Algorithms for triple-word arithmetic

Nicolas Fabiano, Jean-Michel Muller, Joris Picot

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 explicited 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 precisions 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 $\mathrm{RN}(t)$ stands for t rounded to the nearest FP number, ties-to-even, and $\mathrm{RU}(t)$ (resp. $\mathrm{RD}(t)$) stands for t rounded towards $+\infty$ (resp. $-\infty$). We will use three classical functions of the floating-point literature: ulp (*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:

- $\mathrm{ufp}(x) = 2^{\lfloor \log_2 |x| \rfloor}$;
- $\mathrm{ulp}(x) = \mathrm{ufp}(x) \cdot 2^{-p+1}$;
- $\mathrm{uls}(x)$ 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, $\mathrm{ufp}(x)$ is the weight of its most

- *N. Fabiano is with École Normale Supérieure, 45 Rue d'Ulm, 75005 Paris, France. E-mail: nicolabiano22@yahoo.fr*
- *J.-M. Muller is with Univ. Lyon, CNRS, EnsL, Inria, UCBL, LIP, F-69342, LYON Cedex 07, France. E-mail: jean-michel.muller@ens-lyon.fr*
- *J. Picot is with Univ. Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France. E-mail: joris.picot@ens-lyon.fr*

1. See http://www.metalibm.org/lutetia.html

significant bit, $\mathrm{ulp}(x)$ is the weight of its least significant bit, and $\mathrm{uls}(x)$ is the weight of its rightmost nonzero bit. For instance, assuming binary64/double precision arithmetic ($p = 53$), if

$$x = -1.01101_2 \times 2^{364}$$

then $\mathrm{ufp}(x) = 2^{364}$, $\mathrm{ulp}(x) = 2^{312}$, and $\mathrm{uls}(x) = 2^{359}$.

The number $u = 2^{-p} = \frac{1}{2}\mathrm{ulp}(1)$ denotes the roundoff error unit. When an arithmetic operation $a \nabla b$ is performed, where a and b are FP numbers, what is effectively computed is $\mathrm{RN}(a \nabla b)$, and if t is a real number and $T = \mathrm{RN}(t)$, then

$$|t - T| \leq \frac{u}{1+u} \cdot |t| \leq u \cdot |t| \text{ and } |t - T| \leq u \cdot |T|,$$

and more precisely

$$|t - T| \leq \frac{1}{2}\mathrm{ulp}(t) \leq \frac{1}{2}\mathrm{ulp}(T).$$

The algorithms presented in this paper (as well as the usual algorithms that manipulate double-words or general expansions) use as basic blocks the classical algorithms Fast2Sum (Algorithm 1), 2Sum (Algorithm 2), and 2Prod (Algorithm 3) given below. Roughly speaking (more detail below), Algorithms 1 and 2 compute the error of a FP addition, and Algorithm 3 computes the error of a FP multiplication.

Algorithm 1 – Fast2Sum(a, b). (3 operations) [4].

Require: $\exists$ integers $k_a \geq k_b, M_a, M_b$ (with $|M_a|, |M_b| \leq 2^p - 1$), s.t. $a = M_a \cdot 2^{k_a}$ and $b = M_b \cdot 2^{k_b}$.
Ensure: $s + e = a + b$
  $s \leftarrow \mathrm{RN}(a + b)$
  $z \leftarrow \mathrm{RN}(s - a)$
  $e \leftarrow \mathrm{RN}(b - z)$
  return (s, e)

If there exist integers $k_a \geq k_b, M_a, M_b$ (with $|M_a|, |M_b| \leq 2^p - 1$), s.t. $a = M_a \cdot 2^{k_a}$ and $b = M_b \cdot 2^{k_b}$ (which holds if $|a| \geq |b|$) then values s and e computed by Algorithm 1 satisfy $s + e = a + b$. Hence, e is the error of the FP addition $s \leftarrow \mathrm{RN}(a + b)$.

Algorithm 2 – 2Sum(a, b). (6 operations) [19], [13].

Ensure: $s + e = a + b$
  $s \leftarrow \mathrm{RN}(a + b)$
  $a' \leftarrow \mathrm{RN}(s - b)$
  $b' \leftarrow \mathrm{RN}(s - a')$
  $\delta_a \leftarrow \mathrm{RN}(a - a')$
  $\delta_b \leftarrow \mathrm{RN}(b - b')$
  $e \leftarrow \mathrm{RN}(\delta_a + \delta_b)$
  return (s, e)

Algorithm 2 requires twice as many operations as Algorithm 1, but its output variables always satisfy $s + e = a + b$: no knowedge of the respective magnitudes of $|a|$ and $|b|$ is needed.

Algorithm 3 – 2Prod(a, b). (2 operations) [11], [23], [21])

Ensure: $\pi + e = a \cdot b$
  $\pi \leftarrow \mathrm{RN}(a \cdot b)$
  $e \leftarrow \mathrm{RN}(a \cdot b - \pi)$ (FMA)
  return (π, e)

The values π and e computed by Algorithm 3 satisfy $\pi + e = ab$. Algorithm 3 uses an FMA instruction (for computing $\mathrm{RN}(a \cdot b - \pi)$).

Sometimes, we know in advance the value of s or π. In that case, e can be computed saving the first operation, with algorithms denoted in the following by for instance $\mathrm{2Sum}_2(s)(x, y)$.

When one defines a number as the unevaluated 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 $\forall i, |x_{i+1}| < \mathrm{ulp}(x_i)$.*

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 $\forall i, |x_{i+1}| \leq \frac{1}{2}\mathrm{uls}(x_i)$.*

Definition 3. *For any definition of nonoverlapping, a sequence (x_i) is said nonoverlapping wIZ (with possible interleaving zeros) when we have a set I_0 such that $\forall i \in I_0, x_i = 0$, and $(x_i)_{i \notin I_0}$ 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 $x_0 = \mathrm{RN}(x_0 + x_1)$.*

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 $x_0 = \mathrm{RN}(x_0 + x_1 + x_2)$ and $x_1 = \mathrm{RN}(x_1 + x_2)$. 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 ($\forall i, |x_{i+1}| \leq \mathrm{ulp}(x_i)$), 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**
    $s_i, e_{i+1} \leftarrow \mathrm{2Sum}(x_i, s_{i+1})$
  end for
  $e_0 \leftarrow s_0$
  return $(e_0, e_1, \ldots, e_{n-1})$

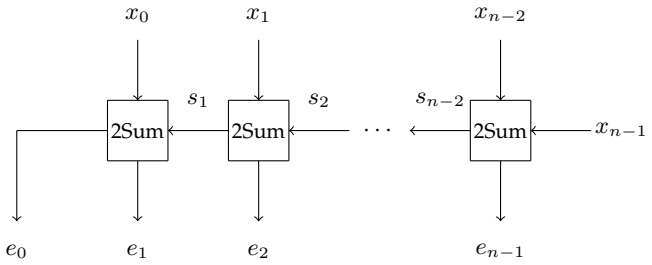

Fig. 1. The VecSum Algorithm [26], [24].

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 $|M_i| \geq 2^{p-1}$) $x_i = M_i 2^{k_i - p + 1}$, $|M_i| < 2^p$, 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 $|s_{i+1}| + |x_i| \leq (4-4u)2^{k_i} \leq (2-2u)2^{k_{i-1}}$ 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 $|e_i| > \frac{1}{2}\mathrm{uls}(e_{i'})$ with $i' < i$. We also suppose without loss of generality that $\mathrm{uls}(e_{i'}) = u$. We easily get by induction that for all i, if $2^k \big| s_i, x_{i-1}, \ldots, x_0$, then $2^k \big| e_i, \ldots, e_0$. Yet $|e_i| \leq \frac{1}{2}\mathrm{ulp}(s_{i-1})$ so $|s_{i-1}| \geq 1$ so s_{i-1} is a multiple of $2u$. Given we want a $e_{i'}$ non-multiple of $2u$, that must be the case for one of the $x_j, j \leq i-2$. In particular, we have $2^{k_j} \leq \frac{1}{2}$, hence $2^{k_{i-2}} \leq \frac{1}{2}$, so $2^{k_{i-1}} \leq \frac{1}{4}$, 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*

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

and

$$\forall i \in I, \mathrm{ufp}(x_{i+1}) \leq 2^{p-2}\mathrm{uls}(x_i) \text{ and } \mathrm{ufp}(x_{i+1}) \leq \frac{1}{4}\mathrm{ufp}(x_{i-1})$$

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 $k_i = e_{x_i}$ the canonical exponent, and for $i \in I$, we take $k_i = \max(k_{i+1} + 1, e_{x_i})$. This is possible because: $2^{k_{i+1}-p+2} \big| x_i$ and $2^{e_{x_i}-p+1} \big| x_i$, which imply $2^{k_i - p + 1} \big| x_i$, and $|x_i| \leq 2 \cdot 2^{e_{x_i}}$, which imply $|x_i| \leq 2 \cdot 2^{k_i}$.

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 $e_{x_i} \leq k_{i-1} - 1$ 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$(e_0, \dots, e_{n-1})$. ($6n-6$ operations & $n-2$ tests)

Ensure: $y_0 + \cdots + y_{n-1} = e_0 + \cdots + e_{n-1}$
  $j \leftarrow 0$
  $\epsilon_0 \leftarrow e_0$
  for $i = 0$ **to** $n-3$ **do**
    $(r_i, \epsilon_{i+1}^t) \leftarrow \mathsf{2Sum}(\epsilon_i, e_{i+1})$
    if $\epsilon_{i+1}^t \neq 0$ **then**
      $y_j \leftarrow r_i$
      $\epsilon_{i+1} \leftarrow \epsilon_{i+1}^t$
      incr j
    else
      $\epsilon_{i+1} \leftarrow r_i$
    end if
  end for
  $y_j, y_{j+1} \leftarrow \mathsf{2Sum}(\epsilon_{n-2}, e_{n-1})$
  $y_{j+2}, \dots, y_{n-1} \leftarrow 0$
  return $(y_0, y_1, \dots, y_{n-1})$

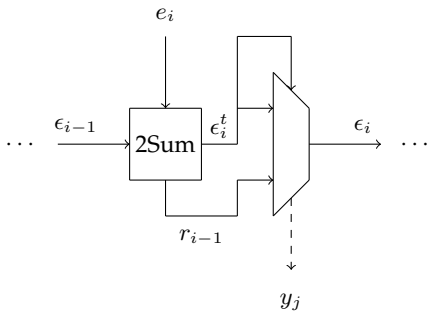

Fig. 2. The VecSumErrBranch (VSEB) Algorithm.

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

Theorem 2. *If (e_i) is F-nonoverlapping wIZ, then VSEB$(x_0, \dots, x_{n-1})$ is P-nonoverlapping with the same sum, provided that $p \geq n-1$. 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 $e_i = M_i \cdot 2^{k_i}$ with $|M_i| < 2^p$ odd so that $|e_{i+1}| \leq \frac{1}{2} 2^{k_i}$. By an easy induction, for all i, r_{i-1} and ϵ_i are multiples of 2^{k_i}.

- Let i_0 be such that $|\epsilon_{i_0}| = 2^{k_{i_0}}$. Let us show by induction that for all $i_0 \leq i \leq n-2$, we have $|r_i| \leq 2^{k_{i_0}}(2 - 2^{i_0 - i - 1})$.
  - We can initialize with $i = i_0 - 1$, because what is transmitted to next step (playing the role of r_{i_0-1}) is ϵ_{i_0}, which exactly satisfies the condition.
  - We suppose that $|r_{i-1}| \leq 2^{k_{i_0}}(2 - 2^{i_0 - i})$.
    We have $|e_{i+1}| \leq \frac{1}{2} 2^{k_i} \leq \frac{1}{4} 2^{k_{i-1}} \leq \cdots \leq 2^{i_0 - i - 1} 2^{k_{i_0}}$.
    Thus $|r_{i-1}| + |e_{i+1}| \leq 2^{k_{i_0}}(2 - 2^{i_0 - i - 1})$, and this is a FP number because $i_0 \geq 1$ and $i \leq n-2$ give $i_0 - i - 1 \geq -n+2 \geq -p+1$, so that $|r_i| \leq 2^{k_{i_0}}(2 - 2^{i_0 - i - 1})$. This gives the result by induction.

  In particular, with j such that $y_j = r_{i_0-1}$, we have $\mathrm{ulp}(y_j) \geq 2|\epsilon_{i_0}| = 2 \cdot 2^{k_{i_0}}$ so $|y_{j+1}| < \mathrm{ulp}(y_j)$.
- For i_0 such that $|\epsilon_{i_0}| > 2^{k_{i_0}}$, we similarly get that for all $i \geq i_0$ such that $\epsilon_{i_0+1}^t, \dots, \epsilon_i^t = 0$, we have $|r_i| \leq |\epsilon_{i_0}| + 2^{k_{i_0}}$.
  Indeed, the same proof replacing $2^{k_{i_0}}(2 - \cdots)$ by $|\epsilon_{i_0}| + 2^{k_{i_0}}(1 - \cdots)$ works, except that the equality case can be reached in case of errors. In this case, this is sufficient given $\mathrm{ulp}(y_j) \geq 2|\epsilon_{i_0}| > |\epsilon_{i_0}| + 2^{k_{i_0}} \geq y_{j+1}$.
- Finally, we can use Fast2Sum. Indeed, we have $2^{k_i} | e_0, \dots, e_i$ so $2^{k_i} | \epsilon_i$, and $|e_{i+1}| \leq 2^{k_i}$.

□

Usually, we do not want to keep the complete output, but only a fixed number of terms. The resulting algorithm is denoted by VSEB$(k)(e_0, \dots, e_{n-1})$. We have

Theorem 3. *The relative error caused by keeping only the first k terms of a P-nonoverlapping sequence is bounded by $2u^k + 4.2 \cdot u^{k+1}$, provided that $p \geq 6$.*

Proof: We have by P-nonoverlapping:

$$\mathrm{ufp}(y_k) \leq u \cdot \mathrm{ufp}(y_{k-1}) \leq \cdots \leq u^k \cdot \mathrm{ufp}(y_0),$$

hence,

$$|y_k| + \cdots + |y_{n-1}| \leq (2 - 2u)(u^k + u^{k+1} + \cdots + u^n)\mathrm{ufp}(y_0),$$

hence $|y_k + \cdots + y_{n-1}| \leq 2u^k \mathrm{ufp}(y_0)$.

We also have,

$$|y_0 + \cdots + y_{n-1}| \geq |y_0| - |y_1 + \cdots + y_{n-1}| \geq (1 - 2u)\mathrm{ufp}(y_0).$$

Therefore,

$$|y_k + \cdots + y_{n-1}| \leq \frac{2u^k}{1 - 2u} |y_0 + \cdots + y_{n-1}|,$$

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 (e_0, e_1) 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:

Algorithm 6 – ToTW(a, b, c). (21 operations & 1 test)

Ensure: $\bar{r}$ TW **and** $\bar{r} = a + b + c$
  $d_0, d_1 \leftarrow$ 2Sum(a, b)
  $e_0, e_1, e_2 \leftarrow$ VecSum(d_0, d_1, c)
  $r_0, r_1, r_2 \leftarrow$ VSEB(e_0, e_1, e_2)
  return (r_0, r_1, r_2)

We have,

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

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

First, $|e_1| \leq \frac{1}{2}\text{ulp}(e_0)$ gives (e_0, e_1) F-nonoverlapping. We denote $s := \text{RN}(d_1 + z)$ the intermediate value in 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 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** $(\star)$ $\text{RN}(-(\frac{3}{2}u - 2u^2) \cdot x_0) \neq x_1$ **then**
    $y \leftarrow \text{RN}(x_0 + x_1)$
  else if $x_2 > 0$ **then**
    $y \leftarrow \text{RU}(x_0 + x_1)$
  else if $x_2 < 0$ **then**
    $y \leftarrow \text{RD}(x_0 + x_1)$
  else
    $y \leftarrow \text{RN}(x_0 + x_1)$
  end if
  return y

At line 1 of Algorithm 7, although the fact that the operation $x_0 + 2x_1$ is exact can in theory be detected by checking a flag, it will be more efficient in practice to compute $(s, e) = \text{Fast2Sum}(x_0, 2x_1)$ and test whether e is zero.

We have,

Theorem 5. *If $\bar{x}$ is a TW, then $RoundTW(\bar{x}) = \text{RN}(\bar{x})$, provided that $p \geq 4$.*

Proof: First, if $x_0 + x_1$ is a FP number, then $y = x_0 + x_1$ anyway, and it is easy to check that $x_0 + x_1 = \text{RN}(\bar{x})$. We suppose for the sequel of the proof that this is not the case.

Given $|x_1| < \text{ulp}(x_0)$, the first condition is false iff $x_0 + x_1$ is halfway between two consecutive FP numbers, or in a special case that can without loss of generality be reduced to $x_0 = 1 + 2u$ and $x_1 = -\frac{3}{2}u$. When that first condition is false, Condition $(\star)$ is designed to be true in the special case, but false elsewise (because of the magnitude of $|x_1|$).

- If $x_0 + x_1$ is halfway between two adjacent FP numbers, then the rounding is decided by the sign of x_2.
- Otherwise, one easily checks that $\text{RN}(x_0 + x_1 + x_2) = \text{RN}(x_0 + x_1)$, given $|x_2| < \text{ulp}(x_1)$.

□

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.

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

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 VSEB (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

$$(x_i) = 1-u, -1+2u, -u+u^2, u-u^2, u^2-u^3, u^2-u^3, u^3-u^4$$

which gives $(e_i) = u, u^2, u^2, -u^3, -u^4$, and finally

$$(y_i) = u, 2u^2, -u^3, -u^4$$

with $2u^2 = \text{ulp}(u)$.

This is why it is reasonable to use the notion of P-nonoverlapping for TW numbers only, but not for general expansions, for which Algorithm 8 preserves ulp-nonoverlapping only [25, page 90].

Sketch of the proof: For space constraints, the proof of Theorem 6 is not detailed. The main steps are:

- prove by induction that $|s_i| \leq 2\text{ufp}(x_{i-1})$ and $|s_i| \leq 4\text{ufp}(x_i)$;
- if $e_i > \frac{1}{2}\text{uls}(e_j)$ for some $j < i$, deduce some conditions on i and the nearby terms in various cases;
- conclude with a case study: $i \leq 3$ and $e_i > \frac{1}{2}\text{uls}(e_j)$; $i \geq 4$ and $e_i > \frac{1}{2}\text{uls}(e_j)$; or $0 < e_i \leq \frac{1}{2}\text{uls}(e_j)$.

□

5.2 Number of operations in Algorithm 8

In the Merge instruction of Algorithm 8, if the last two numbers to sort are x_2 and y_2, there is no need to do it because they play symmetrical roles in 2Sum. Thus this part costs only 4 tests. In VecSum, there are for each block examples where Fast2Sum cannot be used, so it costs 30 operations. One easily checks that Fast2Sum can be used in VSEB, so it costs 12 operations and 4 tests.

6 Product of two TW numbers

To compute the product of two TW numbers, we simply distribute the sub-products and aggregate the terms ensuring P-nonoverlapping, with an error as small as possible. The algorithms presented below guarantee commutativity, even if it is rarely useful in practice.

Algorithm 9 – 3Prod$^{acc}_{3,3}(x_0, x_1, x_2, y_0, y_1, y_2)$. (46 operations & 2 tests)

Require: $\bar{x}$ and $\bar{y}$ TW ; $p \geq 6$
Ensure: $\bar{r}$ TW **and** $\left|\frac{\bar{r}-\bar{x}\bar{y}}{\bar{x}\bar{y}}\right| \leq 28u^3 + 107u^4$
$z^+_{00}, z^-_{00} \leftarrow 2\text{Prod}(x_0, y_0)$
$z^+_{01}, z^-_{01} \leftarrow 2\text{Prod}(x_0, y_1)$
$z^+_{10}, z^-_{10} \leftarrow 2\text{Prod}(x_1, y_0)$
$b_0, b_1, b_2 \leftarrow \text{VecSum}(z^-_{00}, z^+_{01}, z^+_{10})$
$c \leftarrow \text{RN}(b_2 + x_1y_1)$ (FMA)
$z_{3,1} \leftarrow \text{RN}(z^-_{10} + x_0y_2)$ (FMA)
$z_{3,2} \leftarrow \text{RN}(z^-_{01} + x_2y_0)$ (FMA)
$z_3 \leftarrow \text{RN}(z_{3,1} + z_{3,2})$
$e_0, e_1, e_2, e_3, e_4 \leftarrow \text{VecSum}(z^+_{00}, b_0, b_1, c, z_3)$
$r_0 \leftarrow e_0$
$r_1, r_2 \leftarrow \text{VSEB}(2)(e_1, e_2, e_3, e_4)$
return (r_0, r_1, r_2)

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:

$$1 \leq |z^+_{00}| < 4$$
$$|z^-_{00}| \leq 2u \ ; \ \text{uls}(z^-_{00}) \geq 4u^2$$

$$|z^+_{01}|, |z^+_{10}| < 4u \qquad |b_2| \leq 4u^2$$
$$|x_1y_1| < 4u^2 - 4u^3 \qquad |c| < 8u^2$$

$$|z^-_{01}|, |z^-_{10}| \leq 2u^2 \qquad |x_0y_2|, |x_2y_0| < 4u^2$$
$$|z_{3,1}|, |z_{3,2}| \leq 6u^2 \qquad |z_3| \leq 12u^2$$

$$|s_3| \leq 20u^2$$

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 $3Prod^{acc}_{3,3}(\bar{x}, \bar{y})$ is a TW number, and the relative error committed by $3Prod^{acc}_{3,3}(\bar{x}, \bar{y})$ is bounded by $28u^3 + 107u^4$.*

Theorem 7 uses the following, straightforward, lemma.

Lemma 1. *For all FP numbers x, y, $\frac{1}{2}\text{ulp}(x) \big| \text{RN}(x + y)$.*

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 $|s_1|, |s_2|, |s_3| < 16u \leq \frac{1}{2}\text{ufp}(z^+_{00})$ so that the next nonzero $|e_i|$ is strictly less than $\frac{1}{2}\text{ulp}(e_0)$, which concludes.

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

We denote $a := \text{RN}(z^+_{01} + z^+_{10})$ and $s_3 := \text{RN}(c + z_3)$ intermediate sums in VecSum.

We want to show that $\text{VecSum}(z^+_{00}, b_0, b_1, s_3)$ is F-nonovelapping, with e_4 F-nonovelapping them too.

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

- First, let us show that $(z^+_{00}, b_0, b_1, s_3)$ satisfies the conditions of Theorem 1.

First, we always have $\text{ufp}(z^+_{00}) \geq 1$ much larger than four times any other number computed ; and in case on they are non-zero $\text{ufp}(b_1) \leq \frac{1}{2}\text{ulp}(b_0) < \frac{1}{2}\text{ufp}(b_0)$.

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

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

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

- Case $s_3 = 0$: we use $I = \emptyset$. (nothing more to show)
- Case $s_3 \neq 0$ and $b_0 = 0$ (so $b_1 = 0$): we use $I = \emptyset$. (idem)
- Case $s_3 \neq 0$, $b_0 \neq 0$ and $b_1 = 0$: we use $I = \{1\}$.

Indeed, we have $\mathrm{ufp}(s_3) \le \frac{1}{4}\mathrm{ufp}(z_{00}^+)$, and $\mathrm{ufp}(s_3) \le 2^{p-2}\mathrm{uls}(b_0)$. (using $p \ge 6$)

- Case $s_3 \neq 0$, $b_0 \neq 0$ and $b_1 \neq 0$: we use $I = \{2\}$. Indeed, we have $\mathrm{ufp}(s_3) \le 16\mathrm{ufp}(b_1) \le \frac{1}{4}\mathrm{ufp}(b_0)$ and $\mathrm{ufp}(s_3) \le 2^{p-2}\mathrm{uls}(b_1)$. (using $p \ge 6$)

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

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

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:

$$\begin{aligned}
|\epsilon_0| &:= |x_1y_2 + x_2y_1 + x_2y_2| \\
&\le 2(2u - 2u^2)(2u^2 - 2u^3) + (2u^2 - 2u^3)^2 \\
&\le 8u^3 - 11.9u^4
\end{aligned}$$

$$\begin{aligned}
|\epsilon_1| &:= |(z_{10}^- + x_0y_2) - z_{3,1}| \\
&\le u \cdot \mathrm{ufp}(z_{10}^- + x_0y_2) \\
&\le u \cdot \mathrm{ufp}(2u^2 + 4u^2) \\
&\le 4u^3
\end{aligned}$$

$$|\epsilon_2| := |(z_{01}^- + x_2y_0) - z_{3,2}| \le 4u^3$$

$$|\epsilon_3| := |(z_{3,1} + z_{3,2}) - z_3| \le 8u^3$$

$$|\epsilon_4| := |(b_2 + x_1y_1) - c| \le 4u^3$$

$$\begin{aligned}
|\epsilon_5| &:= |(z_{00}^+ + b_0 + b_1 + c + z_3) - (r_0 + r_1 + r_2)| \\
&\le (2u^3 + 4.2u^4)|z_{00}^+ + b_0 + b_1 + c + z_3|
\end{aligned}$$

Yet, $\bar{x}, \bar{y} \ge 1 - (2u - 2u^2) - (2u^2 - 2u^3) \ge 1 - 2u$ so $\bar{x}\bar{y} \ge 1 - 4u$. We eventually obtain that the error cannot be too large if $\epsilon_5 \neq 0$ (not detailed here), and finally:

$$\left|\frac{\bar{r} - \bar{x}\bar{y}}{\bar{x}\bar{y}}\right| \le \frac{28u^3 - 11.9u^4}{1 - 4u} \le 28u^3 + 107u^4. \qquad (1)$$

□

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

$$\begin{aligned}
&(x_0, x_1, x_2) \\
&= (1 + (13 \cdot 2^{26} + 28)u, 2u - 2^{27}u^2, 2u^2 - 4u^3) \\
&(y_0, y_1, y_2) \\
&= (1 + 7 \cdot 2^{27}u, 2u - (2^{28} - 8)u^2, 2u^2 - 4u^3),
\end{aligned} \qquad (2)$$

we obtain a relative error around $(28 - 10^{-5})u^3$.

6.3 Number of operations of Algorithm 9

Before the last 3 lines, we count 22 operations. There are 3 Fast2Sum in $\mathrm{VecSum}(z_{00}^+, b_0, b_1, c, z_3)$, so that it costs 15 operations. Finally, the call to VSEB costs 9 operations and 2 tests. Thus the total is 46 operations and 2 tests.

Note that the optimization that directly uses $e_0 = r_0$ does not save any operation, but it saves the first branching, which is very interesting.

6.4 Faster version

In this version, e_4 is not computed.

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

Require: $\bar{x}$ and $\bar{y}$ TW ; $p \ge 6$
Ensure: $\bar{r}$ TW **and** $\left|\frac{\bar{r} - \bar{x}\bar{y}}{\bar{x}\bar{y}}\right| \le 44u^3 + 176u^4$
[same 5 first lines as Algorithm 9]
$z_{3,1} \leftarrow \mathrm{RN}(z_{10}^- + x_0y_2)$ (FMA)
$z_{3,2} \leftarrow \mathrm{RN}(z_{01}^- + x_2y_0)$ (FMA)
$z_3 \leftarrow \mathrm{RN}(z_{3,1} + z_{3,2})$
$s_3 \leftarrow \mathrm{RN}(c + z_3)$
$e_0, e_1, e_2, e_3 \leftarrow \mathrm{VecSum}(z_{00}^+, b_0, b_1, s_3)$
$r_0 \leftarrow e_0$
$r_1, r_2 \leftarrow \mathrm{VSEB}(2)(e_1, e_2, e_3)$
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 – $\mathrm{3Prod}_{2,3}^{acc}(x_0, x_1, y_0, y_1, y_2)$. (45 operations & 2 tests)

Require: $\bar{x}$ DW and $\bar{y}$ TW ; $p \ge 6$
Ensure: $\bar{r}$ TW **and** $\left|\frac{\bar{r} - \bar{x}\bar{y}}{\bar{x}\bar{y}}\right| \le 10.5u^3 + 39u^4$
[same 5 first lines as Algorithm 9]
$z_{3,1} \leftarrow \mathrm{RN}(z_{10}^- + x_0y_2)$ (FMA)
$z_3 \leftarrow \mathrm{RN}(z_{3,1} + z_{01}^-)$
$e_0, e_1, e_2, e_3, e_4 \leftarrow \mathrm{VecSum}(z_{00}^+, b_0, b_1, c, z_3)$
$r_0 \leftarrow e_0$
$r_1, r_2 \leftarrow \mathrm{VSEB}(2)(e_1, e_2, e_3, e_4)$
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 $p \ge 6$. 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 $3Prod_{2,3}^{acc}(\bar{x},\bar{y})$ 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

$$\begin{array}{c}(x_0,x_1)\\ =(1+3\cdot 2^{27}u,u-2^{27}u^2)\\ (y_0,y_1,y_2)\\ =(1+(3\cdot 2^{26}+6)u,2u-5\cdot 2^{27}u^2,2u^2-26u^3)\end{array} \tag{3}$$

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$_{2,3}^{fast}(x_0,x_1,y_0,y_1,y_2)$. (37 operations & 1 test)

Require: $\bar{x}$ DW and $\bar{y}$ TW ; $p\geq 6$
Ensure: $\bar{r}$ TW **and** $\left|\frac{\bar{r}-\bar{x}\bar{y}}{\bar{x}\bar{y}}\right|\leq 18u^3+75u^4$
[same 5 first lines as Algorithm 9]
$z_{3,1}\leftarrow \mathrm{RN}(z_{10}^{-}+x_0y_2)$ (FMA)
$z_3\leftarrow \mathrm{RN}(z_{3,1}+z_{01}^{-})$
$s_3\leftarrow \mathrm{RN}(c,z_3)$
$e_0,e_1,e_2,e_3\leftarrow \mathrm{VecSum}(z_{00}^{+},b_0,b_1,s_3)$
$r_0\leftarrow e_0$
$r_1,r_2\leftarrow \mathrm{VSEB}(2)(e_1,e_2,e_3)$
return (r_0,r_1,r_2)

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

$$r_{n+1}=r_n(2-r_nx)$$

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 ; $p\geq 10$
$a\leftarrow \mathrm{RN}\left((1+2u)/x_0\right)$
$h_{1,1}\leftarrow 2\mathrm{Prod}_2(1+2u)(a,x_0)=\mathrm{RN}(ax_0-(1+2u))$ (FMA)
$h_1\leftarrow \mathrm{RN}(-h_{1,1}-ax_1)$ (FMA)
$b_{0,1},b_{1,1}\leftarrow 2\mathrm{Prod}(a,1-2u)$
$b_{1,2}\leftarrow \mathrm{RN}(b_{1,1}+ah_1)$ (FMA)
$\bar{b}\leftarrow \mathrm{Fast2Sum}(b_{0,1},b_{1,2})$
$\bar{i}\leftarrow 2-3\mathrm{Prod}_{2,3}(\bar{b},\bar{x})$
$\bar{y}\leftarrow 3\mathrm{Prod}_{2,3}(\bar{b},\bar{i})$
return (y_0,y_1,y_2)

A natural starting point for the Newton-Raphson iterations would be $\mathrm{RN}(1/x_0)$, obtained through a floating-point division. However, Algorithm 13 uses $\mathrm{RN}((1+2u)/(x_0))$ instead of $\mathrm{RN}(1/x_0)$ to start the calculations in order to take profit from the fact that for any FP number x, $\mathrm{RN}(x\times\mathrm{RN}(\frac{1+2u}{x}))=1+2u$ (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 DW×TW products. Then, further simplification is possible: to compute $2-3\mathrm{Prod}_{2,3}(\bar{b},\bar{x})$ 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 (accurate version), and by $19u^3+1502u^4$ if 3Prod$_{2,3}$ is Algorithm 12 (fast version).*

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 $\bar{z}/\bar{x}$, the first idea that springs in mind is to compute the reciprocal of $\bar{x}$ and then to multiply it by $\bar{z}$. However (using the notation $\bar{b}$ that appears in Algorithm 13), this would mean that we compute something like $\bar{z}\times(\bar{b}\times(2-\bar{b}\bar{x}))$, while it is significantly better to compute $(\bar{z}\times\bar{b})\times(2-\bar{b}\bar{x})$. In particular, this allows to parallelize the computations of $\bar{z}\bar{b}$ and $2-\bar{b}\bar{x}$. 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]
$\bar{b} \leftarrow$ Fast2Sum$(b_{0,1}, b_{1,2})$
$\bar{i} \leftarrow 2 - 3\text{Prod}_{2,3}(\bar{b}, \bar{x})$
$\bar{a} \leftarrow 3\text{Prod}_{2,3}(\bar{b}, \bar{z})$
$\bar{y} \leftarrow 3\text{Prod}_{3,3}(\bar{a}, \bar{i})$
return (y_0, y_1, y_2)

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 " $\bar{i} \leftarrow 2 - 3\text{Prod}_{2,3}(\bar{b}, \bar{x})$" 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 $\bar{x}$,$\bar{z}$ are TW numbers and if $p \geq 10$, then the relative error committed by $3Div(\bar{z}, \bar{x})$ 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

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

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. $39u^3 + 10333u^4$
$a \leftarrow \text{RN}\left((1+4u)/\text{RN}(\sqrt{x_0})\right)$
$a' = \frac{1}{2}a$ (exact)
$h_0^{(1)}, h_{1,1}^{(1)} \leftarrow 2\text{Prod}(a, x_0)$
$h_1^{(1)} \leftarrow \text{RN}(h_{1,1}^{(1)} + a x_1)$ (FMA)
$h_{0,1}^{(2)}, h_{1,1}^{(2)} \leftarrow 2\text{Prod}(a', h_0^{(1)})$
$h_0^{(2)} \leftarrow 1.5 - h_{0,1}^{(2)}$ (exact)
$h_1^{(2)} \leftarrow -\text{RN}(h_{1,1}^{(2)} + a' h_1^{(1)})$ (FMA)
$b_{0,1}, b_{1,1} \leftarrow 2\text{Prod}(a, h_0^{(2)})$
$b_{1,2} \leftarrow \text{RN}(b_{1,1} + a h_1^{(2)})$ (FMA)
$\bar{b} \leftarrow$ Fast2Sum$(b_{0,1}, b_{1,2})$
$\bar{b}' = \frac{1}{2}\bar{b}$ (exact)
$\overline{i^{(1)}} \leftarrow 3\text{Prod}_{2,3}(\bar{b}, \bar{x})$
$\overline{i^{(2)}} \leftarrow 1.5 - 3\text{Prod}_{2,3}(\bar{b}', \overline{i^{(1)}})$
$\bar{y} \leftarrow 3\text{Prod}_{3,3}(\overline{i^{(1)}}, \overline{i^{(2)}})$
return (y_0, y_1, y_2)

We have,

Theorem 11. *If $\bar{x}$ is a TW, then the relative error committed by $3SqRt^{acc}(\bar{x})$ (resp. $3SqRt^{fast}(\bar{x})$) 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 Campary library [25]. We compare them to algorithms based on "general" floating-point expansions used in the special case $n = 3$, also implemented in the Campary 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

Name	Definition	Known error bound
mpfr_mul		
truncatedMul	[25, Alg 31]	$8u^3 + 48u^4$ [25, Thm 3.4.4.]
baileyMul	[25, Alg 33]	$8u^3 + 49u^4$ [25, Thm 3.4.5.]
3Prod^{acc}	Alg 9	$28u^3 + 107u^4$ Thm 7
3Prod^{fast}	Alg 10	$44u^3 + 176u^4$
mpfr_div		
division	[25, Alg 36]	
3Div^{acc}	Alg 14	$24u^3 + 1509u^4$ Thm 10
3Div^{fast}	Alg 14	$39u^3 + 1582u^4$ Thm 10

TABLE 1
Algorithms considered in our tests.

Name	Relative error	Ratio to bound	Speed (Mop/s) mean	deviation
mpfr_mul	n/a		37.0	0.9
truncatedMul	$3.42 \cdot 10^{-49}$	0.031	48.3	0.9
baileyMul	$3.42 \cdot 10^{-49}$	0.031	64.5	1.2
3Prod^{acc}	$2.12 \cdot 10^{-47}$	0.554	66.4	1.2
3Prod^{fast}	$2.69 \cdot 10^{-47}$	0.446	80.3	1.7
mpfr_div	n/a		10.4	0.3
division	$6.43 \cdot 10^{-49}$		9.3	0.2
3Div^{acc}	$1.56 \cdot 10^{-47}$	0.476	13.2	0.3
3Div^{fast}	$1.31 \cdot 10^{-47}$	0.245	16.8	0.3

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).

of MPFR FP numbers has been set to $3 \times 53 = 159$. Computations have been performed with an Intel® Core™ i5-7440HQ CPU.

The input operands were generated using the MPFR URandom algorithm. More precisely, each FP component of a TW operand $x = (x_0, x_1, x_2)$ was obtained as:

$$x_{i+1} = \mathrm{URandom}() \times \mathrm{ulp}(x_i)\,,$$

where URandom has an uniform density probability in the range $[0, 1]$.

11.2 Correctness and performance

We checked the various considered TW and FP expansion algorithms over a large set of around 4×10^6 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 ratio 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^{acc} 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^{acc}) seems less accurate than the other version. This results from the fact that 3Div^{acc} is 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.

Hence, compared to standard n-word floating-point expansion algorithms with $n = 3$, our triple-word algorithms 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 Campary 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 computation: Mathematical physics and dynamics.*, Applied Mathematics and Computation **218** (2012), 10106–10121.

[2] Sylvie Boldo, Mioara Joldeş, Jean-Michel Muller, and Valentina Popescu, *Formal verification of a floating-point expansion renormalization algorithm*, 8th International Conference on Interactive Theorem Proving (ITP) (Brasilia, Brazil), 2017.

[3] Florent de Dinechin, Alexey V. Ershov, and Nicolas Gast, *Towards the post-ultimate libm*, 17th IEEE Symposium on Computer Arithmetic (ARITH-17), 2005, pp. 288–295.

[4] T. J. Dekker, *A floating-point technique for extending the available precision*, Numerische Mathematik **18** (1971), no. 3, 224–242.

[5] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, *MPFR: A multiple-precision binary floating-point library with correct rounding*, ACM Transactions on Mathematical Software **33** (2007), no. 2, 15 pages. Available at http://www.mpfr.org/.

[6] Y. Hida, X. S. Li, and D. H. Bailey, *Algorithms for quad-double precision floating-point arithmetic*, 15th IEEE Symposium on Computer Arithmetic (ARITH-15), June 2001, pp. 155–162.

[7] 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.

[8] M. Joldeş, J.-M. Muller, and V. Popescu, *On the computation of the reciprocal of floating point expansions using an adapted newton-raphson iteration*, 25th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'2014), June 2014, pp. 63–67.

[9] Mioara Joldeş, Jean-Michel Muller, and Valentina Popescu, *Tight and rigourous error bounds for basic building blocks of double-word arithmetic*, ACM Transactions on Mathematical Software **44** (2017), no. 2.

[10] Mioara Joldeş, Jean-Michel Muller, Valentina Popescu, and Warwick Tucker, *CAMPARY: Cuda multiple precision arithmetic library and applications*, 5th International Congress on Mathematical Software (ICMS), July 2016.

[11] W. Kahan, *Lecture notes on the status of IEEE-754*, Available at http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF, 1997.

[12] ———, *A logarithm too clever by half*, Available at http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT, 2004.

[13] D. E. Knuth, *The art of computer programming*, 3rd ed., vol. 2, Addison-Wesley, Reading, MA, 1998.

[14] C. Q. Lauter, *Basic building blocks for a triple-double intermediate format*, Tech. Report 2005-38, LIP, École Normale Supérieure de Lyon, September 2005.

[15] C. Q. Lauter, *Arrondi correct de fonctions mathématiques*, Ph.D. thesis, École Normale Supérieure de Lyon, Lyon, France, October 2008, In French, available at http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2008/PhD2008-07.pdf.

[16] V. Lefèvre and J.-M. Muller, *Worst cases for correct rounding of the elementary functions in double precision*, 15th IEEE Symposium on Computer Arithmetic (ARITH-15), June 2001.

[17] X. Li, J. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung, and D. J. Yoo, *Design, implementation and testing of extended and mixed precision BLAS*, Tech. Report 45991, Lawrence Berkeley National Laboratory, 2000, https://publications.lbl.gov/islandora/object/ir\%3A115848.

[18] ———, *Design, implementation and testing of extended and mixed precision BLAS*, ACM Transactions on Mathematical Software **28** (2002), no. 2, 152–205.

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

[20] Daichi Mukunoki and Daisuke Takahashi, *Performance comparison of double, triple and quadruple precision real and complex blas subroutines on gpus*, Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way? (Singapore, Singapore), ATIP '12, A*STAR Computational Resource Centre, 2012, pp. 37:1–37:3.

[21] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres, *Handbook of floating-point arithmetic, 2nd edition*, Birkhäuser Boston, 2018, ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9.

[22] M. Nakata, *A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: SDPA-GMP,-QD and -DD.*, 2010 IEEE International Symposium on Computer-Aided Control System Design, IEEE, 2010, pp. 29–34.

[23] Y. Nievergelt, *Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit*, ACM Transactions on Mathematical Software **29** (2003), no. 1, 27–48.

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

[25] V. Popescu, *Towards fast and certified multiple-precision libraries*, Ph.D. thesis, Université de Lyon, 2017, Available at https://hal.archives-ouvertes.fr/tel-01534090.

[26] D. M. Priest, *Algorithms for arbitrary precision floating point arithmetic*, 10th IEEE Symposium on Computer Arithmetic (ARITH-10), June 1991, pp. 132–143.

[27] D. M. Priest, *On properties of floating-point arithmetics: Numerical stability and the cost of accurate computations*, Ph.D. thesis, University of California at Berkeley, 1992.

[28] Siegfried M. Rump, T. Ogita, and S. Oishi, *Accurate floating-point summation part I: Faithful rounding*, SIAM Journal on Scientific Computing **31** (2008), no. 1, 189–224.

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

[30] S. Yamada, T. Ina, N. Sasa, Y. Idomura, M. Machida, and T. Imamura, *Quadruple-precision blas using bailey's arithmetic with fma instruction: its performance and applications*, 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2017, pp. 1418–1425.

Nicolas 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 Mathématiciennnes et Mathématiciens (TFJM^2) 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 SCAN'97 (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, Birkhauser, 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 developement of computational fluid dynamics software, and he is Research Engineer at École Normale Supérieure de Lyon, France. His research interests are in Rigorous Computing.