On the computation of correctly-rounded sums

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

On the computation of correctly-rounded sums

P. Kornerup V. Lefèvre N. Louvet
J.-M. Muller *

This is LIP research report No 2008-35

October 2008

Abstract

The computation of sums appears in many domains of numerical analysis. We show that among the set of the algorithms with no comparisons performing only floating-point operations, the 2Sum algorithm introduced by Knuth is optimal, both in terms of number of operations and depth of the dependency graph. We also show that, under reasonable conditions, it is impossible to always obtain the correctly rounded-to-nearest sum of $n \geq 3$ floating-point numbers with an algorithm without tests performing only round-to-nearest additions/subtractions. Boldo and Melquiond have proposed an algorithm to compute the rounded-to-nearest sum of three operands, introducing a new rounding mode unavailable on current hardware, *rounding to odd*; but their simulation of rounding to odd requires tests. We show that rounding to odd can be be realized using only floating-point additions/subtractions performed in the standard rounding modes and a multiplication by the constant 0.5, thus allowing the rounded-to-nearest sum of three floating-point numbers to be determined without tests. Starting from the algorithm due to Boldo and Melquiond, we also show that the sum of three floating-point values rounded according to any of the standard directed rounding modes can be determined using only additions/subtractions, provided that the operands are of the same sign.

Keywords: Floating-point arithmetic, summation algorithms, correct rounding, 2Sum and Fast2Sum algorithms.

1 Introduction

The computation of sums appears in many domains of numerical analysis. They occur when performing numerical integration, when evaluating dot products, means, variances and many other functions. When having to add floating point numbers a_1, a_2, ..., a_n, the best we can hope is to get $\circ(a_1 + a_2 + \cdots a_n)$, where $\circ$ is the desired rounding mode. This can always be done at a rather large cost, using multiple-precision arithmetic. The purpose of this paper is to see if this can be done by very simple programs, that only use floating-point additions and subtractions in the target format, and without comparisons, or conditional expressions, or min/max instructions.

*Peter Kornerup is with SDU, Odense, Denmark; Vincent Lefèvre, Nicolas Louvet and Jean-Michel Muller are with CNRS/INRIA/Université de Lyon, Lyon, France.

The original IEEE 754-1985 standard [1] for radix-2 floating-point arithmetic (as well as its follower, the IEEE 854-1987 radix-independent standard [2], and the newly IEEE 754-2008 revised standard [9]) requires that the four arithmetic operations and the square root should be correctly rounded. In a floating-point system that follows one of these standards, the user can choose an *active rounding mode* (called *rounding direction attribute* in IEEE 754-2008), from:

- rounding toward $-\infty$: $RD(x)$ is the largest machine number less than or equal to x;
- rounding toward $+\infty$: $RU(x)$ is the smallest machine number greater than or equal to x;
- rounding toward 0: $RZ(x)$ is equal to $RD(x)$ if $x \geq 0$, and to $RU(x)$ if $x < 0$;
- rounding to nearest: $RN(x)$ is the machine number that is the closest to x (with a special tie-breaking rule if x is exactly between two machine numbers: in IEEE 754-1985 and IEEE 854-1987, the chosen number is the "even" one[1]. IEEE 754-2008 requires this "round-to-nearest even" rule, but also defines a "round-to-away" rule – mainly for financial, decimal, calculations).

When $a \circ b$ is computed, where a and b are floating-point numbers and $\circ$ is $+$, $-$, $\times$ or $\div$, the returned result is what we would get if we computed $a \circ b$ exactly, with "infinite" precision and rounded it according to the active rounding mode. The default rounding mode is round-to-nearest. This requirement is called *correct rounding*. This makes arithmetic deterministic (provided all computations are done in the same format, which might sometimes be difficult to ensure [13]). This allows one to design algorithms and proofs. An example is the following result (due to Dekker [4]).

Theorem 1 (Fast2Sum algorithm)**.** *Assume the radix r of the floating-point system being considered is 2 or 3, and that the used arithmetic provides correct rounding with rounding to nearest. Let a and b be finite floating-point numbers, and assume that the exponent of a is larger than or equal to that of b. The following algorithm computes two floating-point numbers s and t that satisfy:*

- *$s + t = a + b$ exactly;*
- *s is $a + b$ rounded to nearest.*

Algorithm 1 (Fast2Sum(a,b))**.**

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

Note that the condition "the exponent of a is larger than or equal to that of b" might be slow to check in a portable way, but if $|a| \geq |b|$, then that condition will be fulfilled.

If no information on the relative orders of magnitude of a and b is available, there is an alternative algorithm due to Knuth [11] and Møller [12], called 2Sum. It requires 6 operations instead of 3 for the Fast2Sum algorithm, but on current pipelined architectures, an *if* statement with an wrong branch prediction may cause the instruction pipeline to drain:

[1] In binary, the one whose last significand bit is a zero.

due to that, using 2Sum will usually result in much faster software than using a comparison followed by Fast2Sum. The names "2Sum" and "Fast2Sum" seem to have been coined by Shewchuk [16]. We call these algorithms *Error-free additions*.

Algorithm 2 (2Sum(a,b))**.**

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

The problem of computing very accurately the sum of n floating-point numbers arises in many domains, hence it has been dealt with by many authors. Two major techniques have been used: either to try to re-order the input operands, to try to minimize the error of the "straightforward" addition algorithm, or to use methods similar to 2Sum or Fast2Sum to somehow "compensate" the errors of the individual additions by re-injecting these errors at some point in the calculations. For instance, Kahan [10], Pichat [14], and Priest [15] have built clever "compensating" method. Higham [7, 8] gives a very interesting survey. Ogita, Rump and Shin'ichi generalize the "compensated" methods by Kahan and Pichat.

In all the following, we consider a binary, precision-p, floating-point format.

2 2Sum is optimal

Definition 1. *In the following, we call* RN-addition algorithm without branching *an algorithm*

- *without comparisons, or conditional expressions, or min/max instructions;*
- *only based on floating-point additions or subtractions in round-to-nearest mode: at step i the algorithm computes $RN\left(a+b\right)$ or $RN\left(a-b\right)$ where a and b are either one of the input values, or a previously-computed value.*

For instance, 2Sum is an RN-addition algorithm without branching. It 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:

$$\delta_b = RN\left(b-b'\right)$$

and

$$\delta_a = RN\left(a-a'\right)$$

hence we will say that the depth of Algorithm 2Sum is 5. We show the following results, that proves that, among the RN-addition algorithms without branching, 2Sum is optimal in terms of number of operations as well as in terms of depth of the dependency graph.

Theorem 2. *An RN-addition algorithm without branching that computes the same results as 2Sum requires at least* 6 *arithmetic operations.*

Theorem 3. *An RN-addition algorithm without branching that computes the same results as 2Sum has depth at least* 5*.*

Proof. To prove Theorems 2 and 3, we have proceeded as follows. We enumerated all possible RN-addition algorithms without branching with 2 input values, that use 5 additions and/or subtractions or less (for Theorem 2) or that have depth 4 or less (for Theorem 3), eliminating obvious symmetries (for instance, this gave 480756 algorithms for Theorem 2). Each of these algorithms was tried with 2 pairs of well-chosen input values a and b for Theorem 2 (and 3 pairs for Theorem 3), so that every algorithm for which we did not get the right answer for one of the pairs of input values could be immediately eliminated. The idea was that we could then analyze more deeply the very few algorithms that would remain. But there remained none. As an example, the C program that checks that there are no algorithms equivalent to 2Sum with depth ≤ 4 is given in an appendix. □

We used a similar strategy for studying all algorithms with 6 additions and/or subtractions (i.e., the same number as 2Sum) and could conclude that the only algorithms that give the same results as 2Sum are derived from 2Sum through straightforward symmetries (e.g., inversion of the two operations that can be performed in parallel, or replacement of $b' = RN(s-a)$ by $b'' = RN(a-s)$ and, later on, replacement of $s - b'$ by $s + b''$ and of $b - b'$ by $b + b''$). Hence, in a way, 2Sum is the only algorithm that computes $s = RN(a+b)$ and $t = a + b - RN(a+b)$ in 6 operations.

3 On the impossibility of getting $RN(x_1 + x_2 + \cdots + x_n)$ under some conditions

We are interested in the computation of the sum of n floating-point numbers, correctly rounded to nearest. We assume a binary, precision-p, floating-point format.

Theorem 4. *Assume x_1, x_2, x_3, …, x_n ($n \geq 3$) are floating-point numbers of the same format. Assuming an* unbounded exponent range*, an RN-addition algorithm employing only additions and subtractions without branching cannot always return $RN(x_1 + x_2 + \cdots + x_n)$.*

If there exists an RN-addition algorithm to compute the rounded to nearest sum of n floating-point numbers, with $n \geq 3$, then this algorithm must also compute the rounded 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 will show how to construct for any RN-algorithm a set of input data such that the result computed by the algorithm differs from the rounded to nearest result.

Proof of Theorem 4. An RN-addition algorithm without branching can be represented by a directed acyclic graph[2] (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 x_1, x_2, x_3 defined as follows.

[2]Such an algorithm cannot have WHILE loops, since tests are prohibited. It may have FOR loops, that can be unrolled

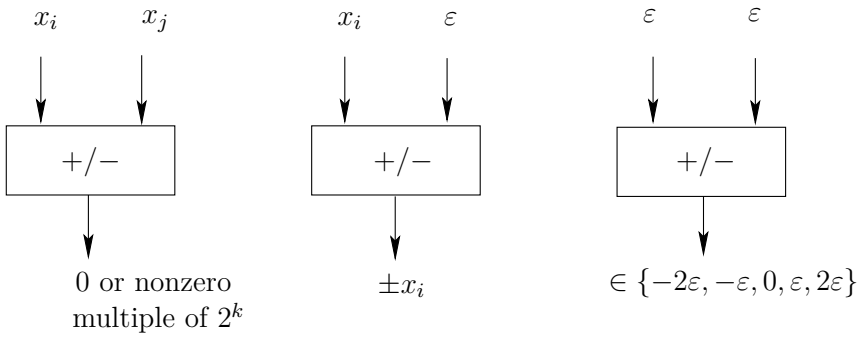
- $x_1 = 2^{k+p}$ and $x_2 = 2^k$: x_1 and x_2 are two nonzero multiples of 2^k whose sum is the exact middle of two consecutive FP numbers;
- $x_3 = \varepsilon$, with $0 \leq 2^{r-1}|\varepsilon| \leq 2^{k-p-1}$.

Note that when $\varepsilon \neq 0$,

$$RN\ (x_1 + x_2 + x_3) = \begin{cases} RD\,(x_1 + x_2 + x_3) & \text{if} \quad \varepsilon < 0 \\ \\ RU\,(x_1 + x_2 + x_3) & \text{if} \quad \varepsilon > 0, \end{cases}$$

where we may also conclude that $RN\ (x_1 + x_2 + x_3) \neq 0$.

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 2^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 2^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 2^k that is the same as if ε were 0, then the other input is either also a nonzero multiple of 2^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 rounded to nearest result differs from the computed one. If both entries of the last operation belong to $\mathcal{S}_{r-1}$, then the result belongs to $\mathcal{S}_r = \{-2^r\varepsilon, \cdots, 0, \cdots, +2^r\varepsilon\}$: if one set $\varepsilon = 0$, then the computed result is 0, contradicting the fact that the rounded to nearest sum must be nonzero. □

In the proof of Theorem 4, 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 $2^{r-1}\varepsilon$ will vanish when added to any multiple of 2^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. Indeed, assuming $\varepsilon = \pm 2^{e_{\min}}$ and $x_1 = 2^{k+p} = 2^{e_{\max}}$, the inequality $2^{r-1}|\varepsilon| \leq 2^{k-p-1}$ gives the following theorem.

Theorem 5. *Assume x_1, x_2, x_3, ..., x_n ($n \geq 3$) are floating-point numbers of the same format. Assuming the extremal exponents of the floating-point format are $e_{\min}$ and $e_{\max}$, an RN-addition algorithm without branching, of depth r, cannot always return $RN(x_1 + x_2 + \cdots + x_n)$, 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 5 shows that an RN-addition algorithm without branching able to always evaluate the rounded to nearest sum of at least 3 floating-point numbers (if such an algorithm exists!) must have depth at least 1939.

4 On the sum of three floating-point numbers

4.1 The Boldo-Melquiond algorithm: another way to round to odd

Floating-point addition allows one to compute $\circ(a + b)$, where a and b are floating-point numbers, and in the case of the round-to-nearest mode, the 2Sum and Fast2Sum algorithms make it possible to compute the error of that floating-point addition. The question that arises is: can we do something similar with the sum of *three* floating-point numbers? Theorem 4 shows that this is impossible to get $RN(a + b + c)$ just by performing rounded-to-nearest additions or subtractions. A natural question is: *what can we do if we allow intermediate use of other rounding modes?*

Boldo and Melquiond [3] introduce a new rounding mode, *round-to-odd*, $\circ_{\text{odd}}$, defined as follows:

- if x is a floating-point number, then $\circ_{\text{odd}}(x) = x$;
- otherwise, $\circ_{\text{odd}}(x)$ is the value among $RD(x)$ and $RU(x)$, whose last significant bit is a one.

This rounding mode is not implemented on current architectures, but that could easily be done. Interestingly enough, Boldo and Melquiond show that using only one rounded-to-odd addition, one can easily compute $RN(a + b + c)$, where a, b and c are floating-point numbers. Their algorithm is given in Figure 1. They explain how to implement rounded-to-odd additions, but unfortunately their method requires testing.

However, if we allow multiplication by the simple constant 0.5 we can implement the odd-rounded addition, $\circ_{\text{odd}}(a+b)$ as follows. Notice that our algorithm requires round-to-nearest *even*, which is the default mode in IEEE-754 binary arithmetic.

Algorithm 3 (OddRoundSum(a,b))**.**

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

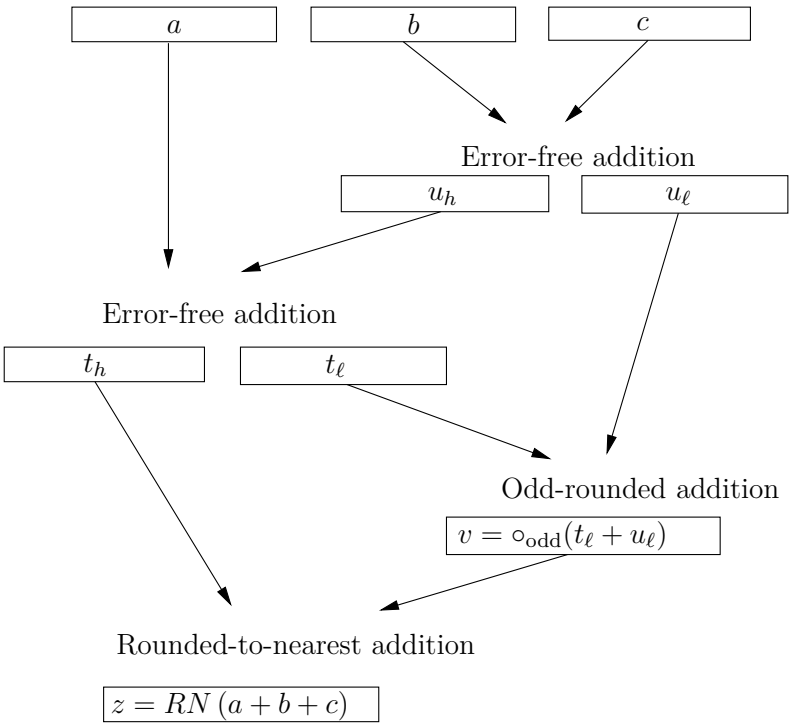


Figure 1: The Boldo-Melquiond algorithm [3] for computing the correctly rounded to nearest sum of 3 floating-point numbers. It requires an "odd-rounded" addition, a rounding mode not available on current floating-point units. The error-free additions are performed using the 2Sum algorithm (unless we know for some reason the ordering of the magnitude of the variables, in which case 2Sum may be used).

Proof. By definition, if $a + b$ is a representable floating-point number, the output o of the algorithm is the odd-rounded sum of $a + b$. Otherwise, provided that there is no overflow or underflow, d and u will be the two closest representable floating-point numbers surrounding $a + b$, with the exact value of $\frac{d+u}{2}$ being the midpoint of the open interval $(d; u)$. Then the last significant bit of $RN\,(d+u)$ is zero, and so is the least significant bit of $RN\,(d+u) \times 0.5$. Therefore, e is either d or u and the last bit of its significand is a zero: this means that, to get the rounded-to-odd sum of a and b, we must return d if $e = u$, and u otherwise. One easily sees that:

- if $e = u$ then $o' = 0$ therefore $o = d$;
- if $e = d$ then $o' = u - d$ (which is exactly representable by Sterbenz lemma), so that $o = u$.

Hence it is trivially seen that the output is the odd-rounded sum of $a + b$. □

Note that d and u may be calculated in parallel, and that the calculation of e' and e may be combined if an FMA instruction is available. But unfortunately, on most floating-point units changing rounding mode requires flushing the pipeline, and hence is very expensive. However, on the Itanium processor such changes costs nothing, provided the algorithm is programmed in assembly code.

Following Boldo-Melquiond we then immediately have:

Theorem 6. *If there are no underflows or overflows in intermediate operations, then the algorithm given in Figure 1, employing addition, subtraction and multiplication by* 0.5*, computes* $RN(a+b+c)$ *for all floating-point numbers* a*,* b *and* c*, employing only available IEEE-754 rounding modes.*

We have no proof that obtaining $RN(a+b+c)$ is impossible using only additions and subtractions with a combination of the four IEEE-754 rounding modes without tests.

4.2 Getting $DR(a+b+c)$ when a, b and c have the same sign

Let us now deal with the problem of computing $DR(a+b+c)$, where DR denotes one of the directed rounding modes (RZ, RD or RU), without branching (yet allowing all four rounding modes of IEEE 754). Our algorithm is depicted in Figure 2. As one immediately sees, it is a simple modification of the Boldo-Melquiond algorithm: the only difference is that the last two operations use a directed rounding mode.

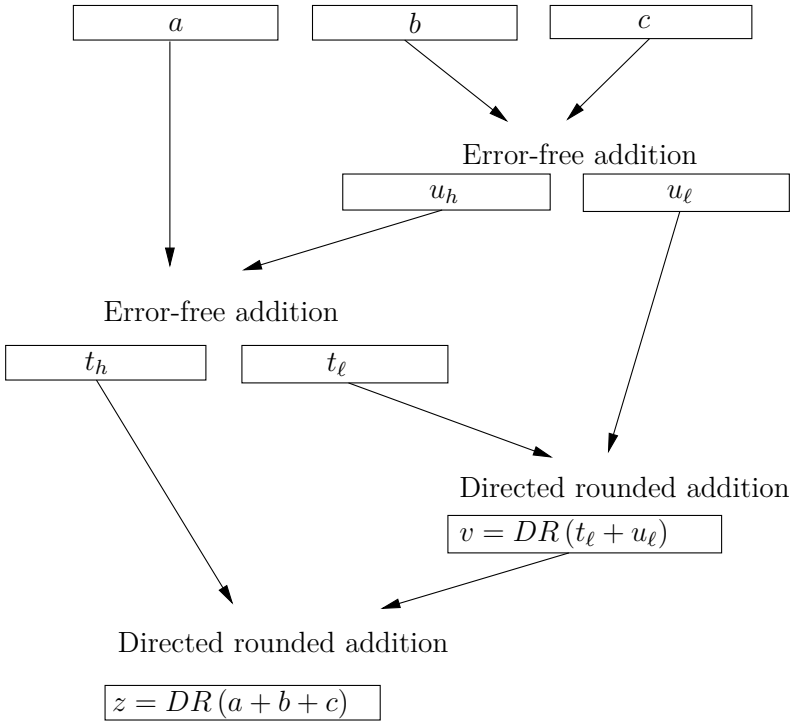


Figure 2: Our modified version of the Boldo-Melquiond algorithm. It computes $DR(a+b+c)$, provided that a, b and c have the same sign.

Theorem 7. *If there are no underflows or overflows in intermediate operations, then the algorithm given in Figure 2 computes* $DR(a+b+c)$ *for all floating-point numbers* a*,* b *and* c *that have the same sign.*

It is worth mentioning that in the general case, even when a, b and c do not have the same sign, the cases for which the algorithm does not return $DR(a+b+c)$ seem very rare. Anyway, the following table provides such counter-examples for the three directed rounding modes.

RD	RZ	RU
$a = -1110001_2 \times 2^{10}$	$a = 1101001_2 \times 2^{6}$	$a = -1111111_2 \times 2^{23}$
$b = 1001011_2 \times 2^{13}$	$b = 1000001_2 \times 2^{0}$	$b = 1111100_2 \times 2^{31}$
$c = -1100001_2 \times 2^{5}$	$c = -1011000_2 \times 2^{8}$	$c = -1010101_2 \times 2^{30}$

We define $\mathrm{ulp}(x)$ as the distance between the closest straddling floating-point numbers a and b (i.e., those with $a < x \leq b$). This particular definition of the ulp, due to Harrison [5, 6], is needed below.

Proof of Theorem 7. Without loss of generality, we assume that a, b and c are all non-negative. The error-free additions (i.e., using 2Sum algorithm, unless we have some information on the ordering of the variables that makes it possible to use Fast2Sum) guarantee that

$$u_h = RN(b+c), \quad u_h + u_\ell = b + c, \quad |u_\ell| \leq \frac{1}{2}\mathrm{ulp}(u_h),$$

and

$$t_h = RN(a+u_h), \quad t_h + t_\ell = a + u_h, \quad |t_\ell| \leq \frac{1}{2}\mathrm{ulp}(t_h).$$

In particular, this implies that u_h and t_h are both non-negative and $a+b+c = t_h + t_\ell + u_\ell$.

Since a and u_h are both non-negative, we have $u_h < a + u_h$. By monotonicity of rounding to the nearest, this implies $u_h \leq RN(a+u_h) = t_h$, and we deduce that $\mathrm{ulp}(u_h) \leq \mathrm{ulp}(t_h)$. Hence $|u_\ell| \leq \frac{1}{2}\mathrm{ulp}(u_h) \leq \frac{1}{2}\mathrm{ulp}(t_h)$, and since $|t_\ell| \leq \frac{1}{2}\mathrm{ulp}(t_h)$, using the triangular inequality, we obtain

$$|t_\ell + u_\ell| \leq \mathrm{ulp}(t_h).$$

If $|t_\ell + u_\ell| = \mathrm{ulp}(t_h)$, since it is a floating-point number, $v = t_\ell + u_\ell$ is computed exactly and $DR(t_h + t_\ell + u_\ell) = DR(t_h + v)$. On the other hand, if $|t_\ell + u_\ell| < \mathrm{ulp}(t_h)$, we distinguish the case of each directed rounding mode.

In the *case of rounding toward zero* ($\mathrm{DR} = \mathrm{RZ}$), from $|t_\ell + u_\ell| < \mathrm{ulp}(t_h)$ we deduce

$$RZ(t_h + t_\ell + u_\ell) = \begin{cases} t_h & \text{if} \quad t_\ell + u_\ell \geq 0 \\ t_h^- & \text{if} \quad t_\ell + u_\ell < 0 \end{cases},$$

and since $|v| = |t_\ell + u_\ell| < \mathrm{ulp}(t_h)$,

$$RZ(t_h + v) = \begin{cases} t_h & \text{if} \quad v \geq 0 \\ t_h^- & \text{if} \quad v < 0 \end{cases}.$$

As $t_\ell + u_\ell$ and v have the same sign, the equality $RZ(t_h + t_\ell + u_\ell) = RZ(t_h + v)$ follows.

In the *case of rounding downward* ($\mathrm{DR} = \mathrm{RD}$), since $|t_\ell + u_\ell| < \mathrm{ulp}(t_h)$ we have

$$RD(t_h + t_\ell + u_\ell) = \begin{cases} t_h & \text{if} \quad t_\ell + u_\ell \geq 0 \\ t_h^- & \text{if} \quad t_\ell + u_\ell < 0 \end{cases}.$$

We use the following fact: if x is a real number, and f is a power of two such that $|x| < f$, then $|RD(x)| \leq f$ and $|RD(x)| = f$ implies $x < 0$. Here, since $|t_\ell + u_\ell| < \mathrm{ulp}(t_h)$, we know that $|v| = RD(t_\ell + u_\ell) \leq \mathrm{ulp}(t_h)$. As a consequence, we distinguish again two cases.

- If $|v| = \mathrm{ulp}(t_h)$, then $t_\ell + u_\ell$ and v are both non-positive, thus $RD(t_h + v) = t_h^-$.

- If $|v| < \mathrm{ulp}(t_h)$, then

$$RD\,(t_h + v) = \begin{cases} t_h & \text{if} \quad v \geq 0 \\ t_h^- & \text{if} \quad v < 0 \end{cases} .$$

Since $t_\ell + u_\ell$ and v share the same sign, the equality $RD\,(t_h + t_\ell + u_\ell) = RD\,(t_h + v)$ holds.

The *case of rounding upward* (DR = RU) is treated using the same arguments as for the case of rounding downward. □

5 Conclusions

We have proved that Knuth's 2Sum algorithm is optimal, both in terms of the number of operations and the depth of the dependency graph. We have also shown that, just by performing rounded-to-nearest floating-point additions and subtractions without any testing, it is impossible to compute the rounded-to-nearest sum of $n \geq 3$ floating-point numbers. We have shown that if changing the rounding mode is allowed, 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 the nearest. We finally proposed an adaptation of the Boldo-Melquiond algorithm for calculating $a + b + c$ rounded according to the standard directed rounding modes when a, b and c share the same sign.

6 Acknowledgement

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

References

[1] 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.

[2] 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.

[3] 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), April 2008.

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

[5] J. Harrison. A machine-checked theory of floating-point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, Berlin, September 1999.

[6] J. Harrison. Floating-point verification using theorem proving. In Marco Bernardo and Alessandro Cimatti, editors, *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM*

2006, volume 3965 of *Lecture Notes in Computer Science*, pages 211–242, Bertinoro, Italy, 2006. Springer-Verlag.

[7] N. J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14(4):783–799, 1993.

[8] N. J. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM, Philadelphia, PA, 2002.

[9] Institute of Electrical and Electronic Engineers. *IEEE Std.* $754^{TM}-2008$ *Standard for Floating-Point Arithmtic*. IEEE, 3 Park Avenue, NY 10016-5997, USA, August 2008.

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

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

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

[13] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM TOPLAS*, 30(3):1–41, 2008.

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

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

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

Appendix: C program that checks that there are no algorithms equivalent to 2Sum with depth ≤ 4.

```
/* Prove that there exist no TwoSum algorithms with depth <= 4. */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#ifndef MAXDEPTH
#define MAXDEPTH 4
#endif

#if MAXDEPTH > 4
/* Note: to reduce the number of searches (though not all algorithms
   may be found), it is advised to define SKIPDOUBLE. */
#define SIZE 90000000
#else
#define SIZE 8000
#endif

#define NNX (0)
#define NNY (1)

typedef struct
{
  int i;     /* node number of 1st operand */
  int j;     /* node number of 2nd operand */
  int n;     /* 0 if addition, 1 if subtraction */
  int d;     /* depth, dag[*].d is an increasing function */
  double r;  /* result 1 */
  double s;  /* result 2 */
  double t;  /* result 3 */
} NODE;

static NODE dag[SIZE];

static void algout (long k, long i, long j, int n)
{
  printf ("v[%ld] = v[%ld] %c v[%ld]\n", k, i, "+-"[n], j);
  if (i > NNY)
    algout (i, dag[i].i, dag[i].j, dag[i].n);
  if (j > NNY)
    algout (j, dag[j].i, dag[j].j, dag[j].n);
}

int main (void)
{
  double res1, res3;
  int d;
```

```
long k;

dag[NNX].r = 4.12310562561766;
dag[NNY].r = 0.31830988618379;

dag[NNX].s = dag[NNY].r;
dag[NNY].s = dag[NNX].r;

dag[NNX].t = 1.5;
dag[NNY].t = nextafter (dag[NNX].t, 2.0);

dag[NNX].d = dag[NNY].d = 0;

res1 = dag[NNX].r + dag[NNY].r;
if (fabs (dag[NNX].r) > fabs (dag[NNY].r))
  {
    res1 = dag[NNX].r - res1;
    res1 += dag[NNY].r;
  }
else
  {
    res1 = dag[NNY].r - res1;
    res1 += dag[NNX].r;
  }

res3 = dag[NNX].t + dag[NNY].t;
if (fabs (dag[NNX].t) > fabs (dag[NNY].t))
  {
    res3 = dag[NNX].t - res3;
    res3 += dag[NNY].t;
  }
else
  {
    res3 = dag[NNY].t - res3;
    res3 += dag[NNX].t;
  }

k = NNY;
for (d = 1; d <= MAXDEPTH; d++)
  {
    long last, i, j;
    int n;

    /* Build the results at depth d (exactly). */
    last = k;
    for (i = 0; i <= last; i++)
      for (j = 0; j <= last; j++)
        for (n = 0; n <= 1; n++)
          {
            NODE res, *p;
```

```
#ifdef SKIPDOUBLE
              if (i == j)
                continue;
#endif

              /* Addition: require i <= j since addition is commutative.
                 Subtraction: require i != j since computing 0 is useless.
                 Also require that the depth of one of the children is
                 equal to d-1. */
              if (n ? (i == j || (dag[i].d != d-1 && dag[j].d != d-1))
                    : (i > j || dag[j].d != d-1))
                continue;

              k++;
              if (d == MAXDEPTH)
                p = &res;
              else
                {
                  if (k == SIZE)
                    {
                      fprintf (stderr, "depth4: array is too short\n");
                      exit (1);
                    }
                  p = dag + k;
                  dag[k].i = i;
                  dag[k].j = j;
                  dag[k].n = n;
                  dag[k].d = d;
                }

              p->r = n ? dag[i].r - dag[j].r : dag[i].r + dag[j].r;
              p->s = n ? dag[i].s - dag[j].s : dag[i].s + dag[j].s;
              p->t = n ? dag[i].t - dag[j].t : dag[i].t + dag[j].t;
              if (p->r == res1 && p->s == res1 && p->t == res3)
                {
                  printf ("depth = %d -> Algorithm %ld\n", d, k);
                  algout (k, i, j, n);
                }
            }

      printf ("depth = %d, %ld results\n", d, k+1);
    }
  return 0;
}
```