Emulation of a FMA and correctly-rounded sums: proved algorithms using rounding to odd

Sylvie Boldo, Guillaume Melquiond

Emulation of FMA and correctly-rounded sums: proved algorithms using rounding to odd

Sylvie Boldo and Guillaume Melquiond

Abstract— **Rounding to odd is a non-standard rounding on floating-point numbers. By using it for some intermediate values instead of rounding to nearest, correctly rounded results can be obtained at the end of computations. We present an algorithm for emulating the fused multiply-and-add operator. We also present an iterative algorithm for computing the correctly rounded sum of a set floating-point numbers under mild assumptions. A variation on both previous algorithms is the correctly rounded sum of any three floating-point numbers. This leads to efficient implementations, even when this rounding is not available. In order to guarantee the correctness of these properties and algorithms, we formally proved them using the Coq proof checker.**

Index Terms— **Floating-point, rounding to odd, accurate summation, FMA, formal proof, Coq.**

I. Introduction

FLOATING-POINT computations and their roundings are described by the IEEE-754 standard [1], [2] followed by every modern general-purpose processor. This standard was written to ensure the coherence of the result of a computation whatever the environment. This is the "correct rounding" principle: the result of an operation is the same as if it was first computed with an infinite precision and then rounded to the precision of the destination format. There may exist higher precision formats though, and it would not be unreasonable for a processor to store all kinds of floating-point result in a single kind of register instead of having as many register sets as it supports floating-point formats. In order to ensure IEEE-754 conformance, care must then be taken that a result is not first rounded to the extended precision of the registers and then rounded to the precision of the destination format.

This "double rounding" phenomenon may happen on processors built around the Intel x86 instruction set for example. Indeed, their floating-point units use 80-bit long registers to store the results of their computations, while the most common format used to store in memory is only 64-bit long (IEEE double precision). To prevent double rounding, a control register allows to set the floating-point precision, so that the results are not first rounded to the register precision. Unfortunately, setting the target precision is a costly operation as it requires the processor pipeline to be flushed. Moreover, thanks to the extended precision, programs generally seem to produce more accurate results. As a consequence, compilers usually do not generate the additional code that would ensure that each computation is correctly rounded in its own precision.

Double rounding can however lead to unexpected inaccuracy. As such, it is considered a dangerous feature. So writing robust floating-point algorithms requires extra care in order to ensure that this potential double rounding will not produce incorrect results [3]. Nevertheless, double rounding is not necessarily a threat. For example, if the extended precision is at least twice as big, then it can be used to emulate correctly rounded basic operations for a smaller precision [4]. Double rounding can also be made innocuous by introducing a new rounding mode and using it for the first rounding. When a real number is not representable, it will be rounded to the adjacent floating-point number with an odd mantissa. In this article, this rounding will be named rounding to odd.

Von Neumann was considering this rounding when designing the arithmetic unit of the EDVAC [5]. Goldberg later used this rounding when converting binary floating-point numbers to decimal representations [6]. The properties of this rounding operator are close to the ones needed when implementing rounded floating-point operators with guards bits [7]. Because of its double rounding property, it has also been studied in the context of multistep gradual rounding [8]. Rounding to odd was never more than an implementation detail though, as two extra bits had to be stored in the floating-point registers. It was part of some hardware recipes that were claimed to give a correct result. Our work aims at giving precise and clear definitions and properties with a strong guarantee on their correctness. We also show that it is worth making rounding to odd a rounding mode in its own rights (it may be computed in hardware or in software). By rounding some computations to odd in an algorithm, more accurate results can be produced without extra precision.

Section II will detail a few characteristics of double rounding and why rounding to nearest is failing us. Section III will introduce the formal definition of rounding to odd, how it solves the double rounding issue, and how to implement this rounding. Its property with respect to double rounding will then be extended to two applications. Section IV will describe an algorithm that emulates the floating-point fused-multiply-and-add operator. Section V will then present algorithms for performing accurate summation. Formal proofs of the lemmas and theorems have been written and included in the Pff library[1] on floating-point arithmetic. Whenever relevant, the names of the properties in the following sections match the ones in the library.

II. Double rounding

A. Floating-point definitions

Our formal proofs are based on the floating-point formalization [9] of Daumas, Rideau, and Théry in Coq [10], and on the corresponding library by Théry, Rideau, and one of the authors [11]. Floating-point numbers are represented by pairs (n, e) that stand for $n \times 2^e$. We use both an integral signed mantissa n and an integral signed exponent e for sake of simplicity.

A floating-point format is denoted by $\mathbb{B}$ and is a pair composed by the lowest exponent $-E$ available and the precision p. We do

S. Boldo is with the INRIA Futurs.
G. Melquiond is with the École Normale Supérieure de Lyon.

[1] See `http://lipforge.ens-lyon.fr/www/pff/`.

not set an upper bound on the exponent as overflows do not matter here (see Section VI). We define a representable pair (n, e) such that $|n| < 2^p$ and $e \geq -E$. We denote by $\mathbb{F}$ the subset of real numbers represented by these pairs for a given format $\mathbb{B}$. Now only the representable floating-point numbers will be referred to; they will simply be denoted as floating-point numbers.

All the IEEE-754 rounding modes are also defined in the Coq library, especially the default rounding: rounding to nearest even, denoted by $\circ$. We have $f = \circ(x)$ if f is the floating-point number closest to x; when x is half way between two consecutive floating-point numbers, the one with an even mantissa is chosen.

A rounding mode is defined in the Coq library as a relation between a real number and a floating-point number, and not a function from real values to floats. Indeed, there may be several floats corresponding to the same real value. For a relation, a weaker property than being a rounding mode is being a faithful rounding. A floating-point number f is a faithful rounding of a real x if it is either the rounded up or rounded down of x, as shown on Figure 1. When x is a floating-point number, it is its own and only faithful rounding. Otherwise there always are two faithful rounded values bracketing the real value when no overflow occurs.

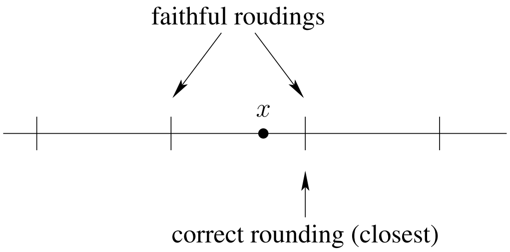


Fig. 1
FAITHFUL ROUNDINGS.

B. Double rounding accuracy

As explained before, a floating-point computation may first be done in an extended precision, and later rounded to the working precision. The extended precision is denoted by $\mathbb{B}_e = (p+k, E_e)$ and the working precision is denoted by $\mathbb{B}_w = (p, E_w)$. If the same rounding mode is used for both computations (usually to nearest even), it can lead to a less precise result than the result after a single rounding.

For example, see Figure 2. When the real value x is in the neighborhood of the midpoint of two consecutive floating-point numbers g and h, it may first be rounded in one direction toward this middle t in extended precision, and then rounded in the same direction toward f in working precision. Although the result f is close to x, it is not the closest floating-point number to x, as h is. When both rounding directions are to nearest, we formally proved that the distance between the given result f and the real value x may be as much as

$$|f - x| \leq \left(\frac{1}{2} + 2^{-k-1}\right) \mathrm{ulp}(f).$$

When there is only one rounding, the corresponding inequality is $|f - x| \leq \frac{1}{2}\mathrm{ulp}(f)$. This is the expected result for a IEEE-754 compatible implementation.

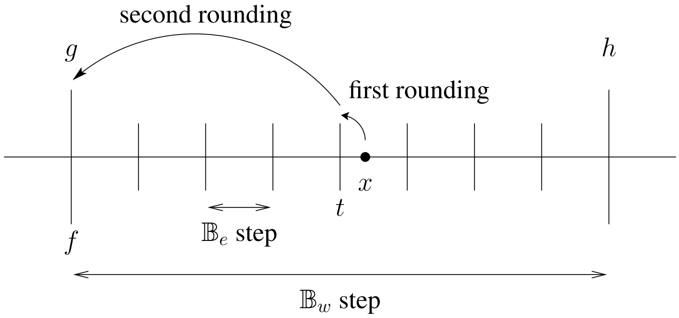


Fig. 2
BAD CASE FOR DOUBLE ROUNDING.

Section IV-B.1 will show that, when there is only one single floating-point format but many computations, trying to get a correctly rounded result is somehow similar to avoiding incorrect double rounding.

C. Double rounding and faithfulness

Another interesting property of double rounding as defined previously is that it is a faithful rounding. We even have a more generic result.

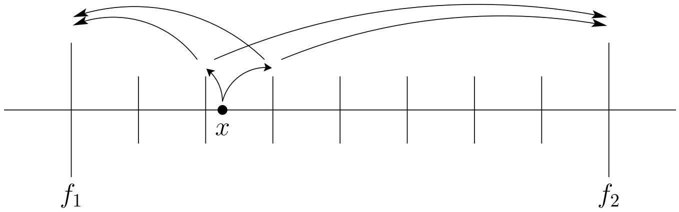


Fig. 3
DOUBLE ROUNDINGS ARE FAITHFUL.

Let us consider that the relations are not required to be rounding modes but only faithful roundings. We formally certified that the rounded result f of a double faithful rounding is faithful to the real initial value x, as shown in Figure 3.

Theorem 1 (DblRndStable): Let R_e be a faithful rounding in extended precision $\mathbb{B}_e = (p+k, E_e)$ and let R_w be a faithful rounding in the working precision $\mathbb{B}_w = (p, E_w)$. If $k \geq 0$ and $k \leq E_e - E_w$, then for all real value x, the floating-point number $R_w(R_e(x))$ is a faithful rounding of x in the working precision.

This is a deep result as faithfulness is the best result we can expect as soon as we consider at least two roundings to nearest. This result can be applied to any two successive IEEE-754 rounding modes (to zero, toward $+\infty$...). The requirements are $k \geq 0$ and $k \leq E_e - E_w$. The last requirement means that the minimum exponents $e_e^{\min}$ and $e_w^{\min}$ — as defined by the IEEE-754 standard — should satisfy $e_e^{\min} \leq e_w^{\min}$. As a consequence, it is equivalent to: Any normal floating-point number with respect to $\mathbb{B}_w$ should be normal with respect to $\mathbb{B}_e$.

This means that any sequence of successive roundings in decreasing precisions gives a faithful rounding of the initial value.

III. ROUNDING TO ODD

As seen in the previous section, rounding two times to nearest may induce a bigger round-off error than one single rounding to nearest and may then lead to unexpected incorrect results. By rounding to odd first, the second rounding will correctly round to nearest the initial value.

A. Formal description

Rounding to odd does not belong to the IEEE-754's or even 754R[2]'s rounding modes. It should not be mixed up with rounding to the nearest odd (similar to the default rounding: rounding to the nearest even).

We denote by $\triangle$ rounding toward $+\infty$ and by $\bigtriangledown$ rounding toward $-\infty$. Rounding to odd is defined by:

$$\square_{\text{odd}}(x) = \begin{cases} x & \text{if } x \in \mathbb{F}, \\ \triangle(x) & \text{if its mantissa is odd,} \\ \bigtriangledown(x) & \text{otherwise.} \end{cases}$$

Note that the result of x rounded to odd can be even only when x is a representable floating-point number. Note also that when x is not representable, $\square_{\text{odd}}(x)$ is not necessarily the nearest floating-point number with an odd mantissa. Indeed, this is wrong when x is close to a power of two. This partly explains why the formal proofs on algorithms involving rounding to odd will have to separate the case of powers of two from other floating-point numbers.

Theorem 2 (To_Odd):* Rounding to odd has the properties of a rounding mode [9]:

- each real can be rounded to odd;
- rounding to odd is faithful;
- rounding to odd is monotone.

Moreover,

- rounding to odd can be expressed as a function: a real cannot be rounded to two different floating-point values;
- rounding to odd is symmetric: if $f = \square_{\text{odd}}(x)$, then $-f = \square_{\text{odd}}(-x)$.

B. Implementing rounding to odd

Rounding to odd the real result x of a floating-point computation can be done in two steps. First round it to zero into the floating-point number $\mathcal{Z}(x)$ with respect to the IEEE-754 standard. And then perform a logical or between the inexact flag ι (or the sticky bit) of the first step and the last bit of the mantissa.

If the mantissa of $\mathcal{Z}(x)$ is already odd, this floating-point number is also the value of x rounded to odd; the logical or does not change it. If the floating-point computation is exact, $\mathcal{Z}(x)$ is equal to x and ι is not set; consequently $\square_{\text{odd}}(x) = \mathcal{Z}(x)$ is correct. Otherwise the computation is inexact and the mantissa of $\mathcal{Z}(x)$ is even, but the final mantissa must be odd, hence the logical or with ι. In this last case, this odd float is the correct one, since the first rounding was toward zero.

Computing ι is not a problem *per se*, since the IEEE-754 standard requires this flag to be implemented, and hardware already uses sticky bits for the other rounding modes. Furthermore, the value of ι can directly be reused to flag the rounded value of x as exact or inexact. As a consequence, on an already IEEE-754 compliant architecture, adding this new rounding has no significant cost.

Another way to round to odd with precision $p + k$ is the following. We first round x toward zero with $p + k - 1$ bits. We then concatenate the inexact bit of the previous operation at the end of the mantissa in order to get a $p + k$-bit float. The justification is similar to the previous one.

Both previous methods are aimed at hardware implementation. They may not be efficient enough to be used in software. Paragraph V-D will present a third way of rounding to odd, more adapted to current architectures and actually implemented. It is portable and available in higher level languages as it does not require changing the rounding direction and accessing the inexact flag.

C. Correct double rounding

Let x be a real number. This number is first rounded to odd in an extended format (precision is $p+k$ bits and 2^{-E_e} is the smallest positive floating-point number). Let t be this intermediate rounded result. It is then rounded to nearest even in the working format (precision is p bits and 2^{-E_w} is the smallest positive floating-point number). Although we are considering a real value x here, an implementation does not need to really handle x. The value x can indeed represent the abstract exact result of an operation between floating-point numbers. Although this sequence of operations is a double rounding, we state that the computed final result is correctly rounded.

Theorem 3 (To_Odd_Even_is_Even): Assuming $p \geq 2$, $k \geq 2$, and $E_e \geq 2 + E_w$,

$$\forall x \in \mathbb{R},\ \circ^p\left(\square_{\text{odd}}^{p+k}(x)\right) = \circ^p(x).$$

The proof is split in three cases as shown in Figure 4. When x is exactly equal to the middle of two consecutive floating-point numbers f_1 and f_2 (case 1), then t is exactly x and f is the correct rounding of x. Otherwise, when x is slightly different from this midpoint (case 2), then t is different from this midpoint: it is the odd value just greater or just smaller than the midpoint depending on the value of x. The reason is that, as $k \geq 2$, the midpoint is even in the $p + k$ precision, so t cannot be rounded into it if it is not exactly equal to it. This obtained t value will then be correctly rounded to f, which is the closest p-bit float from x. The other numbers (case 3) are far away from the midpoint and are easy to handle.

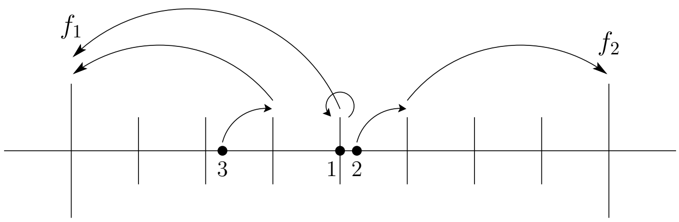


Fig. 4
DIFFERENT CASES OF ROUNDING TO ODD

Note that the hypothesis $E_e \geq 2 + E_w$ is a requirement easy to satisfy. It is weaker than the corresponding one in Theorem 1. In particular, the following condition is sufficient but no longer necessary: Any normal floating-point number with respect to $\mathbb{B}_w$ should be normal with respect to $\mathbb{B}_e$.

While the pen and paper proof is a bit technical, it does seem easy. It does not, however, consider the special cases, especially the ones where $\circ^p(x)$ is a power of two, and subsequently where

[2]See `http://www.validlab.com/754R/`.

$\circ^p(x)$ is the smallest normal floating-point number. We must look into all these special cases in order to ensure that the rounding is always correct, even when underflow occurs. We have formally proved this result using the Coq proof assistant. By using a proof checker, we are sure no cases were forgotten and no mistakes were made in the numerous computations. There are many splittings into subcases; they make the final proof rather long: seven theorems and about one thousand lines of Coq, but we are now sure that every cases (normal/subnormal, power of the radix or not) are supported. Details on this proof were presented in a previous work [12].

Theorem 3 is even more general than what is presented here: it can be applied to any realistic rounding to the closest (meaning that the result of a computation is uniquely defined by the value of the infinitely precise result and does not depend on the machine state). In particular, it handles the new rounding to nearest, ties away from zero, defined by the revision of the IEEE-754 standard.

IV. Emulating the FMA

The fused-multiply-and-add is a recent floating-point operator that is present on a few modern processors like PowerPC or Itanium. This operation will hopefully be standardized in the revision of the IEEE-754 standard. Given three floating-point numbers a, b, and c, it computes the value $z = \circ(a \cdot b + c)$ with one single rounding at the end of the computation. There is no rounding after the product $a \cdot b$. This operator is very useful as it may increase performance and accuracy of the dot product and matrix multiplication. Algorithm 1, page 5, shows how it can be emulated thanks to rounding to odd. This section will describe its principles.

A. The algorithm

Algorithm 1 relies on error-free transformations (ExactAdd and ExactMult) to perform some of the operations exactly. These transformations described below return two floating-point values. The first one is the usual result: the exact sum or product rounded to nearest. The other one is the error term. For addition and multiplication, this term happens to be exactly representable by a floating-point number and computable using only floating-point operations provided neither underflow (for the multiplication) nor overflow occurs. As a consequence, in Algorithm 1, these equalities hold: $a \cdot b = u_h + u_l$ and $c + u_h = t_h + t_l$. And the rounded result is stored in the higher word: $u_h = \circ(a \cdot b)$ and $t_h = \circ(c + u_h)$.

A fast operator for computing the error term of the multiplication is the FMA: $u_l = \circ(a \cdot b + (-u_h))$. Unfortunately, our goal is the emulation of a FMA, so we have to use another method. In IEEE-754 double precision, Dekker's algorithm first splits the 53-bit floating-point inputs into 26-bit parts thanks to the sign bit. These parts can then be multiplied exactly and subtracted in order to get the error term [13]. For the error term of the addition, since we do not know the relative order of $|c|$ and $|u_h|$, we use Knuth's unconditional version of the algorithm [14]. These two algorithms have been formally proved in Coq [9], [15].

Our emulated FMA first computes an approximation of the correct result: $t_h = \circ(\circ(a \cdot b) + c)$. It also computes an auxiliary term v that is added to t_h to get the final result. All the computations are done at the working precision, there is no need for an extended precision. The number v is computed by adding the neglected terms u_l and t_l, and by rounding the result to odd. The following example will show that the answer would be wrong if all the roundings were to nearest instead.

Let us consider $a = 1 + 2^{-27}$ and $b = 1 - 2^{-27}$. The exact product is $a \cdot b = 1 - 2^{-54}$. This real number is exactly the midpoint between 1 and its representable predecessor in double precision. If c is small enough (for example, $|c| \leq 2^{-150}$), it means that the value $\circ(a \cdot b + c)$ will purely depend on the sign of c. If c is negative, it should be $1 - 2^{-53}$, otherwise 1. If our algorithm were to round to nearest instead of rounding to odd, the final result would always be 1, irrespective of the sign of c.

B. Theorem of correctness

Theorem 4 (FmaEmul): Under the notations of Algorithm 1, if u_l is representable and $p \geq 5$, then

$$z = \circ(a \times b + c).$$

The previous theorem states that the algorithm emulates a FMA under two hypotheses. First, the value u_l has to be the error term of the multiplication $a \cdot b$, in order to avoid some degenerate underflow cases: the error term becomes so small that its exponent falls outside the admitted range. The second hypothesis requires the mantissa to have at least 5 bits. This requirement is reasonable since even the smallest format of the IEEE-754 standard has a 24-bit mantissa.

This theorem has been formally proved with a proof checker. This is especially important as it is quite generic. In particular, it does not contain any hypothesis regarding subnormal numbers. The algorithm will behave correctly even if some computed values are not normal numbers, as long as u_l is representable.

1) Adding a negligible yet odd value: We need an intermediate lemma for simplicity and reusability, described by Figure 5.

Lemma 1 (AddOddEven): Let μ be the smallest positive floating-point normal number. Let x be a floating-point number such that $|x| \geq 5 \cdot \mu$. Let z be a real number and $y = \square_{\text{odd}}(z)$. Assuming $5 \cdot |y| \leq |x|$,

$$\circ(x + y) = \circ(x + z).$$

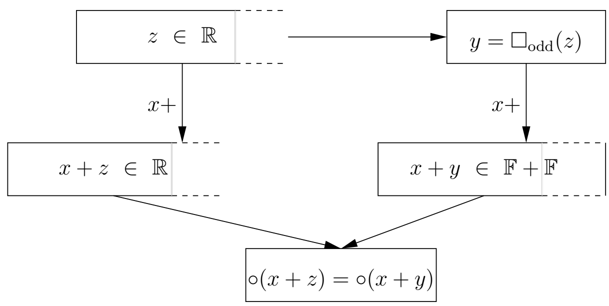
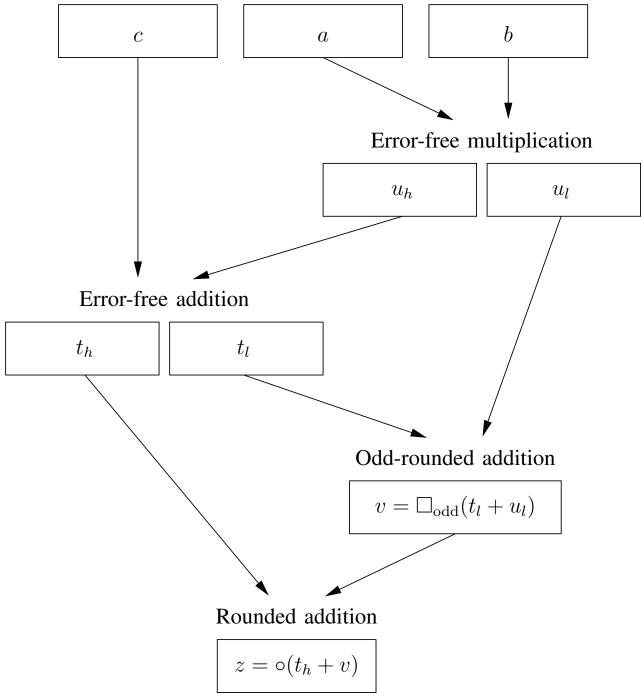


Fig. 5
Lemma AddOddEven.

By uniqueness of the rounded value to nearest even, it is enough to prove that $\circ(x + y)$ is a correct rounding to nearest of $x + z$ with tie breaking to even. By applying Theorem 3, we just have to prove that $x + y$ is equal to $\square_{\text{odd}}^{p+k}(x + z)$ for a k that we might choose as we want (as soon as it is greater than 1).

Algorithm 1 Emulating the FMA.

$$\begin{aligned}(u_h, u_l) &= \text{ExactMult}(a, b)\\ (t_h, t_l) &= \text{ExactAdd}(c, u_h)\\ v &= \square_{\text{odd}}(t_l + u_l)\\ z &= \circ(t_h + v)\end{aligned}$$



The integer k is chosen so that there exists a floating-point number f equal to $x + y$, normal with respect to an extended format on precision $p + k$ and having the same exponent as y. For that, we set $f = (n_x \cdot 2^{e_x - e_y} + n_y, e_y)$. As $|y| \le |x|$, we know that $e_y \le e_x$ and this definition has the required exponent. We then choose k such that $2^{p+k-1} \le |n_f| < 2^{p+k}$. The property $k \ge 2$ is guaranteed by $5 \cdot |y| \le |x|$. The underflow threshold for the extended format is defined as needed thanks to the $5 \cdot \mu \le |x|$ hypothesis. These ponderous details are handled in the machine-checked proof.

So we have defined an extended format where $x + y$ is representable. There is left to prove that $x + y = \square_{\text{odd}}^{p+k}(x + z)$. We know that $y = \square_{\text{odd}}(z)$, thus we have two cases. First, $y = z$, so $x + y = x + z$ and the result holds. Second, y is odd and is a faithful rounding of z. Then we prove (several possible cases and many computations later), that $x + y$ is odd and is a faithful rounding of $x + z$ with respect to the extended format. That ends the proof.

Several variants of this lemma are used in Section V-A. They all have been formally proved too. Their proofs have a similar structure and they will not be detailed here. Please refer to the Coq formal development for in-depth proofs.

2) Emulating a FMA: First, we can eliminate the case where v is computed without rounding error. Indeed, it means that $z = \circ(t_h + v) = \circ(t_h + t_l + u_l)$. Since $u_l = a \cdot b - u_h$ and $t_h + t_l = c + u_h$, we have $z = \circ((c + u_h) + (a \cdot b - u_h)) = \circ(a \cdot b + c)$.

Now, if v is rounded, it means that v is not a subnormal number. Indeed, if the result of a floating-point addition is a subnormal number, then the addition is exact. It also means that neither u_l nor t_l are zero. So neither the product $a \cdot b$ nor the sum $c + u_h$ are representable floating-point numbers.

Since $c + u_h$ is not representable, the inequality $2 \cdot |t_h| \ge |u_h|$ holds. Moreover, since u_l is the error term in $u_h + u_l$, we have $|u_l| \le 2^{-p} \cdot |u_h|$. Similarly, $|t_l| \le 2^{-p} \cdot |t_h|$. As a consequence, both $|u_l|$ and $|t_l|$ are bounded by $2^{1-p} \cdot |t_h|$. So their sum $|u_l + t_l|$ is bounded by $2^{2-p} \cdot |t_h|$. Since v is not a subnormal number, the inequality still holds when rounding $u_l + t_l$ to v. So we have proved that $|v| \le 2^{2-p} \cdot |t_h|$ when the computation of v is inexact.

To summarize, either v is equal to $t_l + u_l$, or v is negligible with respect to t_h. Lemma 1 can then be applied with $x = t_h$, $y = v$, and $z = t_l + u_l$. Indeed $x + z = t_h + t_l + u_l = a \cdot b + c$. We have to verify two inequalities in order to apply it though. First, we must prove $5 \cdot |y| \le |x|$, meaning that $5 \cdot |v| \le |t_h|$. We have just shown that $|v| \le 2^{2-p} \cdot |t_h|$. As $p \ge 5$, this inequality easily holds.

Second, we must prove $5 \cdot \mu \le |x|$, meaning that $5 \cdot \mu \le |t_h|$. We prove it by assuming $|t_h| < 5 \cdot \mu$ and reducing it to the absurd. So t_l is subnormal. More, t_h must be normal: if t_h is subnormal, then $t_l = 0$, which is impossible. We then look into u_l. If u_l is subnormal, then $v = \square_{\text{odd}}(u_l + t_l)$ is computed correctly, which is impossible. So u_l is normal. We then prove that both $t_l = 0$ and $t_l \ne 0$ hold. First, $t_l \ne 0$ as $v \ne u_l + t_l$. Second, we will prove $t_l = 0$ by proving that the addition $c + u_h$ is computed exactly (as $t_h = \circ(c + u_h)$). For that, we will prove that $e_{t_h} < e_{u_h} - 1$ as that implies a cancellation in the computation of $c + u_h$ and therefore the exactness of t_h. There is then left to prove that $2^{e_{t_h}} < 2^{e_{u_h}-1}$. As t_h is normal, $2^{e_{t_h}} \le |t_h| \cdot 2^{1-p}$. As $p \ge 5$ and u_l is normal, $5 \cdot \mu \cdot 2^{1-p} \le \mu \le |u_l|$. Since we have both $|t_h| < 5 \cdot \mu$ and $|u_l| \le 2^{e_{u_h}-1}$, we can deduce $2^{e_{t_h}} < 2^{e_{u_h}-1}$. We have a contradiction in all cases, therefore $5 \cdot \mu \le |t_h|$ holds. So the hypotheses of Lemma 1 are now verified and the proof is completed.

V. Accurate summation

The last steps of the algorithm for emulating the FMA actually compute the correctly rounded sum of three floating-point numbers at once. Although there is no particular assumption on two of the numbers (c and u_h), there is a strong hypothesis on the third

Algorithm 2 Iterated summation

Input: the $(f_i)_{1\le i\le n}$ are suitably ordered and spaced out.

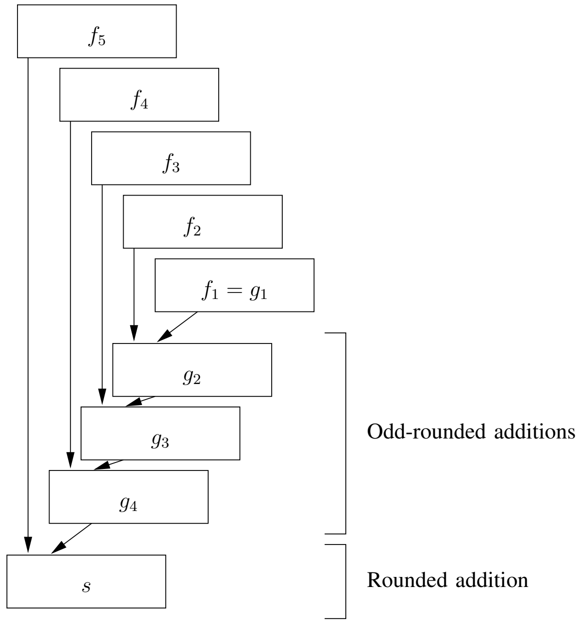
$g_1 = f_1$

For i from 2 to $n-1$,

$\quad g_i = \square_{\mathrm{odd}}(g_{i-1} + f_i)$

$s = \circ(g_{n-1} + f_n)$

Output: $s = \circ(\sum f_i)$.



one: $|u_l| \le 2^{-p} \cdot |u_h|$. We will generalize this summation scheme to an iterated scheme that will compute the correctly rounded sum of a set of floating-point numbers under some strong assumptions. We will then describe a generic adder for three floating-point numbers that rely on rounding to odd to produce the correctly rounded result.

A. Iterated summation

We consider the problem of adding a sequence of floating-point numbers $(f_i)_{1\le i\le n}$. Let us pose $t_j = \sum_{1\le i\le j} f_i$ the exact partial sums. The objective is to compute the correctly rounded sum $s = \circ(t_n)$. This problem is not new: adding several floating-point numbers with good accuracy is an important problem of scientific computing [16]. Demmel and Hida presented a simple algorithm that yields almost correct summation results [17]. And recently Oishi, Rump, and Ogita, presented some other algorithms for accurate summation [18]. Our algorithm requires stronger assumptions, but it is simple, very fast, and will return the correctly rounded result thanks to rounding to odd.

Two approaches are possible. The first one was described in a previous work [12]. The algorithm computes the partial sums in an extended precision format with rounding to odd. In order to produce the final result, it relies on this double rounding property: $\circ(t_n) = \circ(\square_{\mathrm{odd}}^{p+k}(t_n))$. The correctness of the algorithm depends on the following property proved by induction on j for $j < n$:

$$\square_{\mathrm{odd}}^{p+k}(t_{j+1}) = \square_{\mathrm{odd}}^{p+k}\left(f_{j+1} + \square_{\mathrm{odd}}^{p+k}(t_j)\right).$$

Once the value $\square_{\mathrm{odd}}^{p+k}(t_n)$ has been computed, it is rounded to the working precision in order to obtain the correctly rounded result s thanks to the double rounding property. Unfortunately, this algorithm requires that an extended precision format is available in order to compute the intermediate results.

Let us now present a new approach. While similar to the old one, Algorithm 2 does not need any extended precision to perform its intermediate computations.

Theorem 5 (Summation): We use the notations of Algorithm 2 and assume a reasonable floating-point format is used. Let μ be the smallest positive floating-point normal number. If the following properties hold for any j such that $2 < j < n$,

$$|f_j| \ge 2 \cdot |t_{j-1}| \quad \text{and} \quad |f_j| \ge 2 \cdot \mu,$$

and if the most significant term verifies

$$|f_n| \ge 6 \cdot |t_{n-1}| \quad \text{and} \quad |f_n| \ge 5 \cdot \mu,$$

then $s = \circ(t_n)$.

The proof of this theorem has two parts. First, we prove by induction on j that $g_j = \square_{\mathrm{odd}}(t_j)$ holds for all $j < n$. In particular, we have $s = \circ(f_n + \square_{\mathrm{odd}}(t_{n-1}))$. Second, we prove that $\circ(f_n + \square_{\mathrm{odd}}(t_{n-1})) = \circ(f_n + t_{n-1})$. This equality is precisely $s = \circ(t_n)$. Both parts are proved by applying variants of Lemma 1. The correctness of the induction is a consequence of Lemma 2 while the correctness of the final step is a consequence of Lemma 3.

Lemma 2 (AddOddOdd2): Let x be a floating-point number such that $|x| \ge 2 \cdot \mu$. Let z be a real number. Assuming $\frac{1}{2}$ is a normal floating-point and $2 \cdot |z| \le |x|$,

$$\square_{\mathrm{odd}}(x + \square_{\mathrm{odd}}(z)) = \square_{\mathrm{odd}}(x + z).$$

Lemma 3 (AddOddEven2): Let x be a floating-point number such that $|x| \ge 5 \cdot \mu$. Let z be a real number. Assuming $p > 3$ and $6 \cdot |z| \le |x|$,

$$\circ(x + \square_{\mathrm{odd}}(z)) = \circ(x + z).$$

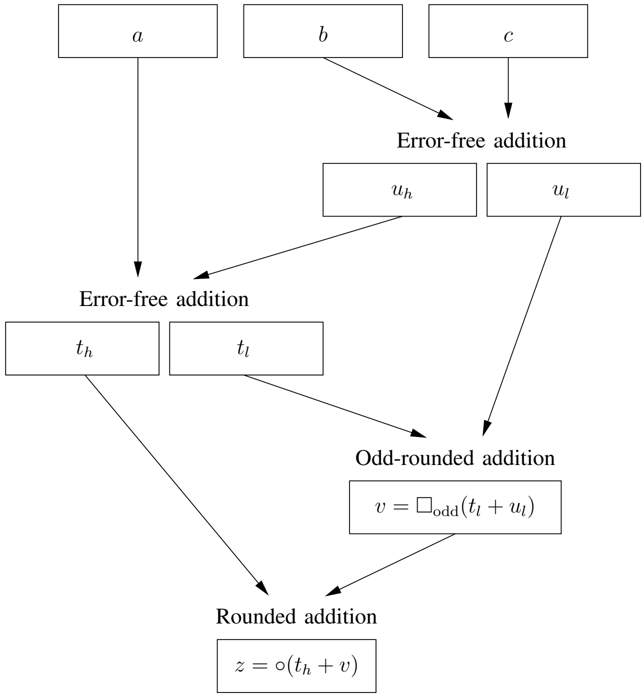
It may generally be a bit difficult to verify that the hypotheses of the summation theorem hold at execution time. So it is interesting to have a sufficient criteria that can be checked with floating-point numbers only:

$$|f_2| \ge 2 \cdot \mu \quad \text{and} \quad |f_n| \ge 9 \cdot |f_{n-1}|,$$

$$\text{for } 1 \le i \le n-2, \quad |f_{i+1}| \ge 3 \cdot |f_i|.$$

Algorithm 3 Adding three numbers

$$
\begin{aligned}
(u_h, u_l) &= \text{ExactAdd}(b, c)\\
(t_h, t_l) &= \text{ExactAdd}(a, u_h)\\
v &= \square_{\text{odd}}(t_l + u_l)\\
z &= \circ(t_h + v)
\end{aligned}
$$



B. Reducing expansions

A floating-point expansion is a list of sorted floating-point numbers, its value being the exact sum of its components [19]. Computations on these multi-precision values are done using only existing hardware and are therefore very fast.

If the expansion is non-overlapping, looking at the three most significant terms is sufficient to get the correct approximated value of the expansion. This can be achieved by computing the sum of these three terms with Algorithm 2. The algorithm requirements on ordering and spacing are easily met by expansions.

Known fast algorithms for basic operations on expansions (addition, multiplication, etc) take as inputs and outputs pseudo-expansions, *i.e.* expansions with a slight overlap (typically a few bits) [19], [20]. Then, looking at three terms only is no longer enough. All the terms up to the least significant one may have an influence on the correctly rounded sum. This problem can be solved by normalizing the pseudo-expansions in order to remove overlapping terms. This process is, however, extremely costly: if the expansion has n terms, Priest's algorithm requires about $6 \cdot n$ floating-point additions in the best case ($9 \cdot n$ in the worst case). In a simpler normalization algorithm with weaker hypotheses [20], the length of the dependency path to get the three most significant terms is $7 \cdot n$ additions.

A more efficient solution is provided by Algorithm 2, as it can directly compute the correctly rounded result with n floating-point additions only. Indeed, the criteria at the end of Section V-A is verified by expansions which overlap by at most $p - 5$ bits, therefore also by pseudo-expansions.

C. Adding three numbers

Let us now consider a simpler situation. We still want to compute a correctly-rounded sum, but there are only three numbers left. In return, we will remove all the requirements on the relative ordering of the inputs. Algorithm 3 shows how to compute this correctly-rounded sum of three numbers.

Its graph looks similar to the graph of Algorithm 1 for emulating the FMA. The only difference lies in its first error-free transformation. Instead of computing the exact product of two of its inputs, this algorithm computes their exact sum. As a consequence, its proof of correctness can directly be derived from the one for the FMA emulation. Indeed, the correctness of the emulation does not depend on the properties of an exact product. The only property that matters is: $u_h + u_l$ is a normalized representation of a number u. As a consequence, both Algorithm 1 and Algorithm 3 are special cases of a more general algorithm that would compute the correctly rounded sum of a floating-point number with a real number exactly represented by the sum of two floating-point numbers.

Note that the three inputs of the adder do not play a symmetric role. This property will be used in the following section to optimize some parts of the adder.

D. A practical use case

CRlibm[3] is an efficient library for computing correctly rounded results of elementary functions in IEEE-754 double precision. Let us consider the logarithm function [21]. In order to be efficient, the library first executes a fast algorithm. This usually gives the correctly rounded result, but in some situations it may be off by one unit in the last place. When the library detects such a situation, it starts again with a slower yet accurate algorithm in order to get the correct final result.

When computing the logarithm $\circ(\log f)$, the slow algorithm will use triple-double arithmetic [22] to first compute an approximation of $\log f$ stored on three double precision numbers $x_h + x_m + x_l$. Thanks to results provided by the table-maker

[3]See `http://lipforge.ens-lyon.fr/www/crlibm/`.

Listing 1 Correctly rounded sum of three ordered values

```
double CorrectRoundedSum3(double xh, double xm, double xl) {
  double th, tl;
  db_number thdb; // thdb.l is the binary representation of th

  // Dekker's error-free adder of two ordered numbers
  Add12(th, tl, xm, xl);

  // round to odd th if tl is not zero
  if (tl != 0.0) {
    thdb.d = th;
    // if the mantissa of th is odd, there is nothing to do
    if (!(thdb.l & 1)) {
      // choose the rounding direction
      // depending on the signs of th and tl
      if ((tl > 0.0) ^ (th < 0.0))
        thdb.l++;
      else
        thdb.l--;
      th = thdb.d;
    }
  }

  // final addition rounded to nearest
  return xh + th;
}
```

dilemma [23], this approximation is known to be sufficiently accurate for the equality $\circ(\log f) = \circ(x_h + x_m + x_l)$ to hold. This means the library just has to compute the correctly rounded sum of the three floating-point numbers x_h, x_m, and x_l.

Computing this sum is exactly the point of Algorithm 3. Unfortunately, rounding to odd is not available on any architecture targeted by CRlibm, so it will have to be emulated. Although such an emulation is costly in software, rounding to odd still allows for a speed-up here. Indeed $x_h + x_m + x_l$ is the result of a sequence of triple-double floating-point operations, so this is precisely the case described in Section V-B. As a consequence, the operands are ordered in such a way that some parts of Algorithm 3 are not necessary. In fact, Lemma 3 implies the following equality:

$$\circ(x_h + x_m + x_l) = \circ(x_h + \square_{\mathrm{odd}}(x_m + x_l)).$$

This means that, at the end of the logarithm function, we just have to compute the rounded-to-odd sum of x_m and x_l, and then do a standard floating-point addition with x_h. Now, all that is left is the computation of $\square_{\mathrm{odd}}(x_m + x_l)$. This is achieved by first computing $t_h = \circ(x_m + x_l)$ and $t_l = x_m + x_l - t_h$ thanks to an error-free adder. If t_l is zero or if the mantissa of t_h is odd, then t_h is already equal to $\square_{\mathrm{odd}}(x_m + x_l)$. Otherwise t_h is off by one unit in the last place. We replace it either by its successor or by its predecessor depending on the signs of t_l and t_h.

Listing 1 shows a cleaned version of a macro used by CRlibm: `ReturnRoundToNearest3Other`. The macro `Add12` is an implementation of Dekker's error-free adder. It is only 3-addition long, and it is correct since the inequality $|x_m| \geq |x_l|$ holds. The successor or the predecessor of t_h is directly computed by incrementing or decrementing the integer `thdb.l` that holds its binary representation. Working on the integer representation is correct, since t_h cannot be zero when t_l is not zero.

CRlibm already contained some code at the end of the logarithm function in order to compute the correctly rounded sum of three floating-point numbers. When the code of Listing 1 is used instead, the slow step of this elementary function gets 25 cycles faster on an AMD Opteron processor. While we only looked at the very last operation of the logarithm, it still amounts to a 2% speed-up on the whole function.

The performance increase would obviously be even greater if we had not to emulate a rounded-to-odd addition. Moreover, this speed-up is not restricted to logarithm: it is available for every other rounded elementary functions, since they all rely on triple-double arithmetic at the end of their slow step.

VI. Conclusion

We first considered rounding to odd as a way of performing intermediate computations in an extended precision and yet still obtaining correctly rounded results at the end of the computations. This is expressed by Theorem 3. Rounding to odd then led us to consider algorithms that could benefit from its robustness. We first considered an iterated summation algorithm that was using extended precision and rounding to odd in order to perform the intermediate additions. The FMA emulation however showed that the extended precision only has to be virtual. As long as we prove that the computations are done as if an extended precision is used, the working precision can be used. This is especially useful when we already compute with the highest available precision. The constraints on the inputs of Algorithm 2 are compatible with floating-point expansions: the correctly rounded sum of an overlapping expansion can easily be computed.

Algorithm 1 for emulating the FMA and Algorithm 3 for adding numbers are similar. They both allow to compute $\circ(a \diamond b + c)$ with a, b, and c three floating-point numbers, as long as $a \diamond b$ is exactly representable as the sum of two floating-point numbers. These algorithms rely on rounding to odd to ensure that the result is correctly rounded. Although this rounding is not available in current hardware, our changes to CRlibm have shown that reasoning on it opens the way to some efficient new algorithms for computing correctly rounded results.

In this paper, we did not tackle at all the problem of overflowing operations. The reason is that overflow does not matter here: on all the algorithms presented, overflow can be detected afterward. Indeed, any of these algorithms will produce an infinity or a NaN as a result in case of overflow. The only remaining problem is that they may create an infinity or a NaN although the result could be represented. For example, let M be the biggest positive floating-point number, and let $a = -M$ and $b = c = M$ in Algorithm 3. Then $u_h = t_h = +\infty$ and $u_l = t_l = v = -\infty$ and $z = NaN$ whereas the correct result is M. This can be misleading, but this is not a real problem when adding three numbers. Indeed, the crucial point is that we cannot create inexact finite results: when the result is finite, it is correct. When emulating the FMA, the error-term of the product is required to be correctly computed. This property can be checked by verifying that the magnitude of the product is big enough.

While the algorithms presented here look short and simple, their correctness is far from trivial. When rounding to odd is replaced by a standard rounding to nearest, there exist inputs such that the final results are no longer correctly rounded. It may be difficult to believe that simply changing one intermediate rounding is enough to fix some algorithms. So we have written formal proofs of their correctness and used the Coq proof-checker to guarantee their validity. This approach is essential to ensure that the algorithms are correct, even in the unusual cases.

References

[1] D. Stevenson *et al.*, "A proposed standard for binary floating point arithmetic," *IEEE Computer*, vol. 14, no. 3, pp. 51–62, 1981.

[2] ——, "An American national standard: IEEE standard for binary floating point arithmetic," *ACM SIGPLAN Notices*, vol. 22, no. 2, pp. 9–25, 1987.

[3] G. Melquiond and S. Pion, "Formally certified floating-point filters for homogeneous geometric predicates," *Theoretical Informatics and Applications*, vol. 41, no. 1, pp. 57–70, 2007.

[4] S. A. Figueroa, "When is double rounding innocuous?" *SIGNUM Newsletter*, vol. 30, no. 3, pp. 21–26, 1995.

[5] J. von Neumann, "First draft of a report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[6] D. Goldberg, "What every computer scientist should know about floating point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–47, 1991.

[7] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[8] C. Lee, "Multistep gradual rounding," *IEEE Transactions on Computers*, vol. 38, no. 4, pp. 595–600, 1989.

[9] M. Daumas, L. Rideau, and L. Théry, "A generic library of floating-point numbers and its application to exact computing," in *14th International Conference on Theorem Proving in Higher Order Logics*, Edinburgh, Scotland, 2001, pp. 169–184.

[10] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[11] S. Boldo, "Preuves formelles en arithmétiques à virgule flottante," Ph.D. dissertation, École Normale Supérieure de Lyon, Nov. 2004.

[12] S. Boldo and G. Melquiond, "When double rounding is odd," in *Proceedings of the 17th IMACS World Congress on Computational and Applied Mathematics*, Paris, France, 2005.

[13] T. J. Dekker, "A floating point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[14] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*. Addison Wesley, 1969, vol. 2.

[15] S. Boldo, "Pitfalls of a full floating-point proof: example on the formal proof of the Veltkamp/Dekker algorithms," in *Third International Joint Conference on Automated Reasoning*, ser. Lecture Notes in Artificial Intelligence, U. Furbach and N. Shankar, Eds., vol. 4130. Seattle, USA: Springer-Verlag, 2006.

[16] N. J. Higham, *Accuracy and stability of numerical algorithms*. SIAM, 1996.

[17] J. W. Demmel and Y. Hida, "Fast and accurate floating point summation with applications to computational geometry," in *Proceedings of the 10th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2002)*, January 2003.

[18] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM Journal on Scientific Computing*, vol. 26, no. 6, pp. 1955–1988, 2005.

[19] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, P. Kornerup and D. W. Matula, Eds. Grenoble, France: IEEE Computer Society, 1991, pp. 132–144.

[20] M. Daumas, "Multiplications of floating point expansions," in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, I. Koren and P. Kornerup, Eds., Adelaide, Australia, 1999, pp. 250–257.

[21] F. de Dinechin, C. Q. Lauter, and J.-M. Muller, "Fast and correctly rounded logarithms in double-precision," *Theoretical Informatics and Applications*, vol. 41, no. 1, pp. 85–102, 2007.

[22] C. Q. Lauter, "Basic building blocks for a triple-double intermediate format," LIP, Tech. Rep. RR2005-38, Sep. 2005.

[23] V. Lefèvre and J.-M. Muller, "Worst cases for correct rounding of the elementary functions in double precision," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, N. Burgess and L. Ciminiera, Eds., Vail, Colorado, 2001, pp. 111–118.

Sylvie Boldo received the MSC and PhD degrees in computer science from the École Normale Supérieure de Lyon, France, in 2001 and 2005. She is now researcher for the INRIA Futurs in the ProVal team (Orsay, France), whose research focuses on formal certification of programs. Her main research interests include floating-point arithmetic, formal methods and formal verification of numerical programs.

Guillaume Melquiond received the MSC and PhD degrees in computer science from the École Normale Supérieure de Lyon, France, in 2003 and 2007. He is now a postdoctoral fellow at the INRIA–Microsoft Research joint laboratory (Orsay, France) in the Mathematical Components team, whose research focuses on developing tools for formally proving mathematical theorems. His interests include floating-point arithmetic, formal methods for certifying numerical software, interval arithmetic, and C++ software engineering.