

7. finally, if $|v_h|$ is three times a power of 2 and is less than or equal to $\frac{3u}{8}$, then $s_h + v_h$ is not a midpoint and one easily checks that

$$\mathrm{RN}\left(s_h + \frac{7}{8}v_h\right) = \mathrm{RN}\left(s_h + \frac{9}{8}v_h\right) = \mathrm{RN}\left(s_h + v_h\right) = s_h = \Sigma.$$

Cases B and C are processed in the same way: the reasoning is exactly the same. It suffices to consider the presented-above sub-cases of Case A, with the values of the variables s_h and v_h multiplied by two (for Case B) or by four (for Case C) – there are in fact less possible sub-cases to be considered, as the bounds on $|v_h|$ and $|v_\ell|$ in case B (respectively $2u$ and u^2) are less than 2 times the bounds of Case A (respectively $\frac{3u}{2}$ and u^2), and the bounds on the same variables in Case C (respectively $3u$ and $2u^2$) are less than four times the bounds of Case A.

3.3 Variant of Algorithm 6: sum of a DW number and a FP number, and error of that sum

Given x_h, x_ℓ and $c \in \mathbb{F}$ such that $|x_\ell| \leq \frac{1}{2}\mathrm{ulp}(x_h)$, Algorithm 6 computes $\Sigma = \mathrm{RN}(x_h + x_\ell + c)$. Let us assume that now we also want to compute the *error* of that addition, namely

$$e = x_h + x_\ell + c - \Sigma.$$

One easily sees that e does not always fit in one FP number only: just consider the case $x_h = 1$, $x_\ell = 2^{-p}$ and $c = 2^{-3p}$, for which $e = -2^{-p} + 2^{-3p}$ needs $2p$ bits to be represented exactly. Below, we give an algorithm (Algorithm 7, derived from Algorithm 6) that expresses e as the unevaluated sum of 2 FP numbers. We assume that $p \geq 5$ so that, from Theorem 3.2, Algorithm 6 can be used.

Let us consider the variables s_h, v_h and v_ℓ defined by Algorithm 6. We know that $x_h + x_\ell + c = s_h + v_h + v_\ell$. As in the proof of Theorem 3.2, we assume without loss of generality that $1 \le x_h \le 2 - 2u$. Define two FP numbers $w_h = \mathrm{RN}(s_h + v_h)$ and $w_\ell = s_h + v_h - w_h$. They can be computed with a Fast2Sum operation:

- when proving Theorem 3.2, we have seen that if $-\frac{x_h}{2} < c \le x_h$ then $s_h \ge \frac{1}{2}$ and $|v_h| \le 3u$, so that $s_h \ge |v_h|$;
- if $-x_h \le c \le -\frac{x_h}{2}$, then x_h and c (and therefore v_h) are multiple of $\frac{1}{2}\mathrm{ulp}(x_h)$ whereas $|v_h| = |x_\ell| \le \frac{1}{2}\mathrm{ulp}(x_h)$, therefore either $s_h = 0$ or $s_h \ge |v_h|$. In both cases $\mathrm{Fast2Sum}(s_h, v_h)$ returns w_h and w_ℓ.

An easy case for computing *e* is when $\Sigma = w_h$: in that case $e = w_\ell + v_\ell$. We have seen in the proof of Theorem 3.2 that this happens:

- when $-x_h \le c \le -\frac{x_h}{2}$ (as $v_\ell = 0$),
- when $-\frac{x_h}{2} < c \le x_h$ and $s_h + v_h$ is not a midpoint (from Remark 3.9).

Let us now assume that $-\frac{x_h}{2} < c \le x_h$ and that $s_h + v_h$ is a midpoint (which implies that v_h is a power of 2, or 3 times a power of 2, from Remark 3.10). In that case, Σ is equal to $\mathrm{RN}(s_h + \mathrm{RN}(\frac{9}{8}v_h)) = \mathrm{RN}(s_h + \frac{9}{8}v_h)$ or to $\mathrm{RN}(s_h + \mathrm{RN}(\frac{7}{8}v_h)) = \mathrm{RN}(s_h + \frac{7}{8}v_h)$, depending on the respective signs of v_h and v_ℓ. Let us consider the three sub-cases A, B, and C of the proof of Theorem 3.2. The number

$$\left|\left(s_h + \frac{9}{8}v_h\right) - \left(s_h + \frac{7}{8}v_h\right)\right| = \left|\frac{v_h}{4}\right|$$

is

$$\begin{cases} < \frac{u}{2} & \text{in case A,} \\ \le \frac{u}{2} & \text{in case B,} \\ \le u & \text{in case C,} \end{cases}$$

and the distance between $s_h + v_h$ and the closest midpoint is

$$\begin{cases} \frac{u}{2} & \text{if} \quad s_h + v_h < \frac{1}{2} & (\textit{may occur in Case A}), \\ \frac{3u}{4} & \text{if} \quad s_h + v_h = \frac{1}{2} + \frac{u}{2} & (\textit{may occur in Case A}), \\ u & \text{if} \quad \frac{1}{2} + \frac{3u}{2} \le s_h + v_h < 1 & (\textit{may occur in Cases A and B}), \\ \frac{3u}{2} & \text{if} \quad s_h + v_h = 1 + u & (\textit{may occur in Case B}), \\ 2u & \text{if} \quad 1 + 3u \le s_h + v_h < 2 & (\textit{may occur in Cases B and C}), \\ 3u & \text{if} \quad s_h + v_h = 2 + 2u & (\textit{may occur in Case C}), \\ 4u & \text{if} \quad s_h + v_h \ge 2 + 6u & (\textit{may occur in Case C}). \end{cases}$$

Therefore $s_h + v_h$ is the only midpoint between $s_h + \frac{9}{8}v_h$ and $s_h + \frac{7}{8}v_h$. Hence $\mathrm{RN}(s_h + \frac{9}{8}v_h)$ and $\mathrm{RN}(s_h + \frac{7}{8}v_h)$ are two adjacent FP numbers, and w_h is one of them. It follows that

$$|\Sigma - w_h| \in \{0, \mathrm{ulp}(s_h + v_h)\}. \tag{3}$$

Let us define $\alpha = \Sigma - w_h$. Equation (3) implies that $\alpha \in \mathbb{F}$, so that the subtraction $\Sigma - w_h$ is exact in FP arithmetic. Also, as $s_h + v_h$ is a midpoint and $w_h = \mathrm{RN}(s_h + v_h)$, we have $|w_\ell| = |w_h - (s_h + v_h)| = \frac{1}{2}\mathrm{ulp}(s_h + v_h)$. Therefore,

$$|w_\ell - \alpha| \in \left\{\frac{1}{2}\mathrm{ulp}(s_h + v_h), \frac{3}{2}\mathrm{ulp}(s_h + v_h)\right\},$$

which implies that $\delta = w_\ell - \alpha$ is a power of 2 or three times a power of 2, so that $\delta \in \mathbb{F}$ (hence it is computed exactly: $\mathrm{RN}(t - \alpha) = w_\ell - \alpha$). We finally obtain

$$e = s_h + v_h + v_\ell - \Sigma = w_h + w_\ell - \Sigma + v_\ell = w_\ell - \alpha + v_\ell = \delta + v_\ell.$$

This expression also holds in the previously considered case $\Sigma = w_h$, since in that case $\delta = w_\ell$.

Finally, one may observe that δ is a multiple of $\mathrm{ulp}(v_h)$ whereas $|v_\ell| \le \frac{1}{2}\mathrm{ulp}(v_h)$, so that (unless $\delta = 0$), $|\delta|$ is larger than $|v_\ell|$: one can use the Fast2Sum algorithm to add δ and v_ℓ if needed.

This gives Algorithm 6 and Theorem 6 below.

```
1:  $(s_h, s_\ell) \leftarrow$ 2Sum$(x_h, c)$
2:  $(v_h, v_\ell) \leftarrow$ 2Sum$(x_\ell, s_\ell)$
3:  $(w_h, w_\ell) \leftarrow$ Fast2Sum$(s_h, v_h)$
4:  if IsNot1or3TimesPowerOf2$(v_h)$ or $v_\ell = 0$ then
5:      $\delta \leftarrow w_\ell$
6:      $z \leftarrow w_h$
7:  else
8:      if $v_\ell$ and $v_h$ have the same sign then
9:          $z \leftarrow \mathrm{RN}\left(s_h + \mathrm{RN}\left(\frac{9}{8}v_h\right)\right)$
10:     else
11:         $z \leftarrow \mathrm{RN}\left(s_h + \mathrm{RN}\left(\frac{7}{8}v_h\right)\right)$
12:     end if
13:     $\alpha \leftarrow \mathrm{RN}(z - w_h)$
14:     $\delta \leftarrow \mathrm{RN}(w_\ell - \alpha)$
15: end if
16: return $(z, \delta, v_\ell)$
```

Algorithm 7 **CR-DWPlusFP-with-error**(x_h, x_ℓ, c). Computes $z = \mathrm{RN}(x_h + x_\ell + c)$, and δ and v_ℓ such that $z + \delta + v_\ell = x_h + x_\ell + c$.

We have,

Theorem 3.11 *If $p \geq 5$ and $|x_\ell| \leq \frac{1}{2} ulp(x_h)$, the numbers z, δ, and v_ℓ returned by Algorithm CR-DWPlusFP-with-error (Algorithm 7) satisfy*

$$z = RN(x_h + x_\ell + c)$$

and

$$\delta + v_\ell = x_h + x_\ell + c - z.$$

The pair (δ, v_ℓ) that represents the error of the addition $(x_h, x_\ell, c) \rightarrow \mathrm{RN}(x_h + x_\ell + c)$ is not, in general, a DW number. To "normalize" that result, the last line of Algorithm 7 can be modified as follows:

- if one just wants the FP number nearest the error of the addition, one can compute $\mathrm{RN}(\delta + v_\ell)$. This gives a possible replacement for the (simpler and faster, yet less accurate) DWPlusFP algorithm of [14];
- if one wants a DW number that represents the error exactly, one can compute $\mathrm{Fast2Sum}(\delta, v_\ell)$.

4 Emulation of ADD3 and the FMA, error of these operations

We now explain how Algorithm 6 can be used to emulate the ADD3 and FMA operations. In the following, we assume that $p \geq 5$, so that Theorem 3.2 can be used. First, let us assume we want to compute $\mathrm{RN}(a + b + c)$, where a, b, and c are FP numbers. The 2Sum algorithm (Algorithm 2) returns a pair (x_h, x_ℓ) such that $x_h = \mathrm{RN}(x_h + x_\ell)$ and $x_h + x_\ell = a + b$. From $x_h = \mathrm{RN}(x_h + x_\ell)$ we deduce $|x_\ell| \leq \frac{1}{2}\mathrm{ulp}(x_h)$. Therefore x_h, x_ℓ and c satisfy the condition of Theorem 3.2, hence Algorithm 6 can be used to compute $\mathrm{RN}(x_h + x_\ell + c) = \mathrm{RN}(a + b + c)$. We obtain:

```
1: $(x_h, x_\ell) \leftarrow \mathrm{2Sum}(a, b)$
2: return  $\mathrm{CR\text{-}DWPlusFP}(x_h, x_\ell, c)$
```

Algorithm 8 **EmulADD3**(a, b, c). Computes $\mathrm{RN}(a + b + c)$.

Theorem 4.1 *In a binary, precision-p (with $p \geq 5$), floating-point arithmetic with an unbounded exponent range, Algorithm 8 returns $RN(a + b + c)$ for all floating-point numbers a, b, and c.*

If we assume now that we want to compute $\mathrm{FMA}(a, b, c) = \mathrm{RN}(ab + c)$, we can in a very similar way use the Dekker-Veltkamp multiplication algorithm (Algorithm 3) to obtain a pair (x_h, x_ℓ) of FP numbers such that $x_h = \mathrm{RN}(x_h + x_\ell)$ and $x_h + x_\ell = ab$, and then use Algorithm 6 to compute $\mathrm{RN}(x_h + x_\ell + c) = \mathrm{RN}(ab + c)$. This gives:

```
1: $(x_h, x_\ell) \leftarrow \mathrm{DekkerProd}(a, b)$
2: return  $\mathrm{CR\text{-}DWPlusFP}(x_h, x_\ell, c)$
```

Algorithm 9 EmulFMA. Computes $\mathrm{FMA}(a, b, c) = \mathrm{RN}(ab + c)$.

Theorem 4.2 *In a binary, precision-p (with $p \geq 5$), floating-point arithmetic with an unbounded exponent range, Algorithm 9 returns $RN(ab + c)$ for all floating-point numbers a, b, and c.*

In Algorithms 8 and 9, if one replaces the call to CR-DWPlusFP(x_h, x_ℓ, c) (Algorithm 6) by a call to CR-DWPlusFP-with-error(x_h, x_ℓ, c) (Algorithm 7), then one will obtain algorithms that compute $\mathrm{RN}(a + b + c)$ and the error of that addition (Algorithm 10), and $\mathrm{RN}(ab + c)$ and the error of that FMA operation (Algorithm 11).

```
1: $(x_h, x_\ell) \leftarrow \mathrm{2Sum}(a, b)$
2: return CR-DWPlusFP-with-error$(x_h, x_\ell, c)$
```

Algorithm 10 **ADD3-with-error**(a, b, c). Computes $z = \mathrm{RN}(a + b + c)$ and δ and v_ℓ such that $a + b + c = z + \delta + v_\ell$.

```
1: $(x_h, x_\ell) \leftarrow \mathrm{DekkerProd}(a, b)$
2: return CR-DWPlusFP-with-error$(x_h, x_\ell, c)$
```

Algorithm 11 **FMA-with-error**(a, b, c). Computes $z = \mathrm{RN}(ab + c)$ and δ and v_ℓ such that $ab + c = z + \delta + v_\ell$.

Algorithm 10 allows one to "normalize" triple-word (TW) numbers (i.e., it transforms any unevaluated sum of three FP numbers into a TW). For that purpose, it is much simpler than Algorithm 6 of [11].

5 Error of the FMA when that instruction is available

Algorithm 11 computes $\mathrm{RN}(ab + c)$ and the error of this operation, namely $e = ab + c - \mathrm{RN}(ab + c)$. On systems where a fast FMA operation is available in hardware, one will not use Algorithm 11 for computing e, for two reasons: first, Algorithm 11 needs many operations just to compute $\mathrm{RN}(ab + c)$, whereas it can be obtained with only one FMA operation, and second, the availability of a fast FMA allows to express the product *ab* as a double-word (x_h, x_ℓ) much faster than by using the Dekker-Veltkamp algorithm. In fact, Algorithm 11 can be simplified considerably, and we obtain Algorithm 12 below.

```
1: $z_h \leftarrow \mathrm{RN}(ab + c)$
2: $x_h = \mathrm{RN}(ab)$
3: $x_\ell = \mathrm{RN}(ab - x_h)$
4: $(s_h, s_\ell) \leftarrow \mathrm{2Sum}(x_h, c)$
5: $(v_h, v_\ell) \leftarrow \mathrm{2Sum}(x_\ell, s_\ell)$
6: $\alpha' \leftarrow \mathrm{RN}(z_h - s_h)$
7: $\delta' \leftarrow \mathrm{RN}(v_h - \alpha')$
8: return $(z, \delta', v_\ell)$
```

Algorithm 12 Faster-FMA-with-error(a, b, c). Computes $z = \mathrm{RN}(ab + c)$ and δ' and v_ℓ such that $ab + c = z + \delta' + v_\ell$.

Let us explain how Algorithm 12 is derived from Algorithms 7 and 11.

- First, to obtain x_h and x_ℓ from a and b, it is no longer necessary to use the Dekker-Veltkamp multiplication algorithm, since $x_h = \mathrm{RN}(ab)$ and $x_\ell = \mathrm{RN}(ab - x_h) = ab - x_h$ are obtained with a multiplication and an FMA;
- second, the various tests needed to compute z in Algorithm 7 are no longer necessary, since $z = \mathrm{RN}(ab + c)$ is obtained with an FMA;
- finally, the computation of the variables w_h and w_ℓ in Algorithm 7 is no longer necessary:
  - When $z_h = \mathrm{RN}(s_h + v_h)$, the computation of α' and δ' (in lines 6 and 7 of Algorithm 12) constitutes the last two operations of Fast2Sum(s_h, v_h). This gives $\delta' = s_h + v_h - z_h$ and therefore $\delta' + v_\ell = s_h + v_h + v_\ell - z_h$, which is the desired error of the FMA;
  - when $z_h \neq \mathrm{RN}(s_h + v_h)$, the number $s_h + v_h$ is a midpoint (from Remark 3.9), and z_h and $\mathrm{RN}(s_h + v_h)$ are the two consecutive FP numbers surrounding this midpoint. It follows that $\alpha' \in \mathbb{F}$. It is even a power of 2, equal to $\pm\mathrm{ulp}(s_h + v_h)$, computed exactly. Furthermore (see Cases A, B, and C in the proof of Theorem 3.2),

$$v_h \in \left\{\pm\frac{1}{2}\mathrm{ulp}(s_h + v_h), \pm\frac{3}{2}\mathrm{ulp}(s_h + v_h)\right\},$$

and therefore,

$$v_h - \alpha' \in \left\{\pm\frac{1}{2}\mathrm{ulp}(s_h + v_h), \pm\frac{3}{2}, \pm\frac{5}{2}\mathrm{ulp}(s_h + v_h)\right\}$$

is a FP number, computed exactly. Therefore

$$\delta' + v_\ell = v_h - \alpha' + v_\ell = v_h + v_\ell + s_h - z_h = ab + c - \mathrm{RN}(ab + c).$$

The obtained Algorithm 12 is an alternative to the algorithm presented by Boldo and Muller in [7].

6 Discussion and comparisons

The primary disadvantage of Algorithms 6, 8, 9, and 11 is the presence of tests. In the event that the branch prediction mechanism of the processor fails,[6] tests can result in a significant reduction in performance. However, to emulate ADD3 and the FMA, tests seem to be unavoidable: the authors of [17] have shown that an algorithm using only rounding-to-nearest additions and subtractions (without any tests) cannot evaluate $\mathrm{RN}(a+b+c)$ for all possible FP numbers a, b, and c. Furthermore, it is important to note that the variable v_h in Algorithm 8 is very unlikely to be a power of 2 or three times a power of 2. Consequently, when a large number of ADD3s or FMAs are computed in a program, branch prediction should work effectively (whereas of course if only a small number of ADD3s or FMAs are computed in a program, the performance loss is of minimal consequence).

We have implemented Algorithms 8, 9, 11, and 12, the FMA and ADD3 algorithms of Boldo and Melquiond, and the algorithm of Boldo and Muller that computes the

[6] In modern processor architectures, the arithmetic operations are *pipelined*. This is assembly-line work adapted to computer architecture: the operations are divided into several steps (typically ranging from 3 to 10), and each step takes one cycle time of the processor. If several independent operations are required in a program, step A of the first operation is executed first, then step B of the first operation and step A of the second operation are executed in parallel, then step C of the first operation, step B of the second operation, and step A of the third operation are executed in parallel, and so on. The advantage is that a new operation can be started at each processor cycle. However, if the nature of the next operation to be performed depends on a test on the result of the current one (i.e., if there is a conditional *branch* in the program), we cannot start that next operation immediately, and we must wait until the current one is finished before we can "feed" the pipeline again, which results in a significant penalty. One way to avoid this is *branch prediction* [22]: the processor tries to predict what the result of the test will be (and thus which "branch" of the program will be executed). This generally works remarkably well (because real programs are not "random", they have some regularity that can be exploited. For example, very often, one branch is much more likely to be taken than the other ones: after a few initial loops this will be easy to detect). The other side of the coin is that if the outcome of the test was wrongly predicted (this is what we call a *branch prediction failure*), the penalty can become very important: the computation must be restarted at the beginning of the branch.

Table 1 Time (in seconds) to perform 5×10^9 ADD3 operations in binary64 arithmetic, using the Boldo-Melquiond algorithm and our algorithm (Algorithm 8), on different computer architectures and with different compilers

Architecture/system	Compiler and options	Boldo-Melquiond	Algorithm8
Intel Corei7	clang (v. 16.0.0)	177	153
under MacOS	clang -O3	22	19
Apple M3Pro	clang (v. 16.0.0)	142	144
under MacOS	clang -O3	7	9
AMD Opteron 6272	gcc (v. 12.2.0)	759	659
under Linux	gcc -O3	127	104
	clang -O3	168	93
Intel Xeon Gold 6444Y	gcc (v. 12.2.0)	95	84
under Linux	gcc -O3	18	20
	clang -O3 (v. 14.0.6)	18	15

Each operand is a random `double` of the form $K \times s \times F$, where F is uniform in [0, 1], s is -1 or $+1$ (each with probability 1/2), and $K \in \{1, 2^{20}, 2^{-20}, 2^{40}, 2^{-40}, 2^{60}, 2^{-60}, 2^{80}, 2^{-80}\}$ (each with probability 1/9)

Table 2 Time (in seconds) to perform 5×10^9 FMA operations in binary64 arithmetic, using the Boldo-Melquiond (BM) algorithm, our algorithm (Algorithm 9), and the FMA provided by the environment on different computer architectures and with different compilers

Architecture/system	Compiler and options	BM	Algorithm 9	Environment
Intel Corei7	clang (v. 16.0.0)	258	200	41
under MacOS	clang -O3	31	25	10
Apple M3Pro	clang (v. 16.0.0)	228	162	7
under MacOS	clang -O3	10	9	4
AMD	gcc (v. 12.2.0)	1068	856	75
Opteron 6272	gcc -O3 -lm	190	110	42
under Linux	clang -O3 -lm	181	95	43
Intel Xeon	gcc -lm (v. 12.2.0)	109	98	10
Gold 6444Y	gcc -O3 -lm	25	24	10
under Linux	clang -O3 -lm (v. 14.0.6)	25	21	10

Each operand is a random `double` of the form $K \times s \times F$, where F is uniform in [0, 1], s is -1 or $+1$ (each with probability 1/2), and $K \in \{1, 2^{20}, 2^{-20}, 2^{40}, 2^{-40}, 2^{-40}, 2^{60}, 2^{-60}, 2^{80}, 2^{-80}\}$ (each with probability 1/9)

Table 3 Time (in seconds) to compute 5×10^9 errors of FMA operations in binary64 arithmetic, using the Boldo-Muller (BM) algorithm and our algorithms (Algorithms 11 and 12), on different computer architectures and with different compilers

Architecture/system	Compiler and options	Boldo-Muller	Algorithm 11	Algorithm 12
Intel Corei7	clang (v. 16.0.0)	166	280	168
under MacOS	clang -O3	30	39	33
Apple M3Pro	clang (v. 16.0.0)	151	298	143
under MacOS	clang -O3	7	13	7
AMD Opteron 6272	gcc (v. 12.2.0)	742	1252	736
under Linux	gcc -O3	134	143	140
	clang -O3	117	122	130
Intel Xeon Gold 6444Y	gcc (v. 12.2.0)	89	144	87
under Linux	gcc -O3	23	30	24
	clang -O3 (v. 14.0.6)	22	28	22

Each operand is a random `double` of the form $K \times s \times F$, where F is uniform in [0, 1], s is -1 or $+1$ (each with probability 1/2), and $K \in \{1, 2^{20}, 2^{-20}, 2^{40}, 2^{-40}\ 2^{60}, 2^{-60}, 2^{80}, 2^{-80}\}$ (each with probability 1/9)

error of an FMA in the C language, in binary64 arithmetic, and compared them on several platforms and with two different compilers (clang and gcc, with and without optimization). The programs are included in the appendix of this paper. The results are given in Tables 1 (for the ADD3 operation), 2 (for the FMA), and 3 (for the FMA error). Since an FMA instruction was available on all the platforms used, Table 2 also shows the timings for the FMA provided by the platform.

To compare the performance of these algorithms, the protocol is as follows: each algorithm is called 5×10^9 times. The input values are random binary64 numbers of the form $K \times s \times F$, where F is uniform in [0, 1], s is -1 or $+1$ (each with probability 1/2), and $K \in \{1, 2^{20}, 2^{-20}, 2^{40}, 2^{-40}, 2^{60}, 2^{-60}, 2^{80}, 2^{-80}\}$ (each with probability 1/9). These random inputs are computed beforehand and stored in a table (otherwise we would not be able to discriminate between the various algorithms, since the time of a random sampling is far from negligible in front of the time of an FMA or ADD3 operation). The table must be large enough to ensure that all possible special cases do occur, but small enough to fit in cache memory: we have chosen a table of 1000 elements for each of the variables a, b, and c (thus, to be able to perform 5×10^9 operations, we access the tables repeatedly in a loop).

With regard to the calculation of the FMA and ADD3 (Tables 1 and 2), the timings for Boldo and Melquiond's algorithms and our algorithms (Algorithms 8 and 9) are always quite close and in general our algorithms perform slightly better (there are a few exceptions: ADD3 on an Apple M3Pro using the clang compiler, and on an Intel Xeon Gold using the gcc compiler with the -O3 option). Given the similar performance, on systems without a hardware FMA, the fact that Algorithms 8 and 9 are "high-level" algorithms (they use only FP operations) is a strong argument for choosing them. Interestingly enough, on the platforms used in the comparisons, the performance gain from using the hardware FMAs is just over a factor of 2 (when the emulation algorithms are compiled with the -O3 option). This shows the interest of these emulation algorithms.

Regarding the calculation of the error of the FMA when this instruction is available, the timings for the Boldo and Muller algorithm and Algorithm 12 are very close. This is not surprising, since both algorithms use two FMAs, one multiplication, two 2Sum operations, and two addition/subtractions (but in a different order). Algorithm 11 is significantly slower[7] because it does not use the available hardware FMA of the platform being used, and because it requires tests. If no hardware FMA is available on the platform being used, the only possible choice is to use Algorithm 11, but if a hardware FMA is available, the only advice we can give to a user is to try the Boldo and Muller algorithm and Algorithm 12: which of them performs best depends on the environment.

7 Conclusion

We have presented a novel approach to emulate the fused multiply-add (FMA) instruction and the ADD3 operation, and to compute the error of these operations, using only standard, rounding-to-nearest floating-point additions, multiplications, and comparisons. Our method builds on the foundation laid by previous research, but eliminates the need for less commonly supported rounding functions such as round-to-odd, or the need to perform integer or logical operations on the binary representations of the FP operands, thereby increasing the practical applicability and portability of the algorithm across different computing architectures.

[7] With one surprising exception: the AMD Operon 6270 and the clang compiler with option -O3.

Appendix: C programs for ADD3, the FMA and their errors

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>
#include <time.h>
#pragma STDC FP_CONTRACT OFF

struct DW
{double h, l;};

struct TW
{double h, m, l;};

typedef union
{uint64_t integer;
double real;} INTDOUBLE;

struct DW FastTwoSum(double a, double b){
struct DW z;
z.h = a+b;
z.l = (b-(z.h-a));
return z;}

struct DW TwoSum(double a, double b){
struct DW z;
double aprime;
z.h = a+b;
aprime = z.h-b;
z.l = (a - aprime) + (b - (z.h-aprime));
return z;}

struct DW split(double x){
static const double K = 134217729.;
struct DW splittedx;
double gamma = K*x;
splittedx.h = (gamma+(x-gamma));
splittedx.l = x - splittedx.h;
return splittedx;
}

struct DW DekkerProd(double a, double b){
struct DW splitteda, splittedb, product;
splitteda = split(a);
splittedb = split(b);
product.h = a*b;
product.l = (((-product.h + splitteda.h*splittedb.h)+(splitteda.h*splittedb.l))
      +splitteda.l*splittedb.h)+splitteda.l*splittedb.l;
return product;
}

int IsNot1or3timesPowerOf2 (double x){
static const double P = 2251799813685249.0, Q = 2251799813685248.0;
double Delta;
```

```
Delta = (P*x)-(Q*x);
return (Delta != x);
}

double OddRoundSum (double x, double y){
INTDOUBLE myvh;
struct DW v;
v = TwoSum(x,y);
myvh.real = v.h;
if (v.l != 0.0){
if (!(myvh.integer & 1)){
if ((v.h > 0.0) ^ (v.l < 0.0)){myvh.integer++;}
  else {myvh.integer--;}}
  }
return myvh.real;
}

double BoldoMelquiondADD3(double a, double b, double c){
struct DW x, s;
double v;
x = TwoSum(a,b);
s = TwoSum(x.h,c);
v = OddRoundSum(x.l,s.l);
return(s.h+v);
}

double OurADD3 (double a, double b, double c){
struct DW x, s, v;
x = TwoSum(a,b);
s = TwoSum(x.h,c);
v = TwoSum(x.l,s.l);
if ((IsNot1or3timesPowerOf2(v.h)) || (v.l == 0)){return s.h+v.h;}
else
       {
        if ((v.l < 0) ^ (v.h < 0))
          {return s.h+(0.875*v.h);}
         else
          {return s.h+(1.125*v.h);}
       }
}

double BoldoMelquiondFMA (double a, double b, double c){
struct DW x, s;
double v, z;
x = DekkerProd(a,b);
s = TwoSum(x.h,c);
v = OddRoundSum(x.l,s.l);
z = s.h+v;
return z;
}

double Ourfma (double a, double b, double c){
struct DW x, s, v;
x = DekkerProd(a,b);
s = TwoSum(x.h,c);
v = TwoSum(x.l,s.l);
if ((IsNot1or3timesPowerOf2(v.h)) || (v.l == 0)){return s.h+v.h;}
```

```
else
       {
        if ((v.l < 0) ^ (v.h < 0))
          {return s.h+(0.875*v.h);}
         else
          {return s.h+(1.125*v.h);}
       }
}

struct TW OurFmaWithError (double a, double b, double c){
struct DW x, s, v;
struct TW z;
double delta;
z.h = fma(a,b,c);
x.h = a*b;
x.l = fma(a,b,-x.h);
s = TwoSum(x.h,c);
v = TwoSum(x.l,s.l);
delta = v.h - (z.h - s.h);
v = FastTwoSum (delta,v.l);
z.m = v.h;
z.l = v.l;
return z;
}

struct TW BoldoMullerFmaWithError (double a, double b, double c){
struct DW x, alpha, beta, r;
struct TW z;
double gamma;
z.h = fma(a,b,c);
x.h = a*b;
x.l = fma(a,b,-x.h);
alpha = TwoSum(c,x.l);
beta = TwoSum(x.h,alpha.h);
gamma = (beta.h-z.h)+beta.l;
r = FastTwoSum(gamma,alpha.l);
z.m = r.h;
z.l = r.l;
return z;
}
```

Acknowledgements We are grateful to the (anonymous) reviewers: their comments were a great help in improving this document. We thank Nelson Beebe for helpful comments on a preliminary version of this paper. This work was partly supported by the NuSCAP (ANR-20-CE48-0014) project and the InterFLOP (ANR-20-CE46-0009) project of the French National Agency for Research (ANR).

References

1. IEEE standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008), IEEE, pages 1–84 (2019)

2. Bohlender, G., Walter, W., Kornerup, P., Matula, D.W.: Semantics for exact floating point operations. In 10th IEEE Symposium on Computer Arithmetic, pages 22–26 (1991)
3. Boldo, S., Muller, J.-M.: Some functions computable with a fused-mac. In 17th IEEE Symposium on Computer Arithmetic (ARITH-17), Cape Cod, MA, USA (2005)
4. Boldo, Sylvie., Daumas, Marc.: Representable correcting terms for possibly underflowing floating point operations. In 16th IEEE Symposium on Computer Arithmetic (ARITH-16), pages 79–86, Santiago de Compostela, Spain (2003)
5. Boldo, S., Jeannerod, C.-P., Melquiond, G., Muller, J.-M.: Floating-point arithmetic. Acta Numerica **32**, 203–290 (2023)
6. Boldo, S., Melquiond, G.: Emulation of FMA and correctly rounded sums: proved algorithms using rounding to odd. IEEE Trans. Comput. **57**(4), 462–471 (2008)
7. Boldo, S., Muller, J.-M.: Exact and approximated error of the FMA. IEEE Trans. Comput. **60**(2), 157–164 (2011)
8. Cocke, John, Markstein, V.: The evolution of RISC technology at IBM. IBM Journal of Research and Development **34**(1), 4–11 (1990)
9. Cornea-Hasegan, M.A., Golliver, R.A., Markstein, P.: Correctness proofs outline for Newton–Raphson based floating-point divide and square root algorithms. In 14th IEEE Symposium on Computer Arithmetic (ARITH-14), pages 96–105 (1999)
10. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. **18**(3), 224–242 (1971)
11. Fabiano, N., Muller, J.-M., Picot, J.: Algorithms for triple-word arithmetic. IEEE Trans. Comput. **68**(11), 1573–1583 (2019)
12. Fasi, M., Higham, N., Mikaitis, M., Pranesh, S.: Numerical behavior of NVIDIA tensor cores. Peer J. Computer Science, 7(e330) 2021
13. Graillat, S., Langlois, P., Louvet, N.: Improving the compensated horner scheme with a fused multiply and add. In Proceedings of the 2006 ACM Symposium on Applied Computing, SAC '06, page 1323–1327, New York, NY, USA, 2006. Association for Computing Machinery
14. Joldeş, M., Muller, J.-M., Popescu, V.: Tight and rigorous error bounds for basic building blocks of double-word arithmetic. ACM Transactions on Mathematical Software, 44(2) (2017)
15. Kahan, W.: Pracniques: further remarks on reducing truncation errors. Commun. ACM **8**(1), 40 (1965)
16. Knuth, D.E.: The Art of Computer Programming. Addison-Wesley, vol. 2, 3rd edn. Reading, MA (1998)
17. Kornerup, Peter, Lefèvre, V., Louvet, N., Muller, J.-M.: On the computation of correctly-rounded sums. IEEE Transactions on Computers **61**(2), 289–298 (2012)
18. Lauter, C.: An efficient software implementation of correctly rounded operations extending FMA: $a + b + c$ and $a \times b + c \times d$. In ACSSC Proc. (2017)
19. Lindholm, T., Yellin, F., Bracha, G., Buckley, A., Smith, D.: The Java Virtual Machine Specification, Java SE 23 Edition. Oracle America, Inc.l (2024). Available at https://docs.oracle.com/javase/specs/jvms/se23/jvms23.pdf
20. Louvet, N.: Algorithmes Compensés en Arithmétique Flottante: Précision, Validation, Performances. PhD thesis, Université de Perpignan, Perpignan, France (2007)
21. Markstein, P.: Computation of elementary functions on the IBM RISC System/6000 processor. IBM J. Res. Dev. **34**(1), 111–119 (1990)
22. Mittal, S.: A survey of techniques for dynamic branch prediction. Concurrency and Computation: Practice and Experience **31**(1), e4666 (2019)
23. Møller, O.: Quasi double-precision in floating-point addition. BIT **5**, 37–50 (1965)
24. Muller, J.-M.: Elementary Functions, 3rd edn. Algorithms and Implementation. Birkhäuser Boston, MA (2016)
25. Muller, J.-M., Brunie, N., de Dinechin, F., Jeannerod, C.-P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., Torres, S.: Handbook of Floating-Point Arithmetic, 2nd edition. Birkhäuser Boston (2018)

26. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. **26**(6), 1955–1988 (2005)
27. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation part I: Faithful rounding. SIAM J. Sci. Comput. **31**(1), 189–224 (2008)
28. Sterbenz, P.H.: Floating-Point Computation. Prentice-Hall, Englewood Cliffs, NJ (1974)