Accurate calculation of Euclidean Norms using Double-word arithmetic

Vincent Lefèvre, Nicolas Louvet, Jean-Michel Muller, Joris Picot, Laurence Rideau

Accurate Calculation of Euclidean Norms using Double-Word Arithmetic

Vincent Lefèvre Nicolas Louvet Jean-Michel Muller Joris Picot
Laurence Rideau

October 7, 2022

Keywords: Floating-point arithmetic; Euclidean norms; Double-word arithmetic; Double-double arithmetic; Overflow; Underflow; Square-root; Formalization; Proof assistant; Coq.

Abstract

We consider the computation of the Euclidean (or L2) norm of an n-dimensional vector in floating-point arithmetic. We review the classical solutions used to avoid spurious overflow or underflow and/or to obtain very accurate results. We modify a recently published algorithm (that uses double-word arithmetic) to allow for a very accurate solution, free of spurious overflows and underflows. To that purpose, we use a double-word square-root algorithm of which we provide a tight error analysis. The returned L2 norm will be within very slightly more than 0.5 ulp from the exact result, which means that we will almost always provide correct rounding.

1 Introduction

1.1 Computation of Euclidean norms

We consider the computation of Euclidean norms in binary floating-point arithmetic. The Euclidean (or L2) norm of a vector $(a_0, a_1, a_2, \ldots, a_{n-1}) \in \mathbb{R}^n$ is the number

$$N = \sqrt{\sum_{i=0}^{n-1} a_i^2}. \tag{1}$$

The particular case $n = 2$ (the so-called "hypotenuse" function) has been studied in excellent references [2, 13]. In this paper, we assume that n is larger (more precisely, our algorithms do work

in the cases $n = 1$ or 2, but it is for larger values that good performance is aimed at). Computing Euclidean norms is important in many scientific and engineering applications. A good implementation of the Euclidean norm must be fast and accurate. It must also avoid spurious underflows and overflows. A spurious underflow or overflow is an underflow or overflow that occurs during an intermediate step, resulting in an inaccurate, infinite or NaN returned result, whereas the exact result is well within the domain of normal floating-point numbers.

To illustrate how spurious underflows and overflows can jeopardize the computation of a Euclidean norm, consider the following examples, assuming IEEE 754 binary64/double-precision arithmetic and $n = 3$, with the default round-to-nearest, ties-to-even, rounding function, and suppose that we implement Formula (1) naively by first summing the squares serially and then taking the square-root.

- With $a_0 = 1.5 \times 2^{511}$, $a_1 = 0$, and $a_2 = 2^{512}$, we will obtain an infinite result (because the computation of a_2^2 overflows), whereas the exact result is 5×2^{510}, which is much smaller than the overflow threshold;
- with $a_0 = a_1 = a_2 = (45/64) \times 2^{-537}$, the computed result is 0, whereas the exact result is around 1.2178×2^{-537}, which is much above the underflow threshold.

Note that from an *accuracy* point-of-view, spurious underflow is a problem only if *all* terms a_i are tiny (otherwise, the errors due to underflows that occur when squaring the "tiny" terms vanish in front of the squares of the "big" terms). However, spurious underflow can be very harmful from a *performance* point-of-view on a system on which subnormal numbers are handled in software, through a trapping underflow mechanism.

There are no catastrophic cancellations when computing a Euclidean norm: all added terms are nonnegative. Hence, even a naive use of Formula (1) will be rather accurate when no underflow or overflow occurs. More precisely, Jeannerod and Rump recently showed [30] that the relative error is bounded by

$$\left(\frac{n}{2} + 1\right) \cdot u, \tag{2}$$

where u is the "rounding unit" (see definition below). The bound (2) is *very sharp*: for instance, in binary64/double-precision floating-point arithmetic, if $n = 7$, with $a_0 = 1125899918705907/2^{50}$, $a_1 = 6893812215223557/2^{66}$ and $a_2 = a_3 = \cdots = a_6 = 1592262918131443/2^{77}$, the naive use of Formula (1) leads to a relative error $4.499999839236531787 \cdots u$, which is extremely close to the bound $\left(\frac{n}{2} + 1\right) u = 4.5u$. However, the probability of observing a similar case by chance is almost zero: such examples must be *built*, and in practice, the typical error rather grows like $\sqrt{n} \cdot u$ [25].

However, we can try to take advantage of the absence of catastrophic cancellations to always obtain results *very near* the exact result. Our goal is to obtain a result that is almost always *correctly rounded* (or always correctly rounded, using Ziv's rounding test, see Remark 4.6 below). Correct

rounding enhances the reproducibility of the calculations, which is becoming an important issue: as pointed out by Demmel and NGuyen [16], in the context of ExaScale computing, reproducibility considerably helps debugging and validating numerical programs, it is also sometimes needed for legal reasons when different sides need to agree on the results of some computation. More generally, our contribution could be used in a "highly accurate" arithmetic toolbox.

Note that the Euclidean norm of the *complex vector* $(b_0, b_1, b_2, \ldots, b_{n-1}) \in \mathbb{C}^n$ is equal to the norm of the *real vector* $(a_0, a_1, a_2, \ldots, a_{2n-1}) \in \mathbb{R}^{2n}$ such that $a_{2j} = \Re(b_j)$ and $a_{2j+1} = \Im(b_j)$. Hence, all properties, algorithms and bounds presented in this paper for real vectors are easily generalizable to complex vectors.

Due to the importance of the topic, several solutions have been suggested and analyzed until recently for computing norms accurately and/or without spurious underflows and overflows [4, 24, 18, 19, 1]. We will present them in Section 2.1. Before that, let us present some definitions and properties related to floating-point arithmetic, that will be useful in the sequel of this paper.

1.2 The underlying FP arithmetic

In the following, we assume a radix-2, precision-p, floating-point (FP) arithmetic (where $p \geq 5$), with extremal exponents $e_{\min} < 0$ and $e_{\max} > 0$. We also assume $e_{\min} = 1 - e_{\max}$ (which is a requirement of the IEEE 754-2019 Standard for FP arithmetic [27]). In such a system, a finite floating-point number (FPN) is a number of the form $M \cdot 2^{e-p+1}$, with $M \in \mathbb{Z}$, $|M| \leq 2^p - 1$, and $e \in \mathbb{Z}$, $e_{\min} \leq e \leq e_{\max}$ [38]. A FPN x is *normal* if $|x| \geq 2^{e_{\min}}$ or $x = 0$, and *subnormal* otherwise. The largest representable FP number is $\Omega = 2^{e_{\max}} \cdot (2 - 2^{1-p})$, the smallest positive normal FP number, also called "underflow threshold", is $2^{e_{\min}}$. In the following, we will say that an arithmetic operation *underflows* if its result is both subnormal *and* inexact. This choice may seem strange, but we want to avoid underflows because of accuracy concerns: when the result is exact, there is no need to worry about accuracy (this is why the underflow flag is not raised in such a case under the default exception handling for underflow of the IEEE 754-2019 Standard). The smallest positive FP number is $\alpha = 2^{e_{\min}-p+1}$.

The notation $\mathrm{RN}(t)$ stands for t rounded to the nearest FP number. We do not assume a particular tie-breaking rule in our proofs,[1] and we use the default ties-to-even rule in our examples. For instance $\mathrm{RN}(c \cdot d)$ is the result of the FP multiplication $\mathtt{c} * \mathtt{d}$, assuming round-to-nearest rounding mode (which is the default in IEEE 754-2019). The number $\mathrm{ulp}(x)$, for $x \neq 0$ is

$$\mathrm{ulp}(x) = 2^{\max\{\lfloor \log_2 |x| \rfloor, e_{\min}\} - p + 1},$$

and $u = 2^{-p} = \frac{1}{2}\mathrm{ulp}(1)$ denotes the roundoff error unit. The constraint $p \geq 5$ implies $u \leq 1/32$, which will serve many times in our proofs. The relative error due to rounding to nearest a real

[1] Our proofs, however, are based on the assumptions that $\mathrm{RN}(-x) = -\mathrm{RN}(x)$, and that if $k \in \mathbb{Z}$ and if both x and $2^k x$ are in the normal domain, then $\mathrm{RN}(2^k x) = 2^k \mathrm{RN}(x)$.

number x such that $|x| \in [2^{e_{\min}}, \Omega]$, namely $|(\mathrm{RN}(x) - x)/x|$, is bounded by $u/(1+u)$ [30]. When tightness is not necessary, we will use the simpler yet very slightly looser bound u. We will denote $\mathrm{succ}(t)$ the floating-point successor of t, and η the number $2^{(e_{\min}+p)/2}$ (beware: it is a FP number only when $e_{\min} + p$ is even). Barring overflow, the square of a FPN $\geq \eta$ can be expressed exactly as the sum of two FPNs [6]. We also have the following property (see for instance [38]):

Property 1.1. *If a FP number $\hat{t}$ approximates a real number t with relative error ϵ, then $\hat{t}$ is within $(\epsilon/u) \cdot \mathrm{ulp}(t)$ from t.*

The FP numbers between 2^k and 2^{k+1} are multiples of $2^{k+1}u$: for instance, the FP numbers between 1 and 2 are 1, $1+2u$, $1+4u$, $1+6u$, …, $2-2u$, 2. We call *binade* an interval of the form $[2^k, 2^{k+1}),\ k \in \mathbb{Z}$. Table 1 reminds the values of p, $e_{\min}$ and $e_{\max}$ for the binary interchange formats of IEEE 754-2019 up to 128 bits [27] and the more recent bfloat16 format [21], and Table 2 summarizes our notation for the important FP parameters.

Table 1: Main parameters of the binary interchange formats of size up to 128 bits specified by the 754-2019 standard [27], and the bfloat16 format [21].

name former name	binary16	binary32 (basic) single precision	binary64 (basic) double precision	binary128 (basic)	bfloat16
p	11	24	53	113	8
$e_{\max}$	$+15$	$+127$	$+1023$	$+16383$	$+127$
$e_{\min}$	-14	-126	-1022	-16382	-126

Table 2: Notation for the important FP parameters.

notation	numerical value	explanation
Ω	$2^{e_{\max}} \cdot (2 - 2^{-p+1})$	largest finite FPN
α	$2^{e_{\min}-p+1}$	smallest positive FPN
$2^{e_{\min}}$	$2^{e_{\min}}$	smallest positive normal FPN (underflow threshold)
$\mathrm{succ}(t)$	$\mathrm{succ}(t)$	floating-point successor of t
η	$2^{(e_{\min}+p)/2}$	the square of a FPN $\geq \eta$ is the sum of two FPNs
u	2^{-p}	roundoff error unit
$\mathrm{ulp}(x)$ (for $x \in \mathbb{R}, x \neq 0$)	$2^{\max\{\lfloor \log_2 \lvert x \rvert \rfloor, e_{\min}\}-p+1}$	unit in the last place

A computed result is *faithfully rounded* if $i)$ it is equal to the exact result if this one is a FP number, and $ii)$ it is one of the two FP numbers that surround the exact result otherwise. This implies (barring overflow) that the returned result is within one ulp of the exact result from the exact result.

Figure 1 illustrates the notions presented in this section.

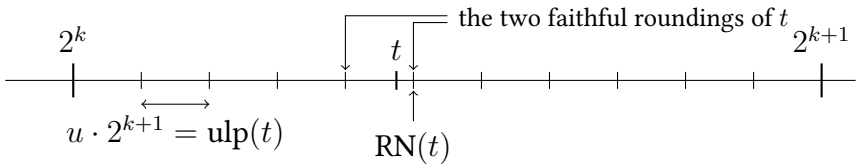

Figure 1: The floating-point numbers between 2^k and 2^{k+1} (assuming $e_{\min} \leq k < e_{\max}$).

1.3 Double-word and pair arithmetics

Evaluating norms with an accuracy significantly better than that of the naive algorithm may require representing intermediate results with a precision higher than the working FP precision. This can be done by representing these intermediate results by a pair of FP numbers. For instance, Graillat et al. [18] recently published an algorithm that computes *faithfully rounded* norms. To achieve that goal, they use double-word (or "double-double") arithmetic in their intermediate calculations. Lange and Rump [43] recently defined a "pair arithmetic" (which is a somehow "relaxed" version of double-word arithmetic), and showed how it can be used, under some conditions, to obtain faithfully rounded results in FP arithmetic. The algorithms used to perform operations with these arithmetics are usually based on the three basic "building blocks" presented in Section 1.3.1: Fast2Sum, 2Sum, and Fast2Mult. It is possible that new operations recently introduced in the IEEE 754 Standard for FP arithmetic [27] and briefly presented in Section 1.3.2 replace these building blocks in a near future. Double-word arithmetic (and more generally, pair and multiple-word arithmetics) is slowly yet steadily gaining importance among numerical methods [32, 43, 35, 17]: this makes a careful study of its error useful.

1.3.1 The basic building blocks: Fast2Sum, 2Sum, and Fast2Mult

Algorithm 1 – Fast2Sum(a, b). The Fast2Sum algorithm [15]. It takes 3 FP operations.

$s \leftarrow \mathrm{RN}(a + b)$
$z \leftarrow \mathrm{RN}(s - a)$
$t \leftarrow \mathrm{RN}(b - z)$

If $|a| \geq |b|$, unless overflow occurs, the two FP numbers s and t returned by Algorithm 1 satisfy $s+t = a+b$. Since s is the result of the conventional floating-point addition of a and b, t is the error of that addition. Also, if the first operation does not overflow, the other operations cannot overflow [7]. For that algorithm, underflow is harmless (this is an immediate consequence of Lemma 1.3).

Algorithm 2 – 2Sum(a, b). The 2Sum algorithm [37, 34]. It takes 6 FP operations.

$s \leftarrow \mathrm{RN}(a + b)$
$a' \leftarrow \mathrm{RN}(s - b)$
$b' \leftarrow \mathrm{RN}(s - a')$
$\delta_a \leftarrow \mathrm{RN}(a - a')$
$\delta_b \leftarrow \mathrm{RN}(b - b')$
$t \leftarrow \mathrm{RN}(\delta_a + \delta_b)$

Unless overflow occurs, the two FP numbers s and t returned by Algorithm 2 satisfy $s+t = a+b$: this algorithm returns the same result as Algorithm 1 without any condition on a and b. On the other hand, it is slightly less overflow-proof: If the first operation does not overflow and if $|a| < \Omega$, then the other operations cannot overflow [7]. Underflow is harmless.

Algorithm 3 – Fast2Mult(a, b)**.** The Fast2Mult algorithm (see for instance [33, 40, 38]). It requires the availability of a fused multiply-add (FMA) instruction for computing $\mathrm{RN}(ab - \pi_h)$.

$\pi_h \leftarrow \mathrm{RN}(a \cdot b)$
$\pi_\ell \leftarrow \mathrm{RN}(a \cdot b - \pi_h)$

In this paper, Algorithm Fast2Mult is used for expressing the square of a FP number as a double-word number. One should keep in mind that, barring overflow, the condition for that algorithm to guarantee that $\pi_h + \pi_\ell = a \cdot b$ is stronger than just requiring the absence of underflow in the first multiplication. Several slightly different conditions appear in the literature (see [6] for a necessary

and sufficient condition). One can show (see for instance [8]) that if $2^{e_{\min}+p} \leq |a \cdot b|$, then $\pi_h + \pi_\ell = a \cdot b$. In the case of the computation of $a \cdot a$, this condition becomes

$$|a| \geq \eta = 2^{(e_{\min}+p)/2}. \tag{3}$$

Algorithm Fast2Mult requires the availability of an FMA instruction. Without an FMA instruction, the calculation of (π_h, π_ℓ) remains possible, but at a significantly higher cost (17 floating-point operations instead of 2 [15]).

1.3.2 An alternative: the new "augmented" arithmetic operations

The latest release of the IEEE Standard for Floating-Point Arithmetic, published in 2019 [27], specifies new "augmented" operations, called *augmentedAddition*, *augmentedSubtraction*, and *augmentedMultiplication* (history and motivation are presented in [41]). These operations use a new "rounding direction", round-to-nearest *ties-to-zero*, denoted RN_0 in this paper, that satisfies [27]:

> $\mathrm{RN}_0(t)$ (where t is a real number) is the FP number nearest t. If the two nearest FP numbers bracketing t are equally near, $\mathrm{RN}_0(t)$ is the one with smaller magnitude. If $|t| > \Omega + 2^{e_{\max}-p}$, then $\mathrm{RN}_0(t) = \pm\infty$, with the same sign as t.

The augmented operations are defined as follows [27, 41]:

- *augmentedAddition*(x, y) delivers (a_0, b_0) such that $a_0 = \mathrm{RN}_0(x + y)$ and, when $a_0 \notin \{\pm\infty, \mathrm{NaN}\}$, $b_0 = (x + y) - a_0$. When $b_0 = 0$, it is required to have the same sign as a_0;
- *augmentedSubtraction*(x, y) is augmentedAddition$(x, -y)$;
- *augmentedMultiplication*(x, y) delivers (a_0, b_0) such that $a_0 = \mathrm{RN}_0(x \cdot y)$ and, where $a_0 \notin \{\pm\infty, \mathrm{NaN}\}$, $b_0 = \mathrm{RN}_0((x \cdot y) - a_0)$. When $(x \cdot y) - a_0 = 0$, the floating-point number b_0 (equal to 0) is required to have the same sign as a_0.

As we are writing these lines, no fast hardware implementation of these operations is offered on widely available platforms. When this happens, in the algorithms presented in this paper, it can be worth replacing 2Sum and Fast2Sum by augmentedAddition, and replacing Fast2Mult by augmentedMultiplication.

1.3.3 Double-word arithmetic

We define a double-word number as follows

Definition 1.2. *A* double-word *(DW) number x is the unevaluated sum $x_h + x_\ell$ of two floating-point numbers x_h and x_ℓ such that $x_h = \mathrm{RN}(x)$.*

Double-word arithmetic goes back to the seminal work of Dekker [15]. Algorithms for manipulating DW numbers have been published and analyzed by Li et al. [36], Hida, Li and Bailey [23, 22], Joldes et al. [32], Muller and Rideau [39]. Let us now give a two classical DW algorithms. Some new results on DW arithmetic necessary for this study are given in Section 3.

Let us first consider the addition of a DW number and a FP number. Consider Algorithm 4 below. It was implemented in the QD library [23].

Algorithm 4 – DWPlusFP(x_h, x_ℓ, y). Algorithm for computing $(x_h, x_\ell) + y$ in binary, precision-p, floating-point arithmetic, implemented in the QD library. The number $x = (x_h, x_\ell)$ is a DW number (i.e., it satisfies Definition 1.2).

1: $(s_h, s_\ell) \leftarrow \text{2Sum}(x_h, y)$
2: $v \leftarrow \text{RN}(x_\ell + s_\ell)$
3: $(z_h, z_\ell) \leftarrow \text{Fast2Sum}(s_h, v)$
4: **return** (z_h, z_ℓ)

That algorithm was analyzed by Joldes et al. [32]. They found that its relative error

$$|((z_h + z_\ell) - (x + y))/\,(x + y)|$$

is bounded by

$$2 \cdot u^2/(1 - 2u) = 2u^2 + 4u^3 + 8u^4 + \cdots \tag{4}$$

They also showed that the bound (4) is asymptotically optimal, by exhibiting "generic" (i.e., parameterized by the precision p) input values for which the ratio between the attained relative error and the bound goes to 1 as p goes to infinity.

Now, let us turn to the addition of two DW numbers. Algorithm 5 below was first given by Dekker [15], under the name of *add2*. It was implemented by Hida, Li, and Bailey in the QD library [23] under the name of "sloppy addition". The reason for that name is that if the input operands have different signs, the relative error can be arbitrarily large. We will *not* use that algorithm, but since it is the algorithm used by Graillat et al. to perform their summations [18], we briefly present it and some of its properties for the sake of completeness and for helping to compare our solutions.

Algorithm 5 – SloppyDWPlusDW$(x_h, x_\ell, y_h, y_\ell)$. "Sloppy" calculation of $(x_h, x_\ell) + (y_h, y_\ell)$ in binary, precision-p, floating-point arithmetic. It takes 11 FP operations.

1: $(s_h, s_\ell) \leftarrow \text{2Sum}(x_h, y_h)$
2: $v \leftarrow \text{RN}(x_\ell + y_\ell)$
3: $w \leftarrow \text{RN}(s_\ell + v)$
4: $(z_h, z_\ell) \leftarrow \text{Fast2Sum}(s_h, w)$
5: **return** (z_h, z_ℓ)

If the inputs operands x_h and y_h have the same sign (which is of course the case when summing squares), the relative error of Algorithm 5 is bounded by $3u^2$ [18]. This bound is asymptotically optimal: consider $x_h = 1 + 2u$, $x_\ell = -u + u^2$, $y_h = 9u$, and $y_\ell = -6u^2 - 8u^3$, for which the double-word number returned by Algorithm 5 is equal to $1 + 10u - 8u^2$ and the exact sum is equal to $1 + 10u - 5u^2 - 8u^3$, resulting in a relative error $u^2 \cdot (3 - 8u)/(1 + 10u - 5u^2 - 8u^3) = 3u^2 - 38u^3 + \mathcal{O}(u^4)$.

1.3.4 Lange and Rump's pair arithmetic

Lange and Rump [43] recently defined a "pair arithmetic" (which is a somehow "relaxed" version of double-word arithmetic), and showed how it can be used, under some conditions, to obtain faithfully rounded results in floating-point arithmetic.

Rewritten with our notation, the pair algorithms used by Lange and Rump [43] for addition and square-root are the following.

Algorithm 6 – Pair_addition$(x_h, x_\ell, y_h, y_\ell)$. Lange and Rump's calculation of $(x_h, x_\ell) + (y_h, y_\ell)$ in binary, precision-p, floating-point arithmetic. It is Algorithm 5 without the last "renormalization". It takes 8 FP operations.

1: $(s_h, s_\ell) \leftarrow \text{2Sum}(x_h, y_h)$
2: $v \leftarrow \text{RN}(x_\ell + y_\ell)$
3: $w \leftarrow \text{RN}(s_\ell + v)$
4: **return** (s_h, w)

Algorithm 7 – Pair_sqrt(x_h, x_ℓ). Lange and Rump's calculation of the square-root of (x_h, x_ℓ) in binary, precision-p, floating-point arithmetic. It is Algorithm 8 without the last "renormalization". It takes 5 FP operations (counting the square-root as one).

1: $s_h \leftarrow \text{RN}(\sqrt{x_h})$
2: $\rho_1 \leftarrow \text{RN}(x_h - s_h^2)$ (with an FMA instruction)
3: $\rho_2 \leftarrow \text{RN}(x_\ell + \rho_1)$
4: $s_\ell \leftarrow \text{RN}(\rho_2/(2 \cdot s_h))$
5: **return** (s_h, s_ℓ)

These algorithms are similar to the DW algorithms presented in this paper, with the difference that they avoid the last "renormalizing" Fast2Sum operation. This makes them significantly faster, but this may sometimes make them less accurate, especially when cancellations occur. When adding squares, however, there are no cancellations: this makes Lange and Rump's pair arithmetic a very good candidate.

1.4 Some results useful later on

The following lemma is frequently used to show that some calculations remain valid even when operands are below the underflow threshold (the proof is straightforward).

Lemma 1.3 (Hauser Lemma [20]). *If x and y are floating-point numbers, and if the number* $\mathrm{RN}(x+y)$ *is subnormal, then $x+y$ is a floating-point number, which implies* $\mathrm{RN}(x+y) = x+y$.

For bounding the error committed during the evaluation of a sum of squares, we will use the following lemma, which is a direct consequence of Lemma 2.1 in [42], due to Lange and Rump.

Lemma 1.4 (Lange and Rump [42]). *Let $\mathbb{F}$ be an arbitrary subset of $\mathbb{R}$ and let $\tilde{+}$ be an operation in $\mathbb{F}$ with the only assumption that $\forall a, b \in \mathbb{F}, |(a\tilde{+}b) - (a+b)| \leq \min\{|a|, |b|\}$. Let $x_1, x_2, \ldots, x_n$ be elements of $\mathbb{F}$ and define numbers s_i and ϵ_i as follows:*

$$\begin{array}{rcl} s_1 & = & x_1, \\ s_i & = & x_i \tilde{+} s_{i-1} = (x_i + s_{i-1})(1+\epsilon_i) \ \text{ for } i = 2, \ldots, n. \end{array}$$

We have $|s_n - \sum_{i=1}^{n} x_i| \leq \sum_{i=2}^{n} |\epsilon_i| \cdot \sum_{i=1}^{n} |x_i|$.

For computing square-roots in double-word arithmetic, we will need the following result, due to Boldo and Daumas [6]. This is Theorem 5 of [6], restricted to binary arithmetic and rewritten with our notation.

Lemma 1.5 (Exact representation of the square-root remainder). *In binary, precision-p, FP arithmetic, let $s = \mathrm{RN}(\sqrt{x})$, where x is a FP number. The correcting term $x - s^2$ is a FPN if and only if there exists a pair of integers (m, e) (with $|m| \leq 2^p - 1$) such that $s = m \cdot 2^{e-p+1}$ and $2e \geq e_{\min} + p - 1$.*

1.5 Aim and organization of this paper

Ideally, one would like to always return *correctly rounded* results (i.e., the computed result is the floating-point number nearest to the exact result, which implies that the error is less than or equal to 0.5 ulp of the exact result). This seems difficult to guarantee without significantly slowing down the calculation. However, we show in this paper that a modification of Graillat et al.'s algorithm [18] can be used to always obtain a maximum error *very slightly* above 0.5 ulp. This means that we *almost always* obtain the correctly rounded result, except in rare cases when the exact norm is very near the middle of two consecutive FP numbers. Furthermore, if needed, we can detect when the result returned by our algorithm may not be correctly rounded (see Remark 4.6).

The sequel of the paper is organized as follows. Section 2 presents the algorithms one can find in the literature. More precisely, in Subsection 2.1 we quickly review the classical solutions suggested for avoiding spurious overflows and underflows, Subsection 2.2 briefly presents the use by

Graillat et al. of double-word arithmetic for obtaining more accurate results, and in Subsection 2.3 we consider applying a recent result by Lange and Rump [43] to obtain faithfully rounded norms in pair arithmetic. In Section 3 we give some new results on DW arithmetic that will be helpful for our study. In particular, we give a new bound (that takes into account the fact that we manipulate positive numbers) for an existing addition algorithm, and we present and analyze an algorithm that computes the square-root of a DW number. Since the proof of that square-root algorithm is long and rather complex, and since the error bound of our Euclidean norm algorithm derives from the error bound of the square-root algorithm, to give more confidence on our result, we have formally proven the square-root algorithm, using the Coq proof assistant (see for instance [9]). This part of the paper continues the work undertaken by two of us on the formal proof of double-word algorithms [39]. Section 4 presents our algorithms for computing Euclidean norms. We first assume in Subsection 4.1 that no underflow or overflow occurs, then we deal with the general case in Subsection 4.2.

Our solution builds on Graillat et al.'s solution [18], with the following differences:

- we introduce a more accurate algorithm for summing the squares of the terms a_i in DW arithmetic;
- once we have obtained an approximation to the sum of squares as a DW number, we directly take its square-root using a specific DW to FP square-root algorithm, whereas Graillat et al. "convert" the sum of squares to floating-point (by just retaining the most significant part) and take its square-root using the conventional FP square-root;
- we use different comparison constants for preventing underflows and overflows.

2 Conventional solutions for computing Euclidean norms

2.1 Avoiding spurious overflows/underflows

Several solutions have been suggested for dealing with spurious underflows and overflows when computing Euclidean norms. A first solution [26] would be to use the exception-handling mechanism provided by the IEEE 754 Standard for FP arithmetic: one could first use the naive method (i.e., straightforward implementation of (1)), check if an underflow or overflow exception occurred, and use a more sophisticated method only in that case. This approach is unlikely to allow good performance on modern highly pipelined processors. All other approaches consist in *scaling* the terms a_i, i.e., we multiply or divide them by one (or several) constant(s) such that computing sums of squares of the scaled values is overflow-free, and that underflow is either impossible or harmless (a good presentation, along with comparisons of existing Fortran codes can be found in [19]). A

straightforward choice is to scale all values by the factor $\max|a_i|$, i.e., to evaluate the norm as

$$\max|a_i| \times \sqrt{\sum_{k=0}^{n-1} (a_k/\max|a_i|)^2}.$$

This approach has several drawbacks:

- it requires two passes over the data (finding the maximum of the $|a_i|$ takes time and no computation can start before that max is found);
- it requires divisions, and FP divisions are in general significantly slower than FP additions and multiplications;
- multiplying and dividing by $\max|a_i|$ are, in general, nonexact operations, which leads to a slightly larger final error than the error of directly using (1) when no underflow/overflow occurs.

An already better approach (at least in terms of accuracy, latency may be another matter) consists in choosing a scale factor equal to a power of 2 close to $\max|a_i|$, obtained for instance by the means of the **scaleB** and **logB** functions[2] specified by the IEEE-754 Standard [27] (when an efficient implementation of these functions is available. If this is not the case, a possible workaround is suggested in [29, Theorem IV.2]).

Higham [24, Pages 500 and 507] attributes to Hammarling a smart algorithm that consists in *dynamically* scaling the data. It was implemented in the LAPACK [19] package released by `netlib`[3]. We start from $s_0 = 1$ and $t_0 = |x_0|$. At step i of the algorithm, we have already computed $s_{i-1} = \sum_{k=0}^{i-1}(x_k/t_{i-1})^2$, where s_{i-1} is the current scaled sum and t_{i-1} is the current value of the scale factor. If $|x_i| \leq t_{i-1}$ then $s_i = s_{i-1} + (x_i/t_{i-1})^2$ and the scale factor does not change: $t_i = t_{i-1}$. If $|x_i| > t_{i-1}$ then we need to update the scale factor. We compute

$$s_i = 1 + s_{i-1} \cdot (t_{i-1}/x_i)^2,$$

and we replace the scale factor by $|x_i|$: $t_i = |x_i|$. After this, one easily checks that $s_i = \sum_{k=0}^{i}(x_k/t_i)^2$. The final result is $t_{n-1}\sqrt{s_{n-1}}$. With this method, a single pass over the data suffices. However, the number of scale factor updates may be large: up to $n-1$ updates if the $|x_i|$'s are in increasing order (although its average value is around $\log(n)$), which may result in delays and additional rounding

[2] **scaleB**(x, k) returns (in a binary format, which is the case considered in this paper) $x \cdot 2^k$ (where x is a FP number and k is an integer), and **logB**(x) returns (in a binary format) $\lfloor \log_2 |x| \rfloor$ (where x is a FP number). In the C programming language, these functions are called **scalbn** and **logb** (resp. **scalbnf** and **logbf**) for binary64/double precision (resp. binary32/single precision) operands.

[3] `www.netlib.org`

errors due to (in general, nonexact) multiplications and divisions. An improvement in terms of accuracy consists in choosing, when $|x_i| > t_{i-1}$, a value t_i equal to a power of two close to (and preferably above) $|x_i|$, and then taking $s_i = (x_i/t_i)^2 + s_{i-1} \cdot (t_{i-1}/t_i)^2$.

With the methods examined so far, a scaling is applied even when not needed.

Blue [4] takes a decisive step by suggesting to split the input numbers into 3 classes (that we will call TINY, MED, and BIG), depending on their order of magnitude:

- numbers of the MED class can be squared, and their squares can be accumulated, without underflows or overflows. A FP number a_i is in the MED class if[4] $a_i = 0$ or $\mathrm{minmed} \leq |a_i| \leq \mathrm{maxmed}$, where the choice of minmed and maxmed depends on the parameters (p, $e_{\min}$ and $e_{\max}$) of the FP arithmetic, and on the largest value of n, say $n_{\max}$, for which a correct behavior is to be guaranteed. We compute $S_{\mathrm{med}} = \sum_{a_i \in \mathrm{MED}} a_i^2$;

- numbers of the BIG class must be "scaled down" to make sure that we can accumulate their squares without overflow. A FP number a_i is in the BIG class if $\mathrm{maxmed} < |a_i|$. All numbers of the BIG class are multiplied by *the same* predefined constant t_{big}, chosen equal to a power of 2 (to make the multiplication errorless), and such that for $a_i \in \mathrm{BIG}$, $t_{\mathrm{big}} \cdot a_i \in \mathrm{MED}$. We compute $S_{\mathrm{big}} = \sum_{a_i \in \mathrm{BIG}} (t_{\mathrm{big}} \cdot a_i)^2$ (usual presentation of the method is with divisions by constants; of course, when actually implementing it, multiplication is preferable for performance reasons);

- numbers of the TINY class must be "scaled up" to make sure that we can compute their squares without underflow: each square must be larger than the subnormal threshold[5] $2^{e_{\min}}$. A FP number a_i is in the TINY class if $|a_i| < \mathrm{minmed}$ and $a_i \neq 0$. All numbers of the TINY class are multiplied by *the same* constant t_{tiny}, chosen equal to a power of 2, and such that for $a_i \in \mathrm{TINY}$, $t_{\mathrm{tiny}} \cdot a_i \in \mathrm{MED}$. We compute $S_{\mathrm{tiny}} = \sum_{a_i \in \mathrm{TINY}} (t_{\mathrm{tiny}} \cdot a_i)^2$.

[4] As a matter of fact, 0 could be as well in the TINY class instead of the MED class if this simplifies the programming. In any case, accumulating 0 in one part or another one will of course not change the result. However, depending on the underlying computer architecture, the choice may have a significant impact on performance, due to branch prediction issues.

[5] We will need a stronger condition when we represent numbers in double-word arithmetic.

Let us summarize the various constraints that the parameters minmed, maxmed, t_{big}, and t_{tiny} must satisfy:

$$\mathrm{minmed}^2 \geq 2^{e_{\min}}, \tag{5a}$$

$$n_{\max} \cdot \mathrm{maxmed}^2 \cdot (1+\kappa) < \Omega + (1/2)\mathrm{ulp}(\Omega) = 2^{e_{\max}+1} - 2^{e_{\max}-p}, \tag{5b}$$

$$\mathrm{maxmed} \cdot t_{\mathrm{big}} \geq \mathrm{minmed}, \tag{5c}$$

$$\Omega \cdot t_{\mathrm{big}} \leq \mathrm{maxmed}, \tag{5d}$$

$$\mathrm{minmed} \cdot t_{\mathrm{tiny}} \leq \mathrm{maxmed}, \tag{5e}$$

$$\alpha \cdot t_{\mathrm{tiny}} \geq \mathrm{minmed}, \tag{5f}$$

where Ω and α are defined in Table 2, and κ is a bound on the relative error of the algorithm used for computing the sum of squares in MED. Assuming that maxmed and $n_{\max}$ are powers of 2, Eq. (5b) can be replaced by $n_{\max} \cdot \mathrm{maxmed}^2 < 2^{e_{\max}+1} - 2^{e_{\max}-p}$. Later on, when we use double-word arithmetic, (5a) will need to be replaced by the stronger condition $\mathrm{minmed} \geq \eta$.

A recent, efficient implementation of Blue's algorithm is given by Anderson [1]. In Blue's original algorithm [4], the three terms S_{tiny}, S_{med}, and S_{big} are all computed, in three accumulators. However, if BIG is nonempty, provided that the ratio $\mathrm{maxmed}/\mathrm{minmed}$ is large enough, the value of S_{tiny} has negligible influence on the final result. Graillat et al.'s algorithm [18] and the variant of Blue's algorithm presented by Hanson and Hopkins in [19] take this into account and use two accumulators only: as soon as an element of BIG is met, we no longer need to accumulate elements of TINY.

Figure 2 illustrates this splitting of the FP numbers into three classes.

The norm $N = \sqrt{\sum_{i=0}^{n-1} a_i^2}$ is equal to $\sqrt{S_{\mathrm{tiny}}/t_{\mathrm{tiny}}^2 + S_{\mathrm{med}} + S_{\mathrm{big}}/t_{\mathrm{big}}^2}$, but obviously that formula cannot be employed since $S_{\mathrm{big}}/t_{\mathrm{big}}^2$—and, more rarely, the sum—could overflow, and $S_{\mathrm{tiny}}/t_{\mathrm{tiny}}^2$ could underflow. Blue suggests obtaining N as follows:

1. if BIG and TINY are empty (i.e., if $S_{\mathrm{big}} = S_{\mathrm{tiny}} = 0$), then $\sqrt{S_{\mathrm{med}}}$ is returned;

2. if BIG is nonempty, then if $\sqrt{S_{\mathrm{big}}}$ is larger than the precomputed constant $\Omega \cdot t_{\mathrm{big}}$, $+\infty$ is returned[6], otherwise we define

$$\begin{aligned} y_{\min} &= \min\left\{\sqrt{S_{\mathrm{med}}}, \tfrac{1}{t_{\mathrm{big}}}\sqrt{S_{\mathrm{big}}}\right\}, \\ y_{\max} &= \max\left\{\sqrt{S_{\mathrm{med}}}, \tfrac{1}{t_{\mathrm{big}}}\sqrt{S_{\mathrm{big}}}\right\}, \end{aligned}$$

   and we go to step 4;

[6] One can check that with an IEEE-754 compliant system, that test is not needed.

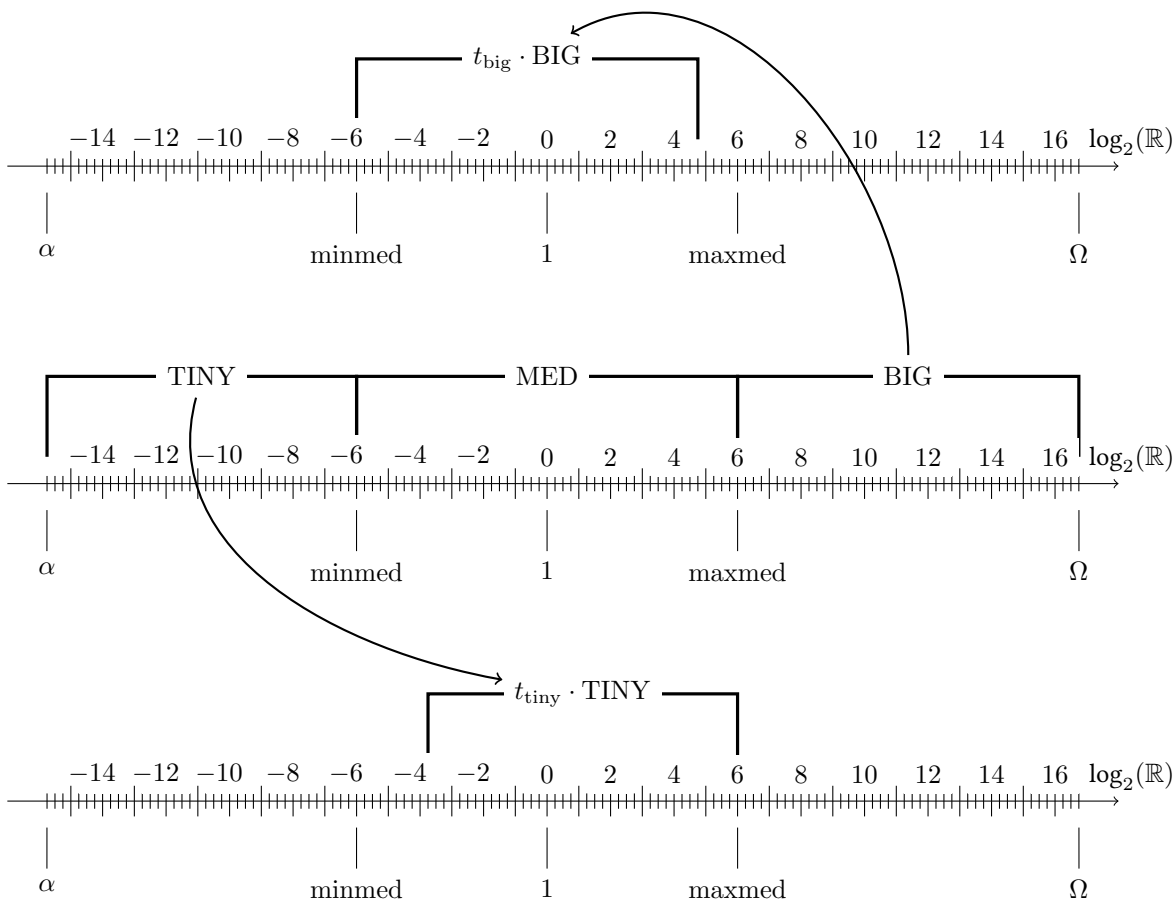

Figure 2: The splitting and the scaling of the FP numbers into 3 classes TINY, MED, and BIG.

3. if BIG is empty and TINY is nonempty, we define

$$\begin{aligned} y_{\min} &= \min\left\{\sqrt{S_{\text{med}}}, \tfrac{1}{t_{\text{tiny}}}\sqrt{S_{\text{tiny}}}\right\}, \\ y_{\max} &= \max\left\{\sqrt{S_{\text{med}}}, \tfrac{1}{t_{\text{tiny}}}\sqrt{S_{\text{tiny}}}\right\}, \end{aligned}$$

   and we go to step 4;

4. if $y_{\min} < \sqrt{u} \cdot y_{\max}$, we return $y_{\max}$, otherwise we return

$$y_{\max} \cdot \left(1 + (y_{\min}/y_{\max})^2\right)^{1/2}. \tag{6}$$

The additional division and square-root that appear in (6) were perhaps unavoidable in the pre-IEEE-754 era. However, they involve additional delay and rounding error in the calculation. Graillat et al. [18] also split the input values into 3 classes, and give a simpler solution for the final reconstruction of the norm N from S_{big}, S_{med}, and S_{tiny}. Their work uses double-word arithmetic (see Section 1.2) for accumulating the sums S_{big}, S_{med}, and S_{tiny}, so the context is slightly different (we will come to that later on in this paper), but let us momentarily present their solution for avoiding under/overflow in the context of simple FP numbers. They choose:

$$t_{\text{big}} = 2^{-E}, \tag{7a}$$

$$t_{\text{tiny}} = 2^{E}, \text{ with} \tag{7b}$$

$$E = 2 \times \lceil (1/2) \cdot \lceil (e_{\max} - e_{\min} + p)/3 \rceil \rceil, \tag{7c}$$

$$\text{minmed} = 2^{e_{\max}+1-2E}, \tag{7d}$$

$$\text{maxmed} = 2^{e_{\max}+1-E}, \tag{7e}$$

so that $t_{\text{big}} = 1/t_{\text{tiny}}$ and $\text{minmed} = t_{\text{big}} \cdot \text{maxmed}$. The choice (7c) indicates that they obtain TINY, MED, and BIG by splitting the exponent range of the FP format into three parts of approximately the same size. Assume $n < 1/u$. Graillat et al. show that:

- If BIG is nonempty, we can neglect the elements of TINY, so that we need to compute

$$\sqrt{S_{\text{med}} + S_{\text{big}}/t_{\text{big}}^2}.$$

  This can be done without under/overflows as follows.

  - If $S_{\text{big}} \geq \text{minmed}^2/u^3$ (i.e., $S_{\text{big}}/t_{\text{big}}^2 \geq \text{maxmed}^2/u^3 \geq n \cdot \text{maxmed}^2/u^2$) or $S_{\text{med}} \leq \text{maxmed}^2 u^2$, then S_{med} is negligible in front of $S_{\text{big}}/t_{\text{big}}^2$ and we can return $(1/t_{\text{big}})\sqrt{S_{\text{big}}} = t_{\text{tiny}}\sqrt{S_{\text{big}}}$;

- if $S_{\text{med}} > \text{maxmed}^2 u^2$ and $S_{\text{big}} < \text{minmed}^2/u^3$, then we can compute

$$\chi = S_{\text{big}}/t_{\text{big}} + t_{\text{big}} S_{\text{med}} = t_{\text{tiny}} S_{\text{big}} + t_{\text{big}} S_{\text{med}}$$

  without overflow or underflow, so that we can return

$$\left(1/\sqrt{t_{\text{big}}}\right) \cdot \sqrt{\chi}, \tag{8}$$

  and the (precomputed) constant $1/\sqrt{t_{\text{big}}} = \sqrt{t_{\text{tiny}}}$ is a power of 2 since (7a) and (7c) imply that t_{big} is an *even* power of 2, so all multiplications in (8) are exact.

- If BIG is empty, we need to compute

$$\sqrt{S_{\text{med}} + S_{\text{tiny}}/t_{\text{tiny}}^2} = (1/t_{\text{tiny}}) \cdot \sqrt{S_{\text{tiny}} + S_{\text{med}}/t_{\text{big}}^2}.$$

  This can be done as follows.

  - If $S_{\text{med}} \geq \text{minmed}/u^3$ or $S_{\text{tiny}} \leq \text{maxmed}^2 u^2$, we can return $\sqrt{S_{\text{med}}}$;
  - otherwise, we can safely compute the desired result as

$$\left(1/\sqrt{t_{\text{tiny}}}\right) \cdot \sqrt{t_{\text{tiny}} S_{\text{med}} + t_{\text{big}} S_{\text{tiny}}}$$

    and the term $1/\sqrt{t_{\text{tiny}}} = \sqrt{t_{\text{big}}}$ is a power of 2.

2.2 Using double-word numbers to improve accuracy: Graillat et al.'s solution

Let us temporarily put aside the problem of avoiding spurious under/overflow, and let us focus on the need for accurately computing the sum of squares and the square-root involved in the computation of the Euclidean norm N. The goal of Graillat et al. [18] was to guarantee *faithful rounding* of N. For that purpose, to save accuracy as much as possible, they compute the sum of squares $\sum_{i=0}^{n-1} a_i^2$ in double-word arithmetic. They first express the squares of the FP numbers a_i as DW numbers using Algorithm 3 (Fast2Mult). Then they sum the obtained DW numbers using Algorithm 5 (SloppyDWPlusDW). As they mention, that summation is easily parallelizable. The obtained result is a double-word approximation (S_h, S_ℓ) to the sum of squares.

After this, they take the square-root of S_h, using the correctly rounded square-root instruction that is available on all IEEE 754 compliant systems. They show that, under reasonable conditions, that square-root is a faithful rounding of the norm. More precisely, the condition they give on n for their algorithm to return a faithful result is $n < 1/(24u + u^2)$, i.e., $n \leq 699050$ in binary32 arithmetic, and $n \leq 3.752 \times 10^{14}$ in binary64 arithmetic.

Assuming sequential addition of the squares of the a_is, and assuming that all numbers are in the MED class (i.e., no scaling is needed), Graillat et al.'s algorithm uses $13n - 10$ floating-point operations.

Let us mention, however, that the choice of "dropping" S_ℓ is tantamount to losing a non-negligible information on the sum of squares.

Incidentally, Graillat et al. make a little and reasonably harmless mistake: they did not realize that the value of minmed they choose (called β_0 in their paper), given in Eq. (7d), is less than η in binary32 arithmetic (it, is however, larger than η in binary64 and binary128 arithmetics). Hence, in very rare cases (computations of norms in binary32 arithmetic with all scaled operands slightly over minmed), some squares will not be expressed exactly as DW numbers. Whether this can lead to errors slightly larger than the claimed bound remains an open question.

They target faithful rounding (i.e., error less than 1 ulp). We have a different goal in mind: we wish to achieve a final error extremely close to 0.5 ulp of the exact result, i.e., we wish to almost always provide a correctly rounded result. This will be done by keeping the sum (S_h, S_ℓ) of the squares in DW arithmetic, and using an algorithm that computes the square-root of a DW number (Algorithm 9). We will also compute (S_h, S_ℓ) more accurately, by using a different summation scheme, based on Algorithm 4.

2.3 An alternative: computing norms with pair arithmetic

In [43], Lange and Rump give conditions for their pair arithmetic to return *faithfully rounded* results. We have applied Theorems 4.2 and 5.4 of [43] to two cases: the computation of N when the squares are added sequentially using the Pair_addition algorithm, and the same computation where the a_i^2 are added *blockwise*: we divide them in k blocks of m terms, with $km = n$, we first add all the terms of each block together, and then we add the k obtained sums. We obtain the following results:

- with the sequential summation, we will obtain a faithfully rounded result if $\left\lceil \frac{4}{5}n + \frac{5}{4} \right\rceil \leq 1/\sqrt{2u - u^2} - 2$;
- with the blockwise summation, we will obtain a faithfully rounded result if $\left\lceil \frac{4}{5}(m + k - 1) + \frac{5}{4} \right\rceil \leq 1/\sqrt{2u - u^2} - 2$.

Table 3 presents the maximum possible values of n allowed by these conditions, for binary32 and binary64 arithmetics. For the blockwise algorithm, we have chosen the "optimal" choice $k = m = \lceil \sqrt{n} \rceil$.

Assuming sequential addition of the squares of the a_is, and assuming that all numbers are in the MED class (i.e., no scaling is needed), computing a norm in pair arithmetic uses $10n - 3$ floating-point operations. Of course, exactly as for conventional or double-word arithmetics, one may need scalings to avoid spurious under/overflows.

Table 3: Maximum values of n for which a pair-arithmetic implementation is guaranteed to return a faithful result. For the blockwise algorithm, we have chosen $k = m = \lceil \sqrt{n} \rceil$.

format	sequential summation	blockwise summation
binary32	3615	3,268,864
binary64	83,886,075	$\approx 8.796 \times 10^{14}$

3 Some results on double-word arithmetic

In this section, let us give a few new results on double-word arithmetic that can be useful for accurately computing norms. All the results of this section have been formally certified using the Coq proof assistant[7] and the Flocq [10, 11] library.

3.1 Properties of DWPlusFP

First, the relative error bound (4) on Algorithm DWPlusFP (Alg. 4) was given in [32] assuming input numbers of *arbitrary sign*. One may wonder if, when the operands have the same sign, we can obtain a better error bound. This would be useful for summing squares, which is the main step of the computation of Euclidean norms. Indeed, we have,

Theorem 3.1. *If $x = (x_h, x_\ell)$ is a nonnegative double-word number and y is a nonnegative FP number, then the relative error of Algorithm 4 is bounded by u^2. That bound is asymptotically optimal.*

The proof is given in the supplementary materials.

Theorem 3.1 does not apply when y is negative. However, for very small values of $|y|$, one can nevertheless obtain an error bound significantly better than (4), which will be useful in Section 4.1. More precisely,

Property 3.2. *Assuming $u \leq 1/16$ (i.e., $p \geq 4$), if $x = x_h + x_\ell$ is positive and $y \geq (-2u - u^2) \cdot x$, then the relative error of Algorithm 4 is bounded by $u^2 + 3u^3$.*

The proof is given in the supplementary materials.

We also have,

Remark 3.3. *When the operands $x = x_h + x_\ell$ and y are positive, Algorithm 4 satisfies the condition of Lemma 1.4 (with $\mathbb{F}$ being the set of the DW numbers), namely:*

$$\left|\mathrm{DWPlusFP}(x_h, x_\ell, y) - (x_h + x_\ell + y)\right| \leq \min\left\{(x_h + x_\ell), y\right\}.$$

[7] `https://coq.inria.fr/`

Proof. We have $|\text{DWPlusFP}(x_h, x_\ell, y) - (x_h + x_\ell + y)| = |v - (x_\ell + s_\ell)|$, and

$$|v - (x_\ell + s_\ell)| \leq |x_\ell| \leq u \cdot (x_h + x_\ell) < x_h + x_\ell,$$

and $|v - (x_\ell + s_\ell)| \leq |s_\ell| \leq y$. Therefore, $|v| \leq \min\{x_h + x_\ell, y\}$. □

Later on, we will compute sums of squares using Algorithm DWPlusFP (Algorithm 4). We will need to bound the computed sum of $n \leq 1/u$ positive numbers less than some power of 2, say 2^k, by $n \cdot 2^k$ (this is of course a straightforward property of the exact sum, but this is far from obvious for the computed sum). This will be ensured by the following lemma.

Lemma 3.4. *If (x_h, x_ℓ) is a DW number and y is a FP number such that $x_h, y \geq 2^{e_{\min}}$, $x_h + x_\ell \leq m_1 \cdot 2^k$ and $y \leq m_2 \cdot 2^k$ where m_1 and m_2 are positive integers satisfying $m_1 + m_2 \leq 2^p$ then, barring overflow, the double-word number (z_h, z_ℓ) returned by Algorithm DWPlusFP satisfies $z_h + z_\ell \leq (m_1 + m_2) \cdot 2^k$.*

The proof is given in the supplementary materials.

3.2 Square-root of a double-word number

Assume that $x = (x_h, x_\ell)$ is a DW number, and that $x_h \geq 2^{2k}$, where k is an integer larger than or equal to $(e_{\min} + p)/2$. The following two algorithms evaluate the square-root of x. Algorithm 8 returns a DW number, and Algorithm 9 returns a FP number.

Algorithm 8 – SQRTDWtoDW(x_h, x_ℓ). Computes the square-root of the DW number (x_h, x_ℓ) in binary, precision-p, floating-point arithmetic and returns a DW number (z_h, z_ℓ). It takes 8 FP operations (counting the FP square-root as one).

1: **if** $x_h = 0$ **then**
2: **return** $(0, 0)$
3: **else**
4: $s_h \leftarrow \text{RN}(\sqrt{x_h})$
5: $\rho_1 \leftarrow \text{RN}(x_h - s_h^2)$ (with an FMA instruction)
6: $\rho_2 \leftarrow \text{RN}(x_\ell + \rho_1)$
7: $s_\ell \leftarrow \text{RN}(\rho_2/(2 \cdot s_h))$
8: $(z_h, z_\ell) \leftarrow \text{Fast2Sum}(s_h, s_\ell)$
9: **return** (z_h, z_ℓ)
10: **end if**

To obtain a floating-point number, one can replace the "Fast2Sum" of Line 8 of Algorithm 8 by a floating-point addition and obtain

Algorithm 9 – SQRTDWtoFP(x_h, x_ℓ). Computes the square-root of the DW number (x_h, x_ℓ) in binary, precision-p, floating-point arithmetic and returns a floating-point number z. It takes 6 FP operations (counting the FP square-root as one).

1: **if** $x_h = 0$ **then**
2: **return** 0
3: **else**
4: $s_h \leftarrow \mathrm{RN}(\sqrt{x_h})$
5: $\rho_1 \leftarrow \mathrm{RN}(x_h - s_h^2)$ (with an FMA instruction)
6: $\rho_2 \leftarrow \mathrm{RN}(x_\ell + \rho_1)$
7: $s_\ell \leftarrow \mathrm{RN}(\rho_2/(2 \cdot s_h))$
8: $z \leftarrow \mathrm{RN}(s_h + s_\ell)$
9: **return** z
10: **end if**

Remark 3.5. *For performance purposes, if the FMA instruction is fast enough, to avoid the multiplication of s_h by 2 that appears in Line 7 of both algorithms, one can somehow "delay" that operation, and replace Lines 7 and 8 of Algorithm 8 by (we inline the Fast2Sum algorithm for the sake of clarity):*

7: $t_\ell \leftarrow \mathrm{RN}(\rho_2/s_h)$ *(t_ℓ equals $2s_\ell$)*
8: $z_h \leftarrow \mathrm{RN}(s_h + 0.5 \cdot t_\ell)$ *(first line of Fast2Sum(s_h, s_ℓ))*
9: $\delta \leftarrow \mathrm{RN}(z_h - s_h)$ *(second line of Fast2Sum(s_h, s_ℓ))*
10: $z_\ell \leftarrow \mathrm{RN}(0.5 \cdot t_\ell - \delta)$ *(third line of Fast2Sum(s_h, s_ℓ))*

Similarly, one can replace Lines 7 and 8 of Algorithm 9 by:

7: $t_\ell \leftarrow \mathrm{RN}(\rho_2/s_h)$ *(t_ℓ equals $2s_\ell$)*
8: $z \leftarrow \mathrm{RN}(s_h + 0.5 \cdot t_\ell)$

By doing so, Algorithm 8 now takes 7 FP operations, and Algorithm 9 takes 5 FP operations. In both cases, the computed results are exactly the same.

Let us now analyze Algorithms 8 and 9. We assume that the input double-word operand $x = (x_h, x_\ell)$ is positive, i.e., $x_h > 0$ (the case $x_h = 0$ is straightforward). We have,

Theorem 3.6. *If $x = (x_h, x_\ell)$ is a double-word number, $p \geq 5$, $x \geq 2^{2k}$, where k is an integer larger than or equal to $(e_{\min} + p)/2$ and no overflow or underflow occurs, then the relative error of Algorithm 8 is bounded by $(25/8) \cdot u^2 = 3.125 \cdot u^2$, and that bound is asymptotically optimal.*

Theorem 3.7. *If $x = (x_h, x_\ell)$ is a double-word number, $p \geq 5$, $x \geq 2^{2k}$, where k is an integer larger than or equal to $(e_{\min} + p)/2$ and no overflow or underflow occurs, then the floating-point number returned by Algorithm 9 is within $(1/2 + (7/4) \cdot 2^{-p}) \cdot \mathrm{ulp}(\sqrt{x_h + x_\ell})$ from $\sqrt{x_h + x_\ell}$, and the relative error of that algorithm is bounded by $u + (17/8) \cdot u^2 + (33/8) \cdot u^3$.*

The proof of Theorem 3.11 below uses the following result.

Remark 3.8. *If $x_h = 2^{2k}$ with $k \in \mathbb{Z}$, then Algorithm 9 returns 2^k.*

The common proof of Theorem 3.6, Theorem 3.7, and Remark 3.8 uses results from [5, 6, 28] and is given in the supplementary materials.

Intuitively, since the square-root of a huge number is less than that number, and the square-root of a tiny number is larger than that number, overflows and underflows are not much of a concern when evaluating square-roots. This does not mean that *intermediate calculations* in Algorithms 8 and 9 cannot underflow or overflow. Let us now quickly address this issue.

Remark 3.9. *Under the conditions of Theorems 3.6 and 3.7 (and assuming $e_{\max} \geq 2$, which always holds in practice[8]), no overflow can occur in Algorithms 8 and 9.*

The proof is given in the supplementary materials.

Remark 3.10. *Under the conditions of Theorems 3.6 and 3.7, with the additional assumption $p + 3 \leq e_{\max}$, underflows in Algorithms 8 and 9 are impossible or harmless.*

The proof is given in the supplementary materials.

Now, let us assume that the input values (x_h, x_ℓ) of Algorithm SQRTDWtoFP (Algorithm 9) approximate some number x with a known relative error bound. Let us see how SQRTDWtoFP(x_h, x_ℓ) approximates $\sqrt{x}$. We have

Theorem 3.11. *If (x_h, x_ℓ) approximates a positive number x with relative error bounded by νu^2 with*

$$\nu u^2 < 1 \tag{9}$$

and if no underflow/overflow occurs, then $R = \mathrm{SQRTDWtoFP}(x_h, x_\ell)$ approximates $\sqrt{x}$ with a relative error bounded by

$$\left(u + \frac{17}{8}u^2 + \frac{33}{8}u^3\right) \cdot \left(1 + \frac{\nu \cdot u^2}{1 - \nu \cdot u^2}\right) + \frac{\nu \cdot u^2}{1 - \nu \cdot u^2}. \tag{10}$$

[8] In fact, Condition $e_{\max} \geq 2$ can be deduced from the conditions of Theorems 3.6 and 3.7. We know that $x \geq 2^{e_{\min}+p}$, so that $x_h \geq 2^{e_{\min}+p}$, which can be representable only if $e_{\max} \geq e_{\min} + p$. And since $p \geq 5$ and $e_{\min} = 1 - e_{\max}$, we have $e_{\max} \geq 6 - e_{\max}$. Therefore, $e_{\max} \geq 3$.

Furthermore, under the more stringent condition

$$2u\nu < 1, \tag{11}$$

we have

$$\left|R - \sqrt{x}\right| \leq \left(\frac{1}{2} + u \cdot \left(\frac{7}{4} + \frac{\nu}{1 - \nu \cdot u^2}\right)\right) \text{ulp}\left(\sqrt{x}\right). \tag{12}$$

The proof is given in the supplementary materials.

Very similarly to what we have done with Algorithm SQRTDWtoFP (Algorithm 9), let us now assume that the input values (x_h, x_ℓ) of Algorithm SQRTDWtoDW (Algorithm 8) approximate some number x with a known relative error bound. Let us see how SQRTDWtoDW(x_h, x_ℓ) approximates $\sqrt{x}$. We have

Theorem 3.12. *If (x_h, x_ℓ) approximates a positive number x with relative error bounded by νu^2 with $\nu u^2 < 1$ and if no underflow/overflow occurs, then $R =$ SQRTDWtoDW(x_h, x_ℓ) approximates $\sqrt{x}$ with a relative error bounded by*

$$u^2 \cdot \left(\frac{25}{8} + \frac{\nu}{1 - \nu u^2} + \frac{25}{8} \cdot \frac{\nu u^2}{1 - \nu u^2}\right). \tag{13}$$

The proof is described in the supplementary materials.

4 Our algorithms for computing Euclidean norms

4.1 Computing a Euclidean norm assuming no underflow or overflow occurs

In this section, we assume that all the terms a_i of (1) are in MED, so that no underflow/overflow occurs and a_i^2 is exactly representable by a DW number for all i. We first approximate the sum of squares $\sum_{i=0}^{n-1} a_i^2$ by a DW number (S_h, S_ℓ), with some relative error νu^2, and then use Algorithm SQRTDWtoFP (Algorithm 9) to approximate the square-root of $S_h + S_\ell$ by a floating-point number R. The final error will be deduced from ν using Theorem 3.11.

Let us now first present two different ways of computing (S_h, S_ℓ).

4.1.1 Sequential computation of the sum of squares

Let us first consider the following, sequential algorithm.

Algorithm 10 Sequential computation of $\sum_{i=0}^{n-1} a_i^2$ assuming no underflow/overflow occurs. It takes $13n - 5$ FP operations.

1. For $i = 0 \dots n-1$, express the terms a_i^2 as double-word numbers (y_i^h, y_i^ℓ), defined as

$$(y_i^h, y_i^\ell) = \text{Fast2Mult}(a_i, a_i). \tag{14}$$

   We have $a_i^2 = y_i^h + y_i^\ell$.

2. Accumulate the terms y_i^h using the DWPlusFP algorithm (Algorithm 4). More precisely, define

$$(x_1^h, x_1^\ell) = \text{2Sum}(y_0^h, y_1^h)$$

   first, then, iteratively compute, for $i = 2 \dots n-1$, the terms

$$(x_i^h, x_i^\ell) = \text{DWPlusFP}(x_{i-1}^h, x_{i-1}^\ell, y_i^h).$$

3. Accumulate the terms y_i^ℓ using the conventional "recursive" summation, i.e., for $i = 0 \dots n-2$, compute

$$\sigma_{i+1} = \text{RN}(\sigma_i + y_{i+1}^\ell),$$

   with $\sigma_0 = y_0^\ell$.

4. Obtain the approximation to $\sum_{i=0}^{n-1} a_i^2$ with one call to DWPlusFP:

$$(S_h, S_\ell) = \text{DWPlusFP}(x_{n-1}^h, x_{n-1}^\ell, \sigma_{n-1}).$$

Algorithm 10 assumes $n \geq 2$. We have,

Lemma 4.1. *Assuming no underflow/overflow occurs, $u \leq 1/16$, and $n \leq 1/u$, the double-word number (S_h, S_ℓ) returned by Algorithm 10 satisfies*

$$\begin{aligned} &\left|(S_h + S_\ell) - \textstyle\sum_{i=0}^{n-1} a_i^2\right| \\ &\leq ((2n-1)u^2 + (n+2)u^3 + (2n-2)u^4 + (7n-7)u^5 + (3n-3)u^6) \cdot \textstyle\sum_{i=0}^{n-1} a_i^2, \end{aligned} \tag{15}$$

which implies

$$\left|(S_h + S_\ell) - \sum_{i=0}^{n-1} a_i^2\right| \leq \left((2n-1)u^2 + (n+5)\,u^3\right) \cdot \sum_{i=0}^{n-1} a_i^2. \tag{16}$$

The proof is given in the supplementary materials.

4.1.2 Blockwise computation of the sum of squares

Now, assume that $n = km$, and that we separate the input numbers a_i into k blocks of m numbers, either for parallelizing the calculation or for obtaining (as it will be clear later on) a more accurate result. Block number j ($j = 0, \ldots, k-1$) contains the elements $a_{mj}, a_{mj+1}, a_{mj+2}, \ldots, a_{m(j+1)-1}$.

We separately sum the elements of each block using Algorithm 10 (which requires $m \leq 1/u$). The results of these "partial" summations are DW numbers (Z_j^h, Z_j^ℓ). A solution could be to sum these numbers using Algorithm SloppyDWPlusDW (Algorithm 5). We obtain, however, a better error bound by summing these terms in the same way as we have summed the terms a_i^2 in Algorithm 10 (which requires $k \leq 1/u$), i.e., we first compute a DW approximation to the sum of the "higher" terms Z_j^h using DWPlusFP (Algorithm 4) iteratively, we then accumulate the "lower" terms Z_j^ℓ using naive summation, and we finally add the obtained results with one call to DWPlusFP. This gives Algorithm 11, presented below.

It could be possible to repeat that block decomposition recursively, resulting in better error bounds. We doubt this would be efficient (except possibly for huge values of n).

For analyzing Algorithm 11, we need the following lemma.

Lemma 4.2. *Let n be a positive integer.*

- *the maximum possible value of $k + m$, where k and m are integers larger than or equal to 2 satisfying $km = n$ is $n/2 + 2$;*
- *the minimum possible value of $k + m$, where k and m are positive integers satisfying $km = n$ is $r_n + n/r_n$, where r_n is the largest divisor of n less than or equal to $\sqrt{n}$. That bound is always larger than or equal to $2\sqrt{n}$.*

Proof. Straightforward by considering the variation of function $t \to t + n/t$. □

Algorithm 11 Blockwise computation of $\sum_{i=0}^{n-1} a_i^2$ assuming no underflow/overflow occurs. It takes $13n + 6k - 5$ FP operations.

1. for $j = 0, 1, \ldots, k-1$, compute an approximation (Z_j^h, Z_j^ℓ) to $\sum_{i=mj}^{m(j+1)-1} a_i^2$ using Algorithm 10 (the sequential summation algorithm) applied to $a_{mj}, a_{mj+1}, a_{mj+2}, \ldots, a_{m(j+1)-1}$;
2. accumulate the terms Z_j^h using Algorithm DWPlusFP (Algorithm 4). More precisely, defining
$$\left(\Sigma_1^h, \Sigma_1^\ell\right) = \text{2Sum}(Z_0^h, Z_1^h),$$
iteratively compute, for $j = 2 \ldots k-1$ the terms
$$\left(\Sigma_j^h, \Sigma_j^\ell\right) = \text{DWPlusFP}\left(\Sigma_{j-1}^h, \Sigma_{j-1}^\ell, Z_j^h\right);$$
3. accumulate the terms Z_j^ℓ using the conventional "recursive" summation, i.e., for $j = 0 \ldots k-2$, compute
$$\tau_{j+1} = \text{RN}(\tau_j + Z_{j+1}^\ell),$$
with $\tau_0 = Z_0^\ell$;
4. obtain the approximation (S_h, S_ℓ) to $\sum_{i=0}^{n-1} a_i^2$ as
$$(S_h, S_\ell) = \text{DWPlusFP}\left(\Sigma_{k-1}^h, \Sigma_{k-1}^\ell, \tau_{k-1}\right).$$

Algorithm 11 assumes $k \geq 2$. It can be applied to the special case $k = 1$ if it stops after step 1 and returns $(S_h, S_\ell) = (Z_0^h, Z_0^\ell)$, so that it reduces to Algorithm 10, taking only $13n - 5$ FP operations.

Lemma 4.3. *Assuming no underflow/overflow occurs, $k \leq 1/u$, $m \leq 1/u$, and $u \leq 1/16$, the double-word number (S_h, S_ℓ) returned by Algorithm 11 satisfies*

$$\frac{\left|(S_h + S_\ell) - \sum_{i=0}^{n-1} a_i^2\right|}{\sum_{i=0}^{n-1} a_i^2} \leq \beta(k) + \beta(m) + \beta(k)\beta(m), \tag{17}$$

where $\beta(t) = (2t-1)u^2 + (t+2)u^3 + (2t-2)u^4 + (7t-7)u^5 + (3t-3)u^6$. Furthermore, if $u \leq 1/32$, we obtain

$$\frac{\left|(S_h + S_\ell) - \sum_{i=0}^{n-1} a_i^2\right|}{\sum_{i=0}^{n-1} a_i^2} \leq (2k + 2m - 2) \cdot u^2 + (0.1290n + 0.9465(k+m) + 10.03) \cdot u^3. \tag{18}$$

The proof is given in the supplementary materials.

Let us now compare the bounds of Algorithm 10 and Algorithm 11, i.e., the bounds (15) and (17), with the bound of Graillat et al.'s algorithm [18] (derived from the relative error bound $3u^2$ of Algorithm 5), namely

$$\frac{3(n-1)u^2}{1-3(n-1)u^2}. \tag{19}$$

We have the following property:

Property 4.4.

- *if ($k = 1$ and $m = n$) or ($k = n$ and $m = 1$), then Algorithm 11 boils down to Algorithm 10;*
- *as soon as $n \geq 3$, $u \leq 1/32$, and $3(n-1)u^2 < 1$ (which is necessary for (19) to make sense), the bound (19) is larger than the bound (16);*
- *if $k \geq 2$ and $m \geq 2$, assuming $u \leq 1/32$ and $n \leq 1/u$, the bound (16) is larger than the bound (18).*

The proof is given in the supplementary materials.

Property 4.4 shows that in all practical cases, our blockwise algorithm has a better error bound than our sequential algorithm, which itself has a better error bound that Graillat et al.'s algorithm.

Assuming $nu \ll 1$, so that the bound (17) is essentially $(2k + 2m - 2) \cdot u^2$, a direct consequence of Lemma 4.2 is that an approximate minimum of the bound (17) is reached when $k = r_n$ or $k = n/r_n$, where r_n is the largest divisor of n less than or equal to $\sqrt{n}$, resulting, if $r_n \approx \sqrt{n}$ in a relative error less than around $(4\sqrt{n} - 2) \cdot u^2$. For instance, in IEEE 754 binary64 arithmetic ($u = 2^{-53}$), with $n = 6000$, the obtained relative error bounds are:

- $1.3322 \times 10^{-12}\, u$ with our sequential algorithm;
- $1.9981 \times 10^{-12}\, u$ with Graillat et al.'s algorithm;
- $3.5306 \times 10^{-14}\, u$ with the blockwise summation algorithm, with the near-optimal choices $k = 60$ and $m = 100$ or $k = 100$ and $m = 60$.

Note that even with a very unbalanced block splitting the blockwise summation is significantly more accurate than the sequential summation. Still with the same values of n and u, the error bound becomes 3.3374×10^{-13} in the case $k = 4$ and $m = 1500$. This is of interest, since for performance reasons, one may wish to choose k equal to the maximum number of floating-point numbers that fit in an SIMD vector, which in practice is not a very large number (see Section 6.2). Beyond being more accurate, the blockwise version exhibits more parallelism than the sequential version, which may lead to better overall performance. Of course, if n is prime or has divisors that

do not allow for a balanced splitting, it may be worth using the blockwise algorithm with a smaller final block (the error bounds still apply, with a "theoretical" number of elements slightly larger than n, corresponding to appending additional zero elements to the vector $(a_0, a_1, \ldots, a_{n-1})$ to complete the final block).

4.1.3 Obtaining the Euclidean norm barring underflow/overflow

We can now combine Lemma 4.3 and Theorem 3.11, and obtain.

Theorem 4.5. *Assume that for all i, $a_i \in \mathrm{MED}$. Assume that $u \leq 1/32$ (i.e., $p \geq 5$) and that Algorithm 10 (sequential summation, with $n \leq 1/u$), or Algorithm 11 (blockwise summation, with $km = n$ and $k, m \leq 1/u$), is used to compute the approximation (S_h, S_ℓ) to $\sum_{i=0}^{n-1} a_i^2$, and that Algorithm SQRTDWtoFP (Algorithm 9) is used to approximate the square-root of $S_h + S_\ell$ by a floating-point number R. Let $\beta(t) = (2t-1)u^2 + (t+2)u^3 + (2t-2)u^4 + (7t-7)u^5 + (3t-3)u^6$, and define a parameter ν as follows:*

$$\nu = \beta(n)/u^2 \quad \textit{if the sequential summation algorithm is used}; \tag{20a}$$

$$\nu = \big(\beta(k) + \beta(m) + \beta(k)\beta(m)\big)/u^2 \quad \textit{if the blockwise summation algorithm is used.} \tag{20b}$$

If $\nu < 1/(2u)$, we have:

$$\left| R - \sqrt{\sum_{i=0}^{n-1} a_i^2} \right| \leq \left(\frac{1}{2} + u \cdot \left(\frac{7}{4} + \frac{\nu}{1 - \nu \cdot u^2} \right) \right) \mathrm{ulp} \left(\sqrt{\sum_{i=0}^{n-1} a_i^2} \right). \tag{21}$$

Note that since $u \leq 1/32$, we can use in Theorem 4.5 the following simpler expressions for ν (see Lemmas 4.1 and 4.3):

$$\nu = (2n-1) + (n+5)\, u \tag{22}$$

with the sequential summation algorithm, and

$$\nu = (2k + 2m - 2) + (0.1290n + 0.9465(k+m) + 10.03) \cdot u \tag{23}$$

with the blockwise summation algorithm. Condition $\nu < 1/(2u)$ is not that restrictive: in the binary32 format of the IEEE 754 Standard, assuming we use the sequential algorithm for summing the terms a_i^2, it is satisfied for $n \leq 4194304$, and in the binary64, it is satisfied for $n \leq 2{,}251{,}799{,}813{,}685{,}248$. Even larger values of n can be reached if we use the blockwise algorithm: in the binary32 format, choosing k equal to the largest divisor of n less than or equal to $\sqrt{n}$, the condition is satisfied for $n = 500{,}000{,}000{,}000$.

If we use our algorithm for computing the Euclidean norm of a vector of 10000 binary64 elements, the returned result will be within 0.500000000002221 ulp from the exact result with the

sequential summation, and within 0.5000000000000444 ulp from the exact result with the block-wise summation and the choice $k = m = 100$. This means that we will almost always obtain a correctly rounded result.

Remark 4.6. *Note that instead of computing* $R = \text{SQRTDWtoFP}(S_h, S_\ell)$ *one may decide to compute* $(R_h, R_\ell) = \text{SQRTDWtoDW}(S_h, S_\ell)$ *(i.e., one chooses to obtain a double-word final result, using Algorithm 8 instead of Algorithm 9). By combining Lemma 4.3 and Theorem 3.12, one easily finds that* (R_h, R_ℓ) *approximates the euclidean norm* N *with relative error*

$$\epsilon = u^2 \cdot \left(\frac{25}{8} + \frac{\nu}{1 - \nu u^2} + \frac{25}{8} \cdot \frac{\nu u^2}{1 - \nu u^2} \right),$$

where ν *is the same as in Theorem 4.5. This can be of interest in two cases:*

- *if subsequent double-word calculations are to be performed;*
- *if one wishes to be certain of obtaining a correctly rounded norm: Ziv's rounding test [14] can be applied to the pair* (R_h, R_ℓ) *to check if* $R_h = \text{RN}(N)$*. More precisely (see [14, Theorem 2.1]), if* e *is a FP number larger than or equal to* $(1 + u)/(1 - \epsilon - 2\epsilon/u)$*, then* $\text{RN}(R_h + \text{RN}(R_\ell \cdot e)) = R_h$ *implies that* $R_h = \text{RN}(N)$*.*

4.2 Computing Euclidean norms in the general case

4.2.1 Choice of the parameters minmed, maxmed, t_{tiny}, and t_{big}

Let us compute Euclidean norms, still using DW arithmetic, now in the general case (i.e., we no longer assume that underflow and overflow cannot occur). We will use the three-class approach presented in Section 2.1, inherited from Blue, very much like what is done by Graillat et al. [18], but with different choices for the parameters minmed, maxmed, t_{tiny}, and t_{big}, for two reasons:

- first, from (3), if we wish to express the squares of the elements of MED as DW numbers without error, we need to replace Constraint (5a) of Section 2.1 by $\mathrm{minmed} \geq \eta$;
- then, to make the necessity of scaling as infrequent as possible, and to make as small as possible the error committed when we neglect the elements of TINY because BIG is nonempty, we try to have maxmed as large as possible and minmed as small as possible.

Exactly as is done by Graillat et al., we choose maxmed and minmed equal to powers of 2, and to avoid introducing additional rounding errors in the scalings, we choose t_{big} and t_{tiny} equal to *even* powers of 2. To simplify the analysis we also choose $t_{\mathrm{big}} = 1/t_{\mathrm{tiny}}$. We also assume $n_{\max} = 1/u = 2^p$, i.e., we wish to guarantee a correct behavior of the algorithms for vectors of dimension up to 2^p.

To simplify the analysis, we also assume $u \leq 1/32$, which always holds in practice and is needed anyway for the error bound of the SQRTDWtoFP algorithm (given by Theorem 3.6) to hold. In particular, this makes it possible to use the simpler expression (23) for variable ν in Theorem 4.5.

Lemma 3.4 implies that if maxmed is a power of 2, when summing, using Algorithm 10 and/or Algorithm 11, n numbers less than maxmed^2, with $n \leq 1/u = 2^p$, the computed result is less than $n \cdot \mathrm{maxmed}^2$. This gives the following constraint on maxmed: $2^p \cdot \mathrm{maxmed}^2 \leq \Omega$.

This leads us to the following choices:

- minmed is the power of 2 just above or equal to η, i.e., $\mathrm{minmed} = 2^{\lceil (e_{\min}+p)/2 \rceil}$ (with this choice we still can observe rare cases where the low-order element of a DW number generated by the computation of a square is subnormal, but this has no influence on accuracy, even if it can have one on performance).

- maxmed is the power of 2 just below $\sqrt{\Omega/2^p}$, i.e., $\mathrm{maxmed} = 2^{\lfloor (e_{\max}-p)/2 \rfloor}$. With these definitions, we have $\mathrm{minmed} \cdot \mathrm{maxmed} \in \{1, 2\}$, and

$$\mathrm{minmed}/\mathrm{maxmed} = 2^{-e_{\max}+p+1}, \tag{24}$$

  which will be useful later on.

- Concerning t_{tiny} and t_{big}, the possible values of these parameters are induced by the choices of minmed and maxmed, the constraints (5c), (5d), (5e), and (5f) presented in Section 2.1, and the additional constraint that $t_{\mathrm{tiny}} = 1/t_{\mathrm{big}}$ is an even power of 2.

  We assume

$$3p + 1 \leq e_{\max}, \text{ i.e., } e_{\min} \leq -3p. \tag{25}$$

  Table 4 gives the various parameters and constraints associated with our algorithm for the binary16, binary32, binary64 and binary128 formats of the IEEE 754-2019 Standard for FP Arithmetic [27], and the bfloat16 format [21]. Among all these formats, binary16 is the only one for which the various constraints required by our algorithm are not satisfied.

  Our choice $t_{\mathrm{tiny}} = 1/t_{\mathrm{big}}$ constraints even more the possible values of t_{big} and t_{tiny}. Table 5 compares the obtained values for the binary32, binary64, and binary128 formats with the values used by Graillat et al. [18].

4.2.2 Obtaining the result from the intermediate sums of squares

The sum $S_{\mathrm{med}} = \sum_{a_i \in \mathrm{MED}} a_i^2$ of the elements of MED is approximated by a double-word $(S_{\mathrm{med}}^h, S_{\mathrm{med}}^\ell)$, obtained using Algorithm 10 (sequential summation) or Algorithm 11 (blockwise summation with k blocks of m elements, where $km = n$). This approximation satisfies

Table 4: The various parameters of our algorithm for the binary16 format of IEEE 754-2019 [27], the bfloat16 format [21], and the binary32, binary64, and binary128 formats of IEEE 754-2019.

parameters	binary16	bfloat16	binary32	binary64	binary128
p	11	8	24	53	113
$e_{\max}$	15	127	127	1023	16383
α	2^{-24}	2^{-133}	2^{-149}	2^{-1074}	2^{-16494}
η	$2^{-3/2}$	2^{-59}	2^{-51}	$2^{-969/2}$	$2^{-16269/2}$
Ω	65504	$2^{128}-2^{120}$ $\approx 3.390\times 10^{38}$	$2^{128}-2^{104}$ $\approx 3.403\times 10^{38}$	$2^{1024}-2^{971}$ $\approx 1.798\times 10^{308}$	$2^{16384}-2^{16271}$ $\approx 1.190\times 10^{4932}$
minmed	$1/2$	2^{-59}	2^{-51}	2^{-484}	2^{-8134}
maxmed	4	2^{59}	2^{51}	2^{485}	2^{8135}
Constraints on t_{big} ((5c), (5d) and even power of 2)	$1/4 \le t_{\text{big}} \le 2^{-14}$ (IMPOSSIBLE)	2^{-118} $\le t_{\text{big}}$ $\le 2^{-70}$	2^{-102} $\le t_{\text{big}}$ $\le 2^{-78}$	2^{-968} $\le t_{\text{big}}$ $\le 2^{-540}$	2^{-16268} $\le t_{\text{big}}$ $\le 2^{-8250}$
Constraints on t_{tiny} ((5e), (5f) and even power of 2)	$2^{24} \le t_{\text{tiny}} \le 4$ (IMPOSSIBLE)	2^{74} $\le t_{\text{tiny}}$ $\le 2^{118}$	2^{98} $\le t_{\text{tiny}}$ $\le 2^{102}$	2^{590} $\le t_{\text{tiny}}$ $\le 2^{968}$	2^{8360} $\le t_{\text{tiny}}$ $\le 2^{16268}$
Constraint $3p+1 \le e_{\max}$	NOT SATISFIED	satisfied	satisfied	satisfied	satisfied

$\left|\left(S^h_{\text{med}} + S^\ell_{\text{med}}\right) - \sum_{a_i \in \text{MED}} a_i^2\right| \le \nu u^2 \sum_{a_i \in \text{MED}} a_i^2$, where ν is defined in (22) if we use Algorithm 10, and (23) if we use Algorithm 11. These algorithms are also applied to the elements of BIG and TINY pre-multiplied by t_{big} and t_{tiny} respectively. Similarly, this gives double words $(S^h_{\text{big}}, S^\ell_{\text{big}})$ and $(S^h_{\text{tiny}}, S^\ell_{\text{tiny}})$ that satisfy

$$\left|\left(S^h_{\text{big}} + S^\ell_{\text{big}}\right) - \sum_{a_i \in \text{BIG}} (t_{\text{big}} a_i)^2\right| \le \nu u^2 \sum_{a_i \in \text{BIG}} (t_{\text{big}} a_i)^2$$

and

$$\left|\left(S^h_{\text{tiny}} + S^\ell_{\text{tiny}}\right) - \sum_{a_i \in \text{TINY}} (t_{\text{tiny}} a_i)^2\right| \le \nu u^2 \sum_{a_i \in \text{TINY}} (t_{\text{tiny}} a_i)^2.$$

This gives

$$\left|\frac{1}{t^2_{\text{tiny}}}\left(S^h_{\text{tiny}} + S^\ell_{\text{tiny}}\right) + \left(S^h_{\text{med}} + S^\ell_{\text{med}}\right) + \frac{1}{t^2_{\text{big}}}\left(S^h_{\text{big}} + S^\ell_{\text{big}}\right) - \sum_{i=0}^{n-1} a_i^2\right| \le \nu u^2 \sum_{i=0}^{n-1} a_i^2. \tag{26}$$

Now, we can follow a reasoning similar to that of Graillat et al. [18]. Denote $S_{\text{med}} = S^h_{\text{med}} + S^\ell_{\text{med}}$,

Table 5: The parameters minmed, maxmed, t_{big}, and t_{tiny} in binary32 arithmetic, for our algorithm and for Graillat et al.'s algorithm [18].

	binary32		binary64		binary128	
	our solution	Graillat et al.	our solution	Graillat et al.	our solution	Graillat et al.
minmed	2^{-51}	2^{-60}	2^{-484}	2^{-376}	2^{-8134}	2^{-5536}
maxmed	2^{51}	2^{34}	2^{485}	2^{324}	2^{8135}	2^{5424}
t_{tiny}	2^{γ}, where $\gamma \in \{98, 100, 102\}$	2^{94}	2^{γ}, where γ is even and $590 \leq \gamma \leq 968$	2^{700}	2^{γ}, where γ is even and $8360 \leq \gamma \leq 16268$	2^{10960}
t_{big}	$1/t_{\mathrm{tiny}}$	2^{-94}	$1/t_{\mathrm{tiny}}$	2^{-700}	$1/t_{\mathrm{tiny}}$	2^{-10960}

$S_{\mathrm{big}} = S_{\mathrm{big}}^{h} + S_{\mathrm{big}}^{\ell}$, and $S_{\mathrm{tiny}} = S_{\mathrm{tiny}}^{h} + S_{\mathrm{tiny}}^{\ell}$. As the expression

$$\frac{1}{t_{\mathrm{tiny}}^2}\left(S_{\mathrm{tiny}}^{h} + S_{\mathrm{tiny}}^{\ell}\right) + \left(S_{\mathrm{med}}^{h} + S_{\mathrm{med}}^{\ell}\right) + \frac{1}{t_{\mathrm{big}}^2}\left(S_{\mathrm{big}}^{h} + S_{\mathrm{big}}^{\ell}\right)$$

cannot be used directly, we operate on a case-by-case basis according to whether BIG, MED, and TINY contain elements or not. The strategy for obtaining the norm from S_{big}, S_{med}, and S_{tiny} is described by Algorithm 12.

Algorithm 12 Obtaining the norm from S_{big}, S_{med}, and S_{tiny} without spurious overflow or underflow.

if BIG is nonempty **then**
  if $S^h_{\text{med}} < \text{minmed}^2 u^2 / t^2_{\text{big}}$ **or** $S^h_{\text{big}} > \text{maxmed}^2 \cdot t^2_{\text{big}} / u^3$ **then**
    return $\frac{1}{t_{\text{big}}} \cdot \text{SQRTDWtoFP}(S^h_{\text{big}}, S^\ell_{\text{big}}) = t_{\text{tiny}} \cdot \text{SQRTDWtoFP}(S^h_{\text{big}}, S^\ell_{\text{big}})$
  else
    compute $\hat{\chi} = \text{SloppyDWPlusDW}(t_{\text{tiny}} S^h_{\text{big}}, t_{\text{tiny}} S^\ell_{\text{big}}, t_{\text{big}} S^h_{\text{med}}, t_{\text{big}} S^\ell_{\text{med}})$
    return $\sqrt{t_{\text{tiny}}} \cdot \text{SQRTDWtoFP}(\hat{\chi})$
  end if
else
  if MED is nonempty **then**
    if $S^h_{\text{tiny}} < \text{minmed}^2 u^2 / t^2_{\text{big}}$ **or** $S^h_{\text{med}} > \text{maxmed}^2 \cdot t^2_{\text{big}} / u^3$ **then**
      return $\text{SQRTDWtoFP}(S^h_{\text{med}}, S^\ell_{\text{med}})$
    else
      compute $\hat{\chi} = \text{SloppyDWPlusDW}(t_{\text{tiny}} S^h_{\text{med}}, t_{\text{tiny}} S^\ell_{\text{med}}, t_{\text{big}} S^h_{\text{tiny}}, t_{\text{big}} S^\ell_{\text{tiny}})$
      return $\sqrt{t_{\text{big}}} \cdot \text{SQRTDWtoFP}(\hat{\chi})$
    end if
  else
    return $t_{\text{big}} \cdot \text{SQRTDWtoFP}(S^h_{\text{tiny}}, S^\ell_{\text{tiny}})$
  end if
end if

Let us explain Algorithm 12. The various cases can be represented by triplets $(a, b, c) \in \{0, 1\}^3$, where 0 means "empty" and 1 means "nonempty" for BIG, MED, and TINY, respectively. For instance, "$(1, 0, 1)$" means "MED is empty, and BIG and TINY are nonempty". In all generality, there are eight cases to consider, but this number can be reduced thanks the following remarks.

- As soon as BIG is not empty,[9] we can neglect the elements of S_{tiny}. So Case $(1, 1, 1)$ reduces to Case $(1, 1, 0)$, and Case $(1, 0, 1)$ reduces to Case $(1, 0, 0)$. Indeed, in these cases, we have $\sum_{i=0}^{n-1} a_i^2 > \text{maxmed}^2$, and $\sum_{a_i \in \text{TINY}} a_i^2 \leq (n-1) \cdot \text{minmed}^2$. Hence,

$$\sum_{a_i \in \text{TINY}} a_i^2 < \frac{(n-1) \cdot \text{minmed}^2}{\text{maxmed}^2} \cdot \sum_{i=0}^{n-1} a_i^2 < 2^p \cdot \frac{\text{minmed}^2}{\text{maxmed}^2} \cdot \sum_{i=0}^{n-1} a_i^2. \tag{27}$$

  Using (24), the term $2^p \cdot \text{minmed}^2 / \text{maxmed}^2 = 2^{-2e_{\max}+3p+2}$ bounds the relative error committed by neglecting the elements of TINY in the summation. From (25), we deduce that it is less than or equal to u^3.

[9] As pointed out by Graillat et al., there is no need to preliminarily check whether BIG is empty or not. One progressively accumulates sums of squares in two registers, initially dedicated to the elements of MED and TINY, and as soon as an element of BIG is met, the accumulation of the terms of TINY is abandoned, and the very same register is now used for accumulating the elements of BIG.

We therefore easily obtain

$$\left| \left(S_{\text{med}} + S_{\text{big}}/t_{\text{big}}^2\right) - \sum_{i=0}^{n-1} a_i^2 \right| \leq u^2 \cdot (\nu + u) \cdot \sum_{i=0}^{n-1} a_i^2. \tag{28}$$

Therefore $\sqrt{S_{\text{med}} + S_{\text{big}}/t_{\text{big}}^2}$ will be a good approximation to $\sqrt{\sum_{i=0}^{n-1} a_i^2}$ (the error of that approximation will be given later on). Hence we will compute

$$\sqrt{S_{\text{med}} + S_{\text{big}}/t_{\text{big}}^2}. \tag{29}$$

- Saying that MED is empty is equivalent to saying that $S_{\text{med}} = 0$. So there is no need to develop Case $(1, 0, 0)$ further provided that what we do on Case $(1, 1, 0)$ is still correct when $S_{\text{med}} = 0$.
- Likewise, saying that TINY is empty is equivalent to saying that $S_{\text{tiny}} = 0$. So there is no need to develop Case $(0, 1, 0)$ further provided that what we do on Case $(0, 0, 1)$ is still correct when $S_{\text{tiny}} = 0$.

We are therefore left with only four cases to consider: $(1, 1, 0)$, $(0, 1, 1)$, $(0, 0, 1)$, and $(0, 0, 0)$.

1. **If BIG is nonempty (Case (1, 1, 0)),** the computation must be carried on without underflows or overflows. More precisely, concerning underflow, we must make sure that no term becomes less than $2^{e_{\min}+p}$, otherwise it could not be represented accurately by a double-word number.

   - If $S_{\text{med}}^h < \text{minmed}^2 u^2/t_{\text{big}}^2$ then $S_{\text{med}} < \text{minmed}^2 u^2/t_{\text{big}}^2$ (because the bound is a FP number), hence, $S_{\text{med}} < (S_{\text{big}}/t_{\text{big}}^2)u^2$. Therefore, the term S_{med} can be neglected in front of the term $S_{\text{big}}/t_{\text{big}}^2$. More precisely,

     $$\left| S_{\text{big}}/t_{\text{big}}^2 - \sum_{i=0}^{n-1} a_i^2 \right| \leq u^2 \cdot \left(1 + \nu + u + u^2\nu + u^3\right) \cdot \sum_{i=0}^{n-1} a_i^2, \tag{30}$$

     and one can return

     $$\frac{1}{t_{\text{big}}} \cdot \text{SQRTDWtoFP}(S_{\text{big}}^h, S_{\text{big}}^\ell) = t_{\text{tiny}} \cdot \text{SQRTDWtoFP}(S_{\text{big}}^h, S_{\text{big}}^\ell).$$

   - If $S_{\text{big}}^h > \text{maxmed}^2 \cdot t_{\text{big}}^2/u^3$ then $S_{\text{big}} > \text{maxmed}^2 \cdot t_{\text{big}}^2/u^3$. Also, Lemma 3.4 implies $S_{\text{med}} \leq n \cdot \text{maxmed}^2 < \text{maxmed}^2/u$, therefore $S_{\text{med}} < (S_{\text{big}}/t_{\text{big}}^2)u^2$, and, as previously, (30) holds and one can return

     $$t_{\text{tiny}} \cdot \text{SQRTDWtoFP}(S_{\text{big}}^h, S_{\text{big}}^\ell).$$

- If $S_{\text{med}}^h \geq \text{minmed}^2 u^2/t_{\text{big}}^2$ and $S_{\text{big}}^h \leq \text{maxmed}^2 \cdot t_{\text{big}}^2/u^3$, then $S_{\text{med}} \geq (\text{minmed}^2 u^2/t_{\text{big}}^2)(1-u)$ and $S_{\text{big}} \leq \text{maxmed}^2 \cdot t_{\text{big}}^2(1+u)/u^3$. Consider

$$\chi = S_{\text{big}}/t_{\text{big}} + t_{\text{big}} S_{\text{med}} = t_{\text{tiny}} S_{\text{big}} + t_{\text{big}} S_{\text{med}}.$$

The number χ can be computed without underflow or overflow:

(a) **Overflow:** we have $t_{\text{big}} S_{\text{med}} \leq t_{\text{big}} \cdot n \cdot \text{maxmed}^2 \leq t_{\text{big}} \cdot \Omega$, and

$$\begin{aligned} t_{\text{tiny}} S_{\text{big}} &\leq t_{\text{tiny}} \text{maxmed}^2 t_{\text{big}}^2 (1+u)/u^3 \\ &\leq t_{\text{big}} \left(\text{maxmed}^2/u\right) \cdot (1+u)/u^2 \\ &\leq t_{\text{big}} \cdot \Omega \cdot (1+u)/u^2. \end{aligned}$$

Therefore,

$$\begin{aligned} \chi &\leq \Omega \cdot t_{\text{big}} \cdot \left(1 + (1+u)/u^2\right) \\ &\leq \text{maxmed} \cdot \left(1 + u + u^2\right) \cdot 2^{2p} \\ &< 2^{(3p+e_{\max})/2} \cdot \left(1 + u + u^2\right), \end{aligned}$$

and since $u \leq 1/32$ implies $1 + u + u^2 < \sqrt{2}$ we deduce $\chi < 2^{(3p+e_{\max}+1)/2}$, which is less than or equal to $2^{e_{\max}} < \Omega$ since $3p + 1 \leq e_{\max}$.

(b) **Underflow:** we have $t_{\text{tiny}} S_{\text{big}} > S_{\text{big}}$, therefore the term $t_{\text{tiny}} S_{\text{big}}$ is larger than minmed^2, which is larger than $\eta^2 = 2^{e_{\min}+p}$. Using (5f), we also have

$$\begin{aligned} t_{\text{big}} S_{\text{med}} &\geq (\text{minmed}^2 \cdot t_{\text{tiny}}) \cdot u^2 \cdot (1-u) \\ &\geq (\text{minmed}^3/\alpha) \cdot u^2 \cdot (1-u) \\ &\geq 2^{3(e_{\min}+p)/2 - (e_{\min}-p+1) - 2p - 1} \\ &\geq 2^{e_{\min}/2 + p/2 - 2}, \end{aligned}$$

and (25) implies $e_{\min}/2 + 3p/2 \leq 0$. Hence

$$t_{\text{big}} S_{\text{med}} \geq 2^{e_{\min}+2p-2} \geq 2^{e_{\min}+p}.$$

Therefore, it suffices to compute χ in double-word arithmetic by summing $t_{\text{big}} \cdot (S_{\text{med}}^h, S_{\text{med}}^\ell)$ and $t_{\text{tiny}} \cdot (S_{\text{big}}^h, S_{\text{big}}^\ell)$ by the means of SloppyDWPlusDW (Algorithm 5). If we call $\hat{\chi}$ the computed result, namely

$$\hat{\chi} = \text{SloppyDWPlusDW}(t_{\text{tiny}} S_{\text{big}}^h, t_{\text{tiny}} S_{\text{big}}^\ell, t_{\text{big}} S_{\text{med}}^h, t_{\text{big}} S_{\text{med}}^\ell),$$

we obtain $|\hat{\chi} - \chi| \leq 3u^2\chi$. Combined with (28) this gives

$$\left|\hat{\chi} - t_{\text{big}} \sum_{i=0}^{n-1} a_i^2\right| \leq u^2 \cdot \left(\nu + 3 + u + 3u^2\nu + 3u^3\right) \cdot \sum_{i=0}^{n-1} a_i^2. \tag{31}$$

we then take the square-root R of $\hat{\chi}$ by the means of SQRTDWtoFP (Algorithm 9), and multiply R by $\sqrt{t_{\text{tiny}}}$ (this last multiplication is errorless since t_{tiny} is an *even* power of two).

2. **If BIG is empty, and MED and TINY are nonempty (Case $(0, 1, 1)$),** we need to compute

$$\sqrt{S_{\text{tiny}}/t_{\text{tiny}}^2 + S_{\text{med}}}$$

without underflows or overflows. Note that this can be rewritten

$$(1/t_{\text{tiny}}) \cdot \sqrt{S_{\text{tiny}} + S_{\text{med}}/t_{\text{big}}^2}. \tag{32}$$

The square-root part in (32) is exactly as (29) (with S_{med} replaced by S_{tiny} and S_{big} replaced by S_{med}). Furthermore, the terms S_{tiny}, S_{med} and S_{big} have the same bounds. Therefore the reasoning is exactly as previously (the error bounds are slightly smaller because we no longer have the error term due to neglecting TINY), and we obtain:

- if $S_{\text{tiny}}^h < \text{minmed}^2 u^2/t_{\text{big}}^2$ or $S_{\text{med}}^h > \text{maxmed}^2 \cdot t_{\text{big}}^2/u^3$ then we can return

$$\text{SQRTDWtoFP}(S_{\text{med}}^h, S_{\text{med}}^\ell);$$

- otherwise, we can compute $\chi = S_{\text{med}}/t_{\text{big}} + t_{\text{big}}S_{\text{tiny}} = t_{\text{tiny}}S_{\text{med}} + t_{\text{big}}S_{\text{tiny}}$ in double-word arithmetic with one call to SloppyDWPlusDW, take its square-root R by the means of SQRTDWtoFP, and multiply R by $\sqrt{t_{\text{big}}}$.

3. **If BIG and MED are empty (Case $(0, 0, 1)$),** then we return $t_{\text{big}} \cdot \text{SQRTDWtoFP}(S_{\text{tiny}}^h, S_{\text{tiny}}^\ell)$, and the error bound of Theorem 4.5 applies.

4. **If BIG and MED and TINY are empty (Case $(0, 0, 0)$),** then we return 0. Note that considering this case is important as Algorithm 9 requires a special treatment for a null input.

Table 6 gives the value of the comparison constants $\text{minmed}^2 u^2/t_{\text{big}}^2$ and $\text{maxmed}^2 \cdot t_{\text{big}}^2/u^3$ (assuming t_{big} is the largest allowed value in Table 5) needed by the algorithm.

Table 6: Value of the comparison constants $\mathrm{minmed}^2 u^2/t_{\mathrm{big}}^2$ and $\mathrm{maxmed}^2 \cdot t_{\mathrm{big}}^2/u^3$ (assuming t_{big} is the largest allowed value, or equivalently t_{tiny} is the smallest allowed value) needed by the algorithm, for the binary32, binary64, and binary128 formats of IEEE 754-2019.

format	binary32 (with $t_{\mathrm{tiny}} = 2^{98}$)	binary64 (with $t_{\mathrm{tiny}} = 2^{590}$)	binary128 (with $t_{\mathrm{tiny}} = 2^{8360}$)
$\mathrm{minmed}^2 u^2/t_{\mathrm{big}}^2$	2^{46}	2^{106}	2^{226}
$\mathrm{maxmed}^2 t_{\mathrm{big}}^2/u^3$	2^{-22}	2^{-51}	2^{-111}

4.2.3 Final error bound

We finally obtain,

Theorem 4.7. *If $n \leq 1/(4u) - 2$ and $u \leq 1/32$, and if the sequential algorithm (Algorithm 10) is used for the summation of squares, then our algorithm computes $\sqrt{\sum_{i=0}^{n-1} a_i^2}$ with an error bounded by*

$$\left(\frac{1}{2} + \frac{(8n+16)\,u + \frac{37}{2}u^2}{4-2u}\right) \mathrm{ulp}\left(\sqrt{\sum_{i=0}^{n-1} a_i^2}\right),$$

without any risk of spurious underflow or overflow.

Theorem 4.8. *If $n \leq 1/u$, $k+m \leq 1/(4u) - 2$ and $u \leq 1/32$, and if the blockwise algorithm (Algorithm 11) is used for the summation of squares, with k blocks of m elements, where $km = n$, then our algorithm computes $\sqrt{\sum_{i=0}^{n-1} a_i^2}$ with an error bounded by*

$$\left(\frac{1}{2} + \frac{(3.116 + 2(k+m))\,u + 9.77u^2}{1-\frac{u}{2}}\right) \mathrm{ulp}\left(\sqrt{\sum_{i=0}^{n-1} a_i^2}\right),$$

without any risk of spurious underflow or overflow.

The proofs of Theorems 4.7 and 4.8 are given in the supplementary materials.

In all practical cases, if the decomposition in blocks is balanced enough, constraint "$k+m \leq 1/(4u)-2$" in Theorem 4.8 is less strong than constraint $n \leq 1/u$. More precisely, assume we choose $k = m = \lceil\sqrt{n}\rceil$ (i.e., we possibly extend the vector $(a_0, a_1, a_2, \ldots, a_{n-1})$ with additional zeros if n is not a perfect square). Constraint $n \leq 1/u$ implies $\lceil\sqrt{n}\rceil < 1/\sqrt{u} + 1$, so that $k + m < 2/\sqrt{u} + 2$, and one easily checks that for all precisions $p \geq 7$ (i.e., $u \leq 1/128$), $2/\sqrt{u} + 2 < 1/(4u) - 2$.

4.2.4 Examples

Table 7 gives the error bounds we obtain in the cases $n = 12$ and $n = 10000$, for the bfloat16, binary32, and binary64 formats.

Table 7: Maximum possible values of n assuming sequential and blockwise summations, and final error, expressed in ulp $\left(\sqrt{\sum_{i=0}^{n-1} a_i^2}\right)$ for $n = 12$ and $n = 10000$, for the bfloat16, binary32, and binary64 formats.

format	bfloat16	binary32	binary64
max. value of n for seq. summation	62	4,194,302	2.2517×10^{15}
max. value of n for blockw. summation	256	16,777,216	9.0072×10^{15}
error bound for $n = 12$ with seq. summation	0.610 ulp	0.5000017 ulp	$0.5\underbrace{0000000000000}_{13 \text{ zeros}}311$ ulp
error bound for $n = 12$ for blockw. summation with $k = 3, m = 4$	0.5672 ulp	0.50000103 ulp	$0.5\underbrace{0000000000000}_{13 \text{ zeros}}191$ ulp
error bound for $n = 10000$ with seq. summation	N/A	0.5012 ulp	$0.5\underbrace{0000000000}_{10 \text{ zeros}}223$ ulp
error bound for $n = 10000$ with blockw. summation with $k = m = 100$	N/A	0.5000241 ulp	$0.5\underbrace{000000000000}_{12 \text{ zeros}}45$ ulp

5 Verification of some of our proofs by the Coq proof assistant

Our proofs, given in the "supplementary material", are quite long and computational. This is not fully satisfactory: long and tedious proofs are seldom read, with the potential risk that errors remain unnoticed. To overcome this difficulty, we have verified a large number of these proofs using the Coq proof assistant with the help of the Flocq library [10, 11]. This verification gives a high confidence in the correctness of the algorithms and error bounds, allowing future users to use them with trust.

The Coq proof assistant (see for example [3]) is based on the computation of inductive constructions. It is an interactive proof construction tool, which provides a language of tactics to help the user to build new proofs. The Flocq library, based on Coq, deals with the arithmetic of floating-point numbers. It provides different models for representing floating-point numbers and a bunch of proofs of related properties. For example, the Flocq library was used by Boldo et al. to prove the correctness of floating-point passes of the verified C compiler of CompCert [12]. Two of us recently used Flocq to consolidate knowledge on the error bounds of basic algorithms for the arithmetic of double-word numbers [39].

Concerning this present work on Euclidean norms, we verified in Coq all the proofs concerning the square-root (Section 3), the proofs of the sum-of-squares algorithms of Section 4.1, and Theorem 4.5 (which establishes the correctness of our algorithms and error bounds for the Euclidean norm assuming no underflow or overflow occurs). For that purpose, we also built a verified proof of the Lange and Rump theorem (Theorem 1.4).

Note that, for the rounding function RN, we have chosen *not to specify* the tie-breaking rule in

our formal proofs (the default in IEEE 754 arithmetic, as said above, is ties-to-even). This makes our proofs more general: they are for instance still valid with the less frequently used (but also specified by IEEE 754) "ties-to-away" tie-breaking rule.

Our formal proofs can be downloaded from the supplementary materials.

6 Experiments

We have checked our algorithm and compared it with other solutions from the literature on two aspects: accuracy and speed. We decided to design two different implementations of our algorithm: an implementation in Julia, used for accuracy testings, and a C implementation, used for performance evaluation and comparison. The reason for this choice is simple: the versatility of Julia makes it much easier to play with different precisions. However, it is exactly the same algorithm that was implemented in both environments. The Julia and C files can be downloaded from the supplementary materials.

6.1 Accuracy testings

We have implemented the Naive algorithm, Hammarling's algorithm, the algorithm of Graillat et al. [18] as well as our algorithm (with the blockwise summation, i.e., Algorithm 11, with $k = 2$) in the Julia programming language, and we have measured the errors obtained with randomly chosen input arrays of increasing sizes. The elements of the arrays are floating-point numbers. Their significands are uniformly generated between 1 and $2 - 2u$, and their exponents are uniformly generated in a range defined by one of the following *profiles*:

- NORMAL_INPUT: the exponents are uniformly generated between $e_{\min}$ and $e_{\max} - p$.
- MED_INPUT: the exponents are uniformly generated between $\lceil (e_{\min} + p)/2 \rceil$ and $\lfloor (e_{\max} - p)/2 \rfloor$.

The reason for choosing the former exponent range is that we want to avoid *non*-spurious underflow or overflow, and the reason for choosing the latter is to avoid *any* underflow or overflow. See Table 1 and Table 4 for the values of the various parameters. We have performed all experiments in the binary32 and binary64 floating-point formats of the IEEE 754 Standard. We initially wanted to perform experiments in the binary128 format too. For that purpose we wanted to use the GNU libquadmath library, which provides a software implementation of the binary128 format, but we discovered that its square-root function (`sqrtq`) is not correctly rounded. As this is a requirement for the accuracy of the algorithm of Graillat et al. [18] as well as ours, the tests could not be performed in that format. Finally, enclosures of the exact results are computed using Johansson's Arb

library [31], with enough accuracy to allow one to determine the correct roundings of the exact results unambiguously.

Tables 8 (NORMAL_INPUT) and 9 (MED_INPUT) present the maximum obtained relative errors, and the percentage of faithfully and correctly rounded results, as well as the percentage of overflows. In Table 8, the Naive algorithm always overflows except in a few cases when $n = 16$. As each input value has probability ≈ 0.25 of being an element of BIG, there is a probability of approximately 0.75^n that the array has no elements of BIG, which is around 0.01 when $n = 16$, around 0.00000001 when $n = 64$, and even less when n is larger. In Table 9, no overflow occurs, so that we can now compare the Naive algorithm with the other ones. Beware: the "100%" in the "correct rounding" columns of the table can be a bit misleading: these results do not show that our algorithm always returns correctly rounded values (indeed, *it cannot*), but that incorrectly rounded values are extremely unlikely in practice. Incorrectly rounded values are much more frequent with the algorithm of Graillat et al. [18], which is not surprising: that algorithm was designed to always return *faithfully rounded* values, and our tests show that this is indeed the case for all the input arrays we have built. The two tables show that rounding errors in the Naive and the Hammaring algorithms clearly increase with n. This is almost not the case with the algorithm of Graillat et al., as well as ours, in the range of sizes between 16 and 4096.

6.2 Performance evaluation

Our experiments have been performed in a similar way to the ones that were reported in [18]. As said above, we implemented our algorithm in the C language for evaluating its performance and for comparing it to four other algorithms:

- the "*Naive* algorithm" is the straightforward implementation of (1). It does not prevent spurious overflow/underflow from happening, and can, in rare cases, be inaccurate when underflows occur;
- a vectorized version of the Naive algorithm;
- *Hammarling's algorithm*, presented in Section 2.1, as implemented by Graillat et al. [18];
- *Graillat et al.'s* algorithm presented in [18].

All five algorithms are compared using the IEEE-754 binary64 format.

We performed our tests on four different machines, that we designate by the microarchitecture they are based on: *ARM ThunderX2*, *Intel Coffee Lake*, *AMD Zen 2* and *Intel Skylake*.

In Section 4.1.2 we have considered possible values of the number of blocks in the blockwise summation (i.e., variable k in Algorithm 11), in order to minimize the error bound. We have seen (see Property 4.4) that even $k = 2$ is a significant improvement, in terms of accuracy, compared to the

Table 8: accuracy test results for vector coefficients chosen in the NORMAL_INPUT profile.

algorithm	n	using binary32				using binary64			
		spurious overflow	relative error / u	rounding faithful	rounding correct	spurious overflow	relative error / u	rounding faithful	rounding correct
Naive	16	94 %	1.8746	99 %	87 %	98 %	1.2317	100 %	95 %
	64	100 %	NaN	0 %	0 %	100 %	NaN	0 %	0 %
	256	100 %	NaN	0 %	0 %	100 %	NaN	0 %	0 %
	1024	100 %	NaN	0 %	0 %	100 %	NaN	0 %	0 %
	4096	100 %	NaN	0 %	0 %	100 %	NaN	0 %	0 %
Hammarling	16	0 %	2.8028	94 %	75 %	0 %	2.5222	98 %	91 %
	64	0 %	3.5873	90 %	59 %	0 %	2.8819	94 %	75 %
	256	0 %	4.4652	88 %	57 %	0 %	3.3164	89 %	58 %
	1024	0 %	7.2813	73 %	43 %	0 %	4.7827	84 %	53 %
	4096	0 %	10.6339	38 %	19 %	0 %	6.4483	76 %	45 %
Graillat et al.	16	0 %	1.4232	100 %	92 %	0 %	1.4351	100 %	97 %
	64	0 %	1.4460	100 %	87 %	0 %	1.4481	100 %	92 %
	256	0 %	1.4664	100 %	87 %	0 %	1.4586	100 %	87 %
	1024	0 %	1.4635	100 %	86 %	0 %	1.4770	100 %	87 %
	4096	0 %	1.4900	100 %	86 %	0 %	1.4624	100 %	87 %
Ours	16	0 %	0.9969	100 %	100 %	0 %	0.9916	100 %	100 %
	64	0 %	0.9962	100 %	100 %	0 %	0.9961	100 %	100 %
	256	0 %	0.9969	100 %	100 %	0 %	0.9975	100 %	100 %
	1024	0 %	0.9986	100 %	100 %	0 %	0.9962	100 %	100 %
	4096	0 %	0.9986	100 %	100 %	0 %	0.9985	100 %	100 %

Table 9: Accuracy test results for vector coefficients chosen in the MED_INPUT profile.

		using binary32				using binary64			
algorithm	n	spurious overflow	relative error / u	rounding faithful	rounding correct	spurious overflow	relative error / u	rounding faithful	rounding correct
Naive	16	0 %	2.4618	99 %	82 %	0 %	1.8825	99 %	93 %
	64	0 %	3.1185	97 %	73 %	0 %	2.3321	99 %	83 %
	256	0 %	5.0218	87 %	56 %	0 %	3.5571	97 %	73 %
	1024	0 %	8.2327	59 %	32 %	0 %	5.1627	89 %	59 %
	4096	0 %	14.7189	15 %	7 %	0 %	8.0136	65 %	37 %
Hammarling	16	0 %	3.1054	91 %	63 %	0 %	2.7093	96 %	84 %
	64	0 %	3.8794	89 %	58 %	0 %	3.0554	91 %	64 %
	256	0 %	4.9390	83 %	51 %	0 %	4.1357	87 %	55 %
	1024	0 %	9.0934	57 %	31 %	0 %	6.2451	81 %	51 %
	4096	0 %	16.7972	16 %	8 %	0 %	8.5663	63 %	35 %
Graillat et al.	16	0 %	1.4743	100 %	88 %	0 %	1.4291	100 %	95 %
	64	0 %	1.4478	100 %	87 %	0 %	1.4722	100 %	89 %
	256	0 %	1.4388	100 %	87 %	0 %	1.4468	100 %	87 %
	1024	0 %	1.4363	100 %	87 %	0 %	1.4845	100 %	87 %
	4096	0 %	1.3335	100 %	87 %	0 %	1.4833	100 %	86 %
Ours	16	0 %	0.9986	100 %	100 %	0 %	0.9919	100 %	100 %
	64	0 %	0.9994	100 %	100 %	0 %	0.9953	100 %	100 %
	256	0 %	0.9987	100 %	100 %	0 %	0.9935	100 %	100 %
	1024	0 %	0.9932	100 %	100 %	0 %	0.9974	100 %	100 %
	4096	0 %	0.9242	100 %	100 %	0 %	0.9982	100 %	100 %

Table 10: The four systems on which we performed our experiments.

machine	CPU	ISA	SIMD	k
ARM ThunderX2	Cavium ThunderX2	ARM v8.1	Neon	2
Intel Coffee Lake	Intel Core i7-8700	x86-64	AVX2	4
AMD Zen 2	AMD EPYC 7282	x86-64	AVX2	4
Intel Skylake	Intel Xeon Gold 6136	x86-64	AVX512	8

sequential summation. Now, for a binary64 implementation, if we reason in terms of performance, the best choice is the maximum number of binary64 FPNs that fit in an SIMD vector. That number varies across the different extensions considered.

The main characteristics of the four used architectures are summarized in Table 10. In this table, we indicate for each system the processor name, the name of the instruction set architecture (column "ISA"), the name of the SIMD extension (column "SIMD") that was used to compile or to program the algorithms, and the chosen number k of blocks in the blockwise summation (taken equal to the number of binary64 FPNs that fit in an SIMD vector).

We retrieved the code of Graillat et al. [18] from

`https://www.christoph-lauter.org/faithfulnorm.tgz`,

and we directly used the plain C code they provide for the "*Naive*" and "*Hammarling*" (called Netlib in their code) algorithms. They also provide an implementation of their Euclidean norm algorithm using intrinsics functions for manipulating AVX2 vectors. In particular, they use these intrinsics functions to make the inner-loop of their algorithm branch-free by using componentwise masking operations.

We have used the same technique for implementing our algorithm, but we have added a small intermediate library to facilitate porting the code to different SIMD extensions. That library contains a type `vec_t` for the SIMD vectors, whose definition depends on the targeted extension. For example, when the code is compiled for the AVX2 SIMD extension, `vec_t` is an alias for the type `__m256d`, and the componentwise addition of two SIMD vectors is defined by

```
inline vec_t vec_add(vec_t v1, vec_t v2) {
  return _mm256_add_pd(v1, v2);
}
```

On the other hand, when the code is compiled for the Neon extension of the ARM architecture, `vec_t` is now an alias for the `float64x2_t` type, and the componentwise addition of two vectors becomes

```
inline vec_t vec_add(vec_t v1, vec_t v2) {
  return vaddq_f64(v1, v2);
}
```

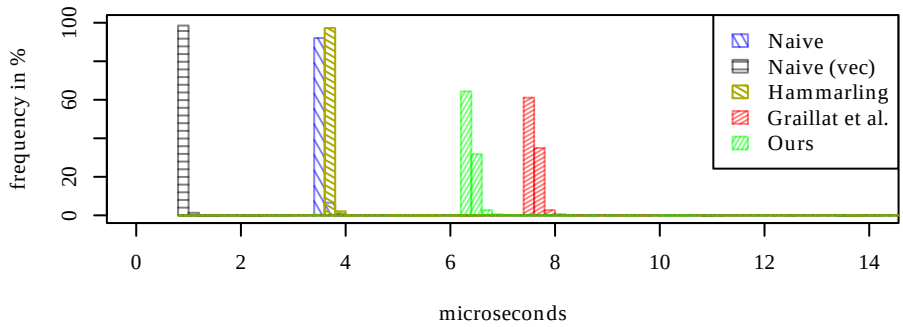

Figure 3: Repartition of the observed timings on Intel Coffee Lake for the MED_INPUT profile.

Note that an `__m256d` vector gathers 4 binary64 numbers, while a `float64x2_t` contains 2 binary64 numbers. Hence, the code we wrote is parameterized by a macro constant `vec_len` that takes for value 8, 4 or 2 depending on the targeted SIMD extension.

We used this small library for implementing our algorithm, and we also used it to re-implement the algorithm proposed by Graillat et al. [18]: on an AVX2 (with FMA) platform, the code we obtain is exactly the same as the one they wrote, but this allowed us to port it easily to the AVX512 and ARM Neon extensions.

The benchmark program of Graillat et al. [18] generates series of random numbers according to a specific *profile*, and gives statistics on the time taken by the different algorithms. Three different profiles are considered here:

- NORMAL_INPUT and MED_INPUT generate floating-point numbers as already explained in Subsection 6.1.
- SPURIOUS_OVERFLOW selects floating-point numbers whose exponent are uniformly chosen between $\lceil e_{\max}/2 \rceil$ and $e_{\max} - p$. As a consequence, squares of the inputs overflow while the exact result is in the normal range.

Table 11 presents the timings obtained on these various systems. The vectorized Naive algorithm is, as expected, the fastest algorithm (but less accurate and prone to spurious overflow/underflow). On Intel and AMD systems, our algorithm is generally slightly faster than Graillat et al.'s algorithm; Hammarling's algorithm is as fast or faster with the MED_INPUT profile, while slower with the NORMAL_INPUT profile. We note that spurious overflows do not slow down the computations significantly. Results on the ARM architecture are quite different: timings do not depend on the input profile, and Hammarling's algorithm is consistently faster.

To give more insights into the timings reported in the previous table, where only average values and standard deviations are given, we present in Figures 3 and 4 the histograms of the timings measured on the Intel Coffee Lake system, with input vectors of size $n = 4096$. With the MED_INPUT

Table 11: Timing comparisons of five algorithms to compute the Euclidean norm, for three different array sizes and three different profiles of inputs (the three profiles are abbreviated by MED_INP, NRM_INP, and SPUR_OVR). For each entry, the mean value and standard deviation of a population of 100,000 runs are given. All times are given in microseconds.

		Intel Coffee Lake (AVX2) @3.2 GHz			*AMD Zen2* (AVX2) $\leq$3.2 GHz		
algorithm	n	MED_INP	NRM_INP	SPUR_OVR	MED_INP	NRM_INP	SPUR_OVR
Naive	256	0.2(0.2)	0.2(0.1)	0.2(0.2)	0.2(0.0)	0.4(0.0)	0.2(0.0)
	1024	0.9(0.3)	0.9(0.3)	0.9(0.3)	1.0(0.0)	1.5(0.1)	1.0(0.0)
	4096	3.6(0.6)	3.6(0.7)	3.6(0.6)	3.9(0.1)	5.8(0.1)	3.9(0.1)
Naive (vec)	256	0.1(0.1)	0.1(0.1)	0.1(0.1)	0.1(0.0)	0.1(0.0)	0.1(0.0)
	1024	0.3(0.2)	0.3(0.2)	0.3(0.2)	0.4(0.0)	0.4(0.0)	0.4(0.0)
	4096	1.0(0.3)	1.0(0.3)	1.0(0.3)	1.6(0.1)	1.6(0.1)	1.6(0.0)
Hammarling	256	0.3(0.1)	0.8(0.4)	0.3(0.2)	0.5(0.0)	1.2(0.0)	0.4(0.0)
	1024	1.0(0.4)	3.1(0.8)	1.0(0.3)	1.7(0.1)	4.4(0.2)	1.7(0.1)
	4096	3.7(0.5)	12.0(1.2)	3.7(0.6)	6.5(0.1)	17.5(0.2)	6.5(0.1)
Graillat et al.	256	0.5(0.3)	0.5(0.2)	0.5(0.3)	0.6(0.0)	0.6(0.0)	0.6(0.0)
	1024	1.9(0.5)	2.0(0.5)	1.9(0.5)	2.1(0.1)	2.1(0.1)	2.1(0.3)
	4096	7.6(0.8)	7.6(0.8)	7.6(0.8)	8.3(0.1)	8.3(0.1)	8.3(0.1)
Ours	256	0.5(0.3)	0.5(0.3)	0.5(0.2)	0.5(0.0)	0.5(0.0)	0.5(0.0)
	1024	1.7(0.5)	1.7(0.5)	1.7(0.4)	1.8(0.1)	1.8(0.1)	1.8(0.1)
	4096	6.4(0.7)	6.4(0.7)	6.4(0.8)	6.7(0.1)	6.7(0.1)	6.7(0.1)
		Intel Skylake (AVX512) @3.0 GHz			*ARM ThunderX2* (Neon) $\leq$2.2 GHz		
algorithm	n	MED_INP	NRM_INP	SPUR_OVR	MED_INP	NRM_INP	SPUR_OVR
Naive	256	0.4(0.0)	0.6(0.1)	0.4(0.0)	0.7(0.0)	0.7(0.0)	0.7(0.0)
	1024	1.5(0.1)	2.3(0.2)	1.5(0.0)	2.8(0.1)	2.8(0.1)	2.8(0.1)
	4096	6.1(0.1)	9.3(0.4)	6.1(0.1)	11.2(0.2)	11.2(0.3)	11.2(0.3)
Naive (vec)	256	0.1(0.0)	0.1(0.0)	0.1(0.0)	0.4(0.0)	0.4(0.0)	0.4(0.0)
	1024	0.2(0.0)	0.2(0.1)	0.2(0.0)	1.4(0.1)	1.4(0.2)	1.4(0.1)
	4096	0.8(0.0)	0.8(0.1)	0.8(0.0)	5.6(0.1)	5.6(0.1)	5.6(0.1)
Hammarling	256	0.5(0.0)	1.3(0.3)	0.5(0.0)	1.7(0.1)	1.6(0.1)	1.7(0.1)
	1024	1.6(0.1)	5.1(0.6)	1.6(0.1)	6.4(0.3)	6.4(0.3)	6.4(0.3)
	4096	6.2(0.1)	20.3(1.2)	6.2(0.1)	25.1(0.4)	25.1(0.4)	25.1(0.4)
Graillat et al.	256	0.5(0.0)	0.5(0.0)	0.5(0.0)	3.5(0.1)	3.5(0.1)	3.5(0.1)
	1024	1.8(0.1)	1.8(0.1)	1.8(0.1)	13.8(0.2)	13.8(0.3)	13.8(0.2)
	4096	6.8(0.1)	6.8(0.1)	6.8(0.1)	55.5(0.4)	55.5(0.5)	55.5(0.5)
Ours	256	0.6(0.0)	0.6(0.0)	0.6(0.0)	3.5(0.1)	3.5(0.1)	3.5(0.1)
	1024	1.7(0.1)	1.7(0.1)	1.7(0.1)	13.5(0.4)	13.5(0.5)	13.5(0.4)
	4096	6.2(0.1)	6.2(0.2)	6.1(0.1)	53.1(0.4)	53.1(0.4)	53.1(0.4)

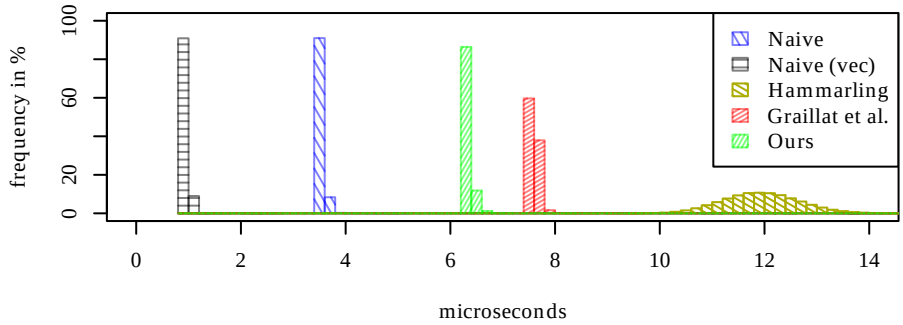

Figure 4: Repartition of the observed timings on Intel Coffee Lake for the NORMAL_INPUT profile.

profile (Fig. 3), the few observed slight variations are probably due to operating system hazards. It is noticeable that Hammarling's algorithm is almost as fast as the nonvectorized Naive algorithm in this profile. However, with the NORMAL_INPUT profile (Fig. 4), the performances of Hammarling's algorithm are clearly degraded and are more scattered. This may be due to frequent changes of the scaling factor needed to prevent overflows/underflows with this profile.

It is clear from Table 11, that the performances of the various algorithms depend much on the platform being used. However, in any case, these experiments show that our algorithm performs quite nicely compared to the other algorithms while being more accurate.

Conclusion

We have presented algorithms that make it possible to compute Euclidean norms of vectors very accurately, and without spurious underflows or overflows, even when these vectors are large. Our tests show that the performance of the "blockwise" version of our algorithm is in general slightly better than the performance of the slightly less accurate algorithm of Graillat et al., and in general better than the significantly less accurate Hammarling's algorithm. Our work on the computation of Euclidean norms also led us to obtain results on double-word arithmetic that can be of interest in other areas:

- we have shown that when the operands are positive, the DWPlusFP algorithm has relative error bound u^2, and that bound is asymptotically optimal;
- we have shown the asymptotic optimality of the already known error bound $3u^2$ for the SloppyDWPlusDW algorithm when the operands are positive;
- we have introduced new algorithms for computing square-roots of double-word numbers (SQRTDWtoDW and SQRTDWtoFP), given an asymptotically optimal relative error bound

for the first one, and an error bound in ulps for the second one.

Furthermore, we have formally proven the critical parts of our algorithms.

Interestingly enough, avoiding spurious underflows and overflows and computing more accurately comes at a reasonable cost: the experiments presented in Section 6.2 show that our algorithm is never more than two times slower than the naive algorithm.

Acknowledgement

This work is partly sponsored by the Agence Nationale de la Recherche (ANR) Nuscap project (https://anr.fr/Projet-ANR-20-CE48-0014).

References

[1] E. Anderson. Algorithm 978: Safe scaling in the level 1 blas. *ACM Trans. Math. Softw.*, 44(1), jul 2017.

[2] N. H. F. Beebe. *The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library.* Springer, 2017.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer, 2004.

[4] J. L. Blue. A portable fortran program to find the euclidean norm of a vector. *ACM Trans. Math. Softw.*, 4(1):15–23, mar 1978.

[5] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula. Semantics for exact floating point operations. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 22–26. IEEE Computer Society Press, Los Alamitos, CA, June 1991.

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

[7] S. Boldo, S. Graillat, and J.-M. Muller. On the robustness of the 2sum and fast2sum algorithms. *ACM Trans. Math. Softw.*, 44(1), July 2017.

[8] S. Boldo, C. Q. Lauter, and J.-M. Muller. Emulating round-to-nearest-ties-to-zero "augmented" floating-point operations using round-to-nearest-ties-to-even arithmetic. *IEEE Transactions on Computers*, 70(7):1046 – 1058, july 2021.

[9] S. Boldo, C. Lelay, and G. Melquiond. Formalization of real analysis: A survey of proof assistants and libraries. *Mathematical Structures in Computer Science*, 26(7):1196–1233, 2016.

[10] S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252, 2011.

[11] S. Boldo and G. Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System.* Elsevier, 2017.

[12] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[13] C. F. Borges. Algorithm 1014: An improved algorithm for hypot(x,y). *ACM Trans. Math. Softw.*, 47(1), December 2020.

[14] F. de Dinechin, C. Lauter, J.-M. Muller, and S. Torres. On Ziv's rounding test. *ACM Transactions on Mathematical Software*, 39(4), 2013.

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

[16] J. Demmel and H. D. Nguyen. Numerical reproducibility and accuracy at exascale. In *21st IEEE Symposium on Computer Arithmetic*, pages 235–237, 2013.

[17] M. Fasi, N. J. Higham, F. Lopez, T. Mary, and M. Mikaitis. Matrix Multiplication in Multiword Arithmetic: Error Analysis and Application to GPU Tensor Cores. working paper or preprint, January 2022.

[18] S. Graillat, C. Lauter, P.T.P. Tang, N. Yamanaka, and S. Oishi. Efficient Calculations of Faithfully Rounded L2-Norms of n-Vectors. *ACM Transactions on Mathematical Software*, 41(4):1–20, October 2015.

[19] R. J. Hanson and T. Hopkins. Remark on algorithm 539: A modern fortran reference implementation for carefully computing the euclidean norm. *ACM Trans. Math. Softw.*, 44(3), December 2017.

[20] J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Trans. Program. Lang. Syst.*, 18(2):139–174, 1996.

[21] G. Henry, P.T.P. Tang, and A. Heinecke. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 69–76, 2019.

[22] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In *ARITH-16*, pages 155–162, June 2001.

[23] Y. Hida, X.S. Li, and D.H. Bailey. C++/fortran-90 double-double and quad-double package, release 2.3.17. Accessible electronically at `http://crd-legacy.lbl.gov/~dhbailey/mpdist/`, March 2012.

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

[25] N. J. Higham and T. Mary. A new approach to probabilistic rounding error analysis. *SIAM Journal on Scientific Computing*, 41(5):A2815–A2835, 2019.

[26] T. E. Hull, T. F. Fairgrieve, and P. T. P. Tang. Implementing complex elementary functions using exception handling. *ACM Transactions on Mathematical Software*, 20(2):215–244, June 1994.

[27] IEEE. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019)*. July 2019.

[28] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux. Midpoints and exact points of some algebraic functions in floating-point arithmetic. *IEEE Transactions on Computers*, 60(2):228–241, February 2011.

[29] C.-P. Jeannerod, J.-M. Muller, and P. Zimmermann. On various ways to split a floating-point number. In *25th IEEE Symposium on Computer Arithmetic, Amherst, MA, USA*, pages 53–60, June 2018.

[30] C.-P. Jeannerod and S. M. Rump. On relative errors of floating-point operations: optimal bounds and applications. *Mathematics of Computation*, 87:803–819, 2018.

[31] F. Johansson. Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66(8):1281–1292, 2017.

[32] M. Joldes, J.-M. Muller, and V. Popescu. Tight and rigorous error bounds for basic building blocks of double-word arithmetic. *ACM Trans. Math. Softw.*, 44(2), oct 2017.

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

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

[35] T. Kouya. Acceleration of lu decomposition supporting double-double, triple-double, and quadruple-double precision floating-point arithmetic with avx2. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, pages 54–61, 2021.

[36] 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. Technical Report 45991, Lawrence Berkeley National Laboratory, 2000. `https://www.netlib.org/lapack/lawnspdf/lawn149.pdf`.

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

[38] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic, 2nd edition.* Birkhäuser Boston, 2018.

[39] J.-M. Muller and L. Rideau. Formalization of double-word arithmetic, and comments on "Tight and rigorous error bounds for basic building blocks of double-word arithmetic". *ACM Transactions on Mathematical Software*, 46(1):1–24, March 2022.

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

[41] E. J. Riedy and J. Demmel. Augmented arithmetic operations proposed for IEEE-754 2018. In *25th IEEE Symposium on Computer Arithmetic, Amherst, MA, USA*, pages 45–52, June 2018.

[42] S. M. Rump and M. Lange. Error estimates for the summation of real numbers with application to floating-point summation. *BIT Numerical Mathematics*, 57:927–941, 2017.

[43] S. M. Rump and M. Lange. Faithfully Rounded Floating-Point Computation. *ACM Transactions on Mathematical Software*, 46(3):1–20, jul 2020.