On various ways to split a floating-point number

Claude-Pierre Jeannerod, Jean-Michel Muller, Paul Zimmermann

On various ways to split a floating-point number

Claude-Pierre Jeannerod*, Jean-Michel Muller†, Paul Zimmermann‡
* Univ Lyon, Inria, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP UMR 5668, F-69007 Lyon, France
† Univ Lyon, CNRS, ENS de Lyon, Inria, Université Claude Bernard Lyon 1, LIP UMR 5668, F-69007 Lyon, France
‡ Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

***Abstract*—We review several ways to split a floating-point number, that is, to decompose it into the exact sum of two floating-point numbers of smaller precision. All the methods considered here involve only a few IEEE floating-point operations, with rounding to nearest and including possibly the fused multiply-add (FMA). Applications range from the implementation of integer functions such as `round` and `floor` to the computation of suitable scaling factors aimed, for example, at avoiding spurious underflows and overflows when implementing functions such as the hypotenuse.**

I. Introduction

Splitting a floating-point number (that is, separating a precision-p floating-point number x into two floating-point numbers x_h and x_ℓ of smaller significand size, and such that $x = x_h + x_\ell$) has several interesting applications in floating-point arithmetic. Among them (the examples below are in radix 2 arithmetic):

- it can be used to implement various integer functions on floating-point numbers: for example, functions such as $\texttt{round}(x)$ (round x to a nearest integer) and $\lfloor x/2^k \rfloor$ are special cases of splittings;
- it can be used for accurately computing sums of several floating-point numbers. The main idea is to split these numbers so that the obtained values can be affected to containers (also called bins), such that all floating-point numbers in the same bin have significand of at most b bits and have their least significant bit of a fixed weight (that depends on the bin). This allows for error-free accumulation of all numbers of the bin provided that we do not add more than 2^{p-b} numbers to one bin. This method seems to appear first in a Matlab program in [1]. It is presented in detail and analyzed in [2];
- it can be used for computing the error of a floating-point multiplication on architectures that do not offer an efficient FMA (fused multiply-add). The underlying idea is to split each of the operands into two $\lfloor p/2 \rfloor$-bit numbers, so the resulting numbers can be multiplied together without error. The first presentation of this method goes back to [3], and the splitting it uses is attributed to Veltkamp;
- it can be used for computing a power of 2 close to $|x|$. This is useful for *scaling* some calculations. Consider for instance the evaluation of $\sqrt{a^2+b^2}$ in floating-point arithmetic. The computation of a^2 or b^2 (or their sum) may underflow or overflow, resulting in a very inaccurate or infinite final result, even when the exact value of $\sqrt{a^2+b^2}$ is far from the underflow and overflow thresholds. A frequently suggested way of dealing with the problem is to divide both operands by the scaling factor $\max\{|a|,|b|\}$, which gives new values a' and b', to compute $\sqrt{a'^2+b'^2}$, and to multiply the obtained value by the same scaling factor. Unfortunately, dividing and multiplying by the scaling factor is in general not exact. A way to avoid this is to choose as scaling factor an integer power of 2 close to $\max\{|a|,|b|\}$.

In the first two examples, we wish to split numbers around a constant, that is, we want x_h to be multiple of some constant (1 in the case of function `round`), and $|x_\ell|$ to be less than this constant (or less than half this constant). We will call these splittings *absolute splittings*. In the last two examples, we wish to split numbers around a value that is somehow proportional to $|x|$ (or to $\text{ulp}(x)$ or $\text{ufp}(x)$—the definition of these functions is recalled at the end of this section). We will call these splittings *relative splittings*. In some languages, these splittings could be done using bit manipulations of the machine representations of the floating-point operands, but this would result in less portable programs.

The main contributions of this paper can be summarized as follows: we propose a new algorithm for computing $\texttt{floor}(x)$ with rounding to nearest mode (Algorithm 2), a new FMA-based relative splitting with a smaller number of flops and latency than Veltkamp's splitting (Algorithm 4), and a survey of different algorithms to compute $\text{sign}(x)\cdot\text{ulp}(x)$ or $\text{sign}(x)\cdot\text{ufp}(x)$ up to a factor 2, some of which are underflow-safe and almost overflow-safe.

Interestingly, most of the algorithms presented in this paper have in common a sequence of two floating-point operations of the form "add some number z to the input, and then subtract z from the obtained sum."

In the following, we assume an IEEE-754 compliant floating-point arithmetic with radix β, precision p, and extremal exponents $e_{\min}$ and $e_{\max}$. (In some cases we will only consider the case $\beta = 2$.) We denote by $\mathbb{F}$ the set of floating-point numbers in such an arithmetic. An element x of $\mathbb{F}$ can be written

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

where M_x and e_x are two integers such that $|M_x| < \beta^p$ and $e_{\min} \leqslant e_x \leqslant e_{\max}$, with $|M_x|$ maximum under these constraints. The *significand* of x is the number $M_x \cdot \beta^{-p+1}$. A nonzero element x of $\mathbb{F}$ is *normal* if $|x| \geqslant \beta^{e_{\min}}$, and

subnormal otherwise. The absolute value of the significand of a normal floating-point number is at least 1 and less than β.

We will write RN to denote rounding to nearest for some given tie-breaking rule (assumed to be either "to even" or "to away", as in IEEE 754-2008). Hence, for $t \in \mathbb{R}$ and in the absence of overflow, $\mathrm{RN}(t)$ satisfies $|t - \mathrm{RN}(t)| \leqslant |t - x|$ for all $x \in \mathbb{F}$. We also write $\mathrm{succ}(t)$ to denote the successor in $\mathbb{F}$ of t (with the implicit assumption that if t is at least the largest element of $\mathbb{F}$, then its successor should be $+\infty$).

Before presenting splitting algorithms, let us recall some definitions of the functions ulp and ufp.

Definition I.1 (classical ulp)**.** *The* unit in the last place *of* $t \in \mathbb{R}$ *is*

$$\mathrm{ulp}(t) = \begin{cases} \beta^{\lfloor \log_\beta |t| \rfloor - p + 1} & \text{if } |t| \geqslant \beta^{e_{\min}}, \\ \beta^{e_{\min} - p + 1} & \text{otherwise.} \end{cases}$$

There are in fact some alternative definitions of ulp, which differ only around the powers of the radix and satisfy slightly different properties. In particular, Harrison's definition [4] is as follows (and, to avoid confusion, we write ulp_H).

Definition I.2 (Harrison's ulp)**.** *Let* $t \in \mathbb{R}$*. The number* $\mathrm{ulp}_H(t)$ *is the distance between the two closest straddling floating-point numbers* x *and* y*, so that* $x \leqslant t \leqslant y$ *and* $x \neq y$*, assuming an unbounded exponent range.*

One easily checks that in radix-2, precision-p arithmetic, if $|t|$ is not a power of 2, then $\mathrm{ulp}(t) = \mathrm{ulp}_H(t)$, and if $|t| = 2^k$, then $\mathrm{ulp}(t) = 2^{k-p+1} = 2\mathrm{ulp}_H(t)$, except in the subnormal range where $\mathrm{ulp}(t) = \mathrm{ulp}_H(t)$.

A related notion is that of ufp (unit in the first place), introduced by Rump, Ogita, and Oishi in [2] and defined as follows.

Definition I.3 (ufp)**.** *The* unit in the first place *of* $t \in \mathbb{R}$ *is*

$$\mathrm{ufp}(t) = \begin{cases} \beta^{\lfloor \log_\beta |t| \rfloor} & \text{if } t \neq 0, \\ 0 & \text{if } t = 0. \end{cases}$$

II. Absolute splittings

The first algorithm we mention splits a floating-point number x into x_h, which is x rounded to a nearest integer, and $x_\ell = x - x_h$:

Require: $C, x \in \mathbb{F}$
$s \leftarrow \mathrm{RN}(C + x)$
$x_h \leftarrow \mathrm{RN}(s - C)$
$x_\ell \leftarrow \mathrm{RN}(x - x_h)$ {optional}
return x_h {or (x_h, x_ℓ)}

Algorithm 1: Rounding to a nearest integer, under conditions specified by Theorem II.1. The input value is x, and the algorithm uses a constant C.

The first occurrence we could find of this algorithm is in [5, p. 22], in radix 2 and with a constant $C = 2^{p-1}$ or $C = 2^{p-1} + 2^{p-2}$; the latter constant aims at accommodating negative values of x and its use is referred to as "the 1.5 trick." A similar example is the code optimization guide of the AMD Athlon processor [6, p. 43], where the so-called "magic number" 6755399441055744, equal to $2^{52} + 2^{51}$, is used to convert double-precision numbers to integers. It is also noted in [7, p. 5] that taking $C = 2^{52}$ yields a nearest integer function for double-precision numbers $x \in [0, 2^{48})$. In [2], this algorithm is introduced (under the name ExtractScalar), still in radix 2 arithmetic, with the constant $C = 2^k$. In particular, it is shown in [2] that if $|x| \leqslant 2^k$, then x_h is a multiple of 2^{k-p} (but needs not be a nearest integer when x is negative) and $|x_\ell| \leqslant 2^{k-p}$.

Of course, Algorithm 1 is nothing more than the so-called Fast2Sum algorithm applied to C and x, but with C considered as a given constant. As noted for example by Linnainmaa [8], the sequence of operations of Fast2Sum already appears in early papers by Kahan [9] (for compensated summation) and Møller [10] (with truncation instead of rounding to nearest) and was analyzed by Dekker [3], who proved that x_ℓ is exactly the error of the addition of C and x when $|C| \geqslant |x|$ and $\beta \leqslant 3$.

Theorem II.1. *Assume* p*-digit floating-point numbers in radix* β*, and* C *integer with* $\beta^{p-1} \leqslant C \leqslant \beta^p$*. If*

$$\beta^{p-1} - C \leqslant x \leqslant \beta^p - C, \tag{1}$$

then the floating-point number x_h *returned by Algorithm 1 is an integer such that* $|x - x_h| \leqslant 1/2$ *(that is,* x_h *is equal to* x *rounded to a nearest integer). Furthermore,* $x = x_h + x_\ell$*.*

Proof. Adding C to Eq. (1) yields

$$\beta^{p-1} \leqslant C + x \leqslant \beta^p,$$

thus $s = \mathrm{RN}(C + x)$ lies in $[\beta^{p-1}, \beta^p]$, which implies s is an integer and $|s - (C + x)| \leqslant 1/2$. Now since C also lies in $[\beta^{p-1}, \beta^p]$, $s - C$ is exactly representable in precision p and thus $x_h = s - C$ is an integer: $|x_h - x| = |(s - C) - x| \leqslant 1/2$.

From $x_h = s - C$ and $s = \mathrm{RN}(C + x)$, we also deduce that $x - x_h$ equals $C + x - \mathrm{RN}(C + x)$, that is, the error of a floating-point addition. Hence $x - x_h$ must be a floating-point number (see for example [11, p. 236]) and we conclude that $x_\ell = x - x_h$. □

Note that Theorem II.1 scales in the obvious way: if $\beta^{k+p-1} \leqslant C \leqslant \beta^{k+p}$ and $\beta^{k+p-1} - C \leqslant x \leqslant \beta^{k+p} - C$, then x_h is a multiple of β^k, and $|x_\ell| = |x - x_h| \leqslant \frac{1}{2}\beta^k$. It is this property that is used in [2] to "extract" parts of floating-point numbers that can be accumulated without error.

When x is nonnegative, the largest interval for x is attained for C minimum, i.e., $C = \beta^{p-1}$, and then the algorithm works for $x \leqslant \beta^p - \beta^{p-1}$. When x can be of either sign, the largest centered interval for x is attained when $\beta^{p-1} - C$ and $\beta^p - C$ are opposite numbers, that is, $C = (\beta^p + \beta^{p-1})/2$, assuming β is even; then the algorithm works for $|x| \leqslant (\beta^{p-1} - \beta^p)/2$.

In particular, for $\beta = 2$ and $p = 53$, this yields the "magic number" $C = 2^{52} + 2^{51}$ mentioned before, together with the range $|x| \leqslant 2^{51}$. Also, for $\beta = 10$ and $p = 16$ (decimal64 IEEE-754 format), the largest range for nonnegative inputs is $x \leqslant 9 \cdot 10^{15}$, obtained with $C = 10^{15}$; and the largest range for signed inputs is $|x| \leqslant 4.5 \cdot 10^{15}$, obtained with $C = 5.5 \cdot 10^{15}$.

A. Computing $\texttt{floor}(x)$

An interesting question is to compute $\lfloor x \rfloor$, or more generally $\lfloor x/2^k \rfloor$, for x a floating-point number. If the rounding mode is to nearest, how can we compute $\lfloor x \rfloor$ efficiently?

Require: $x \in \mathbb{F}$
$y \leftarrow \mathrm{RN}(x - 0.5)$
$C \leftarrow \mathrm{RN}(\beta^p - x)$
$s \leftarrow \mathrm{RN}(C + y)$
$x_h \leftarrow \mathrm{RN}(s - C)$
return x_h

Algorithm 2: Computing $\texttt{floor}(x)$.

Theorem II.2. *Assume β is even, $x \in \mathbb{F}$, $0 \leqslant x \leqslant \beta^{p-1}$. Then Algorithm 2 returns $x_h = \lfloor x \rfloor$.*

Proof. We provide a detailed proof only for "ties to even", since similar arguments apply in the case where RN breaks ties "to away". Since β is even, 0.5 is in $\mathbb{F}$. If $0 \leqslant x < 0.5$, then $-0.5 \leqslant y \leqslant 0$, $C = \beta^p$, thus $s = \beta^p$, and $x_h = 0 = \lfloor x \rfloor$. Now assume $0.5 \leqslant x \leqslant \beta^{p-1}$. In this case, it can be shown that $x - 0.5$ is in $\mathbb{F}$, so that $y = x - 1/2$. Write $x = k + \varepsilon$ with k integer and $0 \leqslant \varepsilon < 1$; C is $\beta^p - k$ if $\varepsilon < 1/2$ or $\varepsilon = 1/2$ and k even, and $\beta^p - (k+1)$ otherwise. In the case $\varepsilon < 1/2$ or $\varepsilon = 1/2$ and k even, we have $s = \mathrm{RN}(\beta^p + \varepsilon - 0.5) = \beta^p$, thus $x_h = k$. In the case $\varepsilon > 1/2$ or $\varepsilon = 1/2$ and k odd, we have $s = \mathrm{RN}(\beta^p + \varepsilon - 1.5) = \beta^p - 1$, thus $x_h = \mathrm{RN}((\beta^p - 1) - (\beta^p - (k+1))) = k$. □

III. Relative splittings

We are now interested in expressing a precision-p floating-point number x as the exact sum of a $(p-s)$-digit number x_h and an s-digit number x_ℓ. The first use of such a splitting in the literature seems to be with $s = \lfloor p/2 \rfloor$: Dekker [3] used it to express the exact product of two floating-point numbers as a double-word. Another use is with $s = p - 1$, so that x_h is a power of β giving the order of magnitude of x. This can be desirable for at least two reasons:

- One may want to evaluate $\mathrm{ulp}(x)$ or $\mathrm{ufp}(x)$. These functions are very useful in the error analysis of floating-point algorithms. In that case, what we want to compute is fully determined (for example, for x nonzero, $\mathrm{ufp}(x)$ is the greatest integer power of β less than or equal to $|x|$).
- One may just want to obtain an integer power of β close to $|x|$. Indeed, for scaling x, such a weaker condition will in general be enough, and one may hope that it can be satisfied using fewer operations than those needed to compute $\mathrm{ufp}(x)$.

A. Veltkamp's splitting and a new variant

We use here the description of Veltkamp's splitting from [12]. Given a floating-point number x of precision p in radix β, and an integer $s < p$, the following algorithm splits x into two non-overlapping floating-point numbers x_h and x_ℓ such that $x = x_h + x_\ell$, with the significand of x_h fitting in $p - s$ digits, and the one of x_ℓ in s digits (or even $s - 1$ when $\beta = 2$ and $s \geqslant 2$).

Require: $C = \beta^s + 1$ and x in $\mathbb{F}$
$\gamma \leftarrow \mathrm{RN}(Cx)$
$\delta \leftarrow \mathrm{RN}(x - \gamma)$
$x_h \leftarrow \mathrm{RN}(\gamma + \delta)$
$x_\ell \leftarrow \mathrm{RN}(x - x_h)$
return (x_h, x_ℓ)

Algorithm 3: Veltkamp's splitting.

Dekker [3] analyzes this algorithm in radix 2, with the implicit assumption that no overflow occurs. This analysis was extended to any radix β by Linnainmaa in [12]. Note that the algorithm works correctly even in the presence of underflows (due to the special shape of C and since underflowing additions are exact, as noted for example in [13]). Moreover, Boldo [14] shows for any radix β that if Cx does not overflow, then no other operation will overflow.

If an FMA instruction is available, we suggest to use the following variant, that requires fewer operations.

Require: $C = \beta^s + 1$ and x in $\mathbb{F}$
$\gamma \leftarrow \mathrm{RN}(Cx)$
$x_h \leftarrow \mathrm{RN}(\gamma - \beta^s x)$
$x_\ell \leftarrow \mathrm{RN}(x - x_h)$ {or $x_\ell \leftarrow \mathrm{RN}(Cx - \gamma)$}
return (x_h, x_ℓ)

Algorithm 4: FMA-based relative splitting.

Theorem III.1. *Let $x \in \mathbb{F}$ and $s \in \mathbb{Z}$ such that $1 \leqslant s < p$. Then, barring underflow and overflow, Algorithm 4 computes $x_h, x_\ell \in \mathbb{F}$ such that $x = x_h + x_\ell$ and, if $\beta = 2$, the significands of x_h and x_ℓ have at most $p - s$ and s bits, respectively. If $\beta > 2$ then their significands have at most $p - s + 1$ and $s + 1$ digits, respectively.*

Proof. The result is clear for $x = 0$ and, for x nonzero, we can restrict with no loss of generality to $1 \leqslant x < \beta$. For $s \geqslant 0$, this implies $\beta^s \leqslant Cx < \beta^{s+1} + \beta \leqslant \beta^{s+2}$, so $\mathrm{ulp}(Cx) \geqslant \beta^{s-p+1}$ and thus $\gamma = \mathrm{RN}(Cx)$ is a multiple of β^{s-p+1}. Since $x \geqslant 1$, $\beta^s x$ is also a multiple of β^{s-p+1}, and thus

$$\gamma - \beta^s x = M \cdot \beta^{s-p+1}$$

for some integer M. Now, let ϵ be such that $\gamma = Cx + \epsilon$. We have $|\epsilon| \leqslant \frac{1}{2}\beta^{s-p+2}$ since $\mathrm{ulp}(Cx) \leqslant \beta^{s-p+2}$, and, on the other hand,

$$\gamma - \beta^s x = x + \epsilon.$$

Therefore, recalling that $|x| < \beta$,

$$|M| \leqslant \frac{|x| + |\epsilon|}{\beta^{s-p+1}} < \beta^{p-s} + \tfrac{1}{2}\beta. \tag{2}$$

This implies $|M| \leqslant \beta^p$ and that $\gamma - \beta^s x$ is in $\mathbb{F}$. Consequently, $x_h = x + \epsilon$ and $x_\ell = \mathrm{RN}(-\epsilon)$. Since x_h is a multiple of β^{s-p+1} and x is a multiple of β^{1-p}, their difference ϵ is also

a multiple of β^{1-p} and, using the bound on $|\epsilon|$ seen above, there exists an integer E such that

$$\epsilon = E \cdot \beta^{1-p}, \quad |E| \leqslant \tfrac{1}{2}\beta^{s+1}. \tag{3}$$

Hence $|E| \leqslant \beta^p$ and thus $\epsilon \in \mathbb{F}$ and $x_\ell = x - x_h$, as wanted.

From (2) and (3) we deduce that $|M| \leqslant \beta^{p-s+1}$ and $|E| \leqslant \beta^{s+1}$, which means that the significands of x_h and x_ℓ fit into $p-s+1$ and $s+1$ digits, respectively. If $\beta = 2$ then the strict inequality in (2) implies $|M| \leqslant 2^{p-s}$, and (3) gives $|E| \leqslant 2^s$. Hence $p-s$ and s bits suffice to represent the significands of x_h and x_ℓ, respectively, in this case.

Finally, note that since $x_h = x + \epsilon$ and $\gamma = Cx + \epsilon$, we have

$$x - x_h = Cx - \gamma,$$

so the suggested variant for x_ℓ can be used as well. □

We conclude this section with several remarks about Algorithm 4:

- If an efficient FMA is available, it can be used for the computation of x_h, and Algorithm 4 takes only 3 flops instead of 4 for Algorithm 3. In fact, if for x_ℓ we use the variant
$$x_\ell \leftarrow \mathrm{RN}(Cx - \gamma),$$
then x_ℓ can be obtained simultaneously with x_h, and this reduces the depth (or latency) of the relative splitting from 4 to 2.
- Even if an efficient FMA is not available, the two floating-point numbers γ and $\beta^s x$ can be computed in parallel, and so, again, the depth of Veltkamp's splitting is reduced, but only from 4 to 3.
- If $\beta > 2$, the bounds $p-s+1$ and $s+1$ on the numbers of digits needed to represent the significands of x_h and x_ℓ can be attained. For example, if $x = \beta - \beta^{1-p}$, then it is easy to check that $\gamma = \beta^{s+1} + \beta$ and
$$x_h = (1 + \beta^{s-p}) \cdot \beta, \qquad x_\ell = -(\beta^s + 1) \cdot \beta^{1-p}.$$
- If $\beta = 2$, the bounds $p-s$ and s specific to this case are attainable as well. To show this, we can consider the same example as before, namely $x = 2 - 2^{1-p}$, and check that we then have $\gamma = (1 + 2^{-s} - 2^{1-p}) \cdot 2^{s+1}$ and
$$x_h = (2^{p-s} - 1) \cdot 2^{s-p+1}, \qquad x_\ell = (2^s - 1) \cdot 2^{1-p}.$$

B. A special case: extracting one bit only

Assuming radix 2, let us now consider the problem of computing $\mathrm{ufp}(x)$ or $\mathrm{ulp}(x)$, and more generally the question of *scaling* x, that is, deducing from x an integer power of 2 that is "close" to $|x|$.

For scaling, a possible solution is to determine a nearest power of two. This can be done by applying Veltkamp's splitting (Algorithm 3) to x with $s = p - 1$: in this case, the resulting x_h has a 1-bit significand and we know from [14, Theorem 1] that it is nearest x in precision $p - s = 1$.

For computing $\mathrm{sign}(x) \cdot \mathrm{ufp}(x)$, we can use the following algorithm, introduced by Rump in [15].

Require: $\beta = 2$, $\varphi = 2^{p-1} + 1$, $\psi = 1 - 2^{-p}$, and $x \in \mathbb{F}$
$q \leftarrow \mathrm{RN}(\varphi x)$
$r \leftarrow \mathrm{RN}(\psi q)$
$\delta \leftarrow \mathrm{RN}(q - r)$
return δ

Algorithm 5: Computing $\mathrm{sign}(x) \cdot \mathrm{ufp}(x)$ for radix 2, as in [15, Algorithm 3.5].

These solutions, however, raise the following issues.

- If $|x|$ is large, then an overflow can occur in the first line of both Algorithms 3 and 5. It is always possible to slightly change Algorithm 5, by scaling it, in order to avoid overflow: it suffices for instance to replace φ by $\frac{1}{2}+2^{-p}$ and to return $2^p\delta$ at the end. However, this variant will not work for subnormal x. Thus, to use Algorithm 5, we somehow need to check the order of magnitude of x.
- If we are only interested in scaling x, then requiring to produce the exact value of $\mathrm{ufp}(x)$ may seem like overkill: maybe one can get sometimes $\mathrm{ufp}(x)$ and sometimes $\frac{1}{2}\mathrm{ufp}(x)$ or $2\mathrm{ufp}(x)$ (or any other convenient value) with a cheaper algorithm.

Based on these remarks, we suggest considering the following algorithms as well. (Recall that ulp_H refers to Harrison's ulp as in Definition I.2.)

Require: $\beta = 2$, $\psi = 1 - 2^{-p}$, and $x \in \mathbb{F}$
$a \leftarrow \mathrm{RN}(\psi x)$
$\delta \leftarrow \mathrm{RN}(x - a)$
return δ

Algorithm 6: Computing $\mathrm{sign}(x) \cdot \mathrm{ulp}_H(x)$ for radix 2 and $|x|$ "large enough" (namely, $|x| > 2^{e_{\min}}$).

Require: $\beta = 2$, $\psi^* = 2^p - 1$, and $x \in \mathbb{F}$
$a \leftarrow \mathrm{RN}(\psi^* x)$
$\delta \leftarrow \mathrm{RN}(2^p x - a)$
return δ

Algorithm 7: Computing $\mathrm{sign}(x) \cdot \mathrm{ufp}(x)$ up to a factor 2, for radix 2 and $|x|$ "small enough" (namely, $|x| < 2^{e_{\max}-p+1}$).

Theorem III.2. *If $|x| > 2^{e_{\min}}$, then Algorithm 6 returns*

$$\mathrm{sign}(x) \cdot \mathrm{ulp}_H(x) = \mathrm{sign}(x) \cdot \begin{cases} \frac{1}{2}\mathrm{ulp}(x) & \text{if } |x| \text{ is a power of 2,} \\ \mathrm{ulp}(x) & \text{otherwise.} \end{cases}$$

If $|x| < 2^{e_{\max}-p+1}$, then Algorithm 7 returns

$$\mathrm{sign}(x) \cdot \begin{cases} \mathrm{ufp}(x) & \text{if } |x| \text{ is a power of 2,} \\ 2\mathrm{ufp}(x) & \text{otherwise.} \end{cases}$$

Proof. The first claim is a direct consequence of [15, Lemma 3.6]. The second claim then follows from the fact that Algorithm 7 is Algorithm 6 applied to $2^p x$. □

Again, let us conclude with several comments.

- Algorithm 6 uses only 2 floating-point operations, and if an FMA is available, this also applies to Algorithm 7. In the absence of an FMA, 3 operations are needed for Algorithm 7, similarly to Algorithm 5.
- Both Algorithms 4 and 7 are based on a sequence of operations of the form

$$\mathrm{RN}\Big(\mathrm{RN}\big((C+1)x\big) - \mathrm{RN}(Cx)\Big)$$

for some prescribed constant C. Specifically, $C = \beta^s$ in Algorithm 4 and $C = 2^p - 1$ in Algorithm 7. Note also that Algorithm 3.6 in [2] is quite similar to Algorithm 7, except that it uses the constant $C = 2^p$ instead of $C = 2^p - 1$, which requires a special treatment when x is a power of 2.
- Note finally that by Theorem III.2, the output of Algorithm 7 is equal to x if and only if $|x|$ is a power of two. Algorithm 7 does not straightforwardly generalize to higher radices: if one sets ψ^* to $\beta^p - 1$ and if δ becomes $\mathrm{RN}(\beta^p x - a)$, then we do not always have δ equal to a power of β close to x. However, we still have the property that $\delta = x$ if and only if x is a power of the radix. Hence Algorithm 7 provides a simple way to check this property using only floating-point operations. More precisely, assuming $\beta^k \leqslant x \leqslant \beta^{k+1} - \beta^{k-p+1}$:
  1) if $x = \beta^k$ then $\delta = x$;
  2) if $\beta^k + \beta^{k-p+1} \leqslant x < \frac{\beta}{2} \cdot \beta^k$ then $\mathrm{ulp}(\psi^* x) = \beta^{k+1}$ and $|\psi^* x - \beta^p x| = |x| < \frac{1}{2}\beta^{k+1}$, so that $\mathrm{RN}(\psi^* x) = \beta^p x$ and $\delta = 0$;
  3) if $x > \frac{\beta}{2} \cdot \beta^k$, we still have $\mathrm{ulp}(\psi^* x) = \beta^{k+1}$, and we deduce that

$$\begin{aligned}\beta^p x - \mathrm{ulp}(\beta^p x) \leqslant \psi^* x = \beta^p x - x \\ < \beta^p x - \frac{1}{2}\mathrm{ulp}(\beta^p x);\end{aligned}$$

  therefore, $\mathrm{RN}(\psi^* x) = \beta^p x - \beta^{k+1}$, so that $\delta = \beta^{k+1} > x$;
  4) the case $x = \frac{\beta}{2} \cdot \beta^k$ (which can occur only when the radix β is even) is similar to case 2) or case 3), depending on the way RN breaks ties.

C. Alternative algorithms for computing $\mathrm{sign}(x) \cdot \mathrm{ulp}(x)$ and $\mathrm{sign}(x) \cdot \mathrm{ufp}(x)$

If we replace the constant φ in Algorithm 5 by $\varphi/2^{p-1}$ and if we ignore the possibility of underflow and overflow, we obtain a way to compute $\mathrm{sign}(x) \cdot \mathrm{ulp}(x)$ which uses three floating-point operations.

An alternative method is given by Algorithm 8 below, which, if an FMA is available, uses only two floating-point operations. This method uses the same scheme as [16, Algorithm 2], outside the subnormal range, except that in [16] the constant $\psi = u(1+2u) = \mathrm{succ}(u)$ is used, whereas we use here $\frac{3}{2}u$, with $u = 2^{-p}$. We determined experimentally that, with or without FMA, $\psi = u(1+2u) = \mathrm{succ}(u)$ is the smallest constant that works, and $\psi = \mathrm{succ}(\frac{3}{2}u)$ is the largest one.

Require: $\beta = 2$, $\psi = 2^{-p} + 2^{-p-1}$, and $x \in \mathbb{F} \setminus \{0\}$
  $a \leftarrow \mathrm{RN}(x + \psi x)$ {or $\mathrm{RN}(x + \mathrm{RN}(\psi x))$ without FMA}
  $\delta \leftarrow \mathrm{RN}(a - x)$
  return δ

Algorithm 8: Computing $\mathrm{sign}(x) \cdot \mathrm{ulp}(x)$ for radix 2 and $x \neq 0$.

Theorem III.3. *For $x \neq 0$ and $p \geqslant 2$, if no underflow or overflow occurs, then Algorithm 8 returns $\mathrm{sign}(x) \cdot \mathrm{ulp}(x)$.*

Proof. For $x < 0$, it is clear that the algorithm returns the opposite of what it returns for $-x$. Let us now assume $x > 0$.

Without loss of generality we can assume $1 \leqslant x < 2$. With $u = \frac{1}{2}\mathrm{ulp}(1) = 2^{-p}$, note that $\psi = \frac{3}{2}u$.

For $x = 1$, $\psi x = \psi$ is exactly representable, thus with or without FMA, we get $a = \mathrm{RN}(1 + \psi) = 1 + 2u$, since the rounding boundaries for $1 + 2u$ are $1 + u$ and $1 + 3u$.

It remains to prove the theorem for $1 + 2u \leqslant x \leqslant 2 - 2u$. We then have

$$\frac{3}{2}u(1 + 2u) \leqslant \psi x \leqslant \frac{3}{2}u(2 - 2u).$$

It follows that $x + u < x + \psi x < x + 3u$, thus with an FMA, $x + \psi x$ is rounded to $x + 2u$, and thus $\delta = 2u$. Now consider the case without an FMA. On the left-hand side, since $\frac{3}{2}u$ is exactly representable, because $p \geqslant 2$, we have $\frac{3}{2}u \leqslant \mathrm{RN}(\psi x)$. On the right-hand side, $\frac{3}{2}u(2 - 2u)$ is strictly less than the midpoint $u(3 - 2u)$, so that $\mathrm{RN}(\psi x) \leqslant u(3 - 4u) < 3u$. Since $x + u < x + \mathrm{RN}(\psi x) < x + 3u$, $x + \mathrm{RN}(\psi x)$ is rounded to $x + 2u$, and $\delta = 2u$. □

To compute $\mathrm{sign}(x) \cdot \mathrm{ufp}(x)$, one can simply scale x by 2^{p-1} within Algorithm 8, which can be performed with 3 FMA instead of 2, but still with a depth of 2: we first compute $a = \mathrm{RN}(x + \psi x)$ and $y = \mathrm{RN}(2^{p-1} x)$ in parallel, and then $\mathrm{RN}(2^{p-1} a - y)$ yields $\mathrm{sign}(x) \cdot \mathrm{ufp}(x)$.

It does not seem possible to compute $\mathrm{sign}(x) \cdot \mathrm{ufp}(x)$ with only 2 FMA. However, perhaps surprisingly, 2 FMA suffice to obtain $\mathrm{sign}(x) \cdot 4\mathrm{ufp}(x)$:

$a \leftarrow \mathrm{RN}((2^{p+1} - 2) \cdot x - x)$
$\delta \leftarrow \mathrm{RN}(2^{p+1} \cdot x - a)$

Scaling this last operation by 2^{-p-1} yields the following variant of Algorithm 8 for computing $\mathrm{sign}(x) \cdot \mathrm{ulp}(x)$, still with 2 FMA:

$a \leftarrow \mathrm{RN}((2^{p+1} - 2) \cdot x - x)$
$\delta \leftarrow \mathrm{RN}(x - 2^{-p-1} \cdot a)$

IV. Underflow-safe and almost overflow-free scaling

In this section we assume that $\beta = 2$ and $p \geqslant 4$ (and we recall that RN breaks ties "to even" or "to away", as defined in IEEE 754-2008). Furthermore, we denote by η the smallest positive element of $\mathbb{F}$:

$$\eta = 2^{e_{\min} - p + 1}.$$

Given a nonzero floating-point number x, we want to compute a scaling factor δ that satisfies the following properties:

- $|x|/\delta$ is much above the underflow threshold and much below the overflow threshold (so that, for example, we can safely square it);
- δ is an integer power of 2 (in order to avoid rounding errors when multiplying or dividing by it).

The algorithms proposed in Section III-B are simple, but underflow or overflow can occur for many inputs x, leading to an output δ that does not satisfy the above properties. Let us now introduce an algorithm that is underflow-safe and almost overflow-free in the sense that only the two extreme values $x = \pm(2 - 2^{1-p}) \cdot 2^{e_{\max}}$ must be excluded.

The first three lines of Algorithm 9 mimic Algorithm 1 of [16], with slightly different notation.

Require: $\beta = 2$, $\Phi = 2^{-p} + 2^{-2p+1}$, $\eta = 2^{e_{\min}-p+1}$, and $x \in \mathbb{F}$.
$y \leftarrow |x|$
$e \leftarrow \mathrm{RN}(\Phi y + \eta)$
{or $e \leftarrow \mathrm{RN}(\mathrm{RN}(\Phi y) + \eta)$ without FMA}
$y_{sup} \leftarrow \mathrm{RN}(y + e)$
$\delta \leftarrow \mathrm{RN}(y_{sup} - y)$
return δ

Algorithm 9: Computing a scaling factor that is overflow-free and underflow-safe, for radix 2 and $|x| \neq (2 - 2^{1-p}) \cdot 2^{e_{\max}}$.

Below we recall a subcase of a result obtained by Rump et al. [16].

Theorem IV.1 (Subcase of [16, Theorem 2.2])**.** *For all $x \in \mathbb{F}$ such that $|x| \notin [2^{e_{\min}}, 2^{e_{\min}+2}]$, $y_{sup} = \mathrm{succ}(|x|)$.*

Theorem 2.2 in [16] was proven for e equal to $\mathrm{RN}(\mathrm{RN}(\Phi|x|) + \eta)$ (that is, for the "without FMA" version of Algorithm 9). However, one can easily check that the proof given in [16] still holds for $e = \mathrm{RN}(\Phi|x| + \eta)$ (the "with FMA" version). An immediate consequence of Theorem IV.1 is that if $|x| \notin [2^{e_{\min}}, 2^{e_{\min}+2}]$ and $|x| \neq (2 - 2^{1-p}) \cdot 2^{e_{\max}}$, then $\delta = \mathrm{ulp}(x)$.

Theorem 2.2 from [16] has been formally checked using the Coq automatic proof checker, which gives much confidence in this result. However, in [16] it is claimed further that if $|x| \in [2^{e_{\min}}, 2^{e_{\min}+2}]$ then y_{sup} is $\mathrm{succ}(\mathrm{succ}(|x|))$. As we are going to see, this property does not always hold. Let $y = |x|$ and consider the following cases:

1) If $2^{e_{\min}} \leqslant y \leqslant 2^{e_{\min}+1} - 2\eta$ then $\mathrm{ulp}(x) = \eta$. A simple calculation shows that
$$\frac{3}{2}\eta + 2^{e_{\min}-2p+1} \leqslant \Phi y + \eta \leqslant 2\eta - 2^{e_{\min}-3p+3},$$
so that $e = \mathrm{RN}(\Phi y + \eta) = 2\eta = 2\mathrm{ulp}(x)$. An immediate consequence is that $\mathrm{RN}(y + e) = y + e = \mathrm{succ}(\mathrm{succ}(|x|))$. Given the domain of x, we deduce that
$$\delta = 2\eta = 2\mathrm{ulp}(x);$$
2) If $y = 2^{e_{\min}+1} - \eta$ then $\Phi y + \eta = 2\eta + 2^{e_{\min}-2p+1} - 2^{e_{\min}-3p+2}$, so $e = \mathrm{RN}(\Phi y + \eta) = 2\eta = 2\mathrm{ulp}(x)$. Therefore $y + e$ is equal to the midpoint $2^{e_{\min}+1} + \eta$ and we have the following:
   - if RN is with "ties to even" then $\mathrm{RN}(y + e) = 2^{e_{\min}+1} = \mathrm{succ}(y)$, so
   $$\delta = \eta = \mathrm{ulp}(x);$$
   - if RN is with "ties to away" then $\mathrm{RN}(y + e) = 2^{e_{\min}+1} + 2\eta = \mathrm{succ}(\mathrm{succ}(y))$, so
   $$\delta = 3\eta = 3\mathrm{ulp}(x).$$
   In this case, δ is not a power of 2. As we will see this is the only case where this happens;
3) If $2^{e_{\min}+1} \leqslant y \leqslant 2^{e_{\min}+2} - 2\eta$ then $\mathrm{ulp}(x) = 2\eta$. A simple calculation shows that
$$2\eta + 2^{e_{\min}-2p+2} \leqslant \Phi y + \eta \leqslant 3\eta + 2^{e_{\min}-2p+2} - 2^{e_{\min}-3p+3}$$
so that $e = \mathrm{RN}(\Phi y + \eta) \in \{2\eta, 3\eta\}$ (and one easily checks that both cases actually do occur). When $e = 2\eta$, we obtain $y_{sup} = \mathrm{succ}(y)$ and
$$\delta = 2\eta = \mathrm{ulp}(x).$$
When $e = 3\eta$, depending on the tie-breaking rule and (for ties-to-even) on whether the last bit of y is a zero or a one, we obtain either $y_{sup} = \mathrm{succ}(y)$ and
$$\delta = 2\eta = \mathrm{ulp}(x),$$
or $y_{sup} = \mathrm{succ}(\mathrm{succ}(y))$ and
$$\delta = 4\eta = 2\mathrm{ulp}(x);$$
4) If $y = 2^{e_{\min}+2}$ then $\Phi y + \eta = 3\eta + 2^{e_{\min}-2p+3}$, so $e = \mathrm{RN}(\Phi y + \eta) = 3\eta$. It follows that $\mathrm{RN}(x + e) = x + 4\eta = \mathrm{succ}(x)$, for any tie-breaking rule, and thus
$$\delta = 4\eta = \mathrm{ulp}(x).$$

Consequently, depending on the tie-breaking rule, if $|x| \in [2^{e_{\min}}, 2^{e_{\min}+2}]$ then Algorithm 9 will return $\mathrm{ulp}(x)$ or $2\mathrm{ulp}(x)$ or, in one case only, $3\mathrm{ulp}(x)$. We deduce the following theorem.

Theorem IV.2. *For $x \in \mathbb{F}$ with $|x| \neq (2 - 2^{1-p}) \cdot 2^{e_{\max}}$, the value δ returned by Algorithm 9 satisfies the following:*
- *if* RN *is with "ties to even" then δ is a power of 2;*
- *if* RN *is with "ties to away" then δ is a power of 2, unless $|x| = 2^{e_{\min}+1} - 2^{e_{\min}-p+1}$, in which case it equals $3 \cdot 2^{e_{\min}-p+1}$;*
- *if $x \neq 0$, then*
$$1 \leqslant \left|\frac{x}{\delta}\right| \leqslant 2^p - 1.$$

Therefore, Algorithm 9 is a very good candidate for performing scalings.

Interestingly enough, in single precision/binary32 arithmetic ($e_{\min} = -126, p = 24$), the number $(2^{e_{\min}+1} - 2^{e_{\min}-p+1})/(3 \cdot 2^{e_{\min}-p+1})$ is a floating-point number, so that even for RN with "ties to away" and $|x| = 2^{e_{\min}+1} - 2^{e_{\min}-p+1}$, the calculation of x/δ is error free. This is not the case in double precision/binary64 arithmetic.

A. An example of application

Let us consider the use of Algorithm 9 for computing $\sqrt{a^2+b^2}$. Assume we wish to evaluate $\sqrt{a^2+b^2}$ without risking spurious overflows and underflows. Assume RN is with "ties to even" (so that the scaling factor provided by Algorithm 9 is, up to two extreme values, always a power of 2) and, in addition, that

$$\begin{cases} e_{\min} < -p, \\ (25/2)\cdot 2^{2p} \leqslant 2^{e_{\max}+1}, \end{cases} \tag{4}$$

which always holds with the usual binary floating-point formats.

In the following, we assume that once the scaling is performed, we just use the naive formula. (Although other evaluation schemes are possible, we do not deal with them in this paper, which focuses only on the scaling part).

Let

$$c = \mathrm{RN}\left(\mathrm{RN}\left|\frac{a}{2}\right| + \mathrm{RN}\left|\frac{b}{2}\right|\right).$$

The computation of c is obviously overflow-free. We will deal with possible loss of precision due to underflow later on.

Define δ_c as the value returned by Algorithm 9 with c as input. We suggest to use δ_c as scaling factor. As a consequence, since we use the naive formula with the scaled inputs, we actually compute

$$\begin{aligned} &\mathrm{RN}\left[\delta_c \times \mathrm{RN}\left(\sqrt{\mathrm{RN}\left(\mathrm{RN}\left(a'^2\right)+\mathrm{RN}\left(b'^2\right)\right)}\right)\right], \\ &\text{with} \quad a' = \mathrm{RN}\left(\tfrac{a}{\delta_c}\right) \quad \text{and} \quad b' = \mathrm{RN}\left(\tfrac{b}{\delta_c}\right). \end{aligned} \tag{5}$$

Since δ_c is a power of 2—we assumed RN is with "ties to even", in which case the situation where $\delta_c = 3\mathrm{ulp}(c)$ cannot happen—, multiplication and division by δ_c can induce a rounding error only when the result is a subnormal number. Let us quickly examine this scaling choice. Without loss of generality, we can assume that $a \geqslant 0$ and $a \geqslant |b|$. Note that if $a \geqslant 2^{e_{\min}}$ then $\mathrm{RN}(a/2) = a/2$, and that if $2\eta \leqslant a \leqslant 2^{e_{\min}}$, we have $|\mathrm{RN}(a/2) - a/2| \leqslant \eta/2$. From this we deduce that if $2\eta \leqslant a$ we always have

$$\frac{2}{5}a \leqslant \mathrm{RN}\left(\frac{a}{2}\right) \leqslant \frac{2}{3}a. \tag{6}$$

(The lower bound is attained for $a = 5\eta$, and the upper bound it attained for $a = 3\eta$.)

1) If $\mathrm{RN}(a/2) \neq 0$ (which implies $a \geqslant 2\eta$) then $\mathrm{RN}(a/2) \leqslant c \leqslant 2\mathrm{RN}(a/2)$, so that, using (6),

$$2a/5 \leqslant c \leqslant 4a/3.$$

Therefore, from Theorem IV.2,

$$\frac{3}{4} \leqslant \frac{3}{4}\cdot\frac{c}{\delta_c} \leqslant \frac{a}{\delta_c} \leqslant \frac{5}{2}\cdot\frac{c}{\delta_c} \leqslant \frac{5}{2}\cdot(2^p-1).$$

Hence, no underflow can result from the computation of $\mathrm{RN}(a/\delta_c)^2$ and, using (4), no overflow can result from the computation of $\mathrm{RN}(\mathrm{RN}(a/\delta_c)^2 + \mathrm{RN}(b/\delta_c)^2)$. If an underflow results from the computation of $(b/\delta_c)^2$ (i.e., if $|b|/\delta_c < 2^{e_{\min}/2}$) then one can show using (4) that $\mathrm{RN}(\mathrm{RN}(a/\delta_c)^2 + \mathrm{RN}(b/\delta_c)^2) = \mathrm{RN}(a/\delta_c)^2$, which implies that the calculation in (5) will return a (see for example [17]), and that this is a very accurate result. If no underflow occurs then the scaling induces no error. As a consequence, if $\mathrm{RN}(a/2) \neq 0$ the usual relative error bound 2^{-p+1} holds unless the final result is subnormal.[1]

2) If $\mathrm{RN}(a/2) = 0$ then a and b are 0 or $\pm\eta$, and $c = 0$. In such a case, we take $\delta_c = \eta$, which is the best possible splitting constant: the final result will be 0 if $a = b = 0$, and η otherwise, i.e., the correct rounding of $\sqrt{a^2+b^2}$.

V. Experimental results

In this section we compare our algorithms with bit manipulation algorithms, and with standard functions of the mathematical library, in the C language. Our experimental framework consists of an Intel i5-4590 processor running at 3.3GHz, under Debian testing (buster), with gcc 7.3.0, and optimization level `-O3`. We also set the FPU control word so that all roundings are done in double precision (and not in double-extended precision as by default). We first generate an array of 10^8 random binary64 numbers in the range $[0, 2^{32}]$, and we then compute the time to call each function on all numbers. The C programs we used are available at http://homepages.loria.fr/PZimmermann/papers/#split.

	`round`	`floor`
Algorithms 1 and 2	**0.106s**	**0.173s**
Bit manipulation	0.302s	0.203s
GNU libm	0.146s	0.209s

Fig. 1. Rounding to an integer. Note: we used the GNU libm `rint` function for rounding to nearest (which is faster than `nearbyint`), and `floor` for rounding towards $-\infty$.

	x_h	x_ℓ	time
Algorithm 3	26 bits	26 bits	0.108s
Algorithm 4	26 bits	27 bits	0.106s
Algorithm 4 with FMA	26 bits	27 bits	0.108s
Bit manipulation	26 bits	27 bits	**0.095s**

Fig. 2. Splitting a double-precision number into x_h and x_ℓ.

	time
Algorithm 5	0.107s
Algorithm 7	0.107s
Bit manipulation	**0.095s**

Fig. 3. Comparison of different ways of computing $\mathrm{sign}(x)\cdot\mathrm{ufp}(x)$ (possibly up to a factor 2 for Algorithm 7).

[1] If the final result is subnormal (i.e., if an underflow happens during the final multiplication by δ: it cannot happen before), then we need to add an additional absolute error $\eta/2$ due to the final rounding, so that we end up with an absolute error bound $3\eta/2$.

These results show that the algorithms we presented — in their respective domain of validity — are competitive with the less portable bit manipulation algorithms, and in some cases faster. Even, in some languages, for example in Javascript, there is no (efficient) way to manipulate the bits of a floating-point number, thus the user is left with no alternative.

VI. Conclusion and future work

We have presented several splitting algorithms that use floating-point arithmetic with rounding to nearest. Some have been part of the floating-point folklore for decades, and some are new. These algorithms have many applications: we show that, beyond the traditional contexts of extended-precision multiplication, accurate summation, and computing the ulp and ufp functions, floating-point splittings can also be used to implement some integer functions such as floor or round, as well as some scalings aimed at reducing the risk of spurious underflows and overflows. In radix-2 floating-point arithmetic, simple algorithms can evaluate $\mathrm{sign}(x) \cdot \mathrm{ulp}(x)$, $\mathrm{sign}(x) \cdot \mathrm{ulp}_H(x)$, and $\mathrm{sign}(x) \cdot \mathrm{ufp}(x)$, and also compute convenient scaling factors. However, generalizing this property to radix β (or, at least, to radix 10) does not seem to be straightforward and will have to be investigated in a future work.

Acknowledgment

We thank Pierrick Gaudry for the initial question that led to that work, and for comments on a first version of this paper. This work was partially supported by the French National Research Agency grants ANR-13-INSE-0007 (MetaLibm project) and ANR-14-CE25-0018-01 (FastRelax project).

References

[1] G. Zielke and V. Drygalla, "Genaue Lösung linearer Gleichungssysteme," *GAMM Mitt. Ges. Angew. Math. Mech.*, vol. 26, pp. 7–107, 2003.

[2] S. M. Rump, T. Ogita, and S. Oishi, "Accurate floating-point summation, Part I: Faithful rounding," *SIAM Journal on Scientific Computing*, vol. 31, no. 1, pp. 189–224, 2008. [Online]. Available: https://doi.org/10.1137/050645671

[3] T. J. Dekker, "A floating-point technique for extending the available precision," *Numer. Math.*, vol. 18, no. 3, pp. 224–242, 1971. [Online]. Available: https://doi.org/10.1007/BF01397083

[4] J. Harrison, "A machine-checked theory of floating point arithmetic," in *12th International Conference in Theorem Proving in Higher Order Logics (TPHOLs)*, ser. Lecture Notes in Computer Science, vol. 1690. Nice, France: Springer-Verlag, Berlin, Sep. 1999, pp. 113–130. [Online]. Available: https://doi.org/10.1007/3-540-48256-3_9

[5] C. Hecker, "Let's get to the (floating) point," *Game Developer Magazine*, vol. 2, pp. 19–24, 1996. [Online]. Available: http://chrishecker.com/images/f/fb/Gdmfp.pdf

[6] "AMD Athlon™ Processor: x86 Code Optimization Guide," Sep. 2000, revision 22007I-0. [Online]. Available: http://www.bartol.udel.edu/mri/sam/Athlon_code_optimization_guide.pdf

[7] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson, "ARPREC: An arbitrary precision computation package," Lawrence Berkeley National Laboratory, Berkeley, CA, Technical Report LBNL-53651, Sep. 2002. [Online]. Available: https://pubarchive.lbl.gov/islandora/object/ir%3A121949

[8] S. Linnainmaa, "Analysis of some known methods of improving the accuracy of floating-point sums," *BIT*, vol. 14, pp. 167–202, 1974. [Online]. Available: https://doi.org/10.1007/BF01932946

[9] W. Kahan, "Further remarks on reducing truncation errors," *Comm. ACM*, vol. 8, no. 1, p. 40, 1965. [Online]. Available: https://doi.org/10.1145/363707.363723

[10] O. Møller, "Quasi double-precision in floating point addition," *BIT*, vol. 5, pp. 37–50, 1965. [Online]. Available: https://doi.org/10.1007/BF01975722

[11] D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1998.

[12] S. Linnainmaa, "Software for doubled-precision floating-point computations," *ACM Trans. Math. Software*, vol. 7, no. 3, pp. 272–283, 1981. [Online]. Available: https://doi.org/10.1145/355958.355960

[13] J. R. Hauser, "Handling floating-point exceptions in numeric programs," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 2, pp. 139–174, 1996. [Online]. Available: https://doi.org/10.1145/227699.227701

[14] S. Boldo, "Pitfalls of a full floating-point proof: Example on the formal proof of the Veltkamp/Dekker algorithms," in *Proceedings of the Third International Joint Conference on Automated Reasoning*, ser. IJCAR'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 52–66. [Online]. Available: http://dx.doi.org/10.1007/11814771_6

[15] S. M. Rump, "Ultimately fast accurate summation," *SIAM J. Sci. Comput.*, vol. 31, no. 5, pp. 3466–3502, 2009. [Online]. Available: http://dx.doi.org/10.1137/080738490

[16] S. M. Rump, P. Zimmermann, S. Boldo, and G. Melquiond, "Computing predecessor and successor in rounding to nearest," *BIT*, vol. 49, no. 2, pp. 419–431, 2009. [Online]. Available: https://doi.org/10.1007/s10543-009-0218-z

[17] W. Kahan, "Mathematics written in sand—the HP-15C, Intel 8087, etc." in *Statistical Computing Section, Proceedings of the American Statistical Association, Toronto*, 1983, pp. 12–26. [Online]. Available: https://people.eecs.berkeley.edu/~wkahan/MathSand.pdf