On the Computation of Correctly-Rounded Sums

Peter Kornerup, Vincent Lefèvre, Nicolas Louvet, Jean-Michel Muller

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the Computation of Correctly-Rounded Sums

Peter Kornerup — Vincent Lefèvre — Nicolas Louvet — Jean-Michel Muller

N° 7262

April 2010

Algorithms, Certification, and Cryptography

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7262--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE – RHÔNE-ALPES

On the Computation of Correctly-Rounded Sums

Peter Kornerup*, Vincent Lefèvre†, Nicolas Louvet†, Jean-Michel Muller†

Theme : Algorithms, Certification, and Cryptography
Équipe-Projet Arénaire

Rapport de recherche n° 7262 — April 2010 — 24 pages

Abstract: This paper presents a study of some basic blocks needed in the design of floating-point summation algorithms. In particular, in radix-2 floating-point arithmetic, we show that among the set of the algorithms with no comparisons performing only floating-point additions/subtractions, the 2Sum algorithm introduced by Knuth is minimal, both in terms of number of operations and depth of the dependency graph. We investigate the possible use of another algorithm, Dekker's Fast2Sum algorithm, in radix-10 arithmetic. We give methods for computing, in radix 10, the floating-point number nearest the average value of two floating-point numbers. Under reasonable conditions, we also prove that no algorithms performing only round-to-nearest additions/subtractions exist to compute the round-to-nearest sum of at least three floating-point numbers. Starting from an algorithm due to Boldo and Melquiond, we also present new results about the computation of the correctly-rounded sum of three floating-point numbers.

Key-words: floating-point arithmetic, summation algorithms, correct rounding, 2Sum and Fast2Sum algorithms

* SDU, Odense, Denmark
† LIP, CNRS/ENS Lyon/INRIA/UCBL/Université de Lyon, Lyon, France

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Sur le calcul de sommes correctement arrondies

Résumé : Ce papier présente une étude de blocs de base nécessaires à la conception d'algorithmes de sommation à virgule flottante. En particulier, en arithmétique flottante en base 2, nous montrons que parmi l'ensemble des algorithmes sans comparaison effectuant seulement des additions/soustractions flottantes, l'algorithme 2Sum introduit par Knuth est minimal, que ce soit en terme de nombre d'opérations ou de profondeur du graphe de dépendances. Nous étudions la possibilité d'utiliser un autre algorithme, l'algorithme Fast2Sum de Dekker, en base 10. Nous donnons des méthodes pour calculer, en base 10, le nombre flottant le plus proche de la valeur moyenne de deux flottants. Sous des conditions raisonnables, nous prouvons aussi qu'il n'existe aucun algorithme effectuant uniquement des additions/soustractions en arrondi au plus près pour calculer la somme arrondie au plus près d'au moins trois nombres flottants. En partant d'un algorithme dû à Boldo et Melquiond, nous présentons aussi de nouveaux résultats concernant le calcul de la somme correctement arrondie de trois nombres flottants.

Mots-clés : arithmétique à virgule flottante, algorithmes de sommation, arrondi correct, algorithmes 2Sum et Fast2Sum

1 Introduction

The computation of sums appears in many domains of numerical analysis. Examples are numerical integration, evaluation of dot products, matrix products, means, variances and many other functions. When computing the sum of n floating-point numbers $a_1, a_2, \ldots, a_n$, the best one can hope is to get $\circ(a_1 + a_2 + \cdots a_n)$, where $\circ$ is the desired rounding function (specified by a *rounding mode*, or by a *rounding direction attribute*, in the terminology of the IEEE 754 Standard for floating-point arithmetic [2, 10]). On current architectures this can always be done in software using multiple-precision arithmetic. This could also be done using a long accumulator, as advocated by Kulisch [4], but such accumulators are not yet available on current processors.

It is well known that the rounding error generated by a round-to-nearest addition is itself a floating-point number. Many accurate and efficient summation algorithms published in the literature (see for instance [1, 17, 18, 16, 5]) are based on this property and implicitly or explicitly use basic blocks such as Dekker's Fast2Sum and Knuth's 2Sum algorithms (Algorithms 1 and 2 below) to compute the rounding error generated by a floating-point addition. Since efficiency is one of the main concerns in the design of floating-point programs, we focus on algorithms using only floating-point additions and subtractions in the target format and without conditional branches, because on current pipelined architectures, a wrong branch prediction may cause the instruction pipeline to drain, with a resulting drastic performance loss. The computation of the correctly-rounded sum of three floating-point numbers is also a basic task needed in many different contexts: in [5], Boldo and Melquiond have presented a new algorithm for this task, with an application in the context of the computation of elementary functions. Hence, it is of great interest to study the properties of these basic blocks.

In this paper, we assume an IEEE 754 [2, 10] arithmetic. We show that among the set of the algorithms with no comparisons performing only floating-point operations, the 2Sum algorithm introduced by Knuth is minimal, both in terms of number of operations and depth of the dependency graph.

The recent revision of the IEEE Standard for floating-point arithmetic considers arithmetics of radices 2 and 10. Some straightforward properties of radix-2 arithmetic have been known for a long time and are taken for granted. And yet, some properties do not hold in radix 10. A simple example is that, in radix 10, computing the average value of two floating-point numbers a and b first by computing $a + b$ rounded to the nearest, and then by computing half the obtained result rounded to the nearest again will not necessarily give the average value rounded to the nearest. We will investigate that problem and suggest some strategies for accurately computing the average value of two numbers in decimal arithmetic.

Under reasonable assumptions, we also show that it is impossible to always obtain the correctly round-to-nearest sum of $n \geq 3$ floating-point numbers with an algorithm performing only round-to-nearest additions/subtractions. The algorithm proposed by Boldo and Melquiond for computing the round-to-nearest sum of three floating-point numbers relies on a non-standard rounding mode, *rounding to odd*. We show that if the radix is even, rounding to odd can be emulated in software using only floating-point additions/subtractions in the standard rounding modes and a multiplication by the constant 0.5, thus allowing the round-to-nearest sum of three floating-point numbers to be determined without tests. We also propose algorithms to compute the correctly-rounded sum of three floating-point values for directed roundings.

In a preliminary version of this paper [11], we gave results valid in radix-2 floating-point arithmetic. We now extend these results to other radices (the most interesting one being radix 10), consider the problem of computing an average value in radix 10, give new summation algorithms, and extend the proofs of Theorems 2 and 3 to any precision.

1.1 Assumptions and notations

We assume a *radix-β and precision-p floating-point arithmetic* as defined (for radices 2 and 10) in the IEEE 754-2008 standard for floating-point arithmetic [10]. Typical examples are the basic formats defined by that standard: precisions 24, 53 or 113 in radix 2, and 7, 16 and 34 in radix 10. The user can choose an *active rounding mode*, also called *rounding direction attribute*: round toward $-\infty$, round toward $+\infty$, round toward 0, round to nearest even, which is the default rounding mode, and round to nearest "TiesToAway". Given a real number x, we denote respectively by $RD(x)$, $RU(x)$, $RZ(x)$ and $RN(x)$ the rounding functions associated to these rounding direction attributes (assuming round to nearest even for $RN(x)$).

Let $\uparrow x$ denote nextUp(x), i.e., the least floating-point number that compares greater than x.

Let us also recall that *correct rounding* is required for the four elementary arithmetic operations and the square root by the above cited IEEE standards: an arithmetic operation is said to be correctly rounded if for any inputs its result is the infinitely precise result rounded according to the active rounding mode. Correct rounding makes arithmetic deterministic, provided all computations are done in the same format, which might be sometimes difficult to ensure [14]. Correct rounding allows one to design portable floating-point algorithms and to prove their correctness, as the results summarized in the next subsection.

1.2 Previous results

The Fast2Sum algorithm (Algorithm 1) was introduced by Dekker [8] in 1971, but the three operations of this algorithm already appeared in 1965 as a part of a summation algorithm, called "Compensated sum method," due to Kahan [1]. The following result is due to Dekker [8, 15].

Theorem 1 (Fast2Sum algorithm) *Assume a radix-β floating-point arithmetic, with $\beta \leq 3$, with subnormal numbers available, that provides correct rounding with rounding to nearest. Let a and b be finite floating-point numbers, both nonzero, such that the exponent of a is larger than or equal to that of b. The following algorithm computes floating-point numbers s and t such that $s = RN(a+b)$ and $s+t = a+b$ exactly.*

Algorithm 1 (Fast2Sum(a,b))

$$\begin{aligned} s &= RN(a+b); \\ z &= RN(s-a); \\ t &= RN(b-z); \end{aligned}$$

Note that instead of having information on the exponents, one may know that $|a| \geq |b|$, but in such a case, the condition of the theorem is fulfilled. Also, the condition $\beta \leq 3$ restricts the use of this algorithm, in practice, to binary arithmetic. That condition is necessary: consider, in radix 10, with precision $p = 4$, the case $a = b = 9999$. By applying Fast2Sum, we would get $s = RN(a+b) = 20000$, $z = RN(s-a) = 10000$, $t = RN(b-z) = -1$. This gives $s+t = 19999$, whereas $a+b = 19998$.

However, one can show that that if a wider internal format is available (one more digit of precision is enough), and if the computation of z is carried on using that wider format, then the condition $\beta \leq 3$ is no longer necessary. This might be useful in decimal arithmetic, when the target format is not the largest one that is available in hardware. We discuss the possible use of Fast2Sum in radix 10 in Section 3.

If no information on the relative orders of magnitude of a and b is available, or if the radix is larger than 3, there is an alternative algorithm due to Knuth [12] and Møller [13], called 2Sum.

Algorithm 2 (2Sum(a,b))

$$\begin{array}{rcl} s & = & RN(a+b); \\ b' & = & RN(s-a); \\ a' & = & RN(s-b'); \\ \delta_b & = & RN(b-b'); \\ \delta_a & = & RN(a-a'); \\ t & = & RN(\delta_a+\delta_b); \end{array}$$

2Sum requires 6 operations instead of 3 for the Fast2Sum algorithm, but on current pipelined architectures, a wrong branch prediction may cause the instruction pipeline to drain. As a consequence, using 2Sum instead of a comparison followed by Fast2Sum will usually result in much faster programs [16]. The names "2Sum" and "Fast2Sum" seem to have been coined by Shewchuk [19]. They are a particular case of what Rump calls "error-free transforms". We call these algorithms *error-free additions* in the sequel.

The IEEE 754-2008 standard [10] describes new operations with two floating-point numbers as operands:

- minNum and maxNum, which deliver respectively the minimum and the maximum;
- minNumMag, which delivers the one with the smaller magnitude (the minimum in case of equal magnitudes);
- maxNumMag, which delivers the one with the larger magnitude (the maximum in case of equal magnitudes).

The operations minNumMag and maxNumMag can be used to sort two floating-point numbers by order of magnitude. In radices less than or equal to three (or when a wider precision is available for computing z), this leads to the following alternative to the 2Sum algorithm.

Algorithm 3 (Mag2Sum(a,b))

$$\begin{array}{rcl} s & = & RN(a+b); \\ a' & = & maxNumMag(a,b); \\ b' & = & minNumMag(a,b); \\ z & = & RN(s-a'); \\ t & = & RN(b'-z); \end{array}$$

Algorithm Mag2Sum consists in sorting the inputs by magnitude before applying Fast2Sum. It requires 5 floating-point operations, but notice that the first three operations can be executed in parallel. Mag2Sum can already be implemented efficiently on the Itanium processor, thanks to the instructions famin and famax available on this architecture [7, p. 291]. Notice that, since it is based on the Fast2Sum algorithm, Algorithm Mag2Sum does not work in radices higher than 3. It therefore cannot be used in decimal arithmetic.

2 Algorithms 2Sum and Mag2Sum are minimal

In the following, we call an *RN-addition algorithm* an algorithm only based on additions and subtractions in the round-to-nearest mode: at step i the algorithm computes $x_i = RN(x_j \pm x_k)$, where x_j and x_k are either one of the input values or a previously computed value. An RN-addition algorithm must not perform any comparison or conditional branch, but may be enhanced with minNum, maxNum, minNumMag or maxNumMag as in Theorem 3.

For instance, 2Sum is an RN-addition algorithm that requires 6 floating-point operations. To estimate the performance of an algorithm, only counting the operations is a rough estimate. On modern architectures, pipelined arithmetic operators and the availability of several FPUs make it possible to perform some operations in parallel, provided they are independent. Hence the depth of the dependency graph of the instructions of the algorithm is an important criterion. In the case of Algorithm 2Sum, two operations only can be performed in parallel, $\delta_b = RN(b - b')$ and $\delta_a = RN(a - a')$. Hence the depth of Algorithm 2Sum is 5. In Algorithm Mag2Sum the first three operations can be executed in parallel, hence this algorithm has depth 3.

In this section, we address the following question: are there other RN-addition algorithms producing the same results as 2Sum, i.e., computing both $RN(a + b)$ and the rounding error $a + b - RN(a + b)$ for any floating-point inputs a and b, and that do not require more operations, or a dependence graph of larger depth?

We show the following result, proving that among the RN-addition algorithms, 2Sum is minimal in terms of number of operations as well as in terms of depth of the dependency graph.

Theorem 2 *Consider a binary arithmetic in precision $p \geq 2$. Among the RN-addition algorithms computing the same results as 2Sum on any inputs,*

- *each one requires at least* 6 *operations;*
- *each one with* 6 *operations reduces to 2Sum through straightforward transformations (symmetries, etc.);*
- *each one has depth at least* 5*.*

As previously mentioned an RN-addition algorithm can also be enhanced with minNum, maxNum, minNumMag and maxNumMag operations [10], which is the case for Algorithm Mag2Sum. The following result proves the minimality of this algorithm.

Theorem 3 *Consider a binary arithmetic in precision $p \geq 2$ and the set of all the RN-addition algorithms enhanced with minNum, maxNum, minNumMag and maxNumMag. Among all such algorithms computing the same results as 2Sum on any inputs,*

- *each one requires at least* 5 *operations;*
- *each one with* 5 *operations reduces to Mag2Sum;*
- *each one has depth at least* 3*.*

Proof: To prove Theorems 2 and 3, we proceeded as follows in precisions p from 2 to 12; we will also show that with our method, the results for precisions $p > 12$ can be deduced from those for precision $p = 12$.

- For the first two items of Theorem 2, we enumerated all the possible RN-addition algorithms with 6 additions/subtractions or less, and each algorithm was tried on 3 ordered pairs of well chosen inputs a and b. In fact, since some algorithms can reduce to other ones through obvious transformations and we are interested in only one among such algorithms, we required some constraints. For instance, we regarded different operation orderings (associated with the same computation DAG) as equivalent. Moreover, since the value $s = RN(a + b)$ must be computed (and this operation does not depend on intermediate values), we assumed that this was the first operation. Indeed, if an RN-addition algorithm computed s in some other way, the corresponding operation could be safely replaced by

$s = RN(a + b)$, then moved to the beginning; this transformation may not be regarded as really obvious, but at least, it does not increase the depth. Similarly we took into account the fact that $RN(x + y) = RN(y + x)$. We also discarded useless operations, such as $RN(x - x)$. The only algorithm that delivered the correct results was 2Sum.

- For the first two items of Theorem 3, we proceeded in a similar way on algorithms with 5 operations or fewer amongst the 6 allowed operations, but with 4 pairs instead of 3.
- For the third item of Theorem 2 (depth), we enumerated all possible results at depth 4 or less on 3 or 4 pairs of well chosen inputs a and b, showing that the expected results could not be obtained on these 3 or 4 pairs at the same time.
- For the third item of Theorem 3 (depth), we could have used the same method as above. However we would like to give a human-checkable proof (valid for any precision $p \geq 2$). Let us assume that there exists an acceptable algorithm whose depth is less than or equal to 2. In the following, we consider the following constraints on the inputs: $a = 2^e$ for some integer e (not too large to avoid overflows), and $0 \leq b - a \leq a \leq b \leq a + b$, so that a minNum, maxNum, minNumMag, or maxNumMag operation at depth at most 2 can statically be simplified to one of its arguments. Moreover, from the cases $b = a$ (which yields $t = 0$) and $b = \uparrow a$ (which yields $t = \text{ulp}(a)$)—we remind that $\uparrow(x) = \text{NextUp}(x)$ is the floating-point successor of x—, we deduce that the expression giving t still depends on both a and b. The expressions $RN(a+b)$, $RN(a-b)$, and $RN(b-a)$ are not possible for t, since they are invalid on $b = 2a = 2^{e+1}$. If $b = a$, then the depth-1 operations are exact and since one must have $t = 0$, the depth-2 operation must be exact too. So, without taking the rounding into account, the expression for t has the form $m.a - m.b$ for some integer m. Therefore, since the computation tree must have at least two addition/subtraction nodes, for parity reasons, it must have exactly three of them. This means that the algorithm does not use the minNum, maxNum, minNumMag, and maxNumMag operations at all. Then from either Theorem 2 or with a manual check of the (up to) 27 remaining cases, we can deduce that there are no such acceptable algorithms whose depth is less than or equal to 2.

The pairs were chosen as follows:

$$\begin{array}{lllclll} a_1 & = & \uparrow 8 & & b_1 & = & \uparrow\uparrow\uparrow 1 \\ a_2 & = & \uparrow\uparrow\uparrow\uparrow\uparrow 1 & & b_2 & = & \uparrow 8 \\ a_3 & = & 3 & & b_3 & = & \uparrow 3 \end{array}$$

and when needed, the fourth pair was:

- for Theorem 3, $a_4 = -a_1$ and $b_4 = -b_1$;
- for Theorem 2 (depth) and $p = 2$, $a_4 = 1$ and $b_4 = 6$;
- for Theorem 2 (depth) and $p = 3$, $a_4 = 10$ and $b_4 = 1$.

In precision $p \geq 4$, the pairs can be written as:

$$\begin{array}{ll} a_1 = 8 + 8\varepsilon_p & b_1 = 1 + 3\varepsilon_p \\ a_2 = 1 + 5\varepsilon_p & b_2 = 8 + 8\varepsilon_p \\ a_3 = 3 & b_3 = 3 + 2\varepsilon_p \\ a_4 = -8 + (-8)\varepsilon_p & b_1 = -1 + (-3)\varepsilon_p \end{array}$$

where $\varepsilon_p = \text{ulp}(1) = 2^{1-p}$. The particular form of these pairs is not important to obtain the results in the tested precisions, but the way of deducing results for larger precisions from the results for precision 12 is based on it. Indeed we will prove that for $p \geq 12$:

- each value computed by any algorithm run in precision p has the form $u + v\varepsilon_p$, where u and v are "small" integers that do not depend on the precision p;
- since the integers u and v are small enough (see below), two values $u_1 + v_1\varepsilon_p$ and $u_2 + v_2\varepsilon_p$ are equal if and only if $u_1 = u_2$ and $v_1 = v_2$.

Let us consider a computation tree of maximum depth n. Assume that $p \geq n + 6$. Under the conditions of the theorems, $n \leq 6$, so that the results will be proved for all $p \geq 6 + 6 = 12$.

First let us introduce some notation associated with a node. Each node has a value of the form $u + v\varepsilon_p$. Pairs (u, v) are ordered by: $(u_1, v_1) < (u_2, v_2)$ if and only if $u_1 < u_2 \ \vee \ (u_1 = u_2 \ \wedge \ v_1 < v_2)$. If the node inputs are $u_i + v_i\varepsilon_p$ and $u_j + v_j\varepsilon_p$, we define the pair $(u_k, \tilde{v}_k)$ as follows:

- add: $u_k = u_i + u_j$ and $\tilde{v}_k = v_i + v_j$;
- sub: $u_k = u_i - u_j$ and $\tilde{v}_k = v_i - v_j$;
- minNum: $(u_k, \tilde{v}_k) = \min((u_i, v_i), (u_j, v_j))$;
- maxNum: $(u_k, \tilde{v}_k) = \max((u_i, v_i), (u_j, v_j))$;
- minNumMag: $(u_k, \tilde{v}_k)$ is (u_i, v_i) if

$$\begin{array}{l} |u_i| < |u_j| \ \vee \\ (u_i = u_j = 0 \ \wedge \ |v_i| < |v_j|) \ \vee \\ (|u_i| = |u_j| \ \wedge \ v_i \times \operatorname{sign}(u_i) < v_j \times \operatorname{sign}(u_j)) \ \vee \\ (|u_i| = |u_j| \ \wedge \ |v_i| = |v_j| \ \wedge \ u_i v_i = u_j v_j \ \wedge \ (u_i, v_i) < (u_j, v_j)), \end{array}$$

  else (u_j, v_j);
- maxNumMag: similar to minNumMag but changing the inequalities;

and we define v_k by: $RN(u_k + \tilde{v}_k\varepsilon_p) = u_k + v_k\varepsilon_p$, giving the *rounded* output pair (u_k, v_k) associated with the node.

Let us prove the following properties by induction on the depth d of a node:

- The pair $(u_k, \tilde{v}_k)$ represents the exact value $u_k + \tilde{v}_k\varepsilon_p$ (i.e., the value of the operation before rounding).
- Unicity of the representation: $|v_k|\,\varepsilon_p < 1/2$.
- $|u_k| \leq 2^{d+3}$ and $|v_k| \leq 2^{d+3}$.
- The values u_k and v_k are integers that do not depend on p.

Any initial value (depth 0) has the form $u + v\varepsilon_p$, where u and v are integers that do not depend on p, such that $|u| \leq 8 = 2^3$ and $|v| \leq 8 = 2^3$. Also, $|v|\,\varepsilon_p \leq 2^{3+1-p} \leq 2^{-n-2} < 1/2$.

Now let us prove the properties for some node, assuming that they are satisfied for both inputs (u_i, v_i) and (u_j, v_j) of this node.

- The first property is proved as follows. For the addition, $(u_i + v_i\varepsilon_p) + (u_j + v_j\varepsilon_p) = (u_i + u_j) + (v_i + v_j)\varepsilon_p = u_k + \tilde{v}_k\varepsilon_p$. The case of the subtraction is similar. Since $|v_i|\,\varepsilon_p < 1/2$ and $|v_j|\,\varepsilon_p < 1/2$ and both u_i and u_j are integers, the ordering of the real values $u_i + v_i\varepsilon_p$ and $u_j + v_j\varepsilon_p$ is the same as the ordering of the pairs (u_i, v_i) and (u_j, v_j) with the above definition; this proves the property for minNum and maxNum. The property for minNumMag and maxNumMag follows from the corresponding definition of $(u_k, \tilde{v}_k)$, where the various cases are checked separately; details are left to the reader.

- From the definition of u_k for each operation, u_k is an integer and one has: $|u_k| \leq 2\max(|u_i|, |u_j|)$. Since the depth of each input is less or equal to $d-1$, it follows that

$$|u_k| \leq 2 \cdot 2^{d-1+3} = 2^{d+3}.$$

Moreover the definition of u_k does not depend on p. This proves the properties on u_k.

- For the same reasons, $|\tilde{v}_k| \leq 2^{d+3}$ and $\tilde{v}_k$ is an integer that does not depend on p. We need to prove that these properties are still satisfied after rounding, i.e., for v_k. If $u_k = 0$ (which does not depend on p), then $v_k = \tilde{v}_k$ (since $|\tilde{v}_k| \leq 2^p$), which proves the properties. Now let us assume that $u_k \neq 0$. Let E be the exponent of $u_k + \tilde{v}_k\varepsilon_p$, i.e. $2^E \leq |u_k + \tilde{v}_k\varepsilon_p| < 2^{E+1}$. Since $|\tilde{v}_k\varepsilon_p| \leq 2^{d+3+1-p} \leq 2^{n+4-p} \leq 1/2$ and u_k is an integer, E depends only on u_k and the sign of $\tilde{v}_k$ (it is the exponent of u_k, minus 1 if $|u_k|$ is a power of 2 and $u_k\tilde{v}_k < 0$); thus is does not depend on p. The significand of $u_k + \tilde{v}_k\varepsilon_p$ as a number in $\left[2^{p-1}, 2^p\right[$ is $(u_k + \tilde{v}_k\varepsilon_p)2^{p-1-E} = u_k 2^{p-1-E} + \tilde{v}_k 2^{-E}$. The rounding of $u_k + \tilde{v}_k\varepsilon_p$ to $u_k + v_k\varepsilon_p$ can be defined as the rounding of its significand to an integer (which will be equal to $u_k 2^{p-1-E} + v_k 2^{-E}$). Since $2^E \leq |u_k| \leq 2^{d+3}$, one has $2^{p-1-E} \geq 2^{p-1-d-3} \geq 2$, so that $u_k 2^{p-1-E}$ is an even integer, thus will not have any influence on the relative rounding error, defined as $\delta = (\tilde{v}_k - v_k)2^{-E}$. If $\{x\}$ denotes the nonnegative fractional part of x, then $\delta = \left\{\tilde{v}_k 2^{-E}\right\} - \Delta$, where $\Delta = 0$ if the rounding is done downward and $\Delta = 1$ if the rounding is done upward; by definition of the rounding-to-nearest with the even rounding rule, $\Delta = 1$ if and only if one of the following two conditions holds:

  1. $\left\{\tilde{v}_k 2^{-E}\right\} > 1/2$;
  2. $\left\{\tilde{v}_k 2^{-E}\right\} = 1/2$ and $\left\lfloor \tilde{v}_k 2^{-E} \right\rfloor$ is an odd integer.

  As $\tilde{v}_k$ and E do not depend on p, the value of δ does not depend on p, and the value of v_k does not depend on p either. Moreover $v_k 2^{-E}$ is equal to $\left\lfloor \tilde{v}_k 2^{-E} \right\rfloor$ or to $\left\lceil \tilde{v}_k 2^{-E} \right\rceil$, so that v_k is an integer. And since $|\tilde{v}_k| \leq 2^{d+3}$ and $E \leq d+3$, it follows that $\left|\tilde{v}_k 2^{-E}\right| \leq 2^{d+3-E}$, which is an integer. Hence $\left|v_k 2^{-E}\right| \leq 2^{d+3-E}$, and $|v_k| \leq 2^{d+3}$. Then $|v_k|\,\varepsilon_p \leq 2^{d+3+1-p} \leq 2^{-2} < 1/2$.

Now let us show that if an algorithm $\mathcal{A}$ (or computation tree) is excluded for some precision $p \geq 12$ because at least on one of the input pairs (a_p, b_p) at this precision, $\mathcal{A}$ does not yield the expected result, then $\mathcal{A}$ must also be excluded for all the other precisions $q \geq 12$. In precision p, let $t_p = a_p + b_p - RN(a_p + b_p) = u + v\varepsilon_p$ be the expected result together with its associated pair (u, v) as defined above, and let $t'_p = u' + v'\varepsilon_p$ the result obtained by running $\mathcal{A}$, together with its associated pair (u', v'). By hypothesis, $t'_p \neq t_p$, so that $(u', v') \neq (u, v)$. And since in precision q, any real can have at most one (u, v) representation satisfying $|v|\,\varepsilon_q < 1/2$, $(u', v') \neq (u, v)$ implies $t'_q = u' + v'\varepsilon_q \neq u + v\varepsilon_q = t_q$. ■

The C programs used to prove all these results are based on the MPFR library [9] (in order to control the precision and the roundings) and can be found on `http://hal.inria.fr/inria-00475279`.

3 On the use of Fast2Sum in radix-10 arithmetic

As explained in Section 1.2, the Fast2Sum algorithm (Algorithm 1) is proven only when the radix is less than or equal to 3. Indeed, there are counterexamples in radix 10 for this algorithm (we gave one in the introduction). And yet, we are going to show that using Fast2Sum may be of interest, since the very few cases for which it does not return the right answer can easily be circumscribed.

3.1 An analysis of Fast2Sum in radix 10

In this section, we consider a radix-10 floating-point system of precision p. We also consider two floating-point numbers a and b, *and we will assume* $|a| \geq |b|$. Without loss of generality, we assume $a > 0$ (just for the sake of simplifying the proofs). Our way of analyzing Fast2Sum will mainly consist in considering Dekker's proof (for radix 2) and locating where it does not generalize to decimal arithmetic.

Notice that when the result of a floating-point addition or subtraction is a subnormal, that operation is performed exactly. Due to this, in the following, we assume no underflow in the operations of the algorithm (in case of an overflow, our previous observation implies that the algorithm is correct).

3.1.1 First operation: $s \leftarrow RN(a + b)$

Assume that $a = M_a \cdot 10^{e_a - p + 1}$, $b = M_b \cdot 10^{e_b - p + 1}$, and $s = M_s \cdot 10^{e_s - p + 1}$, where M_a, M_b, M_s, e_a, e_b, and e_s are integers, with

$$10^{p-1} \leq M_a, |M_b|, |M_s| \leq 10^p - 1.$$

Notice that the IEEE 754-2008 standard for floating-point arithmetic does not define the significand and exponent of a decimal number in a unique way (some numbers have several representations, whose set is called a *cohort*). However, in this paper, we will use, without any loss of generality, the values M_a, M_b, M_s, e_a, e_b, and e_s that satisfy the above given boundings: what is important is the set of values, not the representations. In the paper, M_a and e_a are more a notation that a representation (i.e., their values do not need to match what is stored internally in the processor). Define $\delta = e_a - e_b$.

Since we obviously have $0 \leq a + b \leq 2a < 10a$, e_s is necessarily less than or equal to $e_a + 1$. We now consider two cases.

- **First case:** $e_s = e_a + 1$, in that case,

  $$M_s = \left\lceil \frac{M_a}{10} + \frac{M_b}{10^{\delta+1}} \right\rfloor,$$

  where $\lceil u \rfloor$ is the integer nearest u (with any of the two possible choices in case of a tie, so that our result applies for the default roundTiesToEven rounding attribute, as well as for the roundTiesToAway attribute defined by IEEE 754-2008). Define $\mu = 10M_s - M_a$ (notice that μ is an integer), from

  $$\frac{M_a}{10} + \frac{M_b}{10^{\delta+1}} - \frac{1}{2} \leq M_s \leq \frac{M_a}{10} + \frac{M_b}{10^{\delta+1}} + \frac{1}{2},$$

  we easily deduce

  $$\frac{M_b}{10^{\delta}} - 5 \leq \mu \leq \frac{M_b}{10^{\delta}} + 5,$$

  which implies

  $$|\mu| \leq \left| \frac{M_b}{10^{\delta}} \right| + 5.$$

  From this we conclude that either $s - a$ is exactly representable (with exponent e_a and significand μ), or we are in the case

  $$|M_b| \in \{10^p - 4, 10^p - 3, 10^p - 2, 10^p - 1\} \quad \text{and} \quad \delta = 0.$$

  Notice that if $s - a$ is exactly representable, then it will be computed exactly by the second operation.

- **Second case:** $e_s \le e_a$. We have
$$a + b = \left(10^{\delta} M_a + M_b\right) \cdot 10^{e_b - p + 1}.$$
If $e_s \le e_b$ then $s = a+b$ exactly, since s is obtained by rounding $a+b$ to the nearest multiple of 10^{e_s-p+1}, which divides 10^{e_b-p+1}. Hence, $s - a = b$, and $s - a$ is exactly representable.
If $e_s > e_b$, define $\delta_2 = e_s - e_b$. We have
$$s = \left\lceil 10^{\delta-\delta_2} M_a + 10^{-\delta_2} M_b \right\rfloor \cdot 10^{e_s-p+1},$$
so that
$$\left(10^{-\delta_2} M_b - \frac{1}{2}\right) \cdot 10^{e_s-p+1} \le s - a \le \left(10^{-\delta_2} M_b + \frac{1}{2}\right) \cdot 10^{e_s-p+1},$$
which implies
$$|s - a| \le \left(10^{-\delta_2} |M_b| + \frac{1}{2}\right) \cdot 10^{e_s-p+1}.$$
Moreover, $e_s \le e_a \Rightarrow s - a$ is a multiple of 10^{e_s-p+1}, say $s - a = K \cdot 10^{e_s-p+1}$. We get
$$|K| \le 10^{-\delta_2} |M_b| + \frac{1}{2} \le \frac{10^p - 1}{10} + \frac{1}{2} \le 10^p - 1,$$
therefore $s - a$ is exactly representable.

We therefore deduce the following property on the value s computed after the first step.

Property 1 *The value s computed by the first operation of Algorithm 1 satisfies:*

- *either $s - a$ is exactly representable,*
- *or we simultaneously have*
$$\begin{cases} |M_b| & \in & \{10^p - 4, 10^p - 3, 10^p - 2, 10^p - 1\}, \\ e_b & = & e_a, \\ e_s & = & e_a + 1. \end{cases}$$

3.1.2 Second and third operations: $z \leftarrow RN(s - a)$ and $t \leftarrow RN(b - z)$

That second operation is more easily handled. It suffices to notice that when $s - a$ is exactly representable, then $z = s - a$ exactly, so that
$$b - z = b - (s - a) = (a + b) - s.$$
This means that when $s - a$ is exactly representable, $b - z$ is the error of the floating-point operation $s \leftarrow RN(a + b)$. Since that error is exactly representable (see for instance [6]), it is computed exactly, so that $t = (a + b) - s$.

Therefore, we deduce the following result.

Theorem 4 *If a and b are radix-10 floating-point numbers of precision p, with $|a| \ge |b|$, then the value s computed by the first operation of Algorithm 1 satisfies:*

- *either t is the error of the floating-point addition $a + b$, which means that $s + t = a + b$ exactly,*
- *or we simultaneously have*
$$\begin{cases} |M_b| & \in & \{10^p - 4, 10^p - 3, 10^p - 2, 10^p - 1\}, \\ e_b & = & e_a, \\ e_s & = & e_a + 1. \end{cases}$$

3.2 Some consequences

Let us analyze the few cases for which Algorithm 1 may not work. Notice that since a and b have the same exponent, $|a| \geq |b|$ implies $|M_a| \geq |M_b|$. Also, $|M_a| \leq 10^p - 1$. Hence, when $|M_b| \in \{10^p-4, 10^p-3, 10^p-2, 10^p-1\}$, the possible values of $|M_a|$ are limited to

- 4 cases for $|M_b| = 10^p - 4$;
- 3 cases for $|M_b| = 10^p - 3$;
- 2 cases for $|M_b| = 10^p - 2$;
- 1 case for $|M_b| = 10^p - 1$.

Also, in these cases, when a and b do not have the same sign, Algorithm 1 obviously works (by Sterbenz Lemma, $s = a + b$ exactly, so that $z = b$ and $t = 0$). Therefore, we can assume that a and b have the same sign. Without loss of generality we assume they are positive. It now suffices to check Algorithm 1 with the 10 possible cases. The results are listed in Table 1.

M_a	$M_b = 10^p - 4$	$M_b = 10^p - 3$	$M_b = 10^p - 2$	$M_b = 10^p - 1$
$10^p - 4$	OK	N/A	N/A	N/A
$10^p - 3$	OK	OK	N/A	N/A
$10^p - 2$	OK	Wrong: $t = -3$, $(a+b) - s = -5$	Wrong: $t = -2$, $(a+b) - s = -4$	N/A
$10^p - 1$	Wrong: $t = -4$, $(a+b) - s = -5$	Wrong: $t = -3$, $(a+b) - s = -4$	Wrong: $t = -2$, $(a+b) - s = -3$	Wrong: $t = -1$, $(a+b) - s = -2$

Table 1: Algorithm 1 is checked in the cases $M_b \in \{10^p-4, 10^p-3, 10^p-2, 10^p-1\}$ and $M_b \leq M_a \leq 10^p - 1$.

From these results, we notice that there are only 6 cases where Algorithm 1 does not work. This leads us to the following result.

Theorem 5 *If a and b are radix-10 floating-point numbers of precision p, with $|a| \geq |b|$, then Algorithm 1 always works (i.e., we always have $s + t = a + b$, with $s = RN(a + b)$), unless a and b have the same sign, the same exponent, and their significands M_a and M_b satisfy:*

- $|M_a| = 10^p - 1$ *and* $|M_b| \geq 10^p - 4$*;*
- *or* $|M_a| = 10^p - 2$ *and* $|M_b| \geq 10^p - 3$*.*

Notice that even in the few (6) cases where Algorithm 1 provides a wrong result, the value of t it returns remains an interesting "correcting term" that can be useful in summation algorithms, since $s + t$ is always closer to $a + b$ than s.

Theorem 5 shows that Algorithm Fast2Sum can safely be used in several cases. An example is addition of a constant: for instance, computations of the form "$a \pm 1$", quite frequent, can safely be performed whenever $|a| \geq 1$.

Another very frequent case is when one of the operands is known to be significantly larger than the other one (e.g., we add a small correcting term to some estimate). Concerning that case, an immediate consequence of Theorem 5 is the following result.

Theorem 6 *If a and b are radix-10, precision-p floating-point numbers, and $|b| < \omega_p |a|$, with*

$$\omega_p = \frac{10^p - 4}{10^p - 1},$$

then Algorithm 1 returns a correct result.

For instance, in the decimal32 format of the IEEE 754-2008 standard ($p = 7$), $\omega_p = 0.9999996999999699\cdots$

4 Halving and computing the average of two numbers in radix 10

In radix 2 floating-point arithmetic, if s is a floating-point number, then $s/2$ is computed exactly, provided that no underflow occurs. This is not always the case in radix 10.

Consider a radix-10 number s:

$$s = \pm S \cdot 10^{e-p+1},$$

where S is an integer, $10^{p-1} \le S \le 10^p - 1$, and consider the following two cases:

- if $S < 2 \cdot 10^{p-1}$, then $5S$ is less than 10^p, hence $s/2$ is exactly representable as $5S \cdot 10^{e-p}$: it will be computed exactly, with any rounding mode;
- if $S \ge 2 \cdot 10^{p-1}$, then if S is even, $s/2$ is obviously exactly representable as $S/2 \cdot 10^{e-p+1}$. If S is odd, let k be the integer such that $S = 2k + 1$. From

  $$\frac{s}{2} = \left(k + \frac{1}{2}\right) \cdot 10^{e-p+1},$$

  we deduce that $s/2$ is a rounding breakpoint for the round-to-nearest mode. Therefore (assuming round to nearest *even*), the computed value $RN(s/2)$ will be $k \cdot 10^{e-p+1}$ if k is even, and $(k+1) \cdot 10^{e-p+1}$ otherwise. Let t be that computed result. Notice that $2t$ is either $2k \cdot 10^{e-p+1}$ or $(2k+2) \cdot 10^{e-p+1}$: in any case it is exactly representable, hence it is computed exactly. The same holds for $s - 2t$, which will be $\pm 10^{e-p+1}$. This last result is straightforwardly exactly divisible by 2. We therefore deduce that the sequence of computations

  $$\begin{array}{lcl} t & = & RN(0.5 \times s); \\ twot & = & RN(2 \times t); \\ delta & = & RN(s - twot); \\ r & = & RN(0.5 \times delta); \end{array}$$

  will return a value r equal to the error of the floating-point division of s by two (notice that 0.5 is exactly representable in decimal arithmetic).

  Now, one may easily notice that in all the other cases (that is, when t is exactly $s/2$), the same sequence of operations will return a zero.

This gives us a new error-free transform:

Algorithm 4 (Half-and-error, for radix-10 arithmetic)

$$\begin{array}{lcl} t & = & RN(0.5 \times s); \\ twot & = & RN(2 \times t); \\ delta & = & RN(s - twot); \\ r & = & RN(0.5 \times delta); \end{array}$$

The following theorem summarizes what we have discussed:

Theorem 7 *In radix-*10 *arithmetic, provided that no underflow occurs, Algorithm 4 returns two values t and r such that $t = RN(s/2)$, and $t + r = s/2$ exactly. Also, r is always either* 0 *or* $\pm\frac{1}{2}\mathrm{ulp}(s/2)$.

Let us give an example. Assume a precision-4 radix-10 floating-point system, and let us execute Algorithm 4 with $s = 9367$. Since $s/2 = 4683.5$, we find $t = 4684$. Hence, $\mathit{twot} = 9368$, $\delta = -1$ and $r = -0.5$. We immediately see that we have $t + r = s/2$.

Now, let us focus on the computation of the average value of two floating-point numbers a and b, namely,

$$\mu = \frac{a+b}{2}.$$

Again, in radix-2 arithmetic, the "naive" method that consists in computing $s = RN(a + b)$ and $m = RN(s/2)$ (or rather, equivalently, $m = RN(0.5 \times s)$) will obviously give $m = RN(\mu)$, unless the addition overflows or the division by 2 underflows. This is not the case in radix 10. Consider a toy decimal system of precision $p = 3$, and the two input numbers $a = 1.09$ and $b = 0.195$. We get $s = RN(a + b) = 1.28$, so that $m = RN(s/2) = 0.640$, whereas the exact average value μ is 0.6425: we have an error of 2.5 units in the last place (one can easily show that this is the largest possible error, in the absence of over/underflow).

If a larger precision is available for performing the internal calculations, then we get a better result: if now s is $a + b$ rounded to the nearest in precision $p + d$, then the average value is computed with an error bounded by

$$\left(\frac{1}{2} + \frac{5}{2} \cdot 10^{-d}\right)$$

units in the last place.

If no larger precision is available, we may need to use algorithms such as 2Sum and Half-and-error. Consider for instance,

Algorithm 5 (Average-Radix-10)

$$\begin{array}{lcl}
(s, r) & = & \mathit{2Sum}(a, b); \\
(m_h, m_\ell) & = & \mathit{Half\text{-}and\text{-}error}(s); \\
\rho & = & RN(m_\ell + RN(0.5 \times r)); \\
m & = & RN(m_h + \rho);
\end{array}$$

This algorithm requires 13 arithmetic operations (12 if 2Sum is replaced by Mag2Sum). Using the following properties:

- $s + r = a + b$, with $|r| < \frac{1}{2} \cdot s \cdot 10^{-p+1}$,
- $m_h + m_\ell = s/2$ and $|m_\ell| < \frac{1}{2} \cdot (s/2) \cdot 10^{-p+1}$,
- $t = RN(0.5 \times r) = \frac{r}{2} + \epsilon_1$, with $|\epsilon_1| < (1/2) \cdot (r/2) \cdot 10^{-p+1}$,
- $RN(m_\ell + t) = m_\ell + t + \epsilon_2$, with $|\epsilon_2| < (1/2) \cdot (m_\ell + t) \cdot 10^{-p+1}$,

one can show, after some calculations, that

$$m = RN\left(\frac{a+b}{2} + \epsilon\right),$$

where

$$|\epsilon| \leq |\epsilon_1| + |\epsilon_2| < \frac{3s}{8}10^{-2p+2} + \frac{s}{16}10^{-3p+3}.$$

Still in the "toy example" ($p = 3$, $a = 1.09$, $b = 0.195$) considered above, for which the naive method gave an error of 2.5ulp, we now get $s = 1.28$, $r = 0.005$, $m_h = 0.64$, $m_\ell = 0$, $\rho = 0.0025$, and $m = 0.642$, which is equal to $RN((a + b)/2)$.

This algorithm requires many operations—probably too many for practical purposes. If a and b are sufficiently close, there is a much better solution:

Algorithm 6 (Average value, when a and b are close)

$$\begin{array}{rcl} d & = & RN(a-b); \\ h & = & RN(0.5 \times d); \\ m & = & RN(a-h); \end{array}$$

Theorem 8 *If a and b are two decimal floating-point numbers of the same sign such that $0 \leq b \leq a \leq 2b$ or $2b \leq a \leq b \leq 0$, then the value m returned by Algorithm 6 satisfies $m = RN((a+b)/2)$.*

Proof: Without loss of generality we assume that $0 \leq b \leq a \leq 2b$. Also assume that a and b have been scaled to integers without common factors of 10, where b has at most p digits. By Sterbenz Lemma we have $RN(a - b) = a - b$. The proof is now split in two cases:

$a - b$ is even: $(a - b)/2$ is exactly representable. Hence $m = RN(a - RN((a - b)/2)) = RN((a + b)/2)$.

$a - b$ is odd: $RN((a-b)/2) = (a-b)/2+\delta$ ($\delta = \pm 1/2$) is exactly computable and representable as a p-digit even integer (since mid-points round to even). Now assume that a is a $p + k$ digit integer, with k minimal. Then it follows that $k \leq 1$. Consider:

$$RN((a - b)/2) = (a - b)/2 + \delta,$$

from which it follows that

$$a - RN((a - b)/2) = (a + b)/2 - \delta \; (= (a + b \pm 1)/2),$$

which is an integer representable on at most $p + k$ digits (since a and $RN((a - b)/2)$ have the same sign and both are integers representable on $p + k$ digits).

If $k = 0$, then obviously $m = RN(a - RN((a - b)/2)) = RN((a + b)/2)$.

If $k = 1$, then a is even, hence $a - RN((a - b)/2)$ is even, thus not a mid point. Hence rounding to p digits yields $m = RN(a - RN((a - b)/2)) = RN((a + b)/2)$. ∎

Note that the proof holds for any even radix.

5 On the impossibility of computing a round-to-nearest sum

In this section, we are interested in the computation of the sum of n floating-point numbers, correctly rounded to nearest. We prove the following result.

Theorem 9 *Let $a_1, a_2, \ldots, a_n$ be $n \geq 3$ floating-point numbers of the same format. Assuming an* unbounded exponent range*, and assuming that the radix of the floating-point system is even, an RN-addition algorithm cannot always return $RN(a_1 + a_2 + \cdots + a_n)$.*

If there exists an RN-addition algorithm to compute the round-to-nearest sum of n floating-point numbers, with $n \geq 3$, then this algorithm must also compute the round-to-nearest sum of 3 floating-point values. As a consequence we only consider the case $n = 3$ in the proof of this theorem. We show how to construct for any RN-algorithm a set of input data such that the result computed by the algorithm differs from the round-to-nearest result.

Proof of Theorem 9: Assume a radix-β arithmetic, where β is even.. An RN-addition algorithm can be represented by a directed acyclic graph[1] (DAG) whose nodes are the arithmetic operations. Given such an algorithm, let r be the depth of its associated graph. First we consider the input values a_1, a_2, a_3 defined as follows.

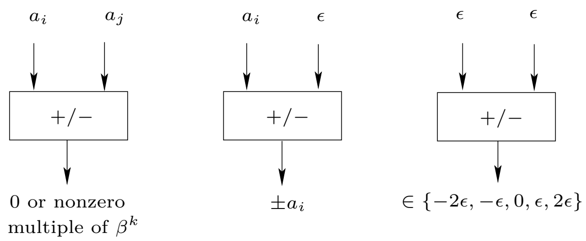
- $a_1 = \beta^{k+p}$ and $a_2 = \left(\frac{\beta}{2}\right)\beta^k$: For any[2] integer k, a_1 and a_2 are two nonzero multiples of β^k whose sum is the exact middle of two consecutive floating-point numbers;
- $a_3 = \varepsilon$, with $0 \leq \beta^{r-1}|\varepsilon| \leq \beta^{k-p-1}$ for $r \geq 1$.

Note that when $\varepsilon \neq 0$,

$$RN\,(a_1 + a_2 + a_3) = \begin{cases} RD(a_1 + a_2 + a_3) & \text{if} \quad \varepsilon < 0 \\ RU(a_1 + a_2 + a_3) & \text{if} \quad \varepsilon > 0, \end{cases}$$

where we may also conclude that $RN\,(a_1+a_2+a_3) \neq 0$.

The various computations that can be performed "at depth 1", i.e., immediately from the entries of the algorithm are illustrated below. The value of ε is so small that after rounding to nearest, every operation with ε in one of its entries will return the same value as if ε were zero, unless the other entry is 0 or ε.



An immediate consequence is that after these computations "at depth 1", the possible available variables are nonzero multiples of β^k that are the same as if ε were 0, and values taken from $\mathcal{S}_1 = \{-2\varepsilon, -\varepsilon, 0, \varepsilon, 2\varepsilon\}$. By induction one easily shows that the available variables after a computation of depth m are either nonzero multiples of β^k that are the same as if ε were 0 or values taken from $\mathcal{S}_m = \{-2^m\varepsilon, \cdots, 0, \cdots, +2^m\varepsilon\}$.

Now, consider the very last addition/subtraction, at depth r in the DAG of the RN-addition algorithm. If at least one of the inputs of this last operation is a nonzero multiple of β^k that is the same as if ε were 0, then the other input is either also a nonzero multiple of β^k or a value belonging to $\mathcal{S}_{r-1} = \{-2^{r-1}\varepsilon, \cdots, 0, \cdots, +2^{r-1}\varepsilon\}$. In both cases the result does not depend on the sign of ε, hence it is always possible to choose the sign of ε so that the round-to-nearest result differs from the computed one. If both entries of the last operation belong to $\mathcal{S}_{r-1}$, then

[1] Such an algorithm cannot have "while" loops, since tests are prohibited. It may have "for" loops that can be unrolled.

[2] Here k is arbitrary. When considering a limited exponent range, we have to assume that $k + p$ is less than the maximum exponent.

the result belongs to $\mathcal{S}_r = \{-2^r\varepsilon, \cdots, 0, \cdots, +2^r\varepsilon\}$. If one sets $\varepsilon = 0$ then the computed result is 0, contradicting the fact that the round-to-nearest sum must be nonzero. ∎

In the proof of Theorem 9, it was necessary to assume an unbounded exponent range to make sure that with a computational graph of depth r, we can always build an ε so small that $2^{r-1}\varepsilon$ vanishes when added to any multiple of β^k. This constraint can be transformed into a constraint on r related to the extremal exponents $e_{\min}$ and $e_{\max}$ of the floating-point system. For instance, in radix 2, assuming $\varepsilon = \pm 2^{e_{\min}}$ and $a_1 = 2^{k+p} = 2^{e_{\max}}$, the inequality $2^{r-1}|\varepsilon| \leq 2^{k-p-1}$ gives the following theorem.

Theorem 10 *Let $a_1, a_2, \ldots, a_n$ be $n \geq 3$ floating-point numbers of the same binary format. Assuming the extremal exponents of the floating-point format are $e_{\min}$ and $e_{\max}$, an RN-addition algorithm of depth r cannot always return $RN(a_1 + a_2 + \cdots + a_n)$ as soon as*

$$r \leq e_{\max} - e_{\min} - 2p.$$

For instance, with the IEEE 754-1985 double precision format ($e_{\min} = -1022$, $e_{\max} = 1023$, $p = 53$), Theorem 10 shows that an RN-addition algorithm able to always evaluate the round-to-nearest sum of at least 3 floating-point numbers (if such an algorithm exists!) must have depth at least 1939.

6 Correctly-rounded sums of three floating-point numbers

We have proved in the previous section that there exist no RN-addition algorithms of acceptable size to compute the round-to-nearest sum of $n \geq 3$ floating-point values. In [5], Boldo and Melquiond presented an algorithm to compute $RN(a + b + c)$ using a round-to-odd addition. Rounding to odd is defined as follows:

- if x is a floating-point number, then $RO(x) = x$;
- otherwise, $RO(x)$ is the value among $RD(x)$ and $RU(x)$ whose least significant digit is odd.

The algorithm of Boldo and Melquiond for computing of $RN(a + b + c)$ is depicted on Fig. 1. Boldo and Melquiond proved their algorithm in radix 2, yet one can check that it also works in radix 10.

Rounding to odd is not a rounding mode available on current architectures, hence a software emulation is proposed in [5] for radix 2: this software emulation requires accesses to the binary representation of the floating-point numbers and conditional branches, both of which are costly on pipelined architectures.

In the next section, we propose a new algorithm for simulating the round-to-odd addition of two floating-point values. This algorithm uses only available IEEE-754 rounding modes and a multiplication by the constant 0.5 (we will use that multiplication in a case where it is exact), and can be used to avoid access to the binary representation of the floating-point numbers and conditional branches in the computation of $RN(a + b + c)$ with the Boldo-Melquiond algorithm. We also study a modified version of the Boldo-Melquiond algorithm to compute $DR(a + b + c)$, where DR denotes any of the IEEE-754 directed rounding modes.

6.1 A new method for rounding to odd

If we allow the multiplication by the constant 0.5 and choosing the rounding mode for each operation, the following algorithm can be used to implement the round-to-odd addition, assuming that the radix β of the floating-point system is even.

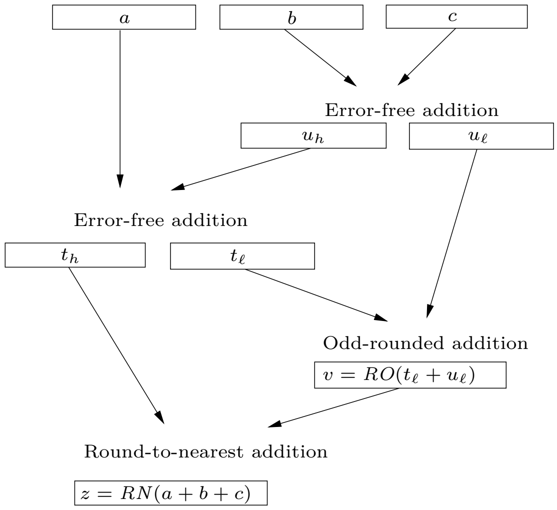


Figure 1: The Boldo-Melquiond algorithm.

For some of the arithmetic operations performed in this algorithm, the result is exactly representable, so it will be exactly computed with any rounding mode: hence, for these operations, we do not have indicated a particular rounding mode.

Algorithm 7 (OddRoundSum(a,b), arbitrary even radix)

$$\begin{array}{lcll} d & = & RD(a+b); & \\ u & = & RU(a+b); & \\ ulp & = & u-d; & \{exact\} \\ hulp & = & 0.5 \times ulp; & \{exact\} \\ e & = & RN(d+hulp); & \\ o' & = & u-e; & \{exact\} \\ o & = & o'+d; & \{exact\} \end{array}$$

For instance, with $\beta = 10$, $p = 4$, $a = 2.355$, and $b = 0.8935$, we successively get

$$\begin{array}{lcl} d & = & 3.248 \\ u & = & 3.249 \\ ulp & = & 0.001 \\ hulp & = & 0.0005 \\ e & = & 3.248 \\ o' & = & 0.001 \\ o & = & 3.249 \end{array}$$

Theorem 11 *Let a and b be two floating-point numbers, and assume that $a+b$ does not overflow and that "RN" means round to nearest* even. *Then Algorithm 7 computes $o = RO(a+b)$.*

Proof: Notice that since the radix is even, $0.5 = 1/2$ is exactly representable. If $a + b$ is exactly representable, then all the operations are exact and $d = u = a+b$, $hulp = ulp = 0$, $e = d$, $o' = 0$, and $o = d = a + b$.

Otherwise d and u are consecutive machine numbers and ulp is a power of the (even) radix, which cannot be the minimum nonzero machine number in magnitude (because the significand of $a + b$ must take at least $p + 1$ digits). Thus $ulp/2$ is exactly representable, so that $d + hulp$ is the exact middle of d and u. Therefore, by the round-to-nearest-even rule, e is the value, among d and u, whose last significand digit is even. Then o is the other one, which is the desired result. ∎

When the radix is 2, it is possible to save an operation, using the following algorithm. Note that if $e' \times 0.5$ is in the subnormal range, this means that $a + b$ is also in the subnormal range, implying that $d = u$, and $e' \times 0.5$ is performed exactly.

Algorithm 8 (OddRoundSum(a,b), radix 2)

$$\begin{array}{lcll}
d & = & RD(a+b); & \\
u & = & RU(a+b); & \\
e' & = & RN(d+u); & \\
e & = & e' \times 0.5; & \{exact\} \\
o' & = & u - e; & \{exact\} \\
o & = & o' + d; & \{exact\}
\end{array}$$

Algorithms 7 or 8 can be used in the algorithm depicted on Fig. 1 to implement the round-to-odd addition. Then we obtain an algorithm using only basic floating-point operations and the IEEE-754 rounding modes to compute $RN(a + b + c)$ for all floating-point numbers a, b and c.

In Algorithms 7 and 8, note that d and u may be calculated in parallel and that the calculation of *hulp* and e (in the general case, i.e., Algorithm 7) or e and o' (in the binary case, i.e., Algorithm 8) may be combined if a fused multiply-add (FMA) instruction is available. On most floating-point units, the rounding mode is dynamic and changing it requires to flush the pipeline, which is expensive. However, on some processors such as Intel's Itanium, the rounding mode of each floating-point operation can be chosen individually [7, Chap. 3]. In this case, choosing the rounding mode has no impact on the running time of a sequence of floating-point operations. Moreover the Itanium provides an FMA instruction, hence the proposed algorithm can be expected to be a very efficient alternative to compute round-to-odd additions on this processor.

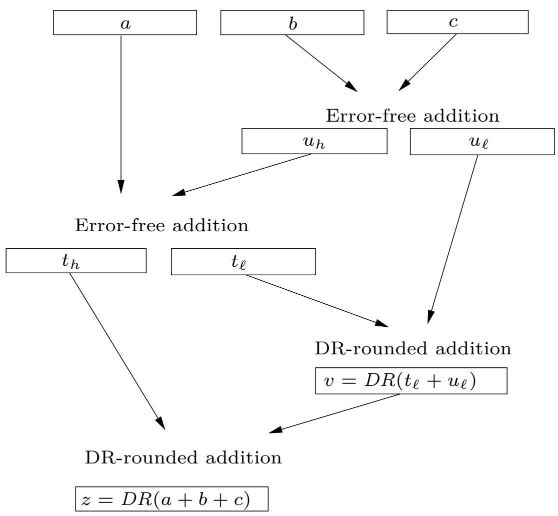


Figure 2: Algorithm to compute $DR(a + b + c)$ with $DR = RD$ or RU, derived from the Boldo-Melquiond Algorithm.

6.2 Computation of $DR(a + b + c)$

We now focus on the problem of computing $DR(a+b+c)$, where DR denotes one of the directed rounding modes (RZ, RD or RU). The algorithm we consider for $DR = RD$ or RU is depicted on Fig. 2 (the case $DR = RZ$ will be dealt with later): it is a variant of the Boldo-Melquiond algorithm. The only difference is that the last two operations use a directed rounding mode. The algorithm can be summarized as follows.

Algorithm 9 (DR3(a,b,c))

$$\begin{array}{rcl} (u_h, u_\ell) & = & \mathit{2Sum}(b, c); \\ (t_h, t_\ell) & = & \mathit{2Sum}(a, u_h); \\ v & = & DR(t_\ell + u_\ell); \\ z & = & DR(t_h + v); \end{array}$$

We will show that Algorithm 9 computes $DR(a + b + c)$ for rounding downward or upward. However, we will see that it may give an incorrect answer for rounding toward zero.

To prove Algorithm 9, we need to distinguish between different precisions. To that purpose, we introduce some notation. Let $\mathcal{F}_{\beta,p}$ denote the set of all radix-β, precision-p floating-point numbers, with an unbounded exponent range (where, obviously, $\beta \geq 2$ and $p \geq 1$). Given a real number $x \neq 0$, $\mathrm{ulp}_p(x)$ denotes the unit in the last place of x, i.e., if $\beta^e \leq |x| < \beta^{e+1}$ with $e \in \mathbb{Z}$, then $\mathrm{ulp}_p(x) = \beta^{e+1-p}$. Given $x \in \mathbb{R}$, we shall denote x rounded downward, rounded upward, rounded toward zero and rounded to nearest in $\mathcal{F}_{\beta,p}$ by $RD_p(x)$, $RU_p(x)$, $RZ_p(x)$ and $RN_p(x)$ respectively. Note that even though these functions depend on the parameter β, we omit β from their indices to make the notation simpler, since β is regarded as fixed; we will even omit the index p when only precision p is considered, just like in the other sections of the paper.

Theorem 12 *Let us assume that the radix β and the precision p satisfy*

- *either* $5 \cdot \beta^{1-p} \leq 1$,
- *or* $\beta = 2^k$, *where* $k \geq 1$ *is an integer, and* $3 \cdot \beta^{1-p} \leq 1$.

Then, given $a, b, c \in \mathcal{F}_{\beta,p}$, and $s = a+b+c$ the exact sum, algorithm DR3 (Algorithm 9) computes $z = DR(s)$.

Notice that for the most common cases, the hypotheses $3\beta^{1-p} \leq 1$ for radix 2 and $5\beta^{1-p} \leq 1$ for radix 10 can be summarized as follows:

$\beta = 2$	$\beta = 10$
$p \geq 3$	$p \geq 2$

For proving Theorem 12, we use the next two lemmata.

Lemma 13 *Let $\beta \geq 2$ and two precisions p and q such that $q \geq p$. Let DR be one of the directed rounding modes (RZ, RD or RU), so that DR_p and DR_q denote the corresponding rounding functions in $\mathcal{F}_{\beta,p}$ and $\mathcal{F}_{\beta,q}$ respectively. Then for all $x \in \mathbb{R}$, one has $DR_p(x) = DR_p(DR_q(x))$.*

The proof of Lemma 13 mainly relies on $\mathcal{F}_{\beta,p} \subset \mathcal{F}_{\beta,q}$ and on the fact that both roundings are done in the same direction.

Lemma 14 *Let $\beta \geq 2$, $p \geq 1$, and $x, y \in \mathcal{F}_{\beta,p}$ such that $x+y \notin \mathcal{F}_{\beta,p}$. We denote $z = RN(x+y)$.*

- *If* $\beta = 2^k$, *where* $k \geq 1$ *is an integer, then* $|y| \leq 2|z|$.

- *For any radix β, one has $|y| \leq 2(1+\beta^{1-p})|z|$.*

Proof: First, since $x+y \notin \mathcal{F}_{\beta,p}$, neither x nor y can be 0. If x and y have the same sign, then $|y| \leq |z| \leq 2|z|$. In the following, let us assume that x and y have different signs. Under this condition, Sterbenz's lemma yields: If $\frac{1}{2}|y| \leq |x| \leq 2|y|$, then $x+y \in \mathcal{F}_{\beta,p}$. Since by assumption $x+y \notin \mathcal{F}_{\beta,p}$,

- either $\frac{1}{2}|y| > |x|$, hence $|x+y| = |y| - |x| > \frac{1}{2}|y|$,
- or $|x| > 2|y|$, hence $|x+y| = |x| - |y| > |y|$.

In both cases, $|x+y| \geq \frac{1}{2}|y|$, hence $|z| = RN(|x+y|) \geq RN(\frac{1}{2}|y|)$. If β is a power of two, then $RN(\frac{1}{2}|y|) = \frac{1}{2}|y|$, hence $|z| \geq \frac{1}{2}|y|$. If no assumption is made on the radix β, then we write $RN(\frac{1}{2}|y|) = (1+\varepsilon)\frac{1}{2}|y|$, with $|\varepsilon| \leq \frac{1}{2}\beta^{1-p}$, which implies $RN(\frac{1}{2}|y|) \geq \frac{1}{2}(1-\frac{1}{2}\beta^{1-p})|y|$. A quick calculation shows that

$$\frac{1}{1-\frac{1}{2}\beta^{1-p}} - (1+\beta^{1-p}) = -\frac{1}{2}\beta^{1-p}\frac{1-\beta^{1-p}}{1-\frac{1}{2}\beta^{1-p}} \leq 0,$$

with equality when $p=1$. As a consequence,

$$\frac{1}{1-\frac{1}{2}\beta^{1-p}} \leq (1+\beta^{1-p}),$$

and $|y| \leq 2(1+\beta^{1-p})|z|$. ■

Proof of Theorem 12: In this proof, let us denote $t_\ell + u_\ell$ by γ, and $t_h + v$ by s'.

Let us first prove the result on the following two special cases:

- If $a + u_h \in \mathcal{F}_{\beta,p}$, then $t_\ell = 0$, which means that $s = t_h + u_\ell$; moreover, $z = DR(t_h + v) = DR(t_h + u_\ell)$, hence $z = DR(s)$.
- If $\gamma = 0$, then $s = t_h$, $v = 0$, and $z = DR(s)$.

In the following of the proof, let us now assume that $a + u_h \notin \mathcal{F}_{\beta,p}$ and $\gamma \neq 0$. Since $(t_h, t_\ell) = 2\text{Sum}(a, u_h)$, one has $|t_\ell| \leq \frac{1}{2}\beta^{1-p}|t_h|$, and from $|\gamma| \leq |u_\ell| + |t_\ell|$ we deduce that $|\gamma| \leq |u_\ell| + \frac{1}{2}\beta^{1-p}|t_h|$. On the other hand, since $(u_h, u_\ell) = 2\text{Sum}(b, c)$, one has $|u_\ell| \leq \frac{1}{2}\beta^{1-p}|u_h|$. As a consequence,

$$|\gamma| \leq \frac{1}{2}\beta^{1-p}|u_h| + \frac{1}{2}\beta^{1-p}|t_h|.$$

As $(t_h, t_\ell) = 2\text{Sum}(a, u_h)$ and $t_h = RN(a + u_h)$, and since $a + u_h$ does not belong to $\mathcal{F}_{\beta,p}$ by hypothesis, Lemma 14 can be used to bound $|u_h|$ with respect to $|t_h|$. We distinguish two cases.

- If β is a power of two, then $|u_h| \leq 2|t_h|$. As a consequence $|\gamma| \leq \frac{3}{2}\beta^{1-p}|t_h|$, and since $3\beta^{1-p} \leq 1$, $|\gamma| \leq |t_h|$. From $|s| = |t_h + t_\ell + u_\ell| \geq |t_h| - |\gamma|$, we also deduce $|s| \geq (\frac{2}{3}\beta^{p-1} - 1)|\gamma|$. Since $3\beta^{1-p} \leq 1$ implies $\frac{2}{3}\beta^{p-1} - 1 \geq 1$, one also has $|\gamma| \leq |s|$.
- If no assumption is made on β, one has $|u_h| \leq 2(1+\beta^{1-p})|t_h|$, which gives $|\gamma| \leq (\frac{3}{2} + \beta^{1-p})\beta^{1-p}|t_h| \leq \frac{5}{2}\beta^{1-p}|t_h|$, and since $5\beta^{1-p} \leq 1$, $|\gamma| \leq |t_h|$ follows. As $|s| \geq |t_h| - |\gamma|$, one also has $|s| \geq \left(\frac{2}{5}\beta^{p-1} - 1\right)|\gamma|$. Since $5\beta^{1-p} \leq 1$ implies $\frac{2}{5}\beta^{p-1} - 1 \geq 1$, it follows that $|\gamma| \leq |s|$.

Therefore, in both cases one has

$$|\gamma| \leq |t_h| \quad \text{and} \quad |\gamma| \leq |s|. \tag{1}$$

We now focus on the last two operations in Algorithm 9. Defining $\rho_{DR}(x)$ by $\rho_{RD}(x) = \lfloor x \rfloor$ and $\rho_{RU}(x) = \lceil x \rceil$, one has

$$s' = t_h + DR_p(t_\ell + u_\ell) = t_h + \rho_{DR}\left(\frac{\gamma}{\mathrm{ulp}_p(\gamma)}\right)\mathrm{ulp}_p(\gamma).$$

From the first inequality in (1) it follows that $\mathrm{ulp}_p(\gamma) \leq \mathrm{ulp}_p(t_h)$, which implies that t_h is an integral multiple of $\mathrm{ulp}_p(\gamma)$. Since $s = t_h + \gamma$, we write

$$\begin{aligned} s' &= \left(\frac{t_h}{\mathrm{ulp}_p(\gamma)} + \rho_{DR}\left(\frac{\gamma}{\mathrm{ulp}_p(\gamma)}\right)\right)\mathrm{ulp}_p(\gamma) \\ &= \rho_{DR}\left(\frac{s}{\mathrm{ulp}_p(\gamma)}\right)\mathrm{ulp}_p(\gamma). \end{aligned}$$

Since $\gamma \neq 0$ and $s \neq 0$, there exists an integer q such that $\mathrm{ulp}_p(\gamma) = \mathrm{ulp}_q(s)$.[3] Furthermore, it follows from the second inequality in (1) that $\mathrm{ulp}_p(\gamma) \leq \mathrm{ulp}_p(s)$, hence $\mathrm{ulp}_q(s) \leq \mathrm{ulp}_p(s)$, which implies $q \geq p$. Hence

$$s' = \rho_{DR}\left(\frac{s}{\mathrm{ulp}_q(s)}\right)\mathrm{ulp}_q(s) = DR_q(s).$$

Since $z = DR_p(s')$, one has $z = DR_p(DR_q(s))$. Then from Lemma 13, we obtain $z = DR_p(s)$. ■

Note that the proof cannot be extended to RZ, due to the fact that the two roundings can be done in opposite directions. For instance, if $s > 0$ (not exactly representable) and $t_\ell + u_\ell < 0$, then one has $RD(s) \leq RD(s')$ as wanted, but $t_\ell + u_\ell$ rounds upward and s' can be $RU(s)$, so that $z = RU(s)$ instead of $RZ(s) = RD(s)$, as shown on the following counter-example. In radix 2 and precision 7, with $a = -3616$, $b = 19200$ and $c = -97$, we have $s = 15487$, $RZ(s) = 15360$ and $RU(s) = 15488$. Running the algorithm depicted on Fig. 2 on this instance gives

$$\begin{aligned} &(u_h, u_\ell) = (19200, -97) \\ &(t_h, t_\ell) = (15616, -32) \\ &v = RZ(-129) = -128 \\ &z = RZ(15488) = 15488 \end{aligned}$$

and $RU(s)$ has been computed instead of $RZ(s)$.

Nevertheless $RZ(s)$ can be obtained by computing both $RD(s)$ and $RU(s)$, then selecting the one closer to zero thanks to the minNumMag instruction [10], as shown in the following algorithm.

Algorithm 10 (RZ3(a,b,c))

$$\begin{aligned} (u_h, u_\ell) &= \mathit{2Sum}(b, c); \\ (t_h, t_\ell) &= \mathit{2Sum}(a, u_h); \\ v_d &= RD(u_\ell + t_\ell); \\ z_d &= RD(t_h + v_d); \\ v_u &= RU(u_\ell + t_\ell); \\ z_u &= RU(t_h + v_u); \\ z &= \mathit{minNumMag}(z_d, z_u); \end{aligned}$$

[3] Notice that q may be negative. We use the same definition of ulp_q as previously: if $\beta^e \leq |x| < \beta^{e+1}$ with $e \in \mathbb{Z}$, then $\mathrm{ulp}_q(x) = \beta^{e+1-q}$.

This algorithm for computing $RZ(a + b + c)$ without branches can already be implemented on the Itanium architecture thanks to the famin instruction [7].

7 Conclusions

We have proved that in binary arithmetic Knuth's 2Sum algorithm is minimal, both in terms of the number of operations and the depth of the dependency graph. We have investigated the possibility of using the Fast2Sum algorithm in radix-10 floating-point arithmetic. We have also shown that, just by performing round-to-nearest floating-point additions and subtractions without any testing, it is impossible to compute the round-to-nearest sum of $n \geq 3$ floating-point numbers in even-radix arithmetic. If changing the rounding mode is allowed, in even-radix arithmetic, we can implement, without testing, the nonstandard *rounding to odd* defined by Boldo and Melquiond, which makes it indeed possible to compute the sum of three floating-point numbers rounded to nearest. We finally proposed an adaptation of the Boldo-Melquiond algorithm for calculating $a + b + c$ rounded according to the standard directed rounding modes.

8 Acknowledgement

We thank Damien Stehlé, who actively participated in our first discussions on these topics.

References

[1] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. New York, 1985.

[3] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Radix Independent Floating-Point Arithmetic, ANSI/IEEE Standard 854-1987*. New York, 1987.

[4] E. Adams and U. Kulisch, editors. *Scientific Computing with Automatic Result Verification*. Academic Press, San Diego, 1993.

[5] S. Boldo and G. Melquiond. Emulation of a FMA and correctly-rounded sums: proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4), Apr. 2008.

[6] S. Boldo and M. Daumas. Representable correcting terms for possibly underflowing floating point operations. In J.-C. Bajard and M. Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, pages 79–86. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[7] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific computing on Itanium based systems*. Intel Press, 2002.

[8] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[9] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007. Available at `http://www.mpfr.org/`.

[10] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008. Available at `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.

[11] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller. On the Computation of Correctly-Rounded Sums *Proceedings of the 19th IEEE Symposium on Computer Arithmetic* Portland, OR, June 2009.

[12] D. Knuth. *The Art of Computer Programming, 3rd edition*, volume 2. Addison-Wesley, Reading, MA, 1998.

[13] O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[14] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM TOPLAS*, 30(3):1–41, 2008. Available at `http://hal.archives-ouvertes.fr/hal-00128124`.

[15] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. Handbook of Floating-Point Arithmetic. Birkhäuser, 2009.

[16] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[17] M. Pichat. Correction d'une somme en arithmétique à virgule flottante (in French). *Numerische Mathematik*, 19:400–406, 1972.

[18] D. Priest. *On Properties of Floating-Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, 1992.

[19] J. R. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399