Some issues related to double roundings

Érik Martin-Dorel, Guillaume Melquiond, Jean-Michel Muller

Some issues related to double roundings

Érik Martin-Dorel
École Normale Supérieure de Lyon, Université de Lyon, Lab. LIP

Guillaume Melquiond
INRIA

Jean-Michel Muller*
CNRS, Université de Lyon, Laboratoire LIP

Draft — July 2011 — please do not distribute

Abstract

Double rounding is a phenomenon that may occur when different floating-point precisions are available on a same system, or when performing scaled operations whose final result is subnormal. Although double rounding is, in general, innocuous, it may change the behavior of some useful small floating-point algorithms. We analyze the potential influence of double roundings on the Fast2Sum and 2Sum algorithms, on some summation algorithms, and Veltkamp's splitting. We also show how to handle possible double roundings when performing scaled Newton-Raphson division iterations (to avoid possible underflow problems).

Keywords: floating-point arithmetic; double roundings; correct rounding; 2Sum; Fast2Sum; summation algorithms; scaled division iterations.

1 Introduction

1.1 Double roundings and similar problems

When several floating-point formats are supported in a given environment, it is sometimes difficult to know in which format some operations are performed. This may make the result of a sequence of arithmetic operations somewhat difficult to predict, unless adequate compilation switches are selected. This is an issue addressed by the recent IEEE 754-2008 standard for floating-point arithmetic [14], so the situation might become more clear in the future. However, current users have to face the problem. For instance, the C99 standard states [15, Section 5.2.4.2.2]:

*This work is partly supported by the TaMaDi project of the French *Agence Nationale de la Recherche*

> the values of operations with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type.

To simplify, assume the various declared variables of a program are of the same format. Two phenomenons may occur when a wider format is available in hardware (a typical example is the "double extended format" available on Intel processors, for variables declared in the double precision/binary64 format):

- for *implicit* variables (such as the result of "`a+b`" in the expression "`d = (a+b)*c`"), it is not clear in which format they are computed. It may be preferable in most cases to compute them in the wider format;
- *explicit* variables may be first computed in the wider format, and then rounded to their destination format. This sometimes leads to a subtle problem called "double rounding" in the literature. Consider the following C program [25]:

```
double a = 1848874847.0;
double b = 19954562207.0;
double c;
c = a * b;
printf("c = %20.19e\n", c);
return 0;
```

Depending on the processor and the compilation options, we will either obtain `3.6893488147419103232e+19` or `3.6893488147419111424e+19` (which is the double-precision/binary64 number closest to the exact product). Let us explain this. The exact value of `a*b` is 36893488147419107329, whose binary representation is

$$\overbrace{\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}}10000000000}^{64\text{ bits}}01$$

If the product is first rounded to the "double-extended precision" format that is available on INTEL processors, we get (in binary)

$$\overbrace{\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}}10000000000}^{64\text{ bits}}\times 4$$

Then, if that intermediate value is rounded to the double-precision destination format, this gives (using the round-to-nearest-even rounding mode)

$$\underbrace{10000000000000000000000000000000000000000000000000000}_{53\text{ bits}} \times 2^{13}$$
$$= 36893488147419103232_{10},$$

In most applications, these phenomenons are innocuous. However, they may make the behavior of some numerical programs difficult to predict (interesting examples are given by Monniaux [24]).

Most compilers offer options that prevent this problem. For instance, on a Linux/Debian Etch 64-bit Intel platform, with GCC, the `-march=pentium4 -mfpmath=sse` compilation switches force the results of operations to be computed and stored in the 64-bit Streaming SIMD Extension (SSE) registers. However, such solutions have drawbacks:

- they significantly restrict the portability of numerical programs: e.g., it is difficult to make sure that one will always use algorithms such as 2Sum or Fast2Sum (see Section 2.2), in a large code, with the right compilation switches;
- they may have a bad impact on the performances of programs, as well as on their accuracy, since in most parts of a numerical program, it is in general more accurate to perform the intermediate calculations in a wider format.

Hence, it is of interest to examine which properties of some basic computer arithmetic building blocks remain true when some intermediate operations may be performed in a wider format and/or when double roundings may occur. When these properties suffice, the obtained programs will be much more portable and "robust". Interestingly enough, as shown by Boldo and Melquiond, double roundings could be avoided if the wider precision calculations were implemented in a special rounding mode called *rounding to odd* [4]. Unfortunately, as we are writing this paper, rounding to odd is not implemented in floating-point units.

Also, with the four arithmetic operations and the square root, one may rather easily find conditions on the precision of the wider format under which double roundings are innocuous. Such conditions have been explicited by Figueroa [9, 10] (who mentions in his paper that they probably have been given by Kahan in a course he gave in 1988). For instance, in binary floating-point arithmetic, if the "target" format is of precision $p \geq 4$ and the wider format is of precision $p+p'$ double roundings are innocuous for addition if $p' \geq p+1$, for multiplication and division if $p' \geq p$, and for square root if $p' \geq p+2$. Notice that in the most frequent case (namely, $p = 53$ and $p' = 11$) these conditions are not satisfied. For Euclidean division, the problem was addressed by Lefèvre [20].

Double roundings may also cause a problem in binary to decimal conversions. Solutions are given by Goldberg [11], and by Cornea et al [6].

A double rounding problem may also occur, even without available wider format, when performing *scaled* division iterations: to avoid overflow or underflow problems, one may multiply one of the operands of a division by some adequately chosen power of 2. A scaling of the final result is therefore required: if the final result is not in the subnormal range, that scaling is errorless. Otherwise, this induces a second rounding. We will give examples of this problem in section 6, as well as a way of avoiding it.

When the rounding mode (or direction) is towards, $+\infty$, $-\infty$ or 0, one may easily check that double roundings cannot change the result of a calculation. As a consequence, in this paper, we will focus on "round to nearest" only.

This paper is organized as follows:

- Section 2 defines some notation, recalls the standard "epsilon model" for bounding errors of floating-point operations, and the classical 2Sum and Fast2Sum algorithms; and finally gives some preliminary remarks that will be useful later on;
- Section 3 analyzes the behavior of the Fast2Sum and 2Sum algorithms in the presence of double roundings. The main results of that section are Theorems 2 and 4, that show that even if Fast2Sum or 2Sum can no longer always return the error of a floating-point addition (because that error is not always exactly representable), they will always return the floating-point number *nearest* that error;
- Fast2Sum and 2Sum are basic building blocks of many summation algorithms. In Section 5, we give some implications of the results obtained in the previous two sections to the behavior of these algorithms;
- Section 4.1 analyzes the behavior of a Veltkamp/Dekker's splitting algorithm in the presence of double roundings. That splitting algorithm allows to express' a precision-p floating-point number as the sum of a precision-s and a precision-$(p-s)$ numbers. This is an important step of Dekker's multiplication algorithm. This may also be useful for some summation algorithms;
- Finally, Section 6 shows how to deal with the double roundings that may occur when scaling Newton-Raphson-like division iterations to avoid underflows.

2 Notation, background material and preliminary remarks

2.1 Notation

2.1.1 Basic parameters of a floating-point format

Assuming extremal exponents $e_{\min}$ and $e_{\max}$, a finite *precision-p binary floating-point (FP) number* x is a number of the form

$$x = M \cdot 2^{e-p+1}, \tag{1}$$

where M and e are integers such that

$$\begin{cases} |M| \leq 2^p - 1 \\ e_{\min} \leq e \leq e_{\max} \end{cases} \tag{2}$$

The number M of largest absolute value such that (1) and (2) hold is called the *integral significand* of x, and (when $x \neq 0$) the corresponding value of e is called the *exponent* of x.

2.1.2 Even, odd, normal and subnormal numbers, underflow

We will say that a finite floating-point number is *even* (resp. *odd*) if its integral significand is even (resp. odd). A FP number is *normal* if the absolute value of its integral significand is larger than or equal to 2^{p-1}. A FP number that is not normal is called *subnormal*. A subnormal number has exponent $e_{\min}$ and its absolute value is less than $2^{e_{\min}}$.

Concerning underflow, we will follow here the rule for raising the underflow flag of the default exception handling of the IEEE 754-2008 standard [14], and say that an operation induces an underflow when the result is both subnormal and inexact.

2.1.3 Target format, wider internal format, roundings

Throughout the paper, we assume a precision-p target binary format, and a precision-$(p+p')$ wider "internal" format. When we just write that a number x is a floating-point number without explicitly giving its precision, we mean that it is a precision-p FP number. We assume that the set of possible exponents of the wider format contains the set of possible exponents of the target format, and in the following, $e_{\min}$ and $e_{\max}$ denote the extremal exponents of the target format.

$\mathrm{RN}_k(u)$ means *u rounded to the nearest precision-k FP number* (assuming round to nearest *even*: if u is exactly halfway between two consecutive precision-k FP numbers, $\mathrm{RN}_k(u)$ is the one of these two numbers that is even). When k is omitted, it means that $k = p$.

We say that $\circ$ is a *faithful rounding* if (i) when x is a FP number, $\circ(x) = x$, and (ii) when x is not a FP number, $\circ(x)$ is one of the two FP numbers that surround x.

We also say that a number x *fits in k bits* if it is equal to a precision-k FP number, or, equivalently, if in the bit string $\mathcal{S}$ constituted by the binary representation of x there is a chain of at most k consecutive bits that contains all the nonzero bits of $\mathcal{S}$.

2.1.4 ulp notation, midpoints

If $|x| \in [2^e, 2^{e+1})$, with $e \leq e_{\max}$ we define $\mathrm{ulp}_p(x)$ as the number

$$2^{\max(e,e_{\min})-p+1}.$$

When there is no ambiguity on the considered precision, we omit the "p" and just write "$\mathrm{ulp}(x)$". Roughly speaking, $\mathrm{ulp}(x)$ is the distance between two consecutive FP numbers in the neighborhood of x (but this last definition lacks rigor when $|x|$ is near a power of 2).

A *precision-p midpoint* is a number exactly halfway between two consecutive precision-p FP numbers. Notice that if x and y are real numbers, with $x \neq y$, if there is no midpoint between x and y then $\mathrm{RN}(x) = \mathrm{RN}(y)$. Notice that:

- if x is a nonzero FP number, and if $|x|$ is not a power of 2, then the two midpoints that surround x are $x - \frac{1}{2}\,\mathrm{ulp}(x)$ and $x + \frac{1}{2}\,\mathrm{ulp}(x)$;
- if $|x|$ is a power of 2 strictly larger than $2^{e_{\min}}$ and less than or equal to $2^{e_{\max}}$ then the two midpoints that surround x are $x - \mathrm{sign}(x) \cdot \frac{1}{4}\,\mathrm{ulp}(x)$ and $x + \mathrm{sign}(x) \cdot \frac{1}{2}\,\mathrm{ulp}(x)$.

2.1.5 Double roundings and double rounding slips

In the literature, the term "double rounding" either just means that two roundings occurred, or means that two roundings occurred *and* that this changed the result. To distinguish between these two events, we will say that, when the arithmetic operation $x \top y$ appears in a program:

- a *double rounding* occurs if what is actually performed is

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(x \top y)\right),$$

- a *double rounding slip* occurs if a double rounding occurs and the obtained result differs from $\mathrm{RN}_p(x \top y)$.

(a very similar definition can be given for an unary function such as $\sqrt{x}$).

2.1.6 The "standard model", or "ϵ-model"

In the FP literature, many properties are shown using the "standard model", or "ϵ-model", i.e., the fact that unless underflow or overflow occur, the computed result of an arithmetic operation $a \top b$ satisfies:

$$\mathrm{RN}(a \top b) = (a \top b)(1 + \epsilon_1) = \frac{a \top b}{1 + \epsilon_2},$$

where $|\epsilon_1|,\ |\epsilon_2| \leq u$, with $u = 2^{-p}$ (when $\top$ is the addition, that property is always true unless overflow occurs). When double roundings may occur, we get a very similar property (just by using the fact that two consecutive roundings, one in precision $p + p'$ and one in precision p, are performed):

Property 1 (ϵ-model with double roundings)**.** *Let a and b be precision-p FP numbers, and let $\top \in \{+, -, \times, \div\}$.*

- *if no underflow occurs, then*

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a \top b)\right) = (a \top b) \cdot (1 + \epsilon_1) = \frac{a \top b}{1 + \epsilon_2}, \tag{3}$$

  where $|\epsilon_1|,\ |\epsilon_2| \leq u'$, with $u' = 2^{-p} + 2^{-p-p'} + 2^{-2p-p'}$;

- *if the result of a floating-point* ***addition*** $a+b$ *has absolute value less than* $2^{e_{\min}}$*, then*

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right) = (a+b)$$

exactly, which implies that for $\top = +$*, (3) always holds, unless overflow occurs.*

Since (when p' is large enough), the bound u' is only slightly larger than u, most properties that can be shown using the ϵ-model only will remain true in the presence of double roundings (possibly with somewhat larger error bounds).

2.1.7 u, θ_k and γ_k notations

In [13, page 67], Higham defines notations θ_k and γ_k that turn out to be very useful in error analysis. We will very slightly adapt them to the context of double roundings.

Define $u = 2^{-p}$ and $u' = 2^{-p} + 2^{-p-p'} + 2^{-2p-p'}$. For any integer k, θ_k will denote a quantity of absolute value bounded by

$$\gamma_k = \frac{ku}{1-ku},$$

and θ'_k will denote a quantity of absolute value bounded by

$$\gamma'_k = \frac{ku'}{1-ku'}.$$

2.2 The 2Sum, Fast2Sum algorithms

The 2Sum algorithm was first introduced by Dekker [7]. It allows one to compute the error of a floating-point addition. Without double roundings, that algorithm is

Algorithm 1 (Fast2Sum(a,b))**.**
$s \leftarrow \mathrm{RN}(a+b)$
$z \leftarrow \mathrm{RN}(s-a)$
$t \leftarrow \mathrm{RN}(b-z)$

We have the following result.

Theorem 1 (Fast2Sum algorithm ([7], and Theorem C of [19], page 236))**.** *Assume the floating-point system being used has radix* $\beta \leq 3$*, subnormal numbers available, and provides correct rounding with rounding to nearest.*

Let a *and* b *be FP numbers, and assume that the exponent of* a *is larger than or equal to that of* b *(this condition might be difficult to check, but of course, if* $|a| \geq |b|$*, it will be satisfied). Algorithm 1 computes two FP numbers* s *and* t *that satisfy the following:*

- $s + t = a + b$ *exactly;*

- *s is the floating-point number that is closest to $a + b$.*

When we do not know in advance whether $e_a \geq e_b$ or not (or when the radix is not 2, but we do not deal with that case in this paper), it may be preferable to use the following algorithm, due to Knuth [19] and Møller [23].

Algorithm 2 ($2\mathrm{Sum}(a, b)$)**.**
$s \leftarrow \mathrm{RN}(a + b)$
$a' \leftarrow \mathrm{RN}(s - b)$
$b' \leftarrow \mathrm{RN}(s - a')$
$\delta_a \leftarrow \mathrm{RN}(a - a')$
$\delta_b \leftarrow \mathrm{RN}(b - b')$
$t \leftarrow \mathrm{RN}(\delta_a + \delta_b)$

Knuth shows that, if a and b are normal FP numbers, then for any radix β, provided that no underflow or overflow occurs, $a + b = s + t$. Boldo et al. [3] give a formal proof of this algorithm in radix 2, and show that underflow does not hinder the result.

2.3 Some preliminary remarks

Let us first notice something about Fast2Sum.

Remark 1. *The proof of Fast2Sum (see for instance the one given in [25]) relies on the fact that if s equals $\mathrm{RN}(a + b)$, then the variables z and t of algorithm Fast2Sum are computed exactly (i.e., $s - a$ and $b - z$ are FP numbers). This implies that the same result will be obtained if these variables are computed in a wider format, or with double roundings (or with a directed rounding mode). Incidently, this shows (at least in common languages) that an explicit declaration of variable z is unnecessary, and that in a program, one may safely replace* `z = s-a; t = b-z` *by* `t = b-(s-a)`.

It is well known that (unless overflow occurs), the error of a rounded-to-nearest floating-point addition of two precision-p numbers is a precision-p number (it is precisely that error that is computed by Algorithms 1 and 2). When double roundings slips occur, the results of sums are very slightly different from rounded to nearest sums. This difference, although it is very small, sometimes suffices to make the error not representable. More precisely,

Remark 2. *Assume a double rounding slip occurs when evaluating the sum s of two precision-p FP numbers a and b, i.e.,*

$$s = RN_p(RN_{p+p'}(a + b)) \neq RN_p(a + b)$$

then, if $p' \geq 1$ and $p' \leq p$ the error $r = a + b - s$ of that floating-point addition may not be exactly representable in precision-p arithmetic.

To show this, it suffices to consider

$$a = 1\underbrace{xxx\cdots x}_{p-3 \text{ bits}}01$$

where $xxxx\cdots x$ is any $(p-3)$-bit bit-chain. The number a is a p-bit integer, thus exactly representable in precision-p FP arithmetic. Also consider,

$$b = 0.0\underbrace{111111\cdots 1}_{p \text{ ones}} = \frac{1}{2} - 2^{-p-1}.$$

The number b is equal to $(2^p-1)\cdot 2^{-p-1}$, hence it is a precision-p floating-pont number too.

We have:

$$a + b = \underbrace{1xxxx...x01}_{p \text{ bits}}.0\underbrace{111111...1}_{p \text{ bits}},$$

so that if $1 \le p' \le p$,

$$u = RN_{p+p'}(a+b) = 1xxxx...x01.100...0,$$

The "round to nearest even" rule thus implies

$$s = RN_p(u) = 1xxxx...x10 = a + 1$$

Therefore,

$$s - (a+b) = a + 1 - (a + \frac{1}{2} - 2^{-p-1}) = \frac{1}{2} + 2^{-p-1} = 0.\underbrace{10000\cdots 01}_{p+1 \text{ bits}},$$

which is not exactly representable in precision-p FP arithmetic. □

Remark 3. *Assume we compute $w = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(u+v))$, where u and v are precision-p, radix-2, FP numbers of exponents e_u and e_v, with $e_u \ge e_v$. If $p' \ge 2$ and a double-rounding slip occurs in that computation, then*

$$e_u - p - 1 \le e_v \le e_u - p', \tag{4}$$

and

$$e_w \ge e_v + p' + 1. \tag{5}$$

Proof of (4).
First, if $e_u - p - 1 > e_v$ (i.e., $e_u - p - 2 \ge e_v$), then

$$|v| < 2^{e_v+1} \le 2^{e_u-p-1} = \frac{1}{4}\,\mathrm{ulp}(u).$$

Also, $p' \ge 2$ implies that $u - \frac{1}{4}\,\mathrm{ulp}(u)$ and $u + \frac{1}{4}\,\mathrm{ulp}(u)$ are precision-$(p+p')$ FP numbers. Therefore,

$$u - \frac{1}{4}\,\mathrm{ulp}(u) \le \mathrm{RN}_{p+p'}(u+v) \le u + \frac{1}{4}\,\mathrm{ulp}(u).$$

Therefore,

- if $|u|$ is not a power of 2 then $\mathrm{RN}_{p+p'}(u+v)$ cannot be a precision-p midpoint. The final result follows from Remark 5 below;
- if $|u| = 2^{e_u}$ exactly, then $|u| - \frac{1}{4}\,\mathrm{ulp}(u)$ is a midpoint (it is the only one between $|u| - \frac{1}{4}\,\mathrm{ulp}(u)$ and $|u| + \frac{1}{4}\,\mathrm{ulp}(u)$), but $e_v \leq e_u - p - 2$ implies $|v| < 2^{e_u-p-1} = \frac{1}{4}\,\mathrm{ulp}(u)$, so that the real value of $u+v$ is between the midpoint $\mu = u - \mathrm{sign}(u) \cdot \frac{1}{4}\,\mathrm{ulp}(u)$ and u. Since $\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(\mu)\right) = u = \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(u)\right)$ and the roundings are monotonic functions, we have $\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(u+v)\right) = \mathrm{RN}(u+v) = u$, so that no double rounding slip occurs.

Second, if $e_v > e_u - p'$, then $u+v$ can be written $2^{e_v-p+1}\left(2^{e_u-e_v}M_u + M_v\right)$, where M_u and M_v are the integral significands of u and v, and the integer $2^{e_u-e_v}M_u + M_v$ satisfies

$$\left|2^{e_u-e_v}M_u + M_v\right| \leq 2^{p'-1}(2^p-1) + (2^p-1) \leq 2^{p+p'} - 1.$$

Therefore $u+v$ is exactly representable in precision $p+p'$, so that no double rounding slip can occur.

Proof of (5).
Let k be the integer such that $2^k \leq |u+v| < 2^{k+1}$. The monotonicity of the rounding functions implies that

$$2^k \leq |\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u+v))| \leq 2^{k+1},$$

therefore, e_w is equal to k or $k+1$. Since $u+v$ does not fit into $p+p'$ bits (otherwise there would not be a double rounding slip) and $u+v$ is a multiple of 2^{e_v-p+1}, we deduce

- if $e_w = k$ then $\mathrm{ulp}_{p+p'}(u+v) > e_b - p + 1$, therefore $e_w - p - p' + 1 > e_b - p + 1$, which implies $e_w \geq e_v + p' + 1$,
- if $e_w = k+1$ then $\mathrm{ulp}_{p+p'}(u+v) > e_b - p + 1$, therefore $(e_w - 1) - p - p' + 1 > e_b - p + 1$, which implies $e_w \geq e_v + p' + 2$.

□

Notice that the condition "$p' \leq p$" in Remark 2 is necessary. More precisely,

Remark 4. *If $p' \geq p+1$ then a double rounding slip cannot occur when computing $a+b$, i.e., we always have*

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right) = \mathrm{RN}_p(a+b).$$

(this is a classical result [9, 10]. A sketch of the proof is the following—Assume, without l.o.g., that $|a| \geq |b|$: if $e_b \geq e_a - p - 1$ then $a+b$ fits in at most $2p+1$ bits, so that as soon as $p' \geq p+1$, $\mathrm{RN}_{p+p'}(a+b) = a+b$ exactly; and if $e_b < e_a - p - 1$, then Equation (4) in Remark 3 implies that no double rounding slip can occur).

Remark 5. *Assume, $p' \geq 1$, if a double rounding slip occurs when evaluating $a \top b$ (where $\top$ is any operation) then* $\mathrm{RN}_{p+p'}(a \top b)$ *is a* precision-p midpoint, *i.e., a number exactly halfway between two consecutive precision-p FP numbers.*

The proof of Remark 5 is common arithmetic folklore, and just uses the fact that roundings are monotonic functions. Let us give it anyway for the sake of completeness. If $\mathrm{RN}_p(a \top b) \neq \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a \top b))$ then there is a precision-p midpoint, say μ, between $a \top b$ and $\mathrm{RN}_{p+p'}(a \top b)$. That number satisfies $|(a \top b) - \mu| \leq |(a \top b) - \mathrm{RN}_{p+p'}(a \top b)|$. μ fits in $(p+1)$ bits. Since $p' \geq 1$, it is a precision-$(p+p')$ FP number. Since by definition $\mathrm{RN}_{p+p'}(a \top b)$ is a a precision-$(p+p')$ FP number nearest $a \top b$, we have:

- either there is only one precision-$(p+p')$ FP number nearest $a \top b$ (i.e., $a \top b$ is not a precision-$(p+p')$ midpoint), in such a case we necessarily have $\mathrm{RN}_{p+p'}(a \top b) = \mu$;
- or $a \top b$ is a precision-$(p+p')$ midpoint. In such a case, if $\mathrm{RN}_{p+p'}(a \top b) \neq \mu$, then either μ is above $a \top b$ and $\mathrm{RN}_{p+p'}(a \top b)$ is below $a \top b$, or μ is below $a \top b$ and $\mathrm{RN}_{p+p'}(a \top b)$ is above $a \top b$: in any case, μ cannot be between $a \top b$ and $\mathrm{RN}_{p+p'}(a \top b)$, which is a contradiction. □

An immediate consequence of Remark 5 (due to the round-to-nearest-even rule) is the following.

Remark 6. *Assume, $p' \geq 1$, if a double rounding slip occurs when evaluating $a \top b$ then the returned result* $\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a \top b)\right)$ *is an even FP number.*

In our proofs, we will also frequently use the following, well-known, result.

Remark 7 (Sterbenz Lemma [35])**.** *If a and b are positive FP numbers, and*

$$\frac{a}{2} \leq b \leq 2a,$$

then $a - b$ is a floating-point number (which implies that it will be computed exactly, whatever the rounding).

Finally, the following result will be used later on to prove that even when the error of a floating-point addition is not exactly representable because of a double rounding slip, as soon as $p' \geq 2$, we are anyway able to compute the floating-point number nearest that error.

Remark 8. *Let a and b be precision-p FP numbers, and define*

$$s = \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}\left(a + b\right)\right).$$

The number $r = a + b - s$ fits in at most $p + 2$ bits, so that as soon as $p' \geq 2$, we have

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}\left(a + b - s\right)\right) = \mathrm{RN}_p(a + b - s). \tag{6}$$

Proof
Without l.o.g., we assume $e_a \geq e_b$. First, we already know that if no double rounding slip occurs when computing s, namely if $\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}\left(a+b\right)\right) = \mathrm{RN}_p(a+b)$, then $a+b-s$ is a precision-p FP number. In such a case, (6) is obviously true. So let us assume that $\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}\left(a+b\right)\right) \neq \mathrm{RN}_p(a+b)$. Equation (4) in Remark 3 implies therefore that $e_b \geq e_a - p - 1$.

Since a and b are both multiple of 2^{e_b-p+1}, we easily deduce that $a+b-s$ too is a multiple of 2^{e_b-p+1}. Also, since $|a|$ and $|b|$ are less than $(2^p-1)\cdot 2^{e_a-p+1}$, $|a+b|$ is less than $(2^p-1)\cdot 2^{e_a-p+2}$, so that (since roundings are monotonic functions and $(2^p-1)\cdot 2^{e_a-p+2}$ is an FP number) s too is of absolute value less than or equal to $(2^p-1)\cdot 2^{e_a-p+2}$, therefore $e_s \leq e_a + 1$.

From all this, we deduce that $a+b-s$ is a multiple of 2^{e_b-p+1} of absolute value less than or equal to

$$2^{-p+e_s} + 2^{-p-p'+e_s} \leq 2^{-p+e_a+1} + 2^{-p-p'+e_a+1} \leq 2^{e_b+2} + 2^{e_b-p'+2}.$$

Hence, $r = a+b-s$ fits in at most $p+2$ bits, therefore, as soon as $p' \geq 2$, $\mathrm{RN}_{p+p'}(a+b-s) = a+b-s$. q.e.d. □

3 Behavior of Fast2Sum and 2Sum in the presence of double roundings

3.1 Fast2Sum and double roundings

Remark 2 implies that Algorithms Fast2Sum and 2Sum cannot always return the exact value of the error when the addition $\mathrm{RN}(a+b)$ is replaced by

$$\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right),$$

i.e., when a double rounding occurs, because that error is not always exactly equal to a floating-point number.

And yet, we may try to bound the difference between the exact error and the returned number t (indeed, we will prove that t is the FP number nearest the exact error). Let us analyze how algorithm Fast2Sum behaves when double roundings are allowed. Assume $e_a \geq e_b$, we will consider that what is actually performed is

Algorithm 3 (Fast2Sum-with-double-roundings(a,b))**.**
$s \leftarrow \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right)$
$z \leftarrow \circ(s-a)$
$t \leftarrow \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(b-z)\right)$ *or* $\mathrm{RN}_p(b-z)$

where $\circ(u)$ means either $\mathrm{RN}_p(u)$, $\mathrm{RN}_{p+p'}(u)$, or $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u))$, or any faithful rounding (this is not important, as we will see that $s-a$ is exactly representable in precision-p FP arithmetic, so that it will be computed exactly,

whatever the rounding: this means that in a program, one may safely replace `z = s-a; t = b-z` by `t = b-(s-a)`).

Define e_a, e_b, and e_s as the floating-point exponents of a, b, and s, and M_a, M_b, and M_s as their significands. We assume $e_a \geq e_b$ (that condition will be satisfied if $|a| \geq |b|$). Notice that this implies $e_s \leq e_a + 1$, since $|s| \leq 2 \cdot \max\{|a|, |b|\} \leq 2 \cdot (2^p - 1) \cdot 2^{e_a - p + 1}$. Without l.o.g., we assume $s \geq 0$ (otherwise if suffices to change the signs of a and b). We have

$$a = M_a \cdot 2^{e_a - p + 1},$$

with $|M_a| \leq 2^p - 1$, and similar relations for b and s. Also, notice that

$$2^{p-1} \leq \frac{s}{2^{e_s - p + 1}} \leq 2^p - 1$$

implies

$$2^{p-1} - \frac{1}{4} \leq \frac{\mathrm{RN}_{p+p'}(a+b)}{2^{e_s - p + 1}} < 2^p - \frac{1}{2},$$

which implies

$$2^{p-1} - \frac{1}{4} - 2^{-p'-2} \leq \frac{a+b}{2^{e_s - p + 1}} < 2^p - \frac{1}{2} + 2^{-p'-1}.$$

1. **if** $e_s = e_a + 1$, define $\delta = e_a - e_b$. We have,

$$a + b = 2^{e_s - p + 1} \left(\frac{M_a}{2} + \frac{M_b}{2^{\delta+1}} \right),$$

from which we deduce

$$\mathrm{RN}_{p+p'}(a+b) = 2^{e_s - p + 1} \left(\frac{M_a}{2} + \frac{M_b}{2^{\delta+1}} + \epsilon \right),$$

where $|\epsilon| \leq 2^{-p'-1}$. Therefore, we have,

$$M_s = \left\lceil \frac{M_a}{2} + \frac{M_b}{2^{\delta+1}} + \epsilon \right\rfloor,$$

where $\lceil u \rfloor$ is the integer nearest to u (with round-to-even choice in case of a tie). Now, define $\mu = 2M_s - M_a$. We have,

$$\frac{M_b}{2^\delta} - 1 - 2^{-p'} \leq \mu \leq \frac{M_b}{2^\delta} + 1 + 2^{-p'}.$$

Since μ is an integer, $\delta \geq 0$, and $|M_b| \leq 2^p - 1$, if $p' \geq 1$, then either $|\mu| \leq 2^p - 1$, or $\mu = \pm 2^p$. In both cases, since $s - a = \mu \cdot 2^{e_a - p + 1}$, $s - a$ is exactly representable in precision p.

2. **if** $e_s \leq e_a$, define $\delta_1 = e_a - e_b$. We have

$$a + b = \left(2^{\delta_1} M_a + M_b\right) \cdot 2^{e_b - p + 1}.$$

- if $e_s \leq e_b$ then $s = a+b$ exactly, since s is obtained by rounding $a+b$ first to the nearest multiple of $2^{e_s-p-p'+1}$—or to the nearest multiple of $2^{e_s-p-p'}$ in case $|a+b|$ is less than $2^{e_s} - 2^{e_s-p-p'-1}$—which is a divisor of 2^{e_b-p+1}, and then to the nearest multiple of 2^{e_s-p+1} (which is a divisor of 2^{e_b-p+1} too). These two rounding operations left it unchanged since it is already a multiple of 2^{e_b-p+1}. Hence in the case $e_s \leq e_b$, $s - a = b$ is exactly representable;
- if $e_s > e_b$, let us define $\delta_2 = e_s - e_b$. We have,

  $$s = \left\lceil 2^{\delta_1-\delta_2} M_a + 2^{-\delta_2} M_b + \epsilon \right\rfloor \cdot 2^{e_s-p+1},$$

  where $|\epsilon| \leq 2^{-p'-1}$. This implies

  $$|s - a| \leq \left(2^{-\delta_2} M_b + \frac{1}{2} + 2^{-p'-1} \right) \cdot 2^{e_s-p+1}.$$

  Also, since $e_s \leq e_a$, $s-a$ is a multiple of 2^{e_s-p+1}. We therefore have,

  $$s - a = K \cdot 2^{e_s-p+1},$$

  where K is an integer satisfying

  $$|K| \leq 2^{-\delta_2}|M_b| + \frac{1}{2} + 2^{-p'-1} < 2^{p-1} + 1 < 2^p - 1.$$

  (as soon as $p \geq 3$ and $p' \geq 1$, which holds in all cases of practical interest). Hence, $s-a$ is exactly representable in precision-p floating-point arithmetic.

We have shown that in all cases, $s-a$ is exactly representable in precision-p FP arithmetic. Hence, variable z of the algorithm will be exactly computed (and the way it is rounded—correct rounding to precision p, correct rounding to an "extended", precision-$(p+p')$ format, double rounding, directed rounding—has no influence on this). Now, from $z = s - a$, we immediately deduce $b - z = (a+b) - s = r$, so that

$$t = \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(r)\right).$$

Using Remark 8, we immediately deduce

$$t = \mathrm{RN}_p\left(r\right).$$

In other words, each time r is exactly representable (which happens every time a double rounding slip does not occur when computing $a+b$), we get it exactly; and when r is not exactly representable, we get the precision-p FP number nearest to r. The following theorem summarizes the obtained results.

Theorem 2. *Assume a binary target floating-point format of precision $p \geq 3$, assume a binary format of precision $p+p'$, with $p' \geq 2$, is available. If a and b are precision-p numbers, with $e_a \geq e_b$ (that condition will be satisfied if $|a| \geq |b|$), and if no overflow occurs, then the sequence of calculations*

$s \leftarrow \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right)$
$z \leftarrow \circ(s-a)$
$t \leftarrow \odot(b-z)$

(where $\circ(u)$ means either $\mathrm{RN}_p(u)$, $\mathrm{RN}_{p+p'}(u)$, or $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u))$, and $\odot(u)$ means either $\mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(u)\right)$ or $\mathrm{RN}_p(u)$) satisfies the following property:

- *$z = s - a$ exactly (this will be useful later on in the paper);*
- *if no double rounding slip occurred when computing s (in other words, if $s = \mathrm{RN}_p(a+b)$), then $t = (a+b-s)$ exactly;*
- *otherwise, $t = \mathrm{RN}_p(a+b-s)$.*

3.2 2Sum and double roundings

The following preliminary lemma is directly adapted from Lemma 4 in Shewchuk's paper [34].

Lemma 3. *Let a and b be precision-p binary floating-point numbers. Let $s = \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right)$ or $\mathrm{RN}_p(a+b)$. If $|s| < \min\{|a|, |b|\}$ then $s = a + b$ (that is, s is computed exactly).*

Proof. Define $\sigma = a+b$. Without loss of generality, we assume that $\min\{|a|, |b|\} = |b|$. Define e_b as the exponent of b and M_b as its integral significand (i.e., $b = M_b \cdot 2^{e_b - p+1}$). Since a and b are both multiples of 2^{e_b-p+1}, σ is a multiple of 2^{e_b-p+1} too. Also, due to the monotonicity of rounding, $|s| < |b|$ implies $|\sigma| < |b|$. An immediate consequence is that $\sigma/2^{e_b-p+1}$ is an integer of absolute value less than $|M_b| \leq 2^p - 1$. This implies that σ is a precision-p floating-point number. Therefore $\mathrm{RN}_{p+p'}(\sigma) = \sigma$, and $\mathrm{RN}(\sigma) = \sigma$, so that $s = \sigma$, q.e.d. □

We will analyze the following algorithm

Algorithm 4 (2Sum-with-double-roundings(a, b))**.**
(1) $s \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a+b))$ *or* $\mathrm{RN}_p(a+b)$
(2) $a' \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(s-b))$ *or* $\mathrm{RN}_p(s-b)$
(3) $b' \leftarrow \circ(s-a')$
(4) $\delta_a \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a-a'))$ *or* $\mathrm{RN}_p(a-a')$
(5) $\delta_b \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(b-b'))$ *or* $\mathrm{RN}_p(b-b')$
(6) $t \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(\delta_a+\delta_b))$ *or* $\mathrm{RN}_p(\delta_a+\delta_b)$

where $\circ(u)$ is either $\mathrm{RN}_p(u)$, $\mathrm{RN}_{p+p'}(u)$, or $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u))$, or any faithful rounding (indeed, we will show that $s - a'$ is a precision-p FP number, so that b' will be computed exactly).

First, let us raise the following point:

Remark 9. *Assuming $p' \geq 2$, if the variables a' and b' of Algorithm 4 satisfy $a' = a$ and $b' = s - a'$ exactly, then $t = \mathrm{RN}(a+b-s)$.*

Proof.
If $a' = a$ then $\delta_a = 0$. Also, $b' = s - a' = s - a$ implies $b - b' = a + b - s$, so that $\delta_b = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a + b - s))$. This gives

$$t = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a + b - s)) = \mathrm{RN}_p(a + b - s),$$

using Remark 8 □

Let us now analyze Algorithm 4.

3.2.1 Behavior of Algorithm 4 in the case $|b| \geq |a|$

In the case $|b| \geq |a|$, lines (1), (2), and (4) of Algorithm 4 constitute Fast2Sum-with-double-roundings(b, a), i.e., Agorithm 3, called with input values (b, a). A consequence of this (see Theorem 2) is that the computation of line (2) is exact, which implies $a' = s - b$ and $\delta_a = \mathrm{RN}_p(a + b - s)$. Also, $a' = s - b$ implies $s - a' = b$, so that $b' = b$ exactly and $\delta_b = 0$. All this implies

$$t = \mathrm{RN}_p(a + b - s).$$

3.2.2 Behavior of Algorithm 4 in the case $|b| < |a|$ and $|s| < |b|$

If $|b| < |a|$ and $|s| < |b|$, then Lemma 3 applies: $s = a + b$ exactly, which implies $a' = a$, $b' = b$, and $\delta_a = \delta_b = t = 0$.

3.2.3 Behavior of Algorithm 4 in the case $|b| < |a|$ and $|s| \geq |b|$

Let us now assume $|b| < |a|$ and $|s| \geq |b|$. These inequalities have two important consequences:

- we have $s = (a + b) \cdot (1 + \epsilon_1)$ and $a' = (s - b) \cdot (1 + \epsilon_2)$, with $|\epsilon_1|, |\epsilon_2| \leq u'$ (we remind the reader that $u' = 2^{-p} + 2^{-p-p'} + 2^{-p-2p'}$, see Section 2.1.7), from which we easily deduce

  $$a' = (a + a\epsilon_1 + b\epsilon_1) \cdot (1 + \epsilon_2) = a \cdot (1 + \epsilon_3),$$

  with $|\epsilon_3| \leq 3u' + 2u'^2$. An immediate consequence is that as soon as $p \geq 4$ and $p' \geq 1$ (which holds in all practical cases), $|a/2| \leq |a'| \leq |2a|$, and a and a' have the same sign. Hence, from Sterbenz Lemma (Remark 7), $a - a'$ is a precision-p floating-point number, which implies $\delta_a = a - a'$ exactly. Also (which will be useful later on), $e'_a \leq e_a + 1$.

- lines (2), (3), and (5) constitute Fast2Sum (with double roundings, i.e., Algorithm 3), called with input values $(s, -b)$. This implies that $b' = s - a'$ exactly, and $\delta_b = \mathrm{RN}_p(a' - (s - b))$. Moreover, Theorem 2 shows that $\delta_b = a' - (s - b)$ exactly if no double rounding slip occurred in line (2): in such a case, $\delta_a + \delta_b = (a + b - s)$, which implies $t = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a + b - s)) = \mathrm{RN}_p(a + b - s)$ (using Remark 8).

So, in the following, we assume that a double rounding slip occurred in line (2), i.e., when computing $s-b$. Notice that this implies from Remark 6 that a' is even. Notice that Equation (5) in Remark 3 implies that $e_b \leq e'_a - p' - 1$.

Also, we know that

- $\delta_a = a - a'$ and $b' = s - a'$ exactly;
- all variables $(s, a', b', \delta_a, \delta_b, t)$ are multiples of $2^{e_b - p + 1}$.

Since a double rounding slip occurred in line (2), we have

$$s - b = a' + i \cdot 2^{e'_a - p} + j \cdot \epsilon,$$

where $i = \pm 1$ (or $\pm\frac{1}{2}$ in the case a' is a power of 2), $j = \pm 1$ and $0 \leq \epsilon \leq 2^{e'_a - p - p'}$. Since $b' = s - a'$ exactly, we deduce

$$b' = b + i \cdot 2^{e'_a - p} + j \cdot \epsilon,$$

hence,

$$b - b' = -i \cdot 2^{e'_a - p} - j \cdot \epsilon,$$

so that, since it is a multiple of $2^{e_b - p + 1}$, $b - b'$ fits in at most $(e'_a - p) - (e_b - p + 1) + 1 = e'_a - e_b \leq e_a - e_b + 1$ bits. Hence, if $e_a - e_b \leq p - 1$ then $b - b'$ is a precision-p floating-point number, therefore $\delta_b = b - b'$ exactly. It follows that $\delta_a + \delta_b = a - a' + b - b' = a - a' + b - (s - a') = a + b - s$, so that $t = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a + b - s)) = \mathrm{RN}_p(a + b - s)$ (using Remark 8).

Furthermore, if $e_a - e_b \geq p + 2$, then one easily checks that $s = a' = a$, $b' = \delta_a = 0$, and $t = \delta_b = b$, which is the desired result.

Hence the last case that remains to be checked is the case $e_a - e_b \in \{p, p + 1\}$. Notice that in that case, if a is not a power of 2, s is necessarily equal to a^-, a, or a^+, where a^- and a^+ are the floating-point predecessor and successor of a. If a is a power of 2, s can also be equal to a^{--} (when $a > 0$) or a^{++} (when $a < 0$). To simplify the presentation, we now assume $a > 0$ (otherwise, it suffices to change the signs of a and b).

1. **if a is not a power of 2,** then s is equal to a^-, a, or a^+. Notice that $s = a^- \Rightarrow a' \geq s$ (because in that case, $b < 0$), and $s = a^+ \Rightarrow a' \leq s$.

   - if $|b|$ is not of the form $\pm 2^{e_a - p} + \epsilon$ with $|\epsilon| \leq 2^{e_a - p - p'}$, then there are no double rounding slips in lines (1) and (2) of the algorithm;
   - otherwise, if a is even, then (due to the round to nearest *even* rounding rule) $s = a$, and $a' = s = a$, therefore Remark 9 implies $t = \mathrm{RN}_p(a + b - s)$;
   - otherwise, if a is odd, then (still due to the round to nearest *even* rounding rule) $s = a^+$ or a^- and $a' = s$, so that $b' = 0$, which implies $\delta_b = b$ and $t = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a - a' + b)) = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a + b - s)) = \mathrm{RN}_p(a + b - s)$.

2. **if a is a power of 2,** i.e., $a = 2^{e_a}$. Notice again that $s < a \Rightarrow a' \geq s$ (because in that case, $b < 0$), and $s > a \Rightarrow a' \leq s$.

   - if $b \geq 0$, then s is equal to a, or a^+.
     - If $s = a^+$ then if $a' = a^+$ there is no double rounding slip in line (2) of the algorithm since a' is odd (using Remark 6), and if $a' = a$ then Remark 9 implies $t = \mathrm{RN}_p(a + b - s)$;
     - now, if $s = a$ then if $a' = a$ Remark 9 implies $t = \mathrm{RN}_p(a+b-s)$, and if $a' = a^-$ then (since a' is odd) there is no double rounding slip in line (2) of the algorithm (using Remark 6).
   - if $b < 0$, then s is equal to a, a^-, or a^{--}.
     - if $s = a$ then $b \geq -2^{e_a-p-1} - 2^{e_a-p-p'-1}$. In such a case, there is no double rounding slip when computing $s - b$, i.e., in line (2) of the algorithm;
     - if $s = a^-$ then if $a' = a^-$ then there is no double rounding slip in line (2) since a^- is odd, and if $a' = a$ then Remark 9 implies $t = \mathrm{RN}_p(a + b - s)$;
     - if $s = a^{--}$ then if $a' = a^-$ then there is no double rounding slip in line (2) since a^- is odd; if $a' = a$ then Remark 9 implies $t = \mathrm{RN}_p(a+b-s)$; and $a' = a^{--}$ is impossible ($s = a^{--}$ implies $-b \geq 3 \cdot 2^{e_a-p-1} - 2^{e_a-p-p'-1}$, from which we deduce $s - b > a^-$, which implies $a' \geq a^-$).

We therefore deduce

Theorem 4. *Assume a radix-2 target floating-point format of precision $p \geq 4$, assume that a format of precision $p + p'$, with $p' \geq 2$, is available. If a and b are precision-p numbers, and if no overflow occurs, then Algorithm 4 satisfies the following property:*

- *if no double rounding slip occurred when computing s (in other words, if $s = \mathrm{RN}_p(a + b)$), then $t = (a + b - s)$ exactly;*
- *otherwise, $t = \mathrm{RN}_p(a + b - s)$.*

Notice that an immediate consequence of Theorems 2 and 4 is

Corollary 1. *The values s and t returned by Algorithms 3 or 4 satisfy*

$$(s + t) = (a + b)(1 + \eta),$$

with $|\eta| \leq 2^{-2p} + 2^{-2p-p'} + 2^{-3p-p'}$.

It may be of interest to notice that, even when a double rounding slip occurred when computing s in fast2Sum or 2Sum, $(a + b) - s$ will very often be exactly representable. More exactly,

Remark 10. *Assume $p' \geq 2$, let a and b be precision-p FP numbers, with $e_a \geq e_b$, such that*

$$s = \mathrm{RN}_p\left(\mathrm{RN}_{p+p'}(a+b)\right) \neq \mathrm{RN}_p(a+b),$$

if $a+b-s$ is not a precision-p FP number, then $e_b = e_a - p - 1$.

Proof. Assume that $a+b-s$ is not a precision-p FP number. Without l.o.g., we assume $a+b \geq 0$. First, Remark 3 implies

$$e_a - p - 1 \leq e_b \leq e_a - p'. \tag{7}$$

Also, Remark 5 implies that $\mathrm{RN}_{p+p'}(a+b)$ has the form $g + \frac{1}{2}\,\mathrm{ulp}_p(g)$, where g is a precision-p FP number, which means that

$$a + b = g + \frac{1}{2}\,\mathrm{ulp}_p(g) + \epsilon,$$

with

$$|\epsilon| \leq \frac{1}{2}\,\mathrm{ulp}_{p+p'}(g) \text{ and } \epsilon \neq 0.$$

Therefore, we have

$$s = \begin{cases} g & \text{if } g \text{ is even} \\ g^+ = g + \mathrm{ulp}(g) & \text{otherwise.} \end{cases}$$

and

$$\mathrm{RN}_p(a+b) = \begin{cases} g & \text{if } \epsilon < 0 \\ g^+ & \text{if } \epsilon > 0. \end{cases}$$

Hence, s and $\mathrm{RN}_p(a+b)$ differ in two cases:

1. if g is even and $\epsilon > 0$, in which case

$$a + b - s = \frac{1}{2}\,\mathrm{ulp}_p(g) + \epsilon;$$

2. if g is odd and $\epsilon < 0$, in which case

$$a + b - s = -\frac{1}{2}\,\mathrm{ulp}_p(g) + \epsilon.$$

Now, notice that ϵ is an integer multiple of $\mathrm{ulp}(b)$. Eq. (7) implies that e_g is $e_a - 1$, e_a, or $e_a + 1$. Furthermore,

- If $e_g = e_a - 1$, then $1/2\,\mathrm{ulp}(e_g) = 2^{e_a - p - 1} \leq 2^{e_b}$, so that $\pm 1/2\,\mathrm{ulp}(e_g) + \epsilon$ is representable in precision-p FP arithmetic;

- If $e_g = e_a$, then the leftmost bit of the binary representation of $\pm\frac{1}{2}\operatorname{ulp}_p(g)+\epsilon$ is of weight $\leq 2^{e_g-p}$, whereas its rightmost nonzero bit has weight $\geq 2^{e_b-p+1}$. Hence, if $\pm\frac{1}{2}\operatorname{ulp}_p(g)+\epsilon$ is not a precision-p FP number then

$$e_b - p + 1 < (e_g - p) - p + 1,$$

which implies

$$e_b \leq e_a - p - 1.$$

Combined with (7) this gives

$$e_b = e_a - p - 1.$$

- If $e_g = e_a + 1$, reasoning as previously, we find that if $\pm\frac{1}{2}\operatorname{ulp}_p(g)+\epsilon$ is not a precision-p FP number then

$$e_b - p + 1 < (e_g - p) - p + 1,$$

which implies

$$e_b \leq e_a - p.$$

However, if $e_b \leq e_a - p$, then $|b| < \operatorname{ulp}_p(a) = 2^{e_a-p+1}$, therefore (since $|a| \leq (2^p - 1)\cdot 2^{e_a-p+1}$), $|a+b| < 2^{e_a+1}$, which implies

$$\operatorname{RN}_{p+p'}(a+b) \leq 2^{e_a+1}.$$

Hence, $\operatorname{RN}_{p+p'}(a+b)$ cannot be of the form $g+\frac{1}{2}\operatorname{ulp}(g)+\epsilon$, with $e_g = e_a+1$ and $|\epsilon| \leq \frac{1}{2}\operatorname{ulp}_{p+p'}(g)$. □

A consequence of Remark 10 will be of interest when discussing the Rump, Ogita and Oishi Splitting algorithm (useful for designing a summation algorithm):

Remark 11. *If a is a power of 2 (say, $a = 2^{e_a}$) and $|b| \leq a$, then the values s and t returned by Algorithm 3 or Algorithm 4 satisfy*

$$t = a + b - s$$

exactly.

Proof. Suppose we have $t \neq a + b - s$. We know that this cannot happen if $s = \operatorname{RN}_p(a+b)$, so we necessarily have

$$\operatorname{RN}_p\left(\operatorname{RN}_{p+p'}(a+b)\right) \neq \operatorname{RN}_p(a+b),$$

and $a + b - s$ is not a FP number. Remark 10 implies $e_b = e_a - p - 1$, so that

$$|b| < \frac{1}{2}\operatorname{ulp}(a).$$

- if $b > 0$ then the only case for which we may have a double rounding slip (according to Remark 5, and the fact that $a + b < a + \frac{1}{2}\,\mathrm{ulp}_p(a) \Rightarrow \mathrm{RN}_{p+p'}(a+b) \leq a + \frac{1}{2}\,\mathrm{ulp}(a)$) is

$$\mathrm{RN}_{p+p'}(a+b) = 2^{e_a} + \frac{1}{2}\,\mathrm{ulp}(a),$$

- if $b < 0$ then then the only case for which we may have a double rounding slip (still according to Remark 5, and the fact that $a + b > a - \frac{1}{2}\,\mathrm{ulp}(a) = a^-$, so that $\mathrm{RN}_{p+p'}(a+b) \geq a^-$) is

$$\mathrm{RN}_{p+p'}(a+b) = 2^{e_a} - \frac{1}{4}\,\mathrm{ulp}(a).$$

In both cases, the round-to-nearest-*even* rounding rule implies $s = 2^{e_a} = a$, so that $a + b - s = b$, which contradicts the assumption that $a + b - s$ is not a FP number. This proof is illustrated by Figure 1. □

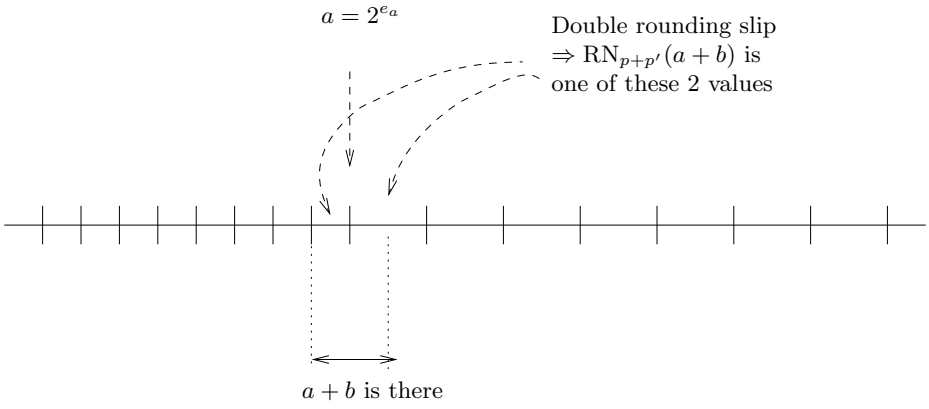

Figure 1: This figure illustrates the fact that when a is a power of 2, we have $t = a + b - s$ exactly. Here, if a double rounding slip occurs, $\mathrm{RN}_{p+p'}(a+b)$ can take two possible values only, and for each of them $s = a$.

4 Splitting algorithms in the presence of double roundings

4.1 Veltkamp's splitting algorithm in the presence of double roundings

In some applications (for instance to implement Dekker's multiplication algorithm, that allows one to express the product of two FP numbers exactly as the

unevaluated sum of two FP numbers), we need to "split" a radix-β, precision-p floating-point number x into two FP numbers x_h and x_ℓ such that, for a given $s \leq p-1$, x_h fits in $p-s$ digits, x_ℓ fits in s digits, and $x = x_h + x_\ell$ exactly.

Veltkamp's algorithm (Algorithm 5) can be used for that purpose [7]. It uses a floating-point constant C equal to $\beta^s + 1$ (which is exactly representable in precision-p floating-point arithmetic, as soon as $s \leq p-1$).

Algorithm 5 (Split(x,s): Veltkamp's algorithm, presented here for a radix-β floating-point format.)**.**
Require: $C = \beta^s + 1$
$\gamma \leftarrow \mathrm{RN}(C \cdot x)$
$\delta \leftarrow \mathrm{RN}(x - \gamma)$
$x_h \leftarrow \mathrm{RN}(\gamma + \delta)$
$x_\ell \leftarrow \mathrm{RN}(x - x_h)$

Boldo [1] shows that for any radix β and any precision p, provided that $C \cdot x$ does not overflow, the algorithm works. Moreover, if $\beta = 2$, then x_ℓ actually fits in $s-1$ bits, which is an important feature when one wishes to implement Dekker's product algorithm with a binary floating-point format of odd precision. Let us see what happens if double roundings may occur. More precisely, assuming $\beta = 2$, we will analyze the following algorithm:

Algorithm 6 (Split(x,s): Veltkamp's algorithm with possible double roundings. x is a radix-2, precision-p floating-point number.)**.**
Require: $C = 2^s + 1$
(1) $\gamma \leftarrow \mathrm{RN}_p(C \cdot x)$ *or* $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(C \cdot x))$
(2) $\delta \leftarrow \mathrm{RN}_p(x - \gamma)$ *or* $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(x - \gamma))$
(3) $x_h \leftarrow \circ(\gamma + \delta)$
(4) $x_\ell \leftarrow \circ(x - x_h)$

where $\circ(u)$ can be either $\mathrm{RN}_p(u)$, $\mathrm{RN}_{p+p'}(u)$, or $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(u))$, or any faithful rounding (indeed, we will show that $\gamma + \delta$ and $x - x_h$ are precision-p floating-point numbers, so that x_h and x_ℓ will be computed exactly).

Theorem 5. *Assuming binary floating-point arithmetic. If $2 \leq s \leq p-2$, $p \geq 5$, $p' \geq 2$ and no overflow occurs, then the values x_h and x_ℓ returned by Algorithm 6 satisfy:*

- $x = x_h + x_\ell$*;*
- x_h *fits in* $p - s$ *bits;*
- x_ℓ *fits in* s *bits.*

Notice that we can no longer be sure that x_ℓ fits in $s-1$ bits (unless no double rounding slip occurs at step (2)). This is illustrated by the following example: assume $p = 11$, $p' = 3$, $s = 5$, and $x = 1041_{10} = 10000010001_2$. We run algorithm 6 with rounding $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(\cdot))$ at steps (1) and (2). We successively find:

- $\gamma = 34368_{10} = 1000011001000000_2$;
- $\delta = -33344_{10} = -1000001001000000_2$;
- $x_h = 1024_{10} = 10000000000_2$;
- $x_\ell = 17_{10} = 10001_2$, which fits in 5 bits, but does not fit in 4 bits.

Proof of Theorem 5
Without loss of generality, we assume $x > 0$. Also, in the following, we assume that x is not a power of 2 (otherwise, the analysis of Algorithm 6 is straightforward). This gives

$$2^{e_x} + 2^{e_x-p+1} \leq x \leq 2^{e_x+1} - 2^{e_x-p+1},$$

where e_x is the floating-point exponent of x. Furthermore, one may easily check that when $x = 2^{e_x+1} - 2^{e_x-p+1}$, the algorithm returns a correct result (if $s+1 \leq p'$, then there is no double rounding slip when computing γ, we get $\gamma = 2^{e_x+s+1} + 2^{e_x+1} - 2^{e_x+s-p+2}$, $\delta = -(2^{e_x+s+1} - 2^{e_x+s-p+2})$, $x_h = 2^{e_x+1}$ and $x_\ell = -2^{e_x-p+1}$; and if $s \geq p'$, there is a double rounding slip when computing γ, we get $\gamma = 2^{e_x+s+1} + 2^{e_x+1}$, $\delta = -2^{e_x+s+1}$, $x_h = 2^{e_x+1}$ and $x_\ell = -2^{e_x-p+1}$). Therefore, in the following, we assume

$$2^{e_x} + 2^{e_x-p+1} \leq x \leq 2^{e_x+1} - 2^{e_x-p+2}.$$

Without difficulty, we find

$$\gamma = (2^s + 1) \cdot x + \epsilon_1,$$

with $|\epsilon_1| \leq 2^{e_x+s-p+1} + 2^{e_x+s-p-p'+1}$. We have

$$\begin{array}{rcl} |x - \gamma| & \leq & 2^s \cdot x + |\epsilon_1| \\ & \leq & 2^s \left(2^{e_x+1} - 2^{e_x-p+2}\right) + 2^{e_x+s-p+1} + 2^{e_x+s-p-p'+1} \\ & < & 2^{e_x+s+1} \end{array}$$

(as soon as $p' \geq 1$), from which we deduce

$$\delta = \mathrm{RN}_p(\mathrm{RN}_{p+p'}(x - \gamma)) = x - \gamma + \epsilon_2,$$

with $|\epsilon_2| \leq 2^{e_x+s-p} + 2^{e_x+s-p-p'}$. Furthermore, $|x - \gamma| \geq 2^s(2^{e_x} + 2^{e_x-p+1}) - (2^{e_x+s-p+1} + 2^{e_x+s-p-p'+1})$. This implies

$$|x - \gamma| \geq 2^{e_x+s}(1 - 2^{-p-p'+1}).$$

Therefore, because of the monotonicity of rounding, as soon as $p' \geq 2$, $|\delta| \geq \mathrm{RN}_p(\mathrm{RN}_{p+p'}[2^{e_x+s} \cdot (1 - 2^{-p-p'+1})]) = 2^{e_x+s}$. This implies that δ is a multiple of $2^{e_x+s-p+1}$.

Let us now focus on the computation of x_h. So far, we have obtained

$$-\delta = -x + \gamma - \epsilon_2 = 2^s x + \epsilon_1 - \epsilon_2,$$

and

$$\gamma = (2^s + 1)x + \epsilon_1,$$

with

$$|\epsilon_1| \leq 2^{e_x+s-p+1} + 2^{e_x+s-p-p'+1} \leq (2^{s-p+1} + 2^{s-p-p'+1}) \cdot x$$

and

$$|\epsilon_2| \leq (2^{s-p} + 2^{s-p-p'}) \cdot x.$$

Therefore

$$\frac{2^s}{2^s+1} \cdot \frac{1 - 2^{-p+2} - 2^{-p-p'+2}}{1 + 2^{-p+1} + 2^{-p-p'+1}} \leq \left|\frac{\delta}{\gamma}\right| \leq \frac{2^s}{2^s+1} \cdot \frac{1 + 2^{-p+2} + 2^{-p-p'+2}}{1 - 2^{-p+1} - 2^{-p-p'+1}},$$

So that as soon as $p \geq 5$, $p' \geq 1$, and $s \geq 2$, $|\gamma|$ and $|\delta|$ are within a factor 2 from each other. Hence, since γ and δ clearly have opposite signs, we deduce from Remark 7 that $x_h = \gamma + \delta$ is computed exactly. Also, since γ and δ are multiples of $2^{e_x+s-p+1}$, x_h is multiple of $2^{e_x+s-p+1}$ too. Moreover,

$$\begin{aligned} x_h &= x + \epsilon_2 \leq \left(2^{e_x+1} - 2^{e_x-p+1}\right) + \left(2^{e_x+s-p} + 2^{e_x+s-p-p'}\right) \\ &< 2^{e_x+1} + 2^{e_x+s-p+1}, \end{aligned}$$

which implies:

- either $x_h < 2^{e_x+1}$, in which case, since x_h is a multiple of $2^{e_x+s-p+1}$, it fits in $p - s$ bits;
- or $x_h \geq 2^{e_x+1}$, but the only possible value that is both multiple of $2^{e_x+s-p+1}$ and less than $2^{e_x+1} + 2^{e_x+s-p+1}$ is 2^{e_x+1}, which fits in 1 bit.

Therefore, in any case, x_h fits in at most $p - s$ bits.

There now remains to focus on the computation of x_ℓ. Since $|x - x_h| = |\epsilon_2| \leq 2^{e_x+s-p} + 2^{e_x+s-p-p'} \leq x \cdot (2^{s-p} + 2^{s-p-p'})$, we easily find that as soon as $s \leq p - 2$, one can apply Sterbenz' Lemma (Remark 7) and deduce that $x_\ell = x - x_h$ is computed exactly.

Hence, we have $x = x_h + x_\ell$. Furthermore,

$$|x_\ell| = |\epsilon_2| \leq 2^{e_x+s-p} + 2^{e_x+s-p-p'},$$

and (since both x and x_h are multiples of 2^{e_x-p+1}), x_ℓ is a multiple of 2^{e_x-p+1} too. From this we deduce that x_ℓ fits in s bits. q.e.d. □

(notice that if $s \leq p' + 1$ there is no multiple of 2^{e_x-p+1} between 2^{e_x+s-p} and $2^{e_x+s-p} + 2^{e_x+s-p-p'}$. In such a case, x_ℓ is necessarily less than or equal to 2^{e_x+s-p}, and we deduce that it fits in $s - 1$ bits).

4.2 Application to the computation of exact products

An important consequence of Theorem 5 is that if the precision p is an even number, then Dekker's product algorithm [7] can be used (the proof of Dekker's product given for instance in [25] pages 137-139 shows that all operations but those of the Veltkamp's splitting in Dekker's algorithm are exact operations, so that they are not hindered by double roundings).

Dekker's product algorithm requires 17 operations. Notice that if a fused multiply-add (FMA) instruction is available,[1] there is a much better way of expressing the product of two FP numbers as the unevaluated sum of two FP numbers, that works even in the presence of double roundings. That way can be traced back at least to Kahan [18]. One may find a good presentation in [27]. More precisely (once adapted to the context of double roundings),

Theorem 6. *Assume a precision-p FP number system of minimum exponent $e_{\min}$. Let x and y be floating-point numbers of exponents e_x and e_y. If $e_x + e_y \geq e_{\min} + p - 1$ and the product xy does not overflow, and if $\circ$ is any faithful rounding (including $\mathrm{RN}_p(\cdot)$, $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(\cdot))$,...), then the sequence of calculations (algorithm 2MultFMA):*

$\pi_h \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(xy))$ *or* $\mathrm{RN}_p(xy)$
$\pi_\ell \leftarrow \circ(xy - \pi_h)$

will return two precision-p floating-point numbers π_h and π_ℓ such that $xy = \pi_h + \pi_\ell$.

The proof just uses the fact (see Theorem 2 in [2]) that for any faithful rounding $\circ$, the number $xy - \circ(xy)$ is a precision-p floating-point number, provided that $e_x + e_y \geq e_{\min} + p - 1$ and that xy does not overflow.

4.3 Rump, Ogita, and Oishi's Extractscalar splitting algorithm

In [33], Rump, Ogita, and Oishi introduce a splitting algorithm, that makes it possible to split an input floating-point number x into two parts y and x', such that $x = y + x'$ exactly, y is a multiple of some given power of 2, and the absolute value of x' is less than or equal to that power of 2. They use that splitting algorithm for designing clever summation methods. We will now see that most properties of their splitting algorithm are preserved in the presence of double roundings. Rewritten with double roundings, Rump, Ogita, and Oishi's Extractscalar splitting algorithm is

Algorithm 7 (Extrascalar-with-dble-rounding(σ, x): Rump, Ogita, and Oishi's splitting algorithm with possible double roundings. $\sigma = 2^k$, and x is a radix-2, precision-p floating-point number.)**.**
$s \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(x + \sigma))$
$y \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(s - \sigma))$

[1] The FMA instruction evaluates expressions of the form $xy + z$ with one final rounding only.

$x' \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(x-y))$
return (y, x')

When there are no double roundings, Rump, Ogita, and Oishi show that if the exponent of x, e_x, satisfies $k-p \leq e_x \leq k$ then $y + x' = x$ exactly, y is a multiple of 2^{k-p}, and $|x'| \leq 2^{k-p}$. The reason behind this is that Algorithm 7 is a variant of Fast2Sum. When double roundings are allowed, we get the following result.

Property 2 (Rump, Ogita, and Oishi's Extractscalar splitting algorithm in the presence of double roundings)**.** *If $k-p \leq e_x \leq k$ then the values y and x' computed by Algorithm 7 satisfy $y + x' = x$ exactly, y is a multiple of 2^{k-p}, and $|x'| \leq 2^{k-p} + 2^{k-p-p'}$.*

Proof. Notice that Algorithm 7 is Fast2Sum-with-double-roundings(σ, x). Theorem 2 and Remark 11 (since σ is a power of 2) imply that $s + x' = \sigma + x$ and $y = s - \sigma$ exactly, so that $y + x' = x$.

Furthermore, s is a multiple of 2^{k-p} (this is obtained by considering that either $|x| \geq 2^{k-1}$, in which case x and therefore $x + \sigma$ are multiple of 2^{k-p}, or $|x| < 2^{k-1}$, in which case $x + \sigma > 2^{k-1}$, hence, $s \geq 2^{k-1}$, which implies that s is a multiple of 2^{k-p}).

An immediate consequence is that y is a multiple of 2^{k-p}. Now, the (possibly double rounded) computed value of s satisfies

$$-2^{k-p} - 2^{k-p-p'} \leq s - (x + \sigma) \leq 2^{k-p} + 2^{k-p-p'},$$

which implies

$$-2^{k-p} - 2^{k-p-p'} \leq x' \leq 2^{k-p} + 2^{k-p-p'}.$$

□

5 Consequences of Theorems 2 and 4 on summation algorithms

Many numerical problems require the computation of sums of lots of many FP numbers. Several *compensated summation* algorithms use, either implicitely or explicitely, the Fast2Sum or 2Sum algorithms [17, 30, 31, 28, 32]. As a consequence, when double roundings may occur, it is of importance to know if the fact that we can only guarantee that we return the FP number nearest the error of a FP addition (instead of that error itself) may have an influence on the behavior of these algorithms. There is a huge literature on summation algorithms: the purpose of this section is not to examine all published algorithms, just to give a few examples.

Before analyzing other summation methods, let us see what happens with the naive, "recursive sum" algorithm.

5.1 The recursive sum algorithm and Kahan's compensated summation algorithm in the presence of double roundings

Let us consider the naive, recursive-sum algorithm, rewritten with double roundings.

Algorithm 8.
$r \leftarrow a_1$
for $i = 2$ *to* n ***do***
  $r \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(r + a_i))$
end for
return r

A straightforward adaptation of the proof for the error bound of the usual recursive sum algorithm without double roundings gives

Property 3. *The final value of the variable r returned by Algorithm 8 satisfies*

$$\left| r - \sum_{i=1}^{n} a_i \right| \leq \gamma'_{n-1} \sum_{i=1}^{n} |a_i|.$$

Without double roundings, the bound is $\gamma_{n-1} \sum_{i=1}^{n} |a_i|$. See Section 2.1.7 for a definition of notations γ_k and γ'_k.

Kahan's compensated summation algorithm, rewritten with double roundings, is as follows

Algorithm 9.
$s \leftarrow a_1$
$c \leftarrow 0$
for $i = 2$ *to* n ***do***
  $y \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(a_i - c))$
  $t \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(s + y))$
  $c \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(\mathrm{RN}_p(\mathrm{RN}_{p+p'}(t - s)) - y))$
  $s \leftarrow t$
end for
return s

Goldberg's proof for this algorithm [11] only uses the ϵ-model, so that adaptation to double roundings is straightforward (it suffices to replace u by u' in the ϵ-model), and we will immediately deduce that the final value s provided by Algorithm 9 satisfies

$$\left| s - \sum_{i=1}^{n} a_i \right| \leq \left(2u' + O(nu'^2)\right) \cdot \sum_{i=1}^{n} |a_i|$$

(see Section 2.1.7 for a definition of notations u and u'). This makes Kahan's compensated summation algorithm very "robust": double roundings have little

influence on the error bound. However, when $\sum_{i=1}^{n} |a_i|$ is very large in front of $|\sum_{i=1}^{n} a_i|$, the relative error of Kahan's compensated summation algorithm becomes large. A solution is to use Priest's *doubly compensated* summation algorithm [31]. For that algorithm, the excellent error bound $2u\,|\sum_{i=1}^{n} |a_i||$ will remain true even in the presence of double roundings (the proof essentially assumes faithfully rounded operations). However, it requires a preliminary sorting of the a_i's by magnitude.

In the following, we investigate the potential influence of double roundings on some sophisticated summation algorithms. For most of these algorithms, the proven error bounds (without double roundings) are of the form

$$\left| \text{computed sum } - \sum_{i=1}^{n} a_i \right| \leq u \cdot \left| \sum_{i=1}^{n} a_i \right| + \alpha \cdot \sum_{i=1}^{n} |a_i|.$$

Rump, Ogita, and Oishi exhibit a family of algorithms for which, without double roundings, α has the form $O(n^K 2^{-Kp})$. As we will see, that property will be (roughly) preserved when $K = 2$. However, for the more subtle algorithms—for which $K \geq 3$—, double roundings may ruin that property.

5.2 Rump, Ogita and Oishi's cascaded summation algorithm in the presence of double roundings

The following algorithm was independently introduced by Pichat [29] and Neumaier [26].

Algorithm 10.
$s \leftarrow a_1$
$e \leftarrow 0$
for $i = 2$ *to* n ***do***
  if $|s| \geq |a_i|$ ***then***
    $(s, e_i) \leftarrow$ *Fast2Sum*(s, a_i)
  else
    $(s, e_i) \leftarrow$ *Fast2Sum*(a_i, s)
  end if
  $e \leftarrow \mathrm{RN}(e + e_i)$
end for
return $\mathrm{RN}(s + e)$

To avoid tests, the algorithm of Pichat and Neumaier can be rewritten using the 2Sum algorithm. This gives the *cascaded summation* algorithm of Rump, Ogita, and Oishi [28]:

Algorithm 11.
$s \leftarrow a_1$
$e \leftarrow 0$
for $i = 2$ *to* n ***do***
  $(s, e_i) \leftarrow$ *2Sum*(s, a_i)

$e \leftarrow \mathrm{RN}(e + e_i)$
end for
return $\mathrm{RN}(s + e)$

Notice that both algorithms will return the same result. In the following, we therefore focus on Algorithm 11 only. More precisely, we will be interested here in analyzing the behaviour of that algorithm, with double roundings allowed. That is, we will consider

Algorithm 12 (Rump, Ogita and Oishi's Cascaded Summation algorithm)**.**

$s \leftarrow a_1$
$e \leftarrow 0$
for $i = 2$ *to* n ***do***
$(s, e_i) \leftarrow$ *2Sum-with-double-roundings*(s, a_i)
$e \leftarrow \mathrm{RN}_p(\mathrm{RN}_{p+p'}(e + e_i))$
end for
return $\mathrm{RN}_p(\mathrm{RN}_{p+p'}(s + e))$

Define s_i as the value of variable s after the loop of index i (namely, $s_1 = a_1$, and for $i \geq 2$, $s_i = 2\mathrm{Sum\text{-}with\text{-}double\text{-}roundings}(s_{i-1}, a_i)$). One easily finds

$$\left\{ \begin{array}{lcl} s_2 + e_2 & = & (a_1 + a_2)(1 + \eta^{(2)}) \\ s_i + e_i & = & (s_{i-1} + a_i)(1 + \eta^{(i)}) \\ |e_i| & \leq & u'(1 + u)|s_i|, \end{array} \right.$$

with $|\eta^{(i)}| \leq 2^{-2p} + 2^{-2p-p'} + 2^{-3p-p'}$ (from Corollary 1). Therefore,

$$\begin{aligned} & s_n + (e_n + e_{n-1} + \cdots + e_2) \\ & = (s_n + e_n) + (e_{n-1} + e_{n-2} + \cdots + e_2) \\ & = (s_{n-1} + a_n)(1 + \eta^{(n)}) + (e_{n-1} + e_{n-2} + \cdots + e_2) \\ & = a_n(1 + \eta^{(n)}) + s_{n-1}\eta^{(n)} + (s_{n-1} + e_{n-1}) + (e_{n-2} + \cdots + e_2) \\ & = a_n(1 + \eta^{(n)}) + s_{n-1}\eta^{(n)} + (s_{n-2} + a_{n-1})(1 + \eta^{(n-1)}) + (e_{n-2} + \cdots + e_2) \\ & = \cdots \\ & = a_n(1 + \eta^{(n)}) + a_{n-1}(1 + \eta^{(n-1)}) + \cdots + a_3(1 + \eta^{(3)}) \\ & + s_{n-1}\eta^{(n)} + s_{n-2}\eta^{(n-1)} + \cdots + s_2\eta^{(3)} \\ & + (s_2 + e_2) \\ & = \sum_{i=3}^{n} a_i(1 + \eta^{(i)}) + \sum_{i=3}^{(n)} s_{i-1}\eta^{(i)} + (a_1 + a_2)(1 + \eta^{(2)}), \end{aligned}$$

From which we deduce

$$s_n + \sum_{i=2}^{n} e_i = \sum_{i=1}^{n} a_i + \eta \cdot \sum_{i=1}^{n} |a_i| + \eta' \sum_{i=2}^{n} |s_i|, \tag{8}$$

with $|\eta|, |\eta'| \leq 2^{-2p} + 2^{-2p-p'} + 2^{-3p-p'}$.

Let $|E|$ be obtained by computing $\sum_{i=2}^{n} e_i$ by the recursive summation algorithm (possibly with double roundings), from Property 3, we have

$$\left| E - \sum_{i=2}^{n} e_i \right| \leq \gamma'_k \cdot \sum_{i=2}^{n} |e_i|. \tag{9}$$

Let us now bound $\sum_{i=2}^{n} |e_i|$. We already have

$$|e_i| \leq \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right) \cdot |s_i|.$$

Now,

$$|s_2| \leq (|a_1| + |a_2|) \cdot (1 + \gamma'_1),$$

so that

$$\begin{array}{rcl} |s_3| & \leq & [(|a_1| + |a_2|)(1 + \gamma'_1) + |a_3|] \cdot (1 + \gamma'_1) \\ & < & (|a_1| + |a_2| + |a_3|) \cdot (1 + \gamma'_2), \end{array}$$

and, by induction

$$\begin{array}{rcl} |s_j| & < & (|a_1| + |a_2| + \cdots + |a_j|) \cdot (1 + \gamma'_{j-1}) \\ & < & (|a_1| + |a_2| + \cdots + |a_n|) \cdot (1 + \gamma'_{n-1}). \end{array}$$

Therefore,

$$\sum_{i=2}^{n} |s_i| \leq (n-1) \cdot (1 + \gamma'_{n-1}) \cdot \sum_{i=1}^{n} |a_i|, \tag{10}$$

which implies

$$\sum_{i=2}^{n} |e_i| \leq (n-1)(1 + \gamma'_{n-1}) \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right) \sum_{i=1}^{n} |a_i|. \tag{11}$$

Hence,

$$\sum_{i=1}^{n} a_i = s_n + E + \rho, \tag{12}$$

where

$$|\rho| < \left(\sum_{i=1}^{n} |a_i| \right) \times \kappa, \tag{13}$$

with

$$\begin{array}{rcl} \kappa & = & \eta + (n-1) \cdot (1 + \gamma'_{n-1}) \\ & & \cdot \left(\eta' + \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right) \gamma'_{n-2}\right). \end{array} \tag{14}$$

We remind the reader that $\gamma'_{n-2} = (n-2)u'/(1-(n-2)u')$. Assuming $p \geq 8$ and $p' \geq 4$, we find

$$\left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right) u' \leq 2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}. \tag{15}$$

Let us now assume $|(n-1)u'| < 1/2$, which implies $1/(1-(n-2)) < 2$. From (15), we deduce

$$\begin{aligned} &\left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right)\gamma'_{n-2} \\ &\leq (2n-4)\cdot\left(2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}\right). \end{aligned} \tag{16}$$

Similarly, still assuming $p \geq 8$ and $p' \geq 4$,

$$\begin{aligned} |\eta'| &\leq 2^{-2p} + 2^{-2p-p'} + 2^{-3p-p'} \\ &< 2^{-2p} + 2^{-2p-p'+1}. \end{aligned} \tag{17}$$

By combining (16) and (17), we obtain

$$\begin{aligned} &\left|\eta' + \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right)\gamma'_{n-2}\right| \\ &< (2n-3)\cdot\left(2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}\right). \end{aligned}$$

Our assumption $|(n-1)u'| < 1/2$ implies $\gamma'_{n-1} < 1$, therefore the term

$$(n-1)(1+\gamma'_{n-1})$$

in (13) is less than $(2n-2)$. From all this, we deduce that the term

$$\left[\eta + (n-1)(1+\gamma'_{n-1})\cdot\left(\eta' + \left(2^{-p} + 2^{-p-p'} + 2^{-2p} + 2^{-2p-p'}\right)\gamma'_{n-2}\right)\right]$$

in (13) is less than

$$\left(4n^2 - 10n - 5\right)\cdot\left(2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}\right). \tag{18}$$

Now, from (12), the final value, say σ, returned by Algorithm 12, satisfies

$$\begin{aligned} \sigma &= (s_n + E)\cdot(1+\theta'), \text{ with } |\theta'| \leq u', \\ &= \left(\sum_{i=1}^{n} a_i - \rho\right)(1+\theta'). \end{aligned}$$

using(18), this implies

$$\begin{aligned} \left|\sigma - \sum_{i=1}^{n} a_i\right| &\leq u'\cdot\left|\sum_{i=1}^{n} a_i\right| \\ &+ (1+u')\left(4n^2-10n-5\right)\left(2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}\right)\sum_{i=1}^{n}|a_i|. \end{aligned}$$

An elementary calculation, still assuming $p \geq 8$ and $p' \geq 4$, shows that

$$(1+u') \cdot \left(2^{-2p} + 2^{-2p-p'+1} + \frac{2^{-2p}}{100}\right) \leq 2^{-2p} + 2^{-2p-p'+1} + \frac{3 \cdot 2^{-2p}}{200},$$

which gives

Theorem 7. *Assuming $p \geq 8$, $p' \geq 4$, and $n < \dfrac{1}{2u'}$, the final value σ returned by Algorithm 12 satisfies*

$$\begin{aligned} \left|\sigma - \sum_{i=1}^{n} a_i\right| &\leq \left(2^{-p} + 2^{-p-p'} + 2^{-2p-p'}\right) \cdot \sum_{i=1}^{n} a_i \\ &+ 2^{-2p} \cdot \left(4n^2 - 10n - 5\right) \cdot \left(1 + 2^{-p'+1} + \frac{3}{200}\right) \cdot \sum_{i=1}^{n} |a_i|. \end{aligned}$$

In that case, the final result is not so different from the classical, double-rounding-free, result (in that classical case, the term if front of $\sum_{i=1}^{n} a_i$ is $u = 2^{-p}$, and the term in front of $\sum_{i=1}^{n} |a_i|$ is γ_{n-1}^2). Hence the Cascaded Summation algorithm is "robust" and can be used safely, even when double roundings may happen: the error bound is slightly larger but remains of the same order of magnitude.

However, more subtle algorithms, that return a more accurate result (assuming no double roundings) when $|\sum_{i=1}^{n} |a_i| / \sum_{i=1}^{n} a_i|$ is very large, may be of less interest when double roundings may happen, unless we have additional information on the input data that allow to make sure there will be no problem. Consider for instance the K-fold summation algorithm of Rump, Ogita and Oishi, defined as follows.

Algorithm 13 (VecSum(a), where $a = (a_1, a_2, \ldots, a_n)$)**.**
$p \leftarrow a$
for $i = 2$ *to* n ***do***
  $(p_i, p_{i-1}) \leftarrow \mathit{2Sum}(p_i, p_{i-1})$
end for
return p

Algorithm 14 (K-fold summation algorithm)**.**
for $k = 1$ *to* $K - 1$ ***do***
  $a \leftarrow \mathit{VecSum}(a)$
end for
$c = a_1$
for $i = 2$ *to* $n - 1$ ***do***
  $c \leftarrow \mathrm{RN}(c + a_i)$
end for
return $\mathrm{RN}(a_n + c)$

If double roundings are not allowed, Rump, Ogita, and Oishi show that if

$4nu < 1$, the final result σ returned by Algorithm 14 satisfies

$$\left|\sigma - \sum_{i=1}^{n} a_i\right| \leq (u + \gamma_{n-1}^2)\left|\sum_{i=1}^{n} a_i\right| + \gamma_{2n-2}^{K} \sum_{i=1}^{n} |a_i|. \tag{19}$$

If a double-rounding slip occurs in the first call to VecSum, an error as large as $2^{-2p} \max |a_i|$ may be produced. Hence, it will not be possible to show a final error bound better than $2^{-2p} \max |a_i| \geq \frac{2^{-2p}}{n} \sum_{i=1}^{n} |a_i|$ when double roundings are allowed. In practice (since double rounding slips are not so frequent, and do not always change the result of 2Sum when they occur), the K-fold summation algorithm will almost always return a result that satisfies a bound close to the one given by (19), but exceptions may occur. Consider the following example (with $n = 5$, but easily generalizable to any larger value of n):

$$(a_1, a_2, a_3, a_4, a_5) = \left(2^{p-1} + 1, \frac{1}{2} - 2^{-p-1}, -2^{p-1}, -2, \frac{1}{2}\right)$$

and assume that Algorithm 14 is run with double roundings, with $1 \leq p' \leq p$. One may easily check that in the first addition of the first 2Sum of the first call to VecSum (i.e., when adding a_1 and a_2), a double rounding slip occurs, so that immediately after this first Fast2Sum, $p_2 = 2^{p-1} + 2$ and $p_1 = -1/2$, so that $p_1 + p_2 \neq a_1 + a_2$. At the end of the first call to VecSum, the returned vector is

$$\left(-\frac{1}{2}, 0, 0, 0, \frac{1}{2}\right)$$

so that Algorithm 14 will return 0 whatever the value of K, whereas the exact sum of the $a_i's$ is -2^{-p-1}. Hence (since $\sum |a_i| = 2^p + 4 - 2^{-p-1} \approx 2^p$), the final error of Algorithm 14 is approximately $2^{-2p-1} \sum |a_i|$, whatever the value of K.

This example shows that if we wish to be sure of getting error bounds of the order of magnitude of the one given by (19) when using the K-fold summation algorithm with $K \geq 3$, we need to select compilation switches that prevent double roundings from occurring, unless we have additional information on the input data (such as all values having the same order of magnitude) that allow to use Remark 10 to show that Fast2Sum and 2Sum will return an exact result, even in the presence of double roundings.

Rump, Ogita, and Oishi suggest another summation algorithm, that returns faithfully rounded sums when run without double roundings [33]. It is based on the splitting algorithm discussed in Section 4.3. Using property 2, one may adapt their summation algorithm, so that it can return faithfully rounded sums, even in the presence of double roundings.

6 Double roundings with scaled division iterations

6.1 Scaled division iterations

As previously, we assume a radix-2, precision-p, floating-point system that is compliant with the IEEE 754-2008 Standard for Floating-Point Arithmetic [14]. We denote $e_{\min}$ and $e_{\max}$ the extremal exponents of that system. We also assume that an FMA instruction is available, and that the ambient rounding mode is *round to nearest* (this is the only one for which double-rounding slips occur).

Many algorithms have been suggested for performing divisions, the most common being digit-recurrence algorithms [8] and variants of the Newton–Raphson iteration [21]. Here, assuming we wish to evaluate the quotient b/a of two floating-point numbers, we focus on algorithms that first provide an approximation y to $1/a$ and an initial approximation q to the quotient b/a, and refine it using the following "correcting step" [22]:

$$\begin{array}{rcl} r & = & \mathrm{RN}\,(b - aq), \\ q' & = & \mathrm{RN}\,(q + ry), \end{array} \tag{20}$$

Under some conditions made explicit in Theorem 9—roughly speaking, if q and y are close enough to b/a and $1/a$, respectively, and no underflow occurs—, then $q' = \mathrm{RN}(b/a)$.

In the applications of that property presented in the literature, the approximations y and q are obtained through variants of the Newton-Raphson iteration (indeed, (20) can be viewed as one Newton-Raphson step), but they might as well result from other means. What matters in this paper is that the correcting step (20) is used.

What makes the method working is the following lemma, which shows that under some conditions, $r = b - aq$ *exactly*. That lemma can be traced back to Kahan [18] or Markstein [22]. The presentation we give here is close to that of Boldo and Daumas [2, 25].

Lemma 8 (Computation of division residuals using an FMA)**.** *Assume a and b are precision-p, radix-2, floating-point numbers, with $a \neq 0$ and $|b/a|$ below the overflow threshold. If q is defined as*

- *b/a if it is exactly representable;*
- *one of the two floating-point numbers that surround b/a otherwise;*

then

$$b - aq$$

is exactly computed using one FMA instruction, with any rounding mode, pro-

vided that

$$e_a + e_q \geq e_{\min} + p - 1,$$
$$\textit{and} \tag{21}$$
$$q \neq \alpha \text{ or } |b/a| \geq \tfrac{\alpha}{2},$$

where e_a and e_q are the exponents of a and q and $\alpha = 2^{e_{\min}-p+1}$ is the smallest positive subnormal number.

For this result to be applicable, we need $e_a + e_q \geq e_{\min} + p - 1$. This condition will be satisfied if $e_b \geq e_{\min} + p$. Other conditions will be needed for the correcting iterations to work (see Theorem 9 below). Also, the intermediate iterations used for computing q and y may require the absence of over/underflow. All this gives somewhat complex conditions on a and b, that can *very* roughly be summarized as "the quotient and the residual r must be far enough from the underflow and overflow thresholds". More precisely,

Theorem 9 (Markstein [22, 5, 21, 12])**.** *Assume a precision-p binary floating-point arithmetic, and let a and b be normal numbers. If*

- *q is a faithful approximation to b/a, and*
- *q is not in the subnormal range, and*
- *$e_b \geq e_{\min} + p$, and*
- *y approximates $1/a$ with a relative error less than 2^{-p}, and*
- *the calculations*
$$r = \circ(b - aq), \qquad q' = \circ(q + ry)$$
are performed using a given rounding mode $\circ$, taken among round to nearest even, round toward zero, round toward $-\infty$, round toward $+\infty$,

then q' is exactly $\circ(b/a)$ (that is, b/a rounded according to the same rounding mode $\circ$).

Notice that if r and q' are first computed in a wider format, of precision $p + p'$, and then rounded to the precision-p destination format, we have no guarantee that $q' = \circ(b/a)$: r will be computed correctly anyway (since $b - aq$ is a precision-p FP number), but a double rounding slip in the computation of q' may suffice to hinder the result. An example is:

- $p = 53$ (double precision/binary64 format), $p + p' = 64$ (Intel "double-extended" format);
- $b = 2^{52} + 2^{51} + 1$ and $a = 2^{53} - 2$;
- $q = (3 \cdot 2^{51} + 3)/(2^{53})$ (notice that $q = \mathrm{RN}_{53}(b/a)$);
- $y = \mathrm{RN}_{53}(1/a) = (2^{52} + 1)/2^{105}$,

for which we get

- $r = b - aq = (-2^{51} + 3)/2^{52}$;
- $\mathrm{RN}_{53}(\mathrm{RN}_{64}(q + ry)) = (3 \cdot 2^{50} + 1)/2^{52}) \neq \mathrm{RN}_{53}(b/a)$.

So it is necessary to make sure that the computation of q' is directly performed in precision p, without a double rounding. But even that may not suffice, as we are now going to see. Let us now assume that we only use precision p.

Given arbitrary FP inputs a and b, a natural way to make sure that the conditions of Theorem 9 be satisfied is to *scale* the iterations. This can be done as follows: a quick preliminary checking on the exponents of a and b determines if the conditions of Theorem 9 may not be satisfied, or if there is some risk of over/underflow in the iterations that compute y and q. If this is the case, operand a, or operand b is multiplied by some adequately chosen power of 2, to get new, *scaled*, operands a^* and b^* such that the division b^*/a^* is performed without any problem. An alternate, possibly simpler, solution is to *always* scale: for instance, we chose a^* and b^* equal to the significands of a and b, i.e., we momentarily set their exponents to zero. In any case, we assume that we now perform a division b^*/a^* such that:

- for that "scaled division", the conditions of Theorem 9 are satisfied;
- the *exact* quotient b/a is equal to $2^\sigma b^*/a^*$, where σ is an integer straightforwardly deduced from the scaling.

Assuming now that the scaled iterations return a scaled approximate quotient q^* and a scaled approximate reciprocal y^*, we perform a scaled correcting step

$$\begin{array}{rcl} r & = & \mathrm{RN}\,(b^* - a^*q^*), \\ q' & = & \mathrm{RN}\,(q^* + ry^*), \end{array}$$

Notice that q' is in the normal range (i.e., its absolute value is larger than or equal to $2^{e_{\min}}$): the scaling was partly done in order to make this sure. If $2^\sigma q'$ is a floating-point number (e.g., if $|2^\sigma q'| \geq 2^{e_{\min}}$), then we clearly should return $2^\sigma q'$. The trouble may occur when $2^\sigma q'$ falls in the subnormal range: if $2^\sigma q'$ is not a FP number, we cannot just return $\mathrm{RN}(2^\sigma q')$ because a *double rounding* slip might occur and lead to the delivery of a wrong result. Consider the following example. Assume the floating-point format being considered is *binary32* (that format was called *single precision* in the previous version of IEEE 754: precision $p = 24$, extremal exponents $e_{\min} = -126$ and $e_{\max} = 127$). Consider the two floating-point input values (the significands are represented in binary):

$$\left\{ \begin{array}{rcll} b & = & 1.00000000001100011001101_2 \times 2^{-113} & = & 8394957_{10} \times 2^{-136}, \\ a & = & 1.00000000000011011001100_2 \times 2^{23} & = & 8390348_{10}. \end{array} \right.$$

The number b/a is equal to

$$0.10000000000100100000000000001011010011111011001100100000010\cdots\times 2^{-135},$$

so that the correctly-rounded, subnormal value that must be returned when computing b/a should be

$$\mathrm{RN}(b/a) = 0.00000000010000000000101 \times 2^{-126}.$$

Now, if, to be able to use Theorem 9, b was scaled, for instance by multiplying it by 2^{128} to get a value b^*, the exact value of b^*/a would be

$$0.10000000000100100000000000001011010011111011001100100000010\cdots \times 2^{-7},$$

which would imply that the computed correctly rounded approximation to b^*/a would be

$$q' = 1.00000000001001000000000 \times 2^{-8}.$$

Multiplied by $2^{\sigma} = 2^{-128}$, this result would be equal equal to

$$1.00000000001001000000000 \times 2^{-136},$$

which means—since it is in the subnormal range: remember that $e_{\min} = -126$—that, after rounding it to the nearest (even) floating-point number, we would get

$$0.00000000010000000000100 \times 2^{-126} \neq \mathrm{RN}(b/a).$$

This phenomenon may appear each time the scaled result q', once multiplied by 2^{σ}, is exactly equal to a (subnormal) *midpoint* [16], i.e., a value exactly halfway between two consecutive floating-point numbers. Notice that if we just use this scaled result q' without any other information, it is impossible to deduce if the exact, infinitely precise, result is above or below the midpoint, so it is hopeless to try to return a correctly rounded value.

Fortunately, intermediate values computed during the last correction iteration contain enough information to allow for a correctly rounded final result, as we are now going to see.

6.2 Avoiding double roundings in scaled division iterations

As stated in the previous section, we assume we have performed the correcting step:

$$\begin{aligned} r &= \mathrm{RN}\,(b^* - a^*q^*), \\ q' &= \mathrm{RN}\,(q^* + ry^*), \end{aligned}$$

and that the scaled operands a^*, b^*, as well as the approximate scaled quotient q^* and scaled reciprocal y^* satisfy the conditions of Theorem 9. We assume that the scaling was such that the exact quotient b/a is equal to $2^{\sigma}b^*/a^*$. We assume that we are interested in quotients rounded to the nearest (our method will work for both tie-breaking rules of IEEE 754-2008: round to nearest "even" as well as round to nearest "away"). Notice that with the other, "directed", rounding modes, there is no double rounding problem. To simplify the presentation, we assume that a and b (and, therefore, a^*, b^*, y^*, q^* and q') are positive

(separately handling the signs of the input operands is straightforward). Since q^* is a faithful approximation to b^*/a^*, we deduce that

$$q^- < \frac{b^*}{a^*} < q^+,$$

where q^- and q^+ are the floating-point predecessor and successor of q^*. Also, since $q' = \mathrm{RN}(b^*/a^*)$, we immediately deduce that $q' \in \{q^-, q, q^+\}$. This is illustrated by Figure 2.

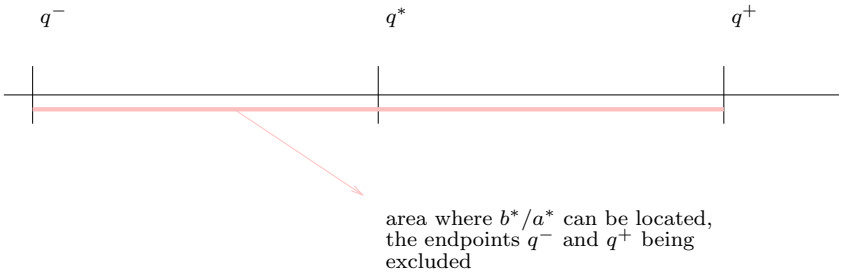

Figure 2: The number q^* is a faithful rounding of b^*/a^*: this means that $q^- < b^*/a^* < q^+$, where q^- and q^+ are the floating-point predecessor and successor of q^*.

As stated before, a double rounding slip may occur when $2^\sigma q'$ is a (subnormal) midpoint of the considered floating-point format. In such a case, in order to return a correctly rounded quotient, one must know if the exact quotient b/a is strictly below, equal to, or strictly above that midpoint. Of, course, this is equivalent to knowing if b^*/a^* is strictly below, equal to, or strictly above q'. Lemma 8 says that $r = b^* - a^*q^*$ exactly. Therefore, *when $2^\sigma q'$ is a midpoint:*

1. if $r = 0$ then $q' = q^* = b^*/a^*$, hence $b/a = 2^\sigma q'$ exactly. Therefore, one should return $\mathrm{RN}(2^\sigma q')$;

2. if $q' \neq q^*$ and $r > 0$ (which implies $q' = q^+$), then q' overestimates b^*/a^*. Therefore, one should return $2^\sigma q'$ rounded down. This is illustrated by Figure 3;

3. if $q' \neq q^*$ and $r < 0$ (which implies $q' = q^-$), then q' underestimates b^*/a^*. Therefore, one should return $2^\sigma q'$ rounded up;

4. if $q' = q^*$ and $r > 0$, then q' underestimates b^*/a^*. Therefore, one should return $2^\sigma q'$ rounded up. This is illustrated by Figure 4;

5. if $q' = q^*$ and $r < 0$, then q' overestimates b^*/a^*. Therefore, one should return $2^\sigma q'$ rounded down.

When $2^\sigma q'$ is not a midpoint, one should of course return $\mathrm{RN}(2^\sigma q')$.
Therefore, in all cases, we are able to find which value is to be returned.

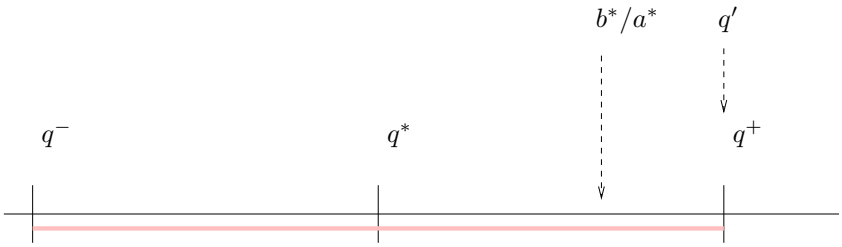

Figure 3: The number q' is equal to q^+. In this case, the "residual" r was positive, and—since $q^- < b^*/a^* < q^+$—, q' is an overestimation of b^*/a^*.

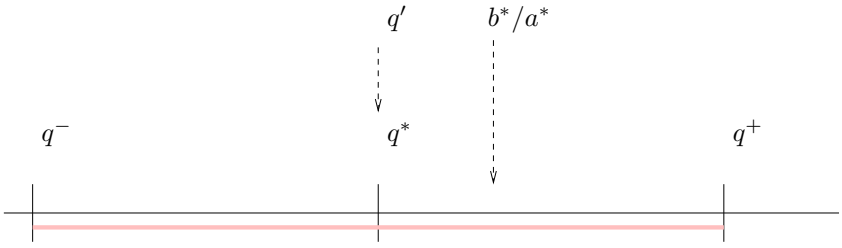

Figure 4: The number q' is equal to q^*. In this case, the "residual" r was positive, and q' is an underestimation of b^*/a^*.

7 Conclusion

We have considered the possible influence of double roundings on several algorithms of the floating-point literature: Fast2Sum, 2Sum, Veltkamp's splitting, 2MultFMA, division correction iterations. Although most of these algorithms do not behave exactly as when there are no double roundings, they still have interesting properties that can be exploited in an useful way. Depending on the considered applications, these properties may suffice, or specific compilation options should be chosen to prevent double roundings.

References

[1] S. Boldo. Pitfalls of a full floating-point proof: example on the formal proof of the Veltkamp/Dekker algorithms. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 52–66, 2006.

[2] 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.

[3] S. Boldo, M. Daumas, C. Moreau-Finot, and L. Théry. Computer validated proofs of a toolset for adaptable arithmetic. Technical report, École Normale Supérieure de Lyon, 2001. Available at `http://arxiv.org/pdf/cs.MS/0107025`.

[4] S. Boldo and G. Melquiond. Emulation of FMA and correctly rounded sums: proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4):462–471, April 2008.

[5] M. Cornea, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton–Raphson-based floating-point divide and square root algorithms. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 96–105. IEEE Computer Society Press, Los Alamitos, CA, April 1999.

[6] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev. A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. *IEEE Transactions on Computers*, 58(2):148–162, 2009.

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

[8] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, MA, 1994.

[9] S. A. Figueroa. When is double rounding innocuous? *ACM SIGNUM Newsletter*, 30(3), July 1995.

[10] S. A. Figueroa. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. PhD thesis, Department of Computer Science, New York University, 2000.

[11] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991. An edited reprint is available at `http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf` from Sun's Numerical Computation Guide; it contains an addendum *Differences Among IEEE 754 Implementations*, also available at `http://www.validlab.com/goldberg/addendum.html`.

[12] J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 2000.

[13] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 2nd edition, 2002.

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

[15] International Organization for Standardization. *Programming Languages – C*. ISO/IEC Standard 9899:1999, Geneva, Switzerland, December 1999.

[16] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux. Midpoints and exact points of some algebraic functions in floating-point arithmetic. *IEEE Transactions on Computers*, 60(2), February 2011.

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

[18] W. Kahan. Lecture notes on the status of IEEE-754. PDF file accessible at `http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF`, 1996.

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

[20] V. Lefèvre. The Euclidean division implemented with a floating-point division and a floor. Research report RR-5604, INRIA, June 2005.

[21] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[22] P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, January 1990.

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

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

[25] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[26] A. Neumaier. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen. *ZAMM*, 54:39–51, 1974. In German.

[27] Y. Nievergelt. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Transactions on Mathematical Software*, 29(1):27–48, 2003.

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

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

[30] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, pages 132–144. IEEE Computer Society Press, Los Alamitos, CA, June 1991.

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

[32] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest. *SIAM Journal on Scientific Computing*, 2005–2008. Submitted for publication.

[33] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[34] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Computational Geometry*, 18:305–363, 1997.

[35] P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.