Algorithms for triple-word arithmetic
Nicolas Fabiano, Jean-Michel Muller, Joris Picot

To cite this version:
Nicolas Fabiano, Jean-Michel Muller, Joris Picot. Algorithms for triple-word arithmetic. IEEE Transactions on Computers, 2019, 68 (11), pp.1573-1583. 10.1109/TC.2019.2918451. hal-01869009v2

HAL Id: hal-01869009
https://hal.science/hal-01869009v2
Submitted on 20 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Algorithms for triple-word arithmetic

Nicolas Fabiano and Jean-Michel Muller and Joris Picot

Abstract—Triple-word arithmetic consists in representing high-precision numbers as the unevaluated sum of three floating-point numbers (with “nonoverlapping” constraints that are explicit in the paper). We introduce and analyze various algorithms for manipulating triple-word numbers: rounding a triple-word number to a floating-point number, adding, multiplying, dividing, and computing square-roots of triple-word numbers, etc. We compare our algorithms, implemented in the Campary library, with other solutions of comparable accuracy. It turns out that our new algorithms are significantly faster than what one would obtain by just using the usual floating-point expansion algorithms in the special case of expansions of length 3.

1 INTRODUCTION AND NOTATION

Double-word and Triple-word arithmetics consist in representing a real number as the unevaluated sum of two and three floating-point numbers, respectively. They are frequently called “double double” and “triple double”, because in practice the underlying format being used is the binary64/double precision format of the IEEE 754 Standard for Floating-Point Arithmetic [7], [21].

A generalization of these arithmetics is the notion of floating-point expansion [26], [29], [10], where a high-precision number is represented as the unevaluated sum of \( n \) floating-point numbers.

Such arithmetics are useful. Numerical computations sometimes require a precision significantly higher than the one offered by the basic floating-point formats. Double-word, triple-word (or even quadruple-word) arithmetics have been used for implementing BLAS [18], [20], [30], for Semidefinite programming [22]. Bailey, Barrio and Borwein [1] give several timely examples in mathematical physics and dynamics where precision higher than double precision/binary64 are needed. Another example is when evaluating transcendental functions with correct rounding: it is almost impossible to guarantee last-bit accuracy in the final result if all intermediate calculations are done in the target precision. For instance, the CRLibm library [3] of correctly rounded elementary functions uses double-word and triple-word [14], [15] operations in the last steps of the evaluation of approximating polynomials. The reason is simple: results on the table maker’s dilemma [16] show that returning a correctly rounded exponential or logarithm requires approximating the considered function with roughly twice the target precision, which in turn requires doing intermediate calculations with significantly more than twice the target precision. The Metalibm Lutetia library,\(^1\) also uses double-word and triple-word arithmetics.

Compared to double-word or triple-word arithmetic, arbitrary precision libraries such as GNU-MPFR [5] have the advantage of being versatile, but may involve a significant penalty in terms of speed and memory consumption if one only requires computations accurate within around 150 bits in a few critical parts of a numerical program.

Algorithms for double-word arithmetic have been presented in [17], [9]. The purpose of this paper is to introduce and analyze efficient algorithms for performing the arithmetic operations in triple-word arithmetic. Our goal is to obtain algorithms that are faster than the ones we could obtain simply by using floating-point expansion algorithms in the particular case \( n = 3 \), for a comparable accuracy.

In the following, we assume a radix-2, precision-\( p \) floating-point (FP) arithmetic system, with unlimited exponent range and correct rounding. As a consequence, our results will apply to “real-world” binary floating-point arithmetic, such as the one specified by the IEEE 754-2008 Standard, provided that underflow and overflow do not occur. We also assume the availability of an FMA (fused multiply-add) instruction. Such an instruction evaluates expressions of the form \( ab + c \) with one final rounding only.

The notation \( a|b \) means “\( a \) divides \( b \)”. The notation RN\((t)\) stands for \( t \) rounded to the nearest FP number, ties-to-even, and RU\((t)\) (resp. RD\((t)\)) stands for \( t \) rounded towards \( +\infty \) (resp. \( -\infty \)). We will use three classical functions of the floating-point literature: upf (unit in the last place) [12], [21]), ufp (unit in the first place) [28]) and uls (unit in the last significant place). They can be defined as follows. If \( x \neq 0 \) is a real number, then:

- \( \text{upf}(x) = 2^{\lfloor \log_2 |x| \rfloor} \);
- \( \text{ulp}(x) = \text{upf}(x) \cdot 2^{-p+1} \);
- \( \text{uls}(x) \) is the largest power of 2 that divides \( x \), i.e., the largest \( 2^k \) \( (k \in \mathbb{Z}) \) such that \( x/2^k \) is an integer.

When \( x \) is a FP number, \( \text{upf}(x) \) is the weight of its most

\(^1\) See http://www.metalibm.org/lutetia.html
Algorithm 2 – 2Sum

Algorithm 1 – Fast2Sum (more detail below), Algorithms 1 and 2 compute the sum of two, three or more FP numbers, one has to explain to which extent they can “overlap”: after all the sum of the three double-precision/binary64 numbers 1, 2, and 4 is just a three-bit number, expressing it as the sum of three FP numbers does not make it more accurate. Several definitions appear in the literature. The first needed in this paper is Priest’s definition:

**Definition 1.** The sequence \( (x_i) \) is P-nonoverlapping (with Priest’s definition [27]) when \( |x_i + 1| < \text{ulp}(x_i) \).

We also introduce the following definition, more restrictive than Shewchuk’s definition [29]:

**Definition 2.** The sequence \( (x_i) \) is F-nonoverlapping (with Fabiano’s definition) when \( |x_i + 1| \leq \frac{1}{2} \text{uls}(x_i) \).

**Definition 3.** For any definition of nonoverlapping, a sequence \( (x_i) \) is said nonoverlapping w1Z (with possible interleaving zeros) when we have a set \( I_0 \) such that \( \forall i \in I_0, x_i = 0, \text{and } (x_i)_{i \notin I_0} \text{ is nonoverlapping}. \)

We can now formally define the double-word and triple-word numbers:

**Definition 4.** We call double-word number (DW) a pair \((x_0, x_1)\) of FP numbers such that \( x_0 = \text{RN}(x_0) + x_1 \).

**Definition 5.** We call triple-word number (TW) a triplet \((x_0, x_1, x_2)\) of FP numbers that is P-nonoverlapping.

Note the deliberate difference between Definitions 4 and 5. To obtain a definition similar to Definition 4, one could define a triple-word number as a 3-uple \((x_0, x_1, x_2)\) of floating-point numbers such that \( x_0 = \text{RN}(x_0) + x_1 + x_2 \) and \( x_1 = \text{RN}(x_1) + x_2 \). Such a requirement would be much stronger than Definition 5, resulting in practice in more complex and less efficient algorithms.

The definition of general expansions in [25] is based on ulp-nonoverlapping \((\forall i, |x_{i+1}| \leq \text{ulp}(x_i))\), which is slightly less restrictive than the one we chose for TW. Therefore algorithms proven for general expansions may not be correct for TW.

The paper is organized as follows. Section 2 presents some other basic blocks that will be used in the rest of the paper, and some original results related to them. Section 3 proves that the classical algorithm to turn
an arbitrary sequence into an expansion works for TW. Section 4 presents an algorithm to correctly round a TW to a FP number, and proves its correctness. Section 5 does the same as section 3 for the sum of two TW. Section 6 presents two versions of an original algorithm for computing the product of two TW, proves their correctness and gives tight error bounds. Sections 7 to 10 provide a similar analysis, in the case of the product of a DW by a TW, the reciprocal of a TW, the quotient of two TW, and the square root of a TW, respectively. Section 11 compares our results to the ones known for general n-word expansions, with \( n = 3 \).

2 OTHER BASIC BLOCKS

The Algorithms on TW presented in this paper use as basic blocks the 2Sum, Fast2Sum and 2Prod algorithms presented in the previous section, as well as the following, less classical, VecSum and VecSumErrBranch algorithms. Many properties of these algorithms have been proven elsewhere [26], [24], [2], but in this paper, we will need specific properties, presented below.

2.1 VecSum

The VecSum algorithm (Algorithm 4) first appears as a part of Priest’s normalization algorithm [26]. The name “VecSum” was coined by Ogita et al [24]. The aim of this algorithm is to turn a sequence that is “slightly” nonoverlapping into one that is “more” nonoverlapping, with no error. It is illustrated Fig. 1.

**Algorithm 4 – VecSum** \((x_0, \ldots, x_{n-1})\). \((6n – 6)\) operations

Ensure: \(e_0 + \cdots + e_{n-1} = x_0 + \cdots + x_{n-1}\)

\[s_{n-1} \leftarrow x_{n-1}\]

for \(i = n - 2\) to 0 do

\[s_i, e_{i+1} \leftarrow 2\text{Sum}(x_i, s_i+1)\]

end for

\(e_0 \leftarrow s_0\)

return \((e_0, e_1, \ldots, e_{n-1})\)

Fig. 1. The VecSum Algorithm [26], [24].

There are several theorems related to this algorithm, that use different definitions of nonoverlapping. In what follows, we will use the following original result:

**Theorem 1.** Assume, after removing possible interleaving zeros, that we can write in a non-necessarily canonical way (i.e., we do not necessarily have \(|M_i| \geq 2^{n-1}\)), \(x_i = M_i2^{e_i+r_i+1}\), \(|M_i| < 2^n\), such that \(\forall i \leq n - 2, k_i - 1 \geq k_i + 1\), and \(k_{n-2} \geq k_{n-1}\). Then VecSum \((x_0, \ldots, x_{n-1})\) is F-nonoverlapping wIZ with the same sum.

In this case, Fast2Sum can be used instead of 2Sum, so that VecSum only costs \(3n – 3\) operations.

**Proof:** Interleaving zeros in the input simply give some interleaving zeros in the output without changing the non-zero terms, so that we can suppose that we have removed them.

Firstly, \(\forall i, |s_i| \leq (2 - 2u)2^{k_i-1}\).

Indeed, if by induction \(|s_{i+1}| \leq (2 - 2u)2^{k_i}\), given \(|x_i| \leq (2 - 2u)2^{k_i}\) we get \(|s_{i+1}| + |x_{i+1}| \leq (4 - 4u)2^{k_i} \leq (2 - 2u)2^{k_i-1}\) so \(|s_i| \leq (2 - 2u)2^{k_i-1}\).

This gives \(|e_i| \leq 2u2^{k_i-1}\), and justifies Fast2Sum being used.

We suppose that \(|e_i| > |s_i+1|\) with \(i' < i\). We also suppose without loss of generality that \(|s_i\) with \(i'\) is a multiple of 2. We get a \(e_i\) non-multiple of 2u, that must be the case for one of the \(x_i, j \leq i - 2\). In particular, we have \(2k_j \leq \frac{1}{2}\), hence \(2^{k_j-2} \leq \frac{1}{2}\), so \(2^{k_i-1} \leq \frac{1}{2}\), which contradicts \(|e_i| \leq 2u2^{k_i-1}\).

The conditions on the input of Theorem 1 are complex, so we will use the following corollary:

**Corollary 1.** Assume that we have \(I \subset [1, n - 2]\) with no 2 consecutive indices such that

\(\forall i \in [0, n - 2], i \notin I\), \(\text{ufp}(x_{i+1}) \leq \frac{1}{2}\text{ufp}(x_i)\), and

\(\forall i \in I, \text{ufp}(x_{i+1}) \leq 2^{n-2}\text{ulp}(x_i)\) and \(\text{ufp}(x_{i+1}) \leq \frac{1}{4}\text{ufp}(x_{i-1})\)

Then VecSum \((x_0, \ldots, x_{n-1})\) is F-nonoverlapping with the same sum. In this case, Fast2Sum can be used instead of 2Sum, so that Algorithm 4 only costs \(3n – 3\) operations.

**Proof:** For \(i \notin I\), we take \(k_i = e_x\) the canonical exponent, and for \(i \in I\), we take \(k_i = \max(k_{i+1} + 1, e_x)\). This is possible because: \(2^{k_i+1}2^{-p+2}\text{ulp}(x_i)\) and \(2^{e_x}2^{p-1}\text{ulp}(x_i)\), which imply \(2^{k_i-1}\text{ulp}(x_i)\) and \(|x_i| \leq 2^{e_x}2^{-p+1},\) which imply \(|x_i| \leq 2^{k_i}2^{-p+1}\).

For \(i, i + 1 \notin I\), we have \(k_{i+1} \leq k_i - 1\). For \(i \in I\), we have on one hand \(k_{i+1} \leq k_i - 1\), and on the other hand \(e_x \leq k_{i-1} - 1\) and \(k_{i+1} \leq k_{i-1} - 2\) so \(k_i \leq k_{i-1} - 1\).

2.2 VecSumErrBranch

VecSumErrBranch (Algorithm 5 below) has similarities with Algorithm 4, but sums are computed starting from the larger terms, and some tests help avoiding to return too many zero terms. It is illustrated Fig. 2. It can be traced back to a part of Hida, Li and Bailey’s normalization algorithm for “quad-double” numbers [6], which itself is a variant of Priest’s normalization algorithm [26].
Algorithm 5 – VSEB(ε₀, . . . , εₙ₋₁). (6n − 6 operations & n − 2 tests)

Ensure: y₀ + · · · + yₙ₋₁ = e₀ + · · · + eₙ−₁

j ← 0
ε₀ ← ε₀
for i = 0 to n − 3 do
  (rᵢ, εᵢ₊₁) ← 2Sum(εᵢ, εᵢ₊₁)
  if εᵢ₊₁ ̸= 0 then
    yⱼ ← rᵢ
    εᵢ+₁ ← εᵢ₊₁
    incr j
  else
    εᵢ₊₁ ← rᵢ
  end if
end for
yⱼ, yⱼ+₁ ← 2Sum(εₙ₋₂, εₙ₋₁)
yⱼ+₂, . . . , yₙ−₁ ← 0
return (y₀, y₁, . . . , yₙ−₁)

Fig. 2. The VecSumErrBranch (VSEB) Algorithm.

We prove the following property, of Algorithm 5, that will be useful later on.

Theorem 2. If (εᵢ) is F-nonoverlapping wIZ, then VSEB(x₀, . . . , xₙ−₁) is P-nonoverlapping with the same sum, provided that p ≥ n − 1. In this case, Fast2Sum can be used instead of 2Sum, so that Algorithm 5 only costs 3n − 3 operations.

Proof: Again, interleaving zeros in the input can be ignored without changing anything, so we suppose that we have removed them. We write εᵢ = Mᵢ · 2ᵏᵢ with |Mᵢ| < 2ᵖ odd so that |εᵢ₊₁| ≤ ½ 2ᵏᵢ. By an easy induction, for all i, rᵢ−₁ and εᵢ are multiples of 2ᵏᵢ.

Let i₀ be such that |εᵢ₀| = 2ᵏᵢ₀. Let us show by induction that for all i₀ ≤ i ≤ n − 2, we have |rᵢ| ≤ 2ᵏᵢ₀(2 − 2ᵖ⁻¹).

We can initialize with i = i₀ − 1, because what is transmitted to next step (playing the role of rᵢ₀−₁) is εᵢ₀, which exactly satisfies the condition.

We suppose that |rᵢ−₁| ≤ 2ᵏᵢ₀(2 − 2ᵖ⁻¹).

We have |εᵢ₊₁| ≤ ½ 2ᵏᵢ ≤ ½ 2ᵏᵢ−₁ ≤ · · · ≤ 2ᵖ⁻² 2ᵏᵢ₀.

Thus |rᵢ−₁| + |εᵢ₊₁| ≤ 2ᵏᵢ₀(2 − 2ᵖ⁻²), and this is a FP number because i₀ ≥ 1 and i ≤ n − 2 give i₀−i−1 ≥ −n+2 ≥ −p+1, so that |rᵢ| ≤ 2ᵏᵢ₀(2 − 2ᵖ⁻¹). This gives the result by induction.

In particular, with j such that yⱼ = rᵢ₀−₁, we have ulp(yⱼ) ≥ 2|εᵢ₀| = 2 · 2ᵏᵢ₀ so |yⱼ+₁| < ulp(yⱼ).

• For i₀ such that |εᵢ₀| > 2ᵏᵢ₀, we similarly get that for all i ≥ i₀ such that εᵢ₊₁, . . . , εᵢ = 0, we have |rᵢ| ≤ |εᵢ₀| + 2ᵏᵢ₀.

Indeed, the same proof replacing 2ᵏᵢ₀(2 · · ·) by |εᵢ₀|+2ᵏᵢ₀(1 · · ·) works, except that the equality case can be reached in case of errors. In this case, this is sufficient given ulp(yⱼ) ≥ 2|εᵢ₀| > |εᵢ₀| + 2ᵏᵢ₀ ≥ yⱼ+₁.

• Finally, we can use Fast2Sum. Indeed, we have 2ᵏᵢ, εᵢ, εᵢ so 2ᵏᵢ|εᵢ|, and |εᵢ₊₁| ≤ 2ᵏᵢ.

Usually, we do not want to keep the complete output, but only a fixed number of terms. The resulting algorithm is denoted by VSEB(k)(ε₀, . . . , εₙ−₁). We have

Theorem 3. The relative error caused by keeping only the first k terms of a P-nonoverlapping sequence is bounded by 2uᵏ + 4.2 · uᵏ⁺¹, provided that p ≥ 6.

Proof: We have by P-nonoverlapping:

ulp(yᵏ) ≤ u · ulp(yᵏ−₁) ≤ · · · ≤ uᵏ · ulp(y₀),

hence,

|yᵏ| + · · · + |yₙ−₁| ≤ (2 − 2u)(uᵏ + uᵏ⁺¹ + · · · + uᵖ)ulp(y₀),

hence |yᵏ + · · · + yₙ−₁| ≤ 2uᵏulp(y₀).

We also have,

|y₀ + · · · + yₙ−₁| ≥ |y₀| − |y₁ + · · · + yₙ−₁| ≥ (1 − 2u)ulp(y₀).

Therefore,

|yᵏ + · · · + yₙ−₁| ≤ 2uᵏ 1−2u |y₀ + · · · + yₙ−₁|,

which implies the theorem.

2.3 Composed algorithm

Algorithms VecSum and VSEB were designed to be composed, in order to obtain a “normalization” algorithm such as Algorithm 6 below. In that case, we note that the first 2Sum in VSEB can be skipped because (ε₀, ε₁) is already a DW. When Fast2Sum can be used everywhere, we get a total of 6n − 9 operations and n − 2 tests.

3 ARBITRARY THREE FP NUMBERS TO TW

Before manipulating Triple-Word numbers, we want to be able to turn any unevaluated sum of three FP numbers into a TW, with no error (this is the equivalent of the 2Sum algorithm for 3 FP numbers). We can use Algorithm 6, which can be found in [6] for quad-word numbers, and in [25, page 87] for general expansions:
We denote

Algorithm 7 – RoundTW($x_0, x_1, x_2$). (3 operations & 4 tests)

Require: $\bar{x}$ TW
Ensure: $y = \text{RN}(\bar{x})$
if $\text{RN}(x_0 + 2x_1)$ inexact operation or (*) $\text{RN}(-\frac{1}{2}u - 2u^2) : x_0 \neq x_1$
then $y = \text{RN}(x_0 + x_1)$
else if $x_2 > 0$
then $y = \text{RU}(x_0 + x_1)$
else if $x_2 < 0$
then $y = \text{RD}(x_0 + x_1)$
else $y = \text{RN}(x_0 + x_1)$
end if
return $y$

We have,

Theorem 4. If $a, b, c$ are FP numbers, then $\text{ToTW}(a, b, c)$ is a TW, provided that $p \geq 4$.

Proof: If $(e_0, e_1, e_2)$ is F-nonoverlapping, then Theorem 2 concludes.

First, $|e_1| \leq \frac{1}{2}\text{ulp}(s)$ gives $(e_0, e_1)$ F-nonoverlapping.
We denote $s := \text{RN}(d_1 + z)$ the intermediate value in $\text{VecSum}(d_0, d_1, c)$.

- If $e_1 \neq 0$, we suppose without loss of generality that $\text{uls}(e_1) = u$, and $e_2 > \frac{1}{2}u$. Then $|e_2| \leq \frac{1}{2}\text{ulp}(s)$ gives $s \geq 1$ so $2u$'s but $e_1$ is not divisible by $2u$ so $d_0$ neither, hence $d_0 < 1$, so $|d_1| \leq \frac{1}{2}\text{ulp}(d_0) \leq \frac{1}{2}u$. Furthermore, $|c + d_1| \geq 1 + \frac{1}{2}u$. Thus $|c| \geq (1 + \frac{1}{2}u) - \frac{1}{2}u = 1$ so $\text{ulp}(c) \geq 2u > 2|d_1|$ so $s = c$ and $e_2 = d_1$, which is impossible since $|e_2| > \frac{1}{2}u \geq |d_1|$. So $(e_1, e_2)$ is F-nonoverlapping too.
- If $e_1 = 0$, then the same reasoning works with $e_0$ instead of $e_1$.

Another typical way of forming a TW consists in using any of the following algorithms, but with inputs that are DW instead of TW (with an implicit third term equal to zero, and simplifications obtained by removing useless operations).

4 Rounding a TW to a FP number

Frequently, TW arithmetic is used in intermediate calculations, but the final result must be a floating-point number. Hence, we need to be able to return the FP number closest to a TW number. This is for typically the case when we use TW numbers in intermediate calculations for implementing correctly rounded elementary functions. This can be done using Algorithm 7.

Algorithm 8 – TWSum($x_0, x_1, x_2, y_0, y_1, y_2$). (42 operations & 8 tests)

Require: $\bar{x}$ and $\bar{y}$ TW
Ensure: $\bar{r}$ TW and $|\bar{r} - (\bar{x} + \bar{y})| \leq 2u^3 + 4.2u^4$
$z_0, \ldots, z_5 \leftarrow \text{Merge}((x_0, x_1, x_2), (y_0, y_1, y_2))$
$e_0, \ldots, e_5 \leftarrow \text{VecSum}(z_0, \ldots, z_5)$
$r_0, r_1, r_2 \leftarrow \text{VSEB}(e_0, \ldots, e_5)$
return $(r_0, r_1, r_2)$

5 Sum of Two TW Numbers

To compute the sum of two TW numbers, we simply use the composition of VecSum and VSEB after a preliminary sorting of the input. This gives Algorithm 8 below. That algorithm would of course also work to compute the sum of a DW and a TW with the same error bound, but with less operations and tests.

5.1 Correctness of Algorithm 8

We have,

Theorem 6. Let $x_0, \ldots, x_5$ be FP numbers such that

$$\forall i, |x_{i+1}| \leq |x_i| \text{ and } \forall i, |x_{i+2}| < \text{ulp}(x_i).$$

Then $\text{VSEB}(\text{VecSum}(x_0, \ldots, x_5))$ is P-nonoverlapping, provided that $p \geq 4$.

Interestingly enough, we can notice that Theorem 6 may not hold for more than 6 floating-point inputs. Indeed, for 7 inputs, we can consider

$$(x_1) = 1 - u, -1 + 2u, -u + u^2, u - w^2, u^2 - w^3, u^3 - w^4$$
Algorithm 9 – 3Prod\(_{x,y}^{a,c}\)(\(x_0, x_1, x_2, y_0, y_1, y_2\)). (46 operations & 2 tests)

Require: \(\bar{x}\) and \(\bar{y}\) TW; \(p \geq 6\)
Ensure: \(\bar{r}\) TW and \(|\frac{x-\bar{x}}{y}\| \leq 28u^3 + 107u^4\)

\[
\begin{align*}
&z_{00}^+ = z_{00}^- \leftarrow 2\text{Prod}(x_0, y_0) \\
&z_{01}^+ = z_{01}^- \leftarrow 2\text{Prod}(x_0, y_1) \\
&z_{10}^+ = z_{10}^- \leftarrow 2\text{Prod}(x_1, y_0) \\
&b_0, b_1, b_2 \leftarrow \text{VecSum}(z_{00}^+, z_{01}^+, z_{10}^+) \\
&c \leftarrow \text{RN}(b_2 + x_1 y_1) \text{(FMA)} \\
&z_{31} \leftarrow \text{RN}(z_{10}^+ + x_0 y_2) \text{(FMA)} \\
&z_{32} \leftarrow \text{RN}(z_{01}^+ + x_2 y_0) \text{(FMA)} \\
&z_3 \leftarrow \text{RN}(z_{31} + z_{32}) \\
&e_0, e_1, e_2, e_3, e_4 \leftarrow \text{VecSum}(z_0^+, b_0, b_1, c, z_3) \\
r_0 \leftarrow e_0 \\
r_1, r_2 \leftarrow \text{VSEB}(2)(e_1, e_2, e_3, e_4) \\
\text{return} \ (r_0, r_1, r_2)
\end{align*}
\]

6.1 Bounds on the different terms

We suppose without loss of generality that \(1 \leq x_0, y_0 < 2\), so that \(|x_1|, |y_1| < 2u\) and \(|x_2|, |y_2| < 2u^2\). Then, we have:

\[
\begin{align*}
1 & \leq |z_{00}^+| < 4 \\
|z_{00}^-| & \leq 2u: \ ulp(z_{00}^-) \geq 4u^2 \\
|z_{01}^+|, |z_{10}^+| & < 4u \quad |b_2| < 4u^2 \\
|x_1 y_1| & < 4u^2 - 4u^3 \quad |c| < 8u^2 \\
|z_{31}, z_{32}| & \leq 6u^2 \quad |z_3| \leq 12u^2 \\
|s_3| & \leq 20u^2
\end{align*}
\]

6.2 Correctness and error bound of Algorithm 9

We have,

Theorem 7. If \(\bar{x}, \bar{y}\) are TW numbers and \(p \geq 6\), then 3Prod\(_{x,y}^{a,c}\)(\(\bar{x}, \bar{y}\)) is a TW number, and the relative error committed by 3Prod\(_{x,y}^{a,c}\)(\(\bar{x}, \bar{y}\)) is bounded by \(28u^3 + 107u^4\).

Proof of the theorem:

1. \((r_0, r_1, r_2)\) is a TW number.
   - First, let us prove that the last 2 lines are equivalent to computing \(r_0, r_1, r_2 = \text{VSEB}(3)(e_0, e_1, e_2, e_3, e_4)\)
     - If \(e_1 \neq 0\), then the fact that \(e_0 = \text{RN}(e_0 + e_1)\) concludes immediately.
     - If \(e_1 = 0\), one easily checks that \(|s_1|, |s_2|, |s_3| < 16u \leq 1 \ulp(z_{00}^+)\) so that the next nonzero \(|e_i|\) is strictly less than \(1 \ulp(e_0)\), which concludes.

   - Then, let us prove that, with this equivalent version, \((r_0, r_1, r_2)\) is P-nonoverlapping.

We denote \(a := \text{RN}(s_0^+, s_2^+ + s_3^+)\) and \(s_3 := \text{RN}(c + z_3)\) intermediate sums in VecSum.

We want to show that VecSum\((s_0^+, b_0, b_1, s_3)\) is P-nonoverlapping, with \(e_2\) F-nonoverlapping them too.

Thanks to Theorem 2, this would imply that \((r_0, r_1, r_2)\) is P-nonoverlapping.

- First, let us show that \((s_0^+, b_0, b_1, s_3)\) satisfies the conditions of Theorem 1.

  First, we always have \(\ulp(s_0^+) \geq 1\) much larger than four times any other number computed; and in case on they are non-zero \(\ulp(b_1) \leq \frac{1}{2} \ulp(b_0) < \frac{1}{2} \ulp(b_0)\).

  For the rest of the proof, we suppose without loss of generality that \(|x_1| \geq |y_1|\).

  On one hand, we easily obtain \(|s_3| \leq 10\ulp(x_1)|. 

On the other hand lemma 1 gives \(\frac{1}{2} \ulp(x_1)| with \(\ulp(x_1) \leq u^2 < \ulp(z_{00}^-) \text{ so } \frac{1}{2} \ulp(x_1)|b_0, b_1|.

  Case \(s_3 = 0\): we use \(I = 0\). (nothing more to show)

  Case \(s_3 \neq 0\) and \(b_0 = 0\) (so \(b_1 = 0\)): we use \(I = 0\). (idem)

  Case \(s_3 \neq 0\), \(b_0 \neq 0\) and \(b_1 = 0\): we use \(I = \{1\}\).
Indeed, we have $\text{ufp}(s_3) \leq \frac{1}{4} \text{ufp}(z_{00}^+)$, and $\text{ufp}(s_4) \leq 2^{p-2}\text{uls}(b_0)$. (using $p \geq 6$)

- Case $s_3 \neq 0$, $b_0 \neq 0$ and $b_1 \neq 0$: we use $I = \{2\}$.

Indeed, we have $\text{ufp}(s_3) \leq 16\text{ufp}(b_1) \leq \frac{1}{4} \text{ufp}(b_0)$ and $\text{ufp}(s_4) \leq 2^{p-2}\text{uls}(b_1)$. (using $p \geq 6$)

It also allows us to call Fast2Sum instead of the three 2Sum in this part of the algorithm, saving FP operations.

- Then, let us show that $e_4$ is F-nonoverlapping with the other $e_i$.

Firstly, $\text{ulp}(s_3) \geq 2|e_4|$. Secondly, we have seen that $\text{uls}(b_0), \text{uls}(b_1) \geq \frac{1}{2} \text{ulp}(x_1) \geq \frac{1}{20}|s_3| \geq \text{ulp}(s_3)$. (using $p \geq 6$). Thirdly, $\text{ulp}(z_{00}^+) \geq 2u > \text{ulp}(s_3)$.

Thus $e_0, e_1$ and $e_2$ are divisible by $\text{ulp}(s_3) \geq 2|e_4|$.

2. The relative error is bounded by $28u^3 + 107u^4$.

There are three sources of error: the terms that are ignored, the roundings in the computation of $z_3$ and $c$ and the terms not kept in VSEB. A naive analysis gives:

$$|\epsilon_0| := |xy + yz(x + yz)| \leq 2(2u - 2u^2)(2u^2 - 2u^3) + (2u^2 - 2u^3)^2 \leq 8u^3 - 11.9u^4$$
$$|\epsilon_1| := |(z_{10}^- + x_0y_0) - z_{3,1}| \leq u \cdot \text{ufp}(z_{10}^- + x_0y_0) \leq u \cdot \text{ufp}(2u^2 + 4u^2) \leq 4u^3$$
$$|\epsilon_2| := |(z_{10}^- + x_2y_0) - z_{3,2}| \leq 4u^3$$
$$|\epsilon_3| := |(z_{3,1} + z_{3,2}) - z_3| \leq 8u^3$$
$$|\epsilon_4| := |(b_2 + x_1y_1) - c| \leq 4u^3$$
$$|\epsilon_5| := |(z_{00}^+ + b_0 + b_1 + c + z_3) - (r_0 + r_1 + r_2)| \leq (2u^3 + 4.2u^4)|z_{00}^+ + b_0 + b_1 + c + z_3|$$

Yet, $x, y \geq 1 - (2u - 2u^2) - (2u^2 - 2u^3) \geq 1 - 2u$ so $xy \geq 1 - 4u$. We eventually obtain that the error cannot be too large if $e_3 \neq 0$ (not detailed here), and finally:

$$\left|\frac{\bar{r} - \bar{xy}}{\bar{xy}}\right| \leq \frac{28u^3 - 11.9u^4}{1 - 4u} \leq 28u^3 + 107u^4. \tag{1}$$

The bound (1) is very tight. Indeed, for instance in binary64 arithmetic, if we take:

$$z_{00}^+ = (x_0, x_1, x_2) = (1 + 13 \cdot 2^{26} + 28u, 2u - 2^{27}u^2, 2u^2 - 4u^3)$$
$$z_{01}^- = (y_0, y_1, y_2) = (1 + 7 \cdot 2^{12}u, 2u - (2^{28} - 8)u^2, 2u^2 - 4u^3),$$

we obtain a relative error around $(28 - 10^{-6})u^3$.

6.4 Faster version

In this version, $e_4$ is not computed.

Algorithm 10 – 3Prod$^{\text{Fast}}_{3,3}$ $(x_0, x_1, x_2, y_0, y_1, y_2)$. (38 operations & 1 test)

Require: $x$ and $y$ TW ; $p \geq 6$

Ensure: $\bar{r}$ TW and $\frac{\bar{r} - \bar{xy}}{\bar{xy}} \leq 44u^3 + 176u^4$

[same 5 first lines as Algorithm 9]

$z_{3,1} \leftarrow \text{RN}(z_{10}^- + x_0y_0)$ (FMA)

$z_{3,2} \leftarrow \text{RN}(z_{01}^- + x_2y_0)$ (FMA)

$z_3 \leftarrow \text{RN}(z_{3,1} + z_{3,2})$

$s_3 \leftarrow \text{RN}(c + z_3)$

$e_0, e_1, e_2, e_3 \leftarrow \text{VecSum}(z_{00}^+, b_0, b_1, s_3)$

$r_0 \leftarrow e_0$

$r_1, r_2 \leftarrow \text{VSEB}(2)(e_1, e_2, e_3)$

return $(r_0, r_1, r_2)$

This saves 8 operations and 1 test. The proof of correctness and the computation of the error bound are similar to those for Algorithm 9. The obtained error bound is $44u^3 + 176u^4$, which is very tight, because with the input values (2) in binary64 arithmetic, we obtain a relative error around $(44 - 10^{-5})u^3$.

7 Multiplying a DW by a TW

To multiply a DW $(x_0, x_1)$ by a TW $(y_0, y_1, y_2)$, we use the same algorithms as previously, slightly simplified to take into account that $x_2 = 0$. Additionally to being interesting in itself, this operation will be used later on to compute reciprocals and quotients, which justifies a separate analysis.

Algorithm 11 – 3Prod$^{\text{Fast}}_{3,3}$ $(x_0, x_1, y_0, y_1, y_2)$. (45 operations & 2 tests)

Require: $x$ DW and $\bar{y}$ TW ; $p \geq 6$

Ensure: $\bar{r}$ TW and $\frac{\bar{r} - \bar{xy}}{\bar{xy}} \leq 10.5u^3 + 39u^4$

[same 5 first lines as Algorithm 9]

$z_{3,1} \leftarrow \text{RN}(z_{10}^- + x_0y_0)$ (FMA)

$z_3 \leftarrow \text{RN}(z_{3,1} + z_{01}^-)$

$e_0, e_1, e_2, e_3, e_4 \leftarrow \text{VecSum}(z_{00}^+, b_0, b_1, c, s_3)$

$r_0 \leftarrow e_0$

$r_1, r_2 \leftarrow \text{VSEB}(2)(e_1, e_2, e_3, e_4)$

return $(r_0, r_1, r_2)$

7.1 Correctness, number of operations and error bound

Since Algorithm 11 is a particular case of Algorithm 9, correctness is directly ensured, provided that $p \geq 6$. Compared to Algorithm 9, we saved 1 operation, so that the total number of operations is 41. The error bound of Theorem 7 holds. However, one can redo the analysis taking into account that $x_2 = 0$ and obtain a better bound, namely,
Theorem 8. If \( \bar{x} \) is a DW number and \( \bar{y} \) is a TW number, then the relative error committed by 3Prod_{2,3}^\text{fast}(\bar{x}, \bar{y}) \) is bounded by 10.5u^3 + 39u^4, provided that \( p \geq 6 \).

The proof is omitted. It is unclear whether the bound of Theorem 8 is very tight, but it is not too bad either: in binary64 arithmetic, the choice

\[
\begin{align*}
(x_0, x_1) &= (1 + 3 \cdot 2^{27} u, u - 2^{27} u^2) \\
y_0, y_1, y_2 &= (1 + (3 \cdot 2^{26} + 6)u, 2u - 5 \cdot 2^{27} u^2, 2u^2 - 26u^3)
\end{align*}
\]

(3)

gives a relative error around \((10 - 2 \cdot 10^{-6})u^3\).

7.2 Faster version

As previously, one obtains a faster yet slightly less accurate algorithm by avoiding the computation of \( e_4 \). This gives Algorithm 12 below.

Algorithm 12 – 3Prod_{2,3}^\text{fast}(x_0, x_1, y_0, y_1, y_2). (37 operations & 1 test)

Require: \( \bar{x} \) DW and \( \bar{y} \) TW ; \( p \geq 6 \)
Ensure: \( \bar{r} \) TW and \( \|\bar{r} - \bar{x}\bar{y}\|/\|\bar{x}\bar{y}\| \leq 18u^3 + 75u^4 \)

[TW with same 5 first lines as Algorithm 9]

\[
\begin{align*}
z_{3,1} &= \text{RN}(z_{10} \times y_2) \quad \text{(FMA)} \\
z_3 &= \text{RN}(z_{3,1} + z_{0,1}) \\
s_3 &= \text{RN}(c, z_3) \\
e_0, e_1, e_2, e_3 &= \text{VecSum}(z_{0,0}^+, b_0, b_1, s_3) \\
r_0 &= e_0 \\
r_1, r_2 &= \text{VSEB}(2)(e_1, e_2, e_3) \\
\text{return} \ (r_0, r_1, r_2)
\end{align*}
\]

Similarly to what was done before, we obtain, provided that \( p \geq 6 \),

\[|\bar{r} - \bar{x}\bar{y}|/|\bar{x}\bar{y}| \leq 18u^3 + 75u^4.\]

This bound is very tight: in binary64 arithmetic, with the input values given by (3), we obtain a relative error around \((18 - 2.4 \cdot 10^{-6})u^3\).

8 Reciprocal of a TW number

To compute the reciprocal of a TW, we use Algorithm 13 below, which is based on the Newton-Raphson iteration, following the idea of [8] for general expansions. This algorithm requires the stronger constraint \( p \geq 10 \). For computing \( 1/x \), the Newton-Raphson iteration is

\[r_{n+1} = r_n(2 - r_n x)\]

which ensures a quadratic convergence towards \( 1/x \) as soon as \( r_0 \) is close enough to \( 1/x \).

Algorithm 13 – 3Reci(x_0, x_1, x_2). (73 operations & 2 tests and relative error bounded by 11.5u^3 + 1465u^4 if 3Prod_{2,3} is Algorithm 11, 65 operations & 1 test and relative error bounded by 19u^3 + 1502u^4 if 3Prod_{2,3} is Algorithm 12.

Require: \( \bar{x} \) TW ; \( p \geq 10 \)
\[a \leftarrow \text{RN}((1 + 2u)/x_0)\]
\[h_{1,1} \leftarrow 2\text{Prod}_2((1 + 2u)(a, x_0) = \text{RN}(ax_0 - (1 + 2u)) \quad \text{(FMA)}\]
\[h_1 \leftarrow \text{RN}(-h_{1,1} - ax_1) \quad \text{(FMA)}\]
\[b_{0,1}, b_{1,1} \leftarrow 2\text{Prod}(a, 1 - 2u) \quad \text{(FMA)}\]
\[b_{1,2} \leftarrow \text{RN}(b_{1,1} + ah_1) \quad \text{(FMA)}\]
\[\bar{b} \leftarrow \text{Fast2Sum}(b_{0,1}, b_{1,2})\]
\[\bar{r} \leftarrow 2 - 3\text{Prod}_{2,3}(\bar{b}, \bar{x})\]
\[\bar{y} \leftarrow 3\text{Prod}_{2,3}(\bar{b}, \bar{r})\]
\[\text{return} \ (y_0, y_1, y_2)\]

A natural starting point for the Newton-Raphson iterations would be \( \text{RN}(1/x_0) \), obtained through a floating-point division. However, Algorithm 13 uses \( \text{RN}((1 + 2u)/x_0) \) instead of \( \text{RN}(1/x_0) \) to start the calculations in order to take profit from the fact that for any FP number \( x \), \( \text{RN}(x \times \text{RN}(1/x_0)) = 1 + 2u \) (the proof is straightforward).

Thus in Algorithm 13, one has to imagine a “virtual” \( h_0 = 2 - (1 + 2u) = 1 - 2u \). This defines a DW number \( \bar{h} = (h_0, h_1) \) that approximates \( 2 - a \cdot (x_0 + x_1) \).

Several variants are possible for implementing the last two lines of Algorithm 13. First, one can choose the accurate (Algorithm 11) or fast (Algorithm 12) version of 3Prod_{2,3} to implement the TW\( \times \)TW products. Then, further simplification is possible: to compute \( 2 - 3\text{Prod}_{2,3}(\bar{b}, \bar{x}) \) there is no need to actually perform a multiplication followed by a subtraction, one can just slightly modify the multiplication algorithm, essentially by replacing the terms \( e_i \) in Algorithm 11 or Algorithm 12 by their opposite by turning some + into − operations and conversely in order not to waste any operation (this gives Algorithm 16 in the appendix, see supplementary material). The last product can also be simplified by taking into account the fact that \( i_0 = 1 \). We have,

Theorem 9. If \( \bar{x} \) is a TW, and if \( p \geq 10 \), then the relative error committed by 3Reci(\( \bar{x} \)) is bounded by 11.5u^3 + 1465u^4 if 3Prod_{2,3} is Algorithm 11, 65 operations & 1 test and relative error bounded by 19u^3 + 1502u^4 if 3Prod_{2,3} is Algorithm 12.

The proof of Theorem 9, with the modified versions of Algorithms 11 and 12 is given in the appendix.

9 Quotient of two TW numbers

To compute \( 1/\bar{x} \), the first idea that springs in mind is to compute the reciprocal of \( \bar{x} \) and then to multiply it by \( \bar{x} \). However (using the notation \( \bar{b} \) that appears in Algorithm 13), this would mean that we compute something like \( \bar{x} \times (\bar{b} \times (2 - \bar{b}x)) \), while it is significantly better to compute \( (\bar{x} \times \bar{b}) \times (2 - \bar{b}x) \). In particular, this allows to parallelize the computations of \( \bar{b} \) and \( 2 - \bar{b}x \). We obtain Algorithm 14 below.
Algorithm 14 – 3Div(\(z_0, z_1, z_2, x_0, x_1, x_2\)). (119 operations & 4 tests and relative error bounded by \(24u^3 + 1509u^4\) if the “accurate” multiplications algorithms are used, and 103 operations & 2 tests and relative error bounded by \(39u^3 + 1582u^4\) if the “fast” multiplications algorithms are used.)

Require: \(\bar{z}, \bar{x}\) TW ; \(p \geq 10\)

[same 5 first lines as Algorithm 13]
\[
\begin{align*}
\hat{b} &\leftarrow \text{Fast2Sum}(b_{\text{0,1}}, b_{\text{1,2}}) \\
\hat{i} &\leftarrow 2 - 3\text{Prod}_{2,3}(\hat{b}, \hat{x}) \\
\bar{a} &\leftarrow 3\text{Prod}_{2,3}(\bar{b}, \bar{x}) \\
\bar{y} &\leftarrow 3\text{Prod}_{3,3}(\bar{a}, \bar{i}) \\
\text{return} \ (y_0, y_1, y_2)
\end{align*}
\]

As for Algorithm 13, several variants are possible for implementing the products \(3\text{Prod}_{2,3}\) and \(3\text{Prod}_{3,3}\) in Algorithm 14. One can choose the accurate or fast version of the DW×TW and TW×TW multiplication algorithms. Again, the subtraction that appears in the line “ \(\hat{i} \leftarrow 2 - 3\text{Prod}_{2,3}(\hat{b}, \hat{x})\) ” does not need to be performed: the multiplication algorithm is slightly modified instead (Algorithm 16 in the appendix, see supplementary material).

For the computation of the final product, one of course can use Algorithm 9 or Algorithm 10. However, as for the reciprocal operation, these algorithms can be simplified significantly by taking into account that \(i_0 = 1\). The corresponding multiplication algorithm in the “fast” case is Algorithm 18 in the appendix. We finally have,

Theorem 10. If \(\bar{z}, \bar{x}\) are TW numbers and if \(p \geq 10\), then the relative error committed by \(3\text{Div}(\bar{z}, \bar{x})\) is bounded by \(24u^3 + 1509u^4\) if the “accurate” versions of the multiplication algorithms are used, and by \(39u^3 + 1582u^4\) if the “fast” versions are used.

The proof is given in the appendix, see supplementary material.

10 SQUARE ROOT OF A TW NUMBER

To compute the square root of a TW number, we use Algorithm 15 below, based again on the Newton-Raphson iteration. Its analysis and the various possible optimizing tricks are very similar to the ones used before for division. The underlying iteration is now

\[ r_{n+1} = r_n \left( \frac{3}{2} - \frac{1}{2} r_n^2 \right), \]

which ensures a quadratic convergence towards \(1/\sqrt{x}\) as soon as \(r_0\) is close enough to \(1/\sqrt{x}\).

Algorithm 15 – 3SqRt(\(x_0, x_1, x_2\)). (127 operations & 4 tests and and relative error bounded by \(24u^3 + 10260u^4\) if the “accurate” multiplications algorithms are used, and 111 operations & 2 tests and relative error bounded by \(39u^3 + 10333u^4\) if the “fast” multiplications algorithms are used.)

Require: \(\bar{x}\) TW ; \(p \geq 11\)

Ensure: \(\bar{y}\) TW and \(\left| \frac{\bar{y}}{\sqrt{\bar{x}}} - \sqrt{\bar{x}} \right| \leq 24u^3 + 10260u^4\), resp. \(39u^3 + 10333u^4\)

\[
\begin{align*}
\hat{a} &\leftarrow \text{RN}(1 + 4u) / \text{RN}(\sqrt{x_0}) \\
\hat{a} &\leftarrow \frac{1}{2} \hat{a} \ (\text{exact}) \\
h_0^{(1)}, h_1^{(1)} &\leftarrow 2\text{Prod}(a, x_0) \\
h_0^{(1)} &\leftarrow \text{RN}(h_1^{(1)} + a x_1) \ (\text{FMA}) \\
h_0^{(2)}, h_1^{(2)} &\leftarrow 2\text{Prod}(a', h_0^{(1)}) \\
h_0^{(2)} &\leftarrow 1.5 - h_0^{(2)} \ (\text{exact}) \\
h_1^{(2)} &\leftarrow -\text{RN}(h_1^{(2)} + a h_0^{(2)}) \ (\text{FMA}) \\
b_0^{(1)}, b_1^{(1)} &\leftarrow 2\text{Prod}(a, h_0^{(2)}) \\
b_1^{(2)} &\leftarrow \text{RN}(b_1^{(2)} + a h_0^{(2)}) \ (\text{FMA}) \\
\hat{b} &\leftarrow \text{Fast2Sum}(b_{\text{0,1}}, b_{\text{1,2}}) \\
\hat{b}' &\leftarrow \frac{1}{2} \hat{b} \ (\text{exact}) \\
\bar{i}^{(1)} &\leftarrow 3\text{Prod}_{2,3}(\hat{b}, \hat{x}) \\
\bar{i}^{(2)} &\leftarrow 1.5 - 3\text{Prod}_{2,3}(\hat{b}', \bar{i}^{(1)}) \\
\bar{y} &\leftarrow 3\text{Prod}_{3,3}(\bar{i}^{(1)}, \bar{i}^{(2)}) \\
\text{return} \ (y_0, y_1, y_2)
\end{align*}
\]

We have,

Theorem 11. If \(\bar{x}\) is a TW, then the relative error committed by \(3\text{SqRt}_N^{\text{acc}}(\bar{x})\) (resp. \(3\text{SqRt}_N^{\text{fast}}(\bar{x})\)) is bounded by \(24u^3 + 10260u^4\) (resp. \(39u^3 + 10333u^4\)).

The major steps of the proof are given in the appendix, see supplementary material.

11 IMPLEMENTATION AND TESTS

11.1 Experimental settings

We have implemented the multiplication and division algorithms presented in this paper (Algorithms 9 and 10 for multiplication, and Algorithm 14 in the “fast” and “accurate” versions for division). Our goal was twofold: to compare the performances of our algorithms with other solutions, and to assess the correctness and tightness of our error bounds. Our algorithms have been implemented in the Campany library [25]. We compare them to algorithms based on “general” floating-point expansions used in the special case \(n = 3\), also implemented in the Campany library, and to the GNU MPFR [5] multiple-precision library. MPFR also provides reference values. The various considered algorithms are given in Table 1.

Source code of this benchmark is available by request to the authors. It has been compiled with GCC 8.2.1 20181127 with compilation options -O3 -march=native, libraries GMP 6.1.2, and MPFR 4.0.2 from Archlinux repositories were used. The precision
We checked the various considered TW and FP expansion where URandom has an uniform density probability in
(3Div
Table 2, the “accurate” version of the division algorithm
variability. In Table 2, we reported the average and
values, their accuracies will not differ significantly.

11.2 Correctness and performance

We checked the various considered TW and FP expansion algorithms over a large set of around \(4 \times 10^6\) randomly
chosen operands. Each result was compared to the result obtained by the corresponding correctly rounded MPFR
algorithm. The second and third columns of Table 2 show for each considered algorithm the largest relative error
obtained during these tests, along with the corresponding
to the proven error bound referenced in Table 1. Interestingly enough, for our multiplication and division
algorithms, the largest encountered errors in our tests are not far from the bounds: for instance, with the 3Prod
algorithm (Algorithm 9), the largest encountered error is only 0.554 times the bound. This shows that our error
bounds are rather tight.

Since in practice there is not much difference in the
accuracy of the “fast” and “accurate” versions of the
multiplication and division algorithms, it makes sense
to use the “fast” versions only. Incidentally, note that in
Table 2, the “accurate” version of the division algorithm
(3Div
seems less accurate than the other version. This
results from the fact that 3Div
more accurate in terms of worst case analysis only: on randomly generated
values, their accuracies will not differ significantly.

The time taken to perform the operations on a large set
of randomly chosen operands is also measured. Each
measure is performed many times to account for its
variability. In Table 2, we reported the average and
standard deviation of our measures. One can see that
our algorithms are significantly faster than MPFR and
the general floating-point expansion algorithms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Known error bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpfr_mul</td>
<td>[25, Alg 31]</td>
<td>8u₃ + 48u₄ [25, Thm 3.4.4.]</td>
</tr>
<tr>
<td>baileyMul</td>
<td>[25, Alg 33]</td>
<td>8u₃ + 49u₄ [25, Thm 3.4.5.]</td>
</tr>
<tr>
<td>3Prod⁹⁹⁹digit</td>
<td>Alg 9</td>
<td>28u₃ + 107u₄ Thm 7</td>
</tr>
<tr>
<td>3Prodfast</td>
<td>Alg 10</td>
<td>44u₃ + 176u₄</td>
</tr>
</tbody>
</table>

**TABLE 1**

Algorithms considered in our tests.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Known error bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpfr_div</td>
<td>[25, Alg 36]</td>
<td>24u₃ + 1509u₄ Thm 10</td>
</tr>
<tr>
<td>division</td>
<td>[25, Alg 36]</td>
<td>39u₃ + 1582u₄ Thm 10</td>
</tr>
</tbody>
</table>

Table 2: Results of correctness and speed tests for multiplication and division algorithms. The speed is measured in Moperations/s (a larger figure corresponds to a faster algorithm).

<table>
<thead>
<tr>
<th>Name</th>
<th>Relative error</th>
<th>Ratio to bound</th>
<th>Speed (Mop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpfr_mul</td>
<td>n/a</td>
<td>37.0</td>
<td>0.9</td>
</tr>
<tr>
<td>truncatedMul</td>
<td>3.42 \times 10^{-49}</td>
<td>0.031</td>
<td>48.3</td>
</tr>
<tr>
<td>baileyMul</td>
<td>3.42 \times 10^{-49}</td>
<td>0.031</td>
<td>64.5</td>
</tr>
<tr>
<td>3Prod⁹⁹⁹digit</td>
<td>2.12 \times 10^{-47}</td>
<td>0.554</td>
<td>66.4</td>
</tr>
<tr>
<td>3Prodfast</td>
<td>2.69 \times 10^{-47}</td>
<td>0.446</td>
<td>80.3</td>
</tr>
<tr>
<td>mpfr_div</td>
<td>n/a</td>
<td>10.4</td>
<td>0.3</td>
</tr>
<tr>
<td>division</td>
<td>6.43 \times 10^{-49}</td>
<td>9.3</td>
<td>0.2</td>
</tr>
<tr>
<td>3Div⁹⁹⁹digit</td>
<td>1.56 \times 10^{-47}</td>
<td>0.476</td>
<td>13.2</td>
</tr>
<tr>
<td>3Divfast</td>
<td>1.31 \times 10^{-47}</td>
<td>0.245</td>
<td>16.8</td>
</tr>
</tbody>
</table>

Hence, compared to standard \(n\)-word floating-point
expansion algorithms with \(n = 3\), our triple-word algo-
rithms offer faster arithmetic, at the price of a slight (yet
carefully bounded) loss in accuracy. This is of interest for
implementing correctly-rounded transcendental functions
in floating-point arithmetic, since in general this requires a
bit more than twice the target precision (i.e., double-word
arithmetic is not enough, and very accurate triple-word
arithmetic is an overkill).

**Conclusion**

We have shown that usual floating-point expansion
algorithms adapted for building and adding triple-word
numbers are correct in the context of triple-words, and
we have introduced algorithms for rounding, multiplying,
reciprocating, dividing and computing square roots of
triple-word numbers, along with their correctness proofs
and error bounds.

Some of the error bounds have been shown to be tight
(at least in binary64 arithmetic). Our algorithms have been
implemented in the Campany library. Our experiments
show that the obtained algorithms are faster than generic
\(n\)-word arithmetics at the price of a slight (yet carefully
bounded) loss in accuracy.

**Acknowledgement**

We are grateful to the anonymous reviewers, whose
comments have been very helpful for revising the original
manuscript.

**References**

[1] D. H. Bailey, R. Barrio, and J. M. Borwein, High precision computa-
tion: Mathematical physics and dynamics., Applied Mathematics and

[2] Sylvie Boldo, Miara Joldes, Jean-Michel Muller, and Valentina
Popescu, Formal verification of a floating-point expansion renormaliza-
tion algorithm, 8th International Conference on Interactive Theorem
Proving (ITP) (Brasilia, Brazil), 2017.
Nicos Fabiano was born in Paris, France, in 1998. He is currently studying Computer Science at the ENS Paris, and he is especially interested in game theory. He is involved in different math associations, especially the scientific part of the Tournoi Français des Jeunes Mathématiciennes et Mathématiciens (TFJM²) and its international version.

Jean-Michel Muller was born in Grenoble, France, in 1961. He received his Ph.D. degree in 1985 from the Institut National Polytechnique de Grenoble. He is Directeur de Recherches (senior researcher) at CNRS, France, and he is the co-head of GDR-IM. His research interests are in Computer Arithmetic. Dr. Muller was co-program chair of the 13th IEEE Symposium on Computer Arithmetic (Asilomar, USA, June 1997), general chair of SCAN97 (Lyon, France, sept. 1997), general chair of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia, April 1999), general chair of the 22nd IEEE Symposium on Computer Arithmetic (Lyon, France, June 2015). He is the author of several books, including “Elementary Functions, Algorithms and Implementation” (3rd edition, Birkhäuser, 2016), and he coordinated the writing of the “Handbook of Floating-Point Arithmetic” (2nd edition Birkhäuser, 2018). He served as an associate editor of the IEEE Transactions on Computers from 2014 to 2018, and he is a fellow of the IEEE.

Joris Picot was born in Troyes, France, in 1985. He received his Ph.D. degree in 2013 from the Université de Toulouse. He spend seven years in the development of computational fluid dynamics software, and he is Research Engineer at École Normale Supérieure de Lyon, France. His research interests are in Rigorous Computing.