Efficient Calculations of Faithfully Rounded l_2-Norms of n-Vectors

STEF GRAILLAT and CHRISTOPH LAUTER, Sorbonne Universités, UPMC Univ Paris 06
PING TAK PETER TANG, Intel Corporation
NAOYA YAMANAKA, Teikyo Heisei University/CREST, JST
SHIN'ICHI OISHI, Waseda University

In this article, we present an efficient algorithm to compute the faithful rounding of the l_2-norm of a floating-point vector. This means that the result is accurate to within 1 bit of the underlying floating-point type. This algorithm does not generate overflows or underflows spuriously, but does so when the final result calls for such a numerical exception to be raised. Moreover, the algorithm is well suited for parallel implementation and vectorization. The implementation runs up to 3 times faster than the netlib version on current processors.

CCS Concepts: • **Mathematics of computing** → **Arbitrary-precision arithmetic**; • **Computing methodologies** → *Linear algebra algorithms*

Additional Key Words and Phrases: Floating-point arithmetic, error-free transformations, faithful rounding, 2-norm, underflow, overflow

ACM Reference Format:
Stef Graillat, Christoph Lauter, Ping Tak Peter Tang, Naoya Yamanaka, and Shin'ichi Oishi. 2015. Efficient calculations of faithfully rounded l_2-norms of n-vectors. ACM Trans. Math. Softw. 41, 4, Article 24 (October 2015), 20 pages.
DOI: http://dx.doi.org/10.1145/2699469

1. INTRODUCTION

Computing the l_2-norm $\|\mathbf{x}\|_2 = \sqrt{\sum_{j=1}^{n} x_j^2}$ of a vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ is prevalent in scientific and engineering applications. This operation is part of the first (lowest) level of the Basic Linear Algebra Subroutine (BLAS1). The simplicity of the formula $\sqrt{\sum_{j=1}^{n} x_j^2}$ is misleading. Summing the squares can cause unwarranted (spurious) overflows or underflows in many instances when, in fact, $\|\mathbf{x}\|_2$ is well within the normal range of the working-precision floating-point arithmetic. Common implementations, such as the

---

Part of this work was done while Professor Shin'ichi Oishi was holding a position of Visiting Professor at UMPC and while Ping Tak Peter Tang was visiting UPMC. Stef Graillat was supported by a CNRS/JSPS Exchange Scientist grant.
Authors' addresses: S. Graillat, Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France and CNRS, UMR 7606, LIP6, F-75005, Paris, France and with CNRS at Laboratoire LIP (CNRS, ENS Lyon, Inria, UCBL), 6 allée d'Italie 69364 Lyon cedex 07, France; email: stef.graillat@lip6.fr; C. Lauter is with Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France and CNRS, UMR 7606, LIP6, F-75005, Paris, France; email: christoph.lauter@lip6.fr; P. T. P. Tang is with Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054, USA; email: peter.tang@intel.com; N. Yamanaka is with Faculty of Modern Life, Teikyo Heisei University 4-21-2 Nakano, Nakano, Tokyo, 164-8530, Japan and CREST, Japan Science and Technology Agency 4-1-8 Honcho, Kawaguchi, Saitama, 332-0012, Japan; email: n.yamanaka@thu.ac.jp; S. Oishi is with Faculty of Science and Engineering, Waseda University, 3-4-1 Okubo, Tokyo 169-8555 Japan and CREST, JST; email: oishi@waseda.jp.

DOI: http://dx.doi.org/10.1145/2699469

public version of LAPACK [Anderson et al. 1999] released by `netlib`[1] essentially compute the l_2-norm as $\widehat{x} \times \|\mathbf{x}/\widehat{x}\|_2$, where $\widehat{x}$ is $\max_j |x_j|$. That implementation requires n divisions in total, which is significantly more expensive than the naïve formula would suggest. Spurious exceptions aside, accuracy is also an issue with implementations that rely on accumulation of squares in working-precision arithmetic. In the worst-case scenario, the last $\log_2(n)$ bits of the binary floating point result could be corrupted. Equivalently, the last $\log_{10}(n)$ digits of the result, when displayed in decimal, could be corrupted. Improving the accuracy of l_2-norm computation enhances the qualities of the larger computational tasks relying on it. Moreover, a highly accurate l_2-norm improves the chances of obtaining reproducible numerical results should the l_2-norm computation be done in parallel, with threads or vector-floating-point (SIMD) instructions[2].

In this article, we present a new division-free l_2-norm algorithm that is amenable to straightforward parallel implementations. We prove that the algorithm always returns a faithfully rounded result, to be defined rigorously later. For now (and informally), this means that the result is accurate to within one bit of the underlying floating-point type. The algorithm also reports overflow and underflow faithfully, a property to be defined later. Loosely speaking, these exceptions are triggered only when the true value $\|\mathbf{x}\|_2$ calls for the event. On current processors (with AVX extensions), our implementation runs at least as fast as the `netlib` version, and up to three times faster when the IEEE754 fused-multiply-add instruction is available.

There are two main features of our algorithm. First, it accumulates the squares, x_j^2, using a pair of floating-point numbers, providing essentially double the underlying floating-point precision. Our technique is similar to the addition operator in the double-double library [Li et al. 2002] but at almost twice the speed by exploiting the nonnegative nature of sum of squares. We provide a rigorous analysis of the accuracy properties of our accumulation process. Second, we eliminate all spurious exceptions by scaling the input data, but without using division. The technique is a "binning" method and is similar to the one proposed in Blue [1978]: the vector elements x_j are grouped—that is, binned—into small, medium, and large inputs, such that, after appropriate scaling up or down, their squares in each bin can be computed without spurious overflow or underflow. However, we improve the binning technique in that [Blue 1978] requires three bins and we use only two bins here, resulting in economy of registers usage and performance improvement. The accumulation and binning are amenable to straightforward parallel implementations. Our reference implementation uses data parallelism through SIMD instructions, but adding thread parallelism is straightforward. Our technique does not incur any memory overhead; in particular, each vector element x_j is read only once, the same way it would be in other approaches.

We organize the rest of the article as follows. Section 2 defines the key technical terms. We formulate our main problem and state our objectives. We present and establish the main theorem in Section 3. The essence is that, if $\|\mathbf{x}\|_2^2$ is computed to enough accuracy as a floating-point pair as "leading"-plus-"correction," then a standard IEEE-conforming square root on the "leading" part yields a faithfully rounded l_2-norm. Section 4 presents a serial and parallel l_2-norm algorithm without binning. In the absence of exceptions, we prove the numerical properties of these two algorithms, which will lay the foundation for the actual binned algorithms that provide the spurious-exception-free property. Section 5 presents the binned versions and the proofs of the faithful rounding and spurious-exception-free nature. Section 6 shows numerical and performance test results to corroborate with the theoretical analysis presented.

[1]www.netlib.org.
[2]Clearly, an l_2-norm routine that returns the correctly rounded floating-point result is always reproducible.

2. BACKGROUND

Throughout this article, we consider a specific IEEE754 binary floating-point type. Let $\mathbb{F}$ denote the entire set of finite values in this type, identified by three parameters $\boldsymbol{\varepsilon}$, $e_{\min}$, and $e_{\max}$:

$$\mathbb{F} = \{\pm 2^{e+1} m\boldsymbol{\varepsilon} \mid m \in \mathbb{N}, e_{\min} \leq e \leq e_{\max}, 0 \leq m\boldsymbol{\varepsilon} < 1\}.$$

For example, $(\boldsymbol{\varepsilon}, e_{\min}, e_{\max})$ is $(2^{-24}, -126, 127)$ for the binary32 format, and $(2^{-53}, -1022, 1023)$ for the binary64 format. Denote the smallest and largest positive normalized numbers by $F_{\text{small}} = 2^{e_{\min}}$ and $F_{\text{large}} = 2^{e_{\max}+1}(1 - \boldsymbol{\varepsilon})$. We assume $\boldsymbol{\varepsilon} \leq 2^{-24}$ throughout this article.

Closely related to $\mathbb{F}$ is the set $\mathbb{F}^\sharp$ where no upper limit of the exponent is imposed:

$$\mathbb{F}^\sharp = \{\pm 2^{e+1} m\boldsymbol{\varepsilon} \mid m \in \mathbb{N}, e_{\min} \leq e, 0 \leq m\boldsymbol{\varepsilon} < 1\}.$$

It is clear that $\mathbb{F} \cap [0, 2^{e_{\max}+1}) = \mathbb{F}^\sharp \cap [0, 2^{e_{\max}+1})$. In particular, numbers $x \in \mathbb{F}^\sharp$ where $0 < |x| < 2^{e_{\min}}$ are also denormalized.

For any real number $\alpha \in \mathbb{R}$, $\circ(\alpha)$ is the IEEE round-to-nearest-even function that maps any finite real number to $\mathbb{F}^\sharp$, $\circ : \mathbb{R} \to \mathbb{F}^\sharp$. The rounding $\circ(\alpha)$ of a real number α is the number in $\mathbb{F}^\sharp$ that is closest to α, with a tie broken by choosing $\circ(\alpha)$ to have an even mantissa (least-significant bit being zero). A crucial property of $\circ(\alpha)$ is best stated in terms of the ulp (units of last place) function defined as follows. For all $\alpha \in \mathbb{R}$,

$$\text{ulp}(\alpha) = \begin{cases} 2^{e+1}\boldsymbol{\varepsilon} & \text{if } |\alpha| \in [2^e, 2^{e+1}), e \geq e_{\min}, \\ 2^{e_{\min}+1}\boldsymbol{\varepsilon} & \text{otherwise.} \end{cases}$$

Note that this definition of ulp reflects the floating-point arithmetic properties in the denormalized range. For any real number α and integer k such that both $|\alpha|$ and $|2^k\alpha| \geq F_{\text{small}}$, then $\text{ulp}(2^k\alpha) = 2^k\text{ulp}(\alpha)$, and $\text{ulp}(\alpha) \leq 2|\alpha|\boldsymbol{\varepsilon}$ where equality $\text{ulp}(\alpha) = 2|\alpha|\boldsymbol{\varepsilon}$ holds if and only if $\log_2(|\alpha|)$ is an integer. It is easy to see that $\circ(\alpha)$ is at worst half a unit of last place away from α: For a real number α, $|\alpha| \in [2^e, 2^{e+1}]$, then $|\circ(\alpha) - \alpha| \leq \text{ulp}(\alpha)/2$. Note that this holds even for $e < e_{\min}$.

The clipping function clip maps floating-point numbers in $\mathbb{F}^\sharp$ to $\mathbb{F} \cup \{-\infty, +\infty\}$: For all $x \in \mathbb{F}^\sharp$,

$$\text{clip}(x) = \begin{cases} x & \text{for } |x| < 2^{e_{\max}+1}, \\ \frac{x}{|x|}\infty & \text{for } |x| \geq 2^{e_{\max}+1}. \end{cases}$$

The four basic IEEE arithmetic operations $\oplus, \ominus, \otimes, \oslash$ can be defined in terms of the rounding and clipping functions:

$$a \circledast b = \text{clip}(\circ(a * b))$$

for any $* \in \{+, -, \times, /\}$ and $a, b \in \mathbb{F}$ (with $b \neq 0$ if the operation "$*$" is division). Similarly, the IEEE square root function `sqrt` is defined as $\texttt{sqrt}(x) = \circ(\sqrt{x})$ for all $x \in \mathbb{F} \cap [0, \infty)$. The clipping function is not needed here, as $\sqrt{x} \leq \sqrt{F_{\text{large}}}$.

For any $\alpha \in \mathbb{R}$, we define α's faithful set of floating-point numbers $\diamond(\alpha)$ as follows. $\diamond(\alpha)$ is the singleton $\{\alpha\}$ if $\alpha \in \mathbb{F}^\sharp$. Otherwise, $\diamond(\alpha)$ is the set of two numbers in $\mathbb{F}^\sharp$ that are closest to α from below and above. In other words,

$$\diamond(\alpha) = \left\{ \max_y\{y \in \mathbb{F}^\sharp \mid y \leq \alpha\},\ \min_y\{y \in \mathbb{F}^\sharp \mid y \geq \alpha\} \right\}.$$

Clipping the set $\diamond(\alpha)$ is by definition the set consisting of the clipping of each of the elements in $\diamond(\alpha)$: $\text{clip}(\diamond(\alpha)) = \{\text{clip}(a) | a \in \diamond(\alpha)\}$.

This article presents a parallelizable and division-free algorithm `AccuNrm2` such that for all vectors $\mathbf{x}$ of practical length, $\texttt{AccuNrm2}(\mathbf{x}) \in \text{clip}(\diamond(\|\mathbf{x}\|_2))$ where $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, $x_j \in \mathbb{F}$. We call this numerical property a faithful rounding of $\|\mathbf{x}\|_2$.

Note that our definition of faithful rounding is slightly stronger than that in Muller et al. [2010], differing with it only in the case when $\|\mathbf{x}\|_2 \geq 2^{e_{\max}+1}$. In this case, our definition requires that $+\infty$ be returned, while the definition in Muller et al. [2010] allows either $+\infty$ or F_{large} to be a faithful rounding.

In addition to computing a faithfully rounded numerical value, `AccuNrm2`($\mathbf{x}$) also reports overflow and underflow faithfully in the following sense. Given an implementation `G`($\mathbf{x}$) of a function $g(\mathbf{x})$, we say `G`($\mathbf{x}$) reports

(1) overflow faithfully if:
  —`G`($\mathbf{x}$) never reports an overflow when $|g(\mathbf{x})| \leq F_{\text{large}}$.
  —`G`($\mathbf{x}$) always reports an overflow when $|g(\mathbf{x})| \geq 2^{e_{\max}+1}$.
(2) underflow faithfully:
  —When $|g(\mathbf{x})| \geq F_{\text{small}}$, `G`($\mathbf{x}$) never reports underflow.
  —When $0 < |g(\mathbf{x})| \leq F_{\text{small}} - 2^{e_{\min}+1}\boldsymbol{\varepsilon}$, then
    —$|$`G`$(\mathbf{x})| \leq F_{\text{small}} - 2^{e_{\min}+1}\boldsymbol{\varepsilon}$ always,
    —if underflow is unmasked, `G`($\mathbf{x}$) reports underflow.
    —if underflow is masked and $g(\mathbf{x}) \in \mathbb{F}$, then `G`($\mathbf{x}$) $= g(\mathbf{x})$ and does not report underflow.

We emphasize that achieving faithful rounding is nontrivial. One can show that $\circ(\sqrt{\circ(\sigma)}) \in \diamond(\|\mathbf{x}\|_2)$, where $\sigma = \sum_j x_j^2 = \mathbf{x}^T\mathbf{x}$ is the exact sum of squares (or inner product). This says that the correctly rounded square root of the correctly rounded sum of squares is a faithfully rounded l_2-norm. Nevertheless, computing the correctly rounded sums of squares is very expensive [Ogita et al. 2005; Rump et al. 2008a, 2008b]. On the other hand, examples exist in which $\circ(\sqrt{S}) \notin \diamond(\|\mathbf{x}\|_2)$ for some $S \in \diamond(\sigma)$. That is, computing the σ to only slightly worse than the correctly rounded sum of squares can cause unfaithful rounding. Our algorithm, in essence, computes a floating-point number $S \approx \sigma$ very accurately and yet efficiently. Theorem 3.3 gives a condition on the accuracy of S that guarantees $\circ(\sqrt{S})$ to be a faithful rounding of $\|\mathbf{x}\|_2$. The fact $\circ(\sqrt{\circ(\sigma)}) \in \diamond(\|\mathbf{x}\|_2)$ alluded to earlier follows trivially from Theorem 3.3 as well.

The fundamental task is that of computing $\sigma = \sum_j x_j^2$ accurately. As we will see later, it is possible to transform this computation into a sum without loss of information (no rounding error). The problem is now transformed into the accurate computation of a sum. There is abundant literature about floating-point summation (see Higham [2002, chap. 4], Knuth [1998], Ogita et al. [2005], Rump et al. [2008a, 2008b], Rump [2009], and Zhu and Hayes [2009, 2010] and references therein). For our purpose, we only need an accurate summation algorithm whose precision is doubled because the entries are nonnegative numbers. For such a precision, a straightforward adaptation of the algorithm `Sum2` [Ogita et al. 2005] is very efficient since it requires $8(n-1)$ floating-point operations (flops), where n is the size of the vector. The resulting sum has a relative error no more than on the order of $n^2\boldsymbol{\varepsilon}^2$. Another choice is to use the double-double arithmetic presented in Li et al. [2002]. The resulting sum is much more accurate, having a relative error no more than $2n\boldsymbol{\varepsilon}^2/(1 - 2n\boldsymbol{\varepsilon}^2)$. The cost, however, is $20(n-1)$ flops. The difference between the two algorithms comes from the "renormalization steps" that are present in the double-double library. As will be shown later, an error bound in the order of $n\boldsymbol{\varepsilon}^2$, as opposed to $n^2\boldsymbol{\varepsilon}^2$, is crucial if faithful rounding is to be guaranteed for a general vector length n. Therefore, we devise an algorithm that is faster than, but of comparable accuracy to, the addition operator in the double-double library. Instead of performing two renormalization steps in the addition of two double-double numbers, we perform only one renormalization step in a careful manner. The resulting error bound is $3n\boldsymbol{\varepsilon}^2/(1 - 3n\boldsymbol{\varepsilon}^3)$, at a cost of $11(n-1)$ flops, which is almost twice as fast as $20(n-1)$ flops.

A naïve computation of the l_2-norm can cause spurious overflow and underflow. The netlib library addresses the overflow issue, but not that of underflow. Spurious underflow can cause significant performance degradation. Blue [1978] uses three bins to eliminate spurious overflows and underflows. Accuracies of the netlib and Blue algorithms are comparable, in which close to $\log_{10}(n)$ digits can be corrupted in the worst case. In summary, our algorithm is accurate to within 1 binary bit, free from spurious over/underflows, and run fasters than the netlib version.

We state without proofs the following elementary facts about floating-point arithmetic. These facts and notations will be used freely in the subsequent sections.

—Let $a, b \in \mathbb{F}^\sharp$. The absolute error bound
$$|\circ(a \text{ op } b) - (a \text{ op } b)| \le \text{ulp}(a \text{ op } b)/2$$
holds for all op $\in \{+, -, \times, /\}$ (excluding division by zero). The relative error bound
$$\circ(a \text{ op } b) = (a \text{ op } b)(1+\delta), \quad |\delta| \le \boldsymbol{\varepsilon}$$
holds for op $\in \{+, -\}$. For op $\in \{\times, /\}$, this relative error bound holds if $F_{\text{small}} \le |a \text{ op } b|$ (excluding division by zero).

—Rounding to nearest is monotonic: Given real numbers $\alpha, \beta \in \mathbb{R}$, $\alpha \le \beta$ implies $\circ(\alpha) \le \circ(\beta)$.

—For $a, b \in \mathbb{F}$, if $S = \circ(a + b) \in \mathbb{F}$, then $a + b - S \in \mathbb{F}$. In particular, the value $s = a + b - S$ satisfies the relationship $S + s = a + b$ and $\circ(S + s) = S$. Furthermore, S and s can be computed from a and b by a sequence of instructions involving $\oplus$ and $\ominus$ (see e.g., Knuth [1998, Theorem B, page 236] and Dekker [1971]). We encapsulate these facts by the two functions `TwoSum`(a, b) and `FastTwoSum`(a, b). They deliver S and s, where $S + s = a + b$ and $\circ(S + s) = S$. The former handles general a and b and requires 6 floating-point operations; the latter requires only 3 floating-point operations, but relies on the assumption that $|a| \ge |b|$. Mapping a, b to S, s is usually called an error-free transformation, as $a + b = S + s$ exactly. Our algorithm and its implementation use both functions.

—Similar to error-free transformations for sum, we have error-free transformations for product. For $a, b \in \mathbb{F}$, if $|ab| \ge F_{\text{small}}/\boldsymbol{\varepsilon}$, and $P = \circ(a \times b) \in \mathbb{F}$, then $a \times b - P \in \mathbb{F}$. In particular, the value $p = a \times b - P$ satisfies the relationship $P + p = a \times b$ and $\circ(P + p) = P$. P and p can be computed from a and b with the IEEE754-2008 fused multiply-add (FMA) instruction: $P := a \otimes b$ and $p := \circ(a \times b - P)$. Alternatively, one can use a sequence of $\oplus$, $\ominus$, and $\otimes$ instructions, as outlined in Dekker [1971]. We denote the exact product function that returns P and p by `TwoProd`(a, b). Our implementation uses either the FMA-based sequence or the sequence given in Dekker [1971], depending on whether the FMA instruction is supported on the available hardware.

3. MAIN THEOREM

The goal is to compute $\|\mathbf{x}\|_2 = \sqrt{\sigma}$, $\sigma = \sum_{j=1}^{n} x_j^2$ faithfully. Our algorithm is built on the core case, when σ is within the "normal" range of $[F_{\text{small}}, F_{\text{large}}]$. We first compute an accurate floating-point approximation, S, to σ. The IEEE square root (correctly rounded) $\circ(\sqrt{S})$ is returned as the final result. This section shows that if S is accurate to within a specific threshold, the final result is faithful: $\circ(\sqrt{S}) \in \diamondsuit(\|\mathbf{x}\|_2) = \text{clip}(\diamondsuit(\|\mathbf{x}\|_2))$.

LEMMA 3.1. *Let α, α' be two real numbers in the interval $[2^e, 2^{e+1}]$ for some integer e. If $|\alpha' - \alpha| < \text{ulp}(2^e)/2$, then $\circ(\alpha') \in \diamondsuit(\alpha)$.*

Proof. Let $a = \circ(\alpha')$. We first note that a and α are close to one another:

$$\begin{aligned} |a-\alpha| &\leq |a-\alpha'| + |\alpha'-\alpha| \\ &\leq \mathrm{ulp}(2^e)/2 + |\alpha'-\alpha|, \quad \text{because } a = \circ(\alpha') \\ &< \mathrm{ulp}(2^e)/2 + \mathrm{ulp}(2^e)/2, \quad \text{by assumption}, \\ |a-\alpha| &< \mathrm{ulp}(2^e). \end{aligned} \tag{1}$$

To prove $a \in \diamond(\alpha)$, we analyze all of the three possibilities of $a = \alpha$, $a < \alpha$ and $a > \alpha$. The case of $a = \alpha$ is trivial because $a \in \mathbb{F}^\sharp$; therefore, $a \in \{a\} = \diamond(a) = \diamond(\alpha)$.

Consider the case of $a < \alpha$. This means that $a < 2^{e+1}$. Hence, $a \in [2^e, 2^{e+1})$ if $e \geq e_{\min}$, and $a \in [0, 2^{e_{\min}})$ if $e < e_{\min}$. Regardless, the next floating-point number in $\mathbb{F}^\sharp$ that is bigger than a is $a + \mathrm{ulp}(2^e)$. We have

$$\begin{aligned} \min\left\{y \in \mathbb{F}^\sharp | y > a\right\} &= a + \mathrm{ulp}(2^e) \\ &> a + |a-\alpha|, \quad \text{by (1)} \\ &= a + (\alpha - a), \quad \text{because } a < \alpha \\ &= \alpha. \end{aligned}$$

Therefore, while $a < \alpha$, the next floating-point number above a is strictly bigger than α. In other words, $a = \max\{y \in \mathbb{F}^\# | y \leq \alpha\}$. This says that $a \in \diamond(\alpha)$ by definition.

Consider now the final case: $a > \alpha$. Since $a > 2^e$, we either have $a \in (2^e, 2^{e+1}]$ if $e \geq e_{\min}$, or $a \in (0, 2^{e_{\min}}]$ if $e < e_{\min}$. In either case, the next number in $\mathbb{F}^\sharp$ that is smaller than a is $a - \mathrm{ulp}(2^e)$.

$$\begin{aligned} \max\{y \in \mathbb{F}^\sharp | y < a\} &= a - \mathrm{ulp}(2^e) \\ &< a - |a-\alpha|, \quad \text{by (1)} \\ &= a - (a - \alpha), \quad \text{because } a > \alpha. \\ &= \alpha. \end{aligned}$$

Therefore, while $a > \alpha$, the next floating-point number below a is strictly less than α. In other words, $a = \min\{y \in \mathbb{F}^\sharp | y \geq \alpha\}$. This says once again that $a \in \diamond(\alpha)$. The proof is now complete. □

Lemma 3.2. *Let $\sigma \in [F_{\text{small}}, F_{\text{large}}]$ be a real number in the interval $[2^e, 2^{e+1})$. In particular, $e_{\min} \leq e \leq e_{\max}$. Let $S, s \in \mathbb{F}^\sharp$ be such that $\circ(S+s) = S$. If $|(S+s)-\sigma| < \sigma\boldsymbol{\varepsilon}/2$, then $S \in [2^e, 2^{e+1}] \cap [0, F_{\text{large}}]$ and $|s| \leq 2^e\boldsymbol{\varepsilon}$.*

Proof. We will first establish the fact that $S \in [2^e, 2^{e+1}] \cap [0, F_{\text{large}}]$. By assumption,

$$\sigma(1-\boldsymbol{\varepsilon}/2) < S + s < \sigma(1+\boldsymbol{\varepsilon}/2).$$

Since $\sigma \in [2^e, 2^{e+1})$, $e \geq e_{\min}$, we have

$$2^e - \mathrm{ulp}(2^e)/4 = 2^e(1-\boldsymbol{\varepsilon}/2) < S + s < 2^{e+1}(1+\boldsymbol{\varepsilon}/2) = 2^{e+1} + \mathrm{ulp}(2^{e+1})/4.$$

Consequently, $\circ(S+s) \in [2^e, 2^{e+1}]$. Moreover, $S + s < \sigma + \sigma\boldsymbol{\varepsilon}/2$ implies also that $S + s < F_{\text{large}} + \mathrm{ulp}(2^{e_{\max}})/2$, which leads to $\circ(S+s) \leq F_{\text{large}}$. But $S = \circ(S+s)$ by assumption, thus we have established that $S \in [2^e, 2^{e+1}] \cap [0, F_{\text{large}}]$.

Turning now to s, there are only two possibilities: $S+s \in [2^e, 2^{e+1}]$ or $S+s \notin [2^e, 2^{e+1}]$. If $S + s \in [2^e, 2^{e+1}]$, then $|s| = |\circ(S+s) - (S+s)| \leq \mathrm{ulp}(2^e)/2 = 2^e\boldsymbol{\varepsilon}$. Consider now $S + s \notin [2^e, 2^{e+1}]$. But since $S = \circ(S+s)$ and we have established previously that $\circ(S+s) \in [2^e, 2^{e+1}]$, we are left with only two cases:

(1) $S = 2^e$ and $s < 0$: In this case, the inequality $2^e - \mathrm{ulp}(2^e)/4 < S + s$ implies that $s > -\mathrm{ulp}(2^e)/4 = -2^e\boldsymbol{\varepsilon}/2$. Thus $|s| < 2^e\boldsymbol{\varepsilon}/2$.

(2) $S = 2^{e+1}$ and $s > 0$: In this case, the inequality $S + s < 2^{e+1} + \mathrm{ulp}(2^{e+1})/4$ implies that $s < \mathrm{ulp}(2^{e+1})/4 = 2^e\boldsymbol{\varepsilon}$. Thus $|s| < 2^e\boldsymbol{\varepsilon}$.

To review, $|s| \le 2^e\boldsymbol{\varepsilon}$ if $S + s \in [2^e, 2^{e+1}]$ and $|s| < 2^e\boldsymbol{\varepsilon}$ if $S + s \notin [2^e, 2^{e+1}]$. Thus $|s| \le 2^e\boldsymbol{\varepsilon}$ always and the proof is complete. □

THEOREM 3.3. *Let* $\sigma \in [F_{\text{small}}, F_{\text{large}}]$ *be a real number and* $S, s \in \mathbb{F}^\sharp$ *where* $\circ(S+s) = S$. *If* $|(S + s) - \sigma| < \boldsymbol{\varepsilon}\sigma/8$, *then* $\circ(\sqrt{S}) \in \diamond(\sqrt{\sigma})$.

PROOF. We establish this theorem by showing that $\sqrt{\sigma}$ and $\sqrt{S}$ satisfy the conditions for α and α' in Lemma 3.1. The assumption $|(S + s) - \sigma| < \boldsymbol{\varepsilon}\sigma/8$ implies that $S + s > (1-\boldsymbol{\varepsilon}/8)\sigma$. $\circ(S+s) = S$ and that $S+s$ is obviously positive imply that $S \ge (S+s)(1-\boldsymbol{\varepsilon})$. Thus

$$\begin{aligned}
S &> (1-\boldsymbol{\varepsilon})(1-\boldsymbol{\varepsilon}/8)\sigma, \\
&> (1-3\boldsymbol{\varepsilon}/2)\sigma, \quad \text{because } \boldsymbol{\varepsilon} \le 2^{-24} \\
\sqrt{S} &> (1-3\boldsymbol{\varepsilon}/2)\sqrt{\sigma}, \quad \text{because } \sqrt{1-3\boldsymbol{\varepsilon}/2} > 1-3\boldsymbol{\varepsilon}/2, \\
\sqrt{S}+\sqrt{\sigma} &> 2(1-3\boldsymbol{\varepsilon}/4)\sqrt{\sigma}. \qquad (2)
\end{aligned}$$

We derive an upper bound of $|\sqrt{S} - \sqrt{\sigma}|$ as follows.

$$\begin{aligned}
|\sqrt{S}-\sqrt{\sigma}| &= \frac{|S-\sigma|}{\sqrt{S}+\sqrt{\sigma}}, \\
&< \frac{|S-\sigma|}{2\sqrt{\sigma}(1-3\boldsymbol{\varepsilon}/4)}, \quad \text{by (2)} \\
&< \frac{|S-\sigma|}{2\sqrt{\sigma}}(1+\boldsymbol{\varepsilon}), \quad \text{because } (1-3\boldsymbol{\varepsilon}/4)^{-1} < (1+\boldsymbol{\varepsilon}) \\
&< \frac{\boldsymbol{\varepsilon}\sigma/8+|s|}{2\sqrt{\sigma}}(1+\boldsymbol{\varepsilon}), \quad \text{by assumption}, \\
|\sqrt{S}-\sqrt{\sigma}| &< \left(\frac{\boldsymbol{\varepsilon}}{16}\sqrt{\sigma}+\frac{|s|}{2\sqrt{\sigma}}\right)(1+\boldsymbol{\varepsilon}). \qquad (3)
\end{aligned}$$

There is a unique interval of the form $[2^{2e}, 2^{2e+2})$ that contains σ. If σ is in the "left" half: $\sigma \in [2^{2e}, 2^{2e+1})$, then $2e \ge e_{\min}$ and Lemma 3.2 shows that $S \in [2^{2e}, 2^{2e+1}]$ and $|s| \le 2^{2e}\boldsymbol{\varepsilon}$. If σ is in the "right" half: $\sigma \in [2^{2e+1}, 2^{2e+2})$, then $2e+1 \ge e_{\min}$ and Lemma 3.2 shows that $S \in [2^{2e+1}, 2^{2e+2}]$ and $|s| \le 2^{2e+1}\boldsymbol{\varepsilon}$. Summarizing, both $\sqrt{\sigma}$ and $\sqrt{S}$ are in $[2^e, 2^{e+1}]$ with $e \ge e_{\min}$. By virtue of Lemma 3.1, we can establish the fact that $\circ(\sqrt{S}) \in \diamond(\sqrt{\sigma})$ provided we can show $|\sqrt{S} - \sqrt{s}| < \mathrm{ulp}(2^e)/2$. Since $e \ge e_{\min}$, $\mathrm{ulp}(2^e) = 2^{e+1}\boldsymbol{\varepsilon}$, this is equivalent to establishing that $|\sqrt{S} - \sqrt{\sigma}| < (2^e\boldsymbol{\varepsilon})$. We accomplish this by using Equation (3) and the simple case analysis tabulated here:

Case of	$\dfrac{\boldsymbol{\varepsilon}\sqrt{\sigma}}{2^e\boldsymbol{\varepsilon}}$	$\dfrac{\lvert s\rvert}{\sqrt{\sigma}2^{e+1}\boldsymbol{\varepsilon}}$	$\dfrac{\lvert\sqrt{S}-\sqrt{\sigma}\rvert}{2^e\boldsymbol{\varepsilon}} < \left(\dfrac{\boldsymbol{\varepsilon}\sqrt{\sigma}}{2^{e+4}\boldsymbol{\varepsilon}}+\dfrac{\lvert s\rvert}{\sqrt{\sigma}2^{e+1}\boldsymbol{\varepsilon}}\right)(1+\boldsymbol{\varepsilon})$
$\sigma \in [2^{2e}, 2^{2e+1})$	$< \sqrt{2}$	$\le \dfrac{2^{2e}\boldsymbol{\varepsilon}}{2^{2e+1}\boldsymbol{\varepsilon}} = \dfrac{1}{2}$	$< \left(\dfrac{\sqrt{2}}{16}+\dfrac{1}{2}\right)(1+\boldsymbol{\varepsilon}) < 1$
$\sigma \in [2^{2e+1}, 2^{2e+2})$	< 2	$\le \dfrac{2^{2e+1}\boldsymbol{\varepsilon}}{2^{2e+1}\sqrt{2}\boldsymbol{\varepsilon}} = \dfrac{1}{\sqrt{2}}$	$< \left(\dfrac{1}{8}+\dfrac{1}{\sqrt{2}}\right)(1+\boldsymbol{\varepsilon}) < 1$

Clearly, $|\sqrt{S} - \sqrt{\sigma}| < 2^e\boldsymbol{\varepsilon}$ always and $\circ(\sqrt{S}) \in \diamond(\sqrt{\sigma})$, as claimed. □

4. FAITHFUL 2-NORM: CORE CASE

Let $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ be the vector in question. The core algorithm considers the case of normal range in the sense that $\sigma = \mathbf{x}^T\mathbf{x}$ is safely away from overflow and that the least significant bit of each of the x_j^2 does not underflow. More formally, we consider vectors $\mathbf{x}$ in safe range, defined as (1) $\sigma \leq F_{\text{large}}/2$, and (2) for all j, x_j is either 0, or in the range $F_{\text{small}}/\boldsymbol{\varepsilon}^2 \leq x_j^2 \leq F_{\text{large}}$.

We use a data type, double-FP, consisting of two floating-point numbers. We denote them, for example, by $\mathbf{A} = [A, a]$. $\mathbf{A}$ is the double-FP variable, and the actual pairs of floating-point numbers are A and a. They have the characteristics $\circ(A+a) = A$, which means that a is a "tail" part to add extra precision to A. The mathematical sum of the two components represents a value of at least twice the precision of the underlying floating-point number. Here is the core algorithm for computing the sum of squares $\sum_j x_j^2$. We make the crucial distinction between an assignment operation ":=" in an algorithm and the mathematical equality sign "=".

```
function SumOfSquares(x) // Accurate accumulation
    S := [0, 0]
    for j = 1, 2, ..., n do:
        P := TwoProd(x_j, x_j)
        // P = [P, p], P + p = x_j^2 exactly
        S := SumNonNeg(S, P)
    return S
end SumOfSquares

function SumNonNeg(A, B) // [A, a] + [B, b]
// error bound of this operation 3ε^2 (Theorem 4.1)
// A = [A, a], B = [B, b] nonnegative: A + a, B + b ≥ 0
    H := TwoSum(A, B)
    // H = [H, h], H + h = A + B exactly
    c := a ⊕ b      // c = a + b + δ_c
    d := h ⊕ c      // d = h + c + δ_d.
    S := FastTwoSum(H, d)
    // S = [S, s], S + s = H + d exactly
    //see Section 2 for TwoSum and FastTwoSum.
    return S
end SumNonNeg
```

Theorem 3.3 guarantees that if $[S, s]$ returned by SumOfSquares satisfies $|(S+s)-\sigma| < \boldsymbol{\varepsilon}\sigma/8$, then $\circ(\sqrt{S})$ is a faithful rounding of $\sqrt{\sigma} = \|\mathbf{x}\|_2$. Since SumOfSquares is summing n double-FP type, standard error analysis (see, e.g., Chapter 3 of Higham [2002]) shows that the relative error is bounded by $\Delta_{n-1}(\delta)$, where $\Delta_\ell(\delta) = \ell\delta/(1-\ell\delta)$ and δ is the relative error bound on the underlying addition operation, which is the SumNonNeg function. Theorem 4.1 shows that this δ is $3\boldsymbol{\varepsilon}^2$. From that, we can deduce the length limit of $\mathbf{x}$ within which $|(S+s)-\sigma| < \boldsymbol{\varepsilon}\sigma/8$.

THEOREM 4.1. *Let $\mathbf{S} = [S, s]$ be the result from applying* SumNonNeg *on nonnegatives $\mathbf{A} = [A, a]$ and $\mathbf{B} = [B, b]$. Let $\alpha = A + a \geq 0$, $\beta = B + b \geq 0$ denote the exact input values, and $\sigma = \alpha + \beta$ denote the exact sum. If $F_{\text{small}}/\boldsymbol{\varepsilon}^2 \leq \sigma \leq F_{\text{large}}/2$, then $|(S+s)-\sigma| \leq 3\boldsymbol{\varepsilon}^2\sigma$.*

PROOF. The theorem clearly holds if α or β is zero. Moreover, it is clear that SumNonNeg is insensitive to the order of its two input arguments. It suffices, therefore, to consider $\alpha \geq \beta > 0$. There are only two rounding errors in the entire function: $\delta_c = c - (a+b) = \circ(a+b) - (a+b)$ and $\delta_d = d - (h+c) = \circ(h+c) - (h+c)$. More precisely, $S + s = H + d$ and

$$H + d = H + h + a + b + \delta_c + \delta_d = \sigma + \delta_c + \delta_d.$$

Thus $|(S+s)-\sigma| \le |\delta_c|+|\delta_d|$. The rest of the proof establishes the fact that $|\delta_c|+|\delta_d| \le 3\boldsymbol{\varepsilon}^2\sigma$.

We use this fact heavily: For any real number $\gamma \in \mathbb{R}$, $|\circ(\gamma)-\gamma| \le \mathrm{ulp}(\gamma)/2$. Let

$$\alpha = 2^{e_\alpha}(1+f_\alpha),\ \ \beta = 2^{e_\beta}(1+f_\beta),\ \ \text{and}\ \ \sigma = 2^{e_\sigma}(1+f_\sigma),$$

where $0 \le f_\alpha, f_\beta, f_\sigma < 1$.

$$\sigma \ge \alpha \ge \beta \Rightarrow \mathrm{ulp}(\sigma) \ge \mathrm{ulp}(\alpha) \ge \mathrm{ulp}(\beta).$$

Because $|a| \le \mathrm{ulp}(\alpha)/2$, $|a| \le \mathrm{ulp}(\sigma)/2$. Similarly, $|b| \le \mathrm{ulp}(\sigma)/2$. Therefore, $A+B = \sigma-(a+b) \le \sigma+\mathrm{ulp}(\sigma)$, which implies that $\mathrm{ulp}(A+B) \le 2\mathrm{ulp}(\sigma)$.

We note that $|\delta_c| = |\circ(a+b)-(a+b)| \le \mathrm{ulp}(a+b)/2$ and $|a+b| \le \mathrm{ulp}(\sigma)$. But $|a+b| = \mathrm{ulp}(\sigma)$ only when $|a| = |b| = \mathrm{ulp}(\sigma)/2$, which implies $a+b$ is representable exactly in $\mathbb{F}$ and $\delta_c = 0$. When $|a+b| < \mathrm{ulp}(\sigma)$, we have $\mathrm{ulp}(a+b) \le \mathrm{ulp}(\mathrm{ulp}(\sigma/2))$. Hence, using basic properties of the ulp function stated in Section 2,

$$|\delta_c| \le \frac{1}{2}\mathrm{ulp}(\mathrm{ulp}(\sigma/2)).$$

Because $\sigma \ge F_{\mathrm{small}}/\boldsymbol{\varepsilon}^2$, $\mathrm{ulp}(\mathrm{ulp}(\sigma/2)) = \mathrm{ulp}(\sigma)\boldsymbol{\varepsilon} \le 2\sigma\boldsymbol{\varepsilon}^2$. Hence,

$$|\delta_c| \le \sigma\boldsymbol{\varepsilon}^2. \tag{4}$$

We now show that $|\delta_d| \le 2\sigma\boldsymbol{\varepsilon}^2$. Indeed

$$\begin{aligned}
|\delta_d| &= |\circ(h+c)-(h+c)| \\
&\le \frac{1}{2}\mathrm{ulp}(h+c) \\
&\le \frac{1}{2}\mathrm{ulp}(\mathrm{ulp}(A+B)/2+|c|), \\
|\delta_d| &\le \frac{1}{2}\mathrm{ulp}(\mathrm{ulp}(\sigma)+|c|). \qquad (5)
\end{aligned}$$

To complete the estimate on $|\delta_d|$, we analyze $|c|$. There are only two possibilities: either $e_\alpha \ge e_\beta+1$ or $e_\alpha = e_\beta$. We show that each situation leads to $|\delta_d| \le 2\sigma\boldsymbol{\varepsilon}^2$.

Consider the case of $e_\alpha \ge e_\beta+1$. We have $|a| \le \mathrm{ulp}(\sigma)/2$ and $|b| \le \mathrm{ulp}(\sigma)/4$. Therefore,

$$|c| = |\circ(a+b)| \le |a+b|(1+\boldsymbol{\varepsilon}) \le \frac{3}{4}(1+\boldsymbol{\varepsilon})\mathrm{ulp}(\sigma).$$

Equation (5) implies that

$$\begin{aligned}
|\delta_d| &\le \frac{1}{2}\mathrm{ulp}\left(\mathrm{ulp}(\sigma)+\frac{3}{4}(1+\boldsymbol{\varepsilon})\mathrm{ulp}(\sigma)\right) \\
&= \frac{1}{2}\mathrm{ulp}(\mathrm{ulp}(\sigma)), \\
|\delta_d| &\le 2\sigma\boldsymbol{\varepsilon}^2. \qquad (6)
\end{aligned}$$

Consider the case of $e_\alpha = e_\beta$. In this situation, we must have $e_\sigma = e_\alpha+1$ and $\mathrm{ulp}(\alpha) = \mathrm{ulp}(\beta) = \mathrm{ulp}(\sigma)/2$. As a result, $|a|+|b| \le \mathrm{ulp}(\sigma)/2$ and $|c| \le (1+\boldsymbol{\varepsilon})\mathrm{ulp}(\sigma)/2$, thus

$$\begin{aligned}
|\delta_d| &\le \frac{1}{2}\mathrm{ulp}\left(\mathrm{ulp}(\sigma)+\frac{1}{2}(1+\boldsymbol{\varepsilon})\mathrm{ulp}(\sigma)\right) \\
&= \frac{1}{2}\mathrm{ulp}(\mathrm{ulp}(\sigma)), \\
|\delta_d| &\le 2\sigma\boldsymbol{\varepsilon}^2. \qquad (7)
\end{aligned}$$

Equations (4), (6), and (7) together show that $|\delta_c| + |\delta_d| \le 3\sigma\boldsymbol{\varepsilon}^2$, and the theorem is proved. □

THEOREM 4.2. *Let n be the length of a vector* $\mathbf{x}$ *in safe range and* σ *denote* $\sum_j x_j^2$. *Let* `SumOfSquares`($\mathbf{x}$) *return the result* $[S, s]$. *Then*

$$|(S+s) - \sigma| \le \Delta_{n-1}(3\boldsymbol{\varepsilon}^2)\sigma,$$

where $\Delta_\ell(\delta) = \ell\delta/(1-\ell\delta)$. *In particular, if the length n satisfies* $n < ((24+\boldsymbol{\varepsilon})\boldsymbol{\varepsilon})^{-1}$, *then*

$$|(S+s) - \sigma| < \boldsymbol{\varepsilon}\sigma/8.$$

PROOF. From Theorem 4.1 and standard error bound on adding n nonnegative floating-point types [Higham 2002] with an addition operation of relative error bounded by $3\boldsymbol{\varepsilon}^2$,

$$|(S+s) - \sigma| \le \Delta_{n-1}(3\boldsymbol{\varepsilon}^2)\sigma.$$

Because $\Delta_\ell(\delta)$ is an increasing function in ℓ in the range $0 \le \ell < 1/\delta$, for $n < ((24 + 3\boldsymbol{\varepsilon})\boldsymbol{\varepsilon})^{-1}$,

$$\Delta_{n-1}(3\boldsymbol{\varepsilon}^2) < \Delta_n(3\boldsymbol{\varepsilon}^2) < \Delta_L(3\boldsymbol{\varepsilon}^2) = \frac{\boldsymbol{\varepsilon}}{8},$$

where $L = ((24+3\boldsymbol{\varepsilon})\boldsymbol{\varepsilon})^{-1}$. □

That maximum vector length bound $L = ((24+3\boldsymbol{\varepsilon})\boldsymbol{\varepsilon})^{-1}$ corresponds to $L = 699050$ for IEEE754 binary32 and $L \le 3.76 \cdot 10^{14}$ for IEEE754 binary64.

We parallelize `SumOfSquares` in an obvious manner: partition the input vector $\mathbf{x}$ to τ subvectors of roughly equal length. Perform the sum of squares on each subvector in parallel. This parallelism can be realized either at the thread level or data level, the latter using SIMD vector instructions such as SSE or AVX. The partial sums of squares are then accumulated in a serial manner.

function `SumOfSquaresP`($\mathbf{x}$) // Parallel `SumOfSquares`
  Partition $\mathbf{x}$ into τ portions, $\mathbf{x}^{(t)}$, $t = 1, 2, \ldots, \tau$
  // length of each $\mathbf{x}^{(t)}$ is no more than $m = \lceil n/\tau \rceil$.
  $\mathbf{S}^{(t)}$:= `SumOfSquares`($\mathbf{x}^{(t)}$), $t = 1, 2, \ldots, \tau$.
  // In parallel, each $\mathbf{S}^{(t)} = [S^{(t)}, s^{(t)}]$ is a double-FP.
  $\mathbf{S}$:= [0, 0]; $\mathbf{S}$:= `SumNonNeg`($\mathbf{S}, \mathbf{S}^{(t)}$), $t = 1, 2, \ldots, \tau$.
  // In serial, summing the τ partial sums of squares
  // $\mathbf{S} = [S, s]$ at this point; $S + s \approx \sum_j^n x_j^2$.
  return S
end `SumOfSquaresP`

THEOREM 4.3. *Let n be the length of* $\mathbf{x}$ *and* $\mathbf{S} = [S, s]$ *be the result of* `SumOfSquaresP`($\mathbf{x}$) *with* τ *portions and* $m = \lceil n/\tau \rceil$. *Then*

$$|(S+s) - \sigma| \le \Delta_{m+\tau}(3\boldsymbol{\varepsilon}^2)\sigma.$$

In particular,

$$|(S+s) - \sigma| \le \Delta_{n-1}(3\boldsymbol{\varepsilon}^2)\sigma$$

whenever $m + \tau \le n - 1$.

PROOF. We document the two stages of errors with the following notations. Let $\sigma^{(t)} = (\mathbf{x}^{(t)})^T\mathbf{x}^{(t)}$, $t = 1, 2, \ldots, \tau$, and $\sigma = \sum_{t=1}^{\tau} \sigma^{(t)}$ denote the exact partial and exact complete inner products, respectively. Let

$$\begin{array}{ll} \widetilde{\sigma}^{(t)} = S^{(t)} + s^{(t)}, & \text{approximate partials } 1 \le t \le \tau, \\ \widetilde{\sigma} = \sum_{t=1}^{\tau} \widetilde{\sigma}^{(t)}, & \text{exact sum of approximate partials,} \\ \widetilde{\widetilde{\sigma}} = S + s, & \text{approximate sum of approximate partials.} \end{array}$$

For the computed partials, for which we are summing no more than $m = \lceil n/\tau \rceil$ double-FP types, we have

$$|\widetilde{\sigma}^{(t)} - \sigma^{(t)}| \le \Delta_{m-1}(3\boldsymbol{\varepsilon}^2)\,\sigma^{(t)}, \quad t = 1, 2, \ldots, \tau,$$

and

$$\left|\sum_{t=1}^{\tau}(\widetilde{\sigma}^{(t)} - \sigma^{(t)})\right| \le \Delta_{m-1}(3\boldsymbol{\varepsilon}^2) \sum_{j=1}^{\tau} \sigma^{(t)}.$$

This implies that

$$|\widetilde{\sigma} - \sigma| \ \le\ \Delta_{m-1}(3\boldsymbol{\varepsilon}^2)\,\sigma, \tag{8}$$

$$\widetilde{\sigma} \ \le\ (1 + \Delta_{m-1}(3\boldsymbol{\varepsilon}^2))\,\sigma. \tag{9}$$

Similarly,

$$|\widetilde{\widetilde{\sigma}} - \widetilde{\sigma}| \le \Delta_{\tau-1}(3\boldsymbol{\varepsilon}^2)\,\widetilde{\sigma}. \tag{10}$$

Combining Equations (8) through (10),

$$\begin{aligned} \frac{|\widetilde{\widetilde{\sigma}} - \sigma|}{\sigma} &\le \Delta_{\tau-1}(3\boldsymbol{\varepsilon}^2)\,(\widetilde{\sigma}/\sigma) + \Delta_m(3\boldsymbol{\varepsilon}^2) \\ &\le \Delta_{\tau-1}(3\boldsymbol{\varepsilon}^2)\,(1 + \Delta_{m-1}(3\boldsymbol{\varepsilon}^2)) + \Delta_m(3\boldsymbol{\varepsilon}^2) \\ &\le \Delta_\tau(3\boldsymbol{\varepsilon}^2) + \Delta_m(3\boldsymbol{\varepsilon}^2) \qquad (11) \\ &\le \Delta_{m+\tau}(3\boldsymbol{\varepsilon}^2). \qquad (12) \end{aligned}$$

Equation (11) follows from its preceding line as long as $3\boldsymbol{\varepsilon}^2 \le (n+m+2)^{-1}$, and Equation (12) follows from Equation (11) because $\Delta_\ell(3\boldsymbol{\varepsilon}^2) + \Delta_{\ell'}(3\boldsymbol{\varepsilon}^2) \le \Delta_{\ell+\ell'}(3\boldsymbol{\varepsilon}^2)$. Finally, whenever $m + \tau \le n - 1$, $\Delta_{m+\tau}(3\boldsymbol{\varepsilon}^2) \le \Delta_{n-1}(3\boldsymbol{\varepsilon}^2)$. The proof is now complete. □

We remark that the number of threads τ is typically much smaller than the vector length n. In a common scenario, τ equals the number of cores that is in the order of 10 or so. Moreover, threading is beneficial only when there is enough work per thread, implying that $m = n/\tau$ is in the order of 100 or more. Thus, $m + \tau \approx n/\tau$, implying that $m + \tau \le n - 1$. This says that a parallel sum of squares is, in general, more accurate than the serial version. In particular, as long as $n < ((24 + \boldsymbol{\varepsilon})\boldsymbol{\varepsilon})^{-1}$ and $[S, s]$ is obtained with `SumOfSquares` or `SumOfSquaresP`, an IEEE conforming square root evaluation `sqrt`(S) produces a faithfully rounded $\|\mathbf{x}\|_2$ for $\mathbf{x}$ whose elements fall in the core range discussed here.

5. FAITHFUL 2-NORM: GENERAL CASE

That the simple accumulation of $\sigma = \sum_{j=1}^{n} x_j^2$ is susceptible to spurious exceptions can be illustrated by the simple example of $n = 8$, $x_j = \circ(2\sqrt{F_{\text{large}}})$ for $j \le 4$ and $x_j = \circ(\sqrt{F_{\text{small}}}/2)$, $j > 4$. While $\|\mathbf{x}\|_2$ is approximately $4\sqrt{F_{\text{large}}}$, overflows in computing x_j^2 for $j \le 4$ leads to a computed σ of $+\infty$, rendering the final computed l_2-norm completely wrong. This is why general-purpose software such as LAPACK's public release essentially computes $\hat{x}\sqrt{\sum_j (x_j/\hat{x})^2}$, where $\hat{x} = \max_j |x_j|$. While this strategy resolves the spurious overflow problem satisfactorily, spurious underflows can still be triggered. If underflow is masked, spurious underflow is harmless to the final numerical results. Nevertheless, these spurious underflows may significantly degrade performance on computing platforms that handle underflow via a trapping mechanism.

We present here an algorithm that returns a value in $\mathrm{clip}(\diamond(\|\mathbf{x}\|_2))$ and reports overflows and underflows faithfully, as defined in Section 2. Let $\sigma = \sum_j x_j^2$ denote the sum of squares. Our algorithm first computes Z, a faithful rounding of a scaled l_2-norm $\|\widehat{\mathbf{x}}\|_2 = \gamma^{-m/2}\|\mathbf{x}\|_2$, that is, $Z \in \diamond(\|\widehat{\mathbf{x}}\|_2)$. The factor $\gamma^{-m/2}$ is chosen so that γ^m is an even power of 2, $\gamma^{-m/2} \in \mathbb{F}$, and $\|\widehat{\mathbf{x}}\|_2 \in [F_{\mathrm{small}}, F_{\mathrm{large}}]$. We shall describe a way to choose γ and to compute m later. The final result is returned, naturally, as $\gamma^{m/2} \otimes Z$. Theorem 5.1 establishes rigorously that not only does the numerical value $\gamma^{m/2} \otimes Z \in \mathrm{clip}(\diamond(\|\mathbf{x}\|_2))$, but the multiplication also reports overflow and underflow faithfully. The remainder of this section focuses on the computation of $Z \in \diamond(\|\widehat{\mathbf{x}}\|_2)$.

THEOREM 5.1. *Let $\zeta \in [F_{\mathrm{small}}, F_{\mathrm{large}}] \cup \{0\}$ be a real number and Z be a floating-point number, where $Z \in \diamond(\zeta)$. Let t be an integer where $2^t \in \mathbb{F}$. Then the IEEE multiplication $2^t \otimes Z$ satisfies $2^t \otimes Z \in \mathrm{clip}(\diamond(2^t\zeta))$ and $2^t \otimes Z$ reports overflow and underflow faithfully as an implementation of $2^t\zeta$.*

PROOF. $2^t \otimes Z \in \mathrm{clip}(\diamond(2^t\zeta))$ follows easily if $\circ(2^t Z) \in \diamond(2^t\zeta)$, which is what we will prove. The case of $\zeta = 0$ is trivial, as $\diamond(\zeta) = \{0\}$, implying that $Z = 0$ as well. Obviously, $\circ(2^t Z) \in \diamond(2^t\zeta)$.

It, therefore, suffices to consider $\zeta \in [F_{\mathrm{small}}, F_{\mathrm{large}}]$. There is a unique integer e, $e \geq e_{\mathrm{min}}$, such that $\zeta \in [2^e, 2^{e+1})$. $Z \in \diamond(\zeta)$ implies that $Z \in [2^e, 2^{e+1}]$ and $|\zeta - Z| < \mathrm{ulp}(2^e)$. There are only two possibilities: $t + e \geq e_{\mathrm{min}}$ and $t + e < e_{\mathrm{min}}$. If $t + e \geq e_{\mathrm{min}}$, we have

$$\circ(2^t Z) = 2^t Z \in \{2^t\alpha \mid \alpha \in \diamond(\zeta)\} = \diamond(2^t\zeta).$$

If $t + e < e_{\mathrm{min}}$, both $2^t\zeta$ and $2^t Z$ lie inside $[2^{t+e}, 2^{t+e+1}]$ and

$$|2^t\zeta - 2^t Z| < 2^t\mathrm{ulp}(2^e) \leq \mathrm{ulp}(2^{t+e})/2.$$

By virtue of Lemma 3.1, $\circ(2^t Z) \in \diamond(2^t\zeta)$.

Now that $W = \circ(2^t Z) \in \diamond(2^t\zeta)$, it is clear that given any $Y \in \mathbb{F}^\sharp$, $2^t\zeta \leq Y$ implies that $W \leq Y$, and $2^t\zeta \geq Y$ implies that $W \geq Y$. Therefore, if $2^t\zeta \in [F_{\mathrm{small}}, F_{\mathrm{large}}]$, $2^t \otimes Z$ will not report overflow or underflow. If $2^t\zeta \geq 2^{e_{\mathrm{max}}+1}$, $2^t \otimes Z$ will definitely report an overflow. Finally, observe that if $2^t\zeta \in F$ for some $t < 0$, then $\zeta \in \mathbb{F}^\sharp$ and thus $Z \in \diamond(\zeta) = \{\zeta\}$, implying that $Z = \zeta$ and $2^t Z = 2^t\zeta$. Consequently, whenever $2^t\zeta \leq F_{\mathrm{small}} - 2^{e_{\mathrm{min}}+1}\boldsymbol{\varepsilon}$, $2^t \otimes Z$ will report underflow faithfully, as defined in Section 2. This completes the proof. □

We turn now to the computation of a scaled l_2-norm. A known strategy [Blue 1978] partitions the input data into three bins: one for small inputs, one for large, and one for "medium" inputs. The data in each bin are scaled by a common, statically chosen scale factor so that sums of squares of elements in each bin incur no spurious underflow or overflow exceptions. The final result is constructed by appropriate combination of the partial sums of squares from the bins. We follow the same approach, but with two enhancements. First, while we will explain our approach using three bins, we will point out later that our implementation keeps only two bins of data at any given time. This is important as the bins are in practice kept in the scarce SIMD vector registers[3].

[3]It is not possible to go down to one bin only as long as the bin boundaries and scaling factors are chosen statically to avoid division operations.

Second, our combination of the binned partial sums of squares are done in a way that guarantees a faithfully rounded scaled l_2-norm.

The basic idea is to divide the entire input range of $|x_j|$ into three "equal" subranges, where sums of squares of data in the middle (interior) subrange does not generate exceptions. Data in the two exterior ranges are scaled into the interior subrange. Now, the specifics. Define the even integer E by

$$E = \min\{e \mid 3e \geq e_{\max} - e_{\min} - \log_2(\boldsymbol{\varepsilon}), e \text{ is even}\}.$$

From E, we define a scale factor $\gamma = 2^{-E}$ and use the following notations.

$$\begin{array}{rclrcl} E_{\text{hi}} & = & e_{\max} + 1 - E, & \beta_{\text{hi}} & = & 2^{E_{\text{hi}}}, \\ E_{\text{lo}} & = & e_{\max} + 1 - 2E, & \beta_{\text{lo}} & = & 2^{E_{\text{lo}}}. \end{array}$$

In particular, $\gamma\beta_{\text{hi}} = \beta_{\text{lo}}$. We tabulate the specific values for binary32 and binary64 here.

	E	E_{hi}	E_{lo}	$\beta_{\text{hi}}\beta_{\text{lo}}$
binary32	94	34	−60	$2^{-26} = \boldsymbol{\varepsilon}/4$
binary64	700	324	−376	$2^{-52} = 2\boldsymbol{\varepsilon}$

Given the input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, the three bins are

$$\begin{array}{rcl} \mathcal{A} & = & \{\ \gamma x_j \mid |x_j| \geq \beta_{\text{hi}}\ \}, \\ \mathcal{B} & = & \{\ x_j \mid \beta_{\text{lo}} \leq |x_j| < \beta_{\text{hi}}\ \}, \\ \mathcal{C} & = & \{\ x_j/\gamma \mid |x_j| < \beta_{\text{lo}}\ \}. \end{array}$$

By design, $\beta_{\text{lo}} \leq |\widehat{x}_j| < \beta_{\text{hi}}$ for $\widehat{x}_j \in \mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$. Denote the partial, scaled, sums of squares as

$$\widehat{\sigma}_{\mathcal{A}} = \sum_{\widehat{x}_j \in \mathcal{A}} \widehat{x}_j^2, \quad \widehat{\sigma}_{\mathcal{B}} = \sum_{\widehat{x}_j \in \mathcal{B}} \widehat{x}_j^2, \quad \text{and} \quad \widehat{\sigma}_{\mathcal{C}} = \sum_{\widehat{x}_j \in \mathcal{C}} \widehat{x}_j^2.$$

Clearly, the "bin sums" are in the range

$$\widehat{\sigma}_{\mathcal{A}}, \widehat{\sigma}_{\mathcal{B}}, \widehat{\sigma}_{\mathcal{C}} \in \{0\} \cup \left[\beta_{\text{lo}}^2, n\beta_{\text{hi}}^2\right) \subseteq \{0\} \cup \left[\beta_{\text{lo}}^2, \beta_{\text{hi}}^2/\boldsymbol{\varepsilon}\right), \tag{13}$$

by assuming $n \leq 1/\boldsymbol{\varepsilon}$. Furthermore,

$$\sigma = \sum_j x_j^2 = \gamma^{-2}\,\widehat{\sigma}_{\mathcal{A}} + \widehat{\sigma}_{\mathcal{B}} + \gamma^2\,\widehat{\sigma}_{\mathcal{C}}. \tag{14}$$

A straightforward implementation can collect all three bins, and invoke the `SumOfSquares` function (or the parallel version) on each of the bins, followed by some appropriate combination method to arrive at the final result. Our implementation, in fact, keeps and processes only two bins. The observation is that $\mathcal{A}$ and $\mathcal{C}$ are never needed simultaneously. If $\mathcal{A}$ is nonempty, then Equation (13) and the fact that $\gamma\beta_{\text{hi}} = \beta_{\text{lo}}$ show that $\gamma^2\widehat{\sigma}_{\mathcal{C}}/(\gamma^{-2}\widehat{\sigma}_{\mathcal{A}}) \leq \gamma^2/\boldsymbol{\varepsilon} \ll \boldsymbol{\varepsilon}^2$. Neglecting $\gamma^2\widehat{\sigma}_{\mathcal{C}}$ altogether incurs a relative error (much) less than $\boldsymbol{\varepsilon}^2$. A similar estimate shows that keeping $\widehat{\sigma}_{\mathcal{B}}$ is nevertheless necessary in order not to lose too much accuracy.

Briefly speaking, our implementation starts the binning process by keeping the interior (middle) bin and one exterior bin. When the first element belonging to $\mathcal{A}$ appears, the existing exterior bin is replaced with that element, and elements from $\mathcal{C}$ are never collected from that point onwards. Let us mention that the logic required to decide whether the first element belonging to $\mathcal{A}$ has already appeared or not can be implemented using nothing but masks, not requiring branches. Denote the two actually maintained bins by $\mathcal{U}$ and $\mathcal{V}$, then

$$\sigma_2 = \gamma^k(\widehat{\sigma}_{\mathcal{U}} + \gamma^2\widehat{\sigma}_{\mathcal{V}}),$$

where $k = -2$ if $\mathcal{U}$ corresponds to $\mathcal{A}$, and $k = 0$ if $\mathcal{U}$ corresponds to $\mathcal{B}$. In either case, the "two-bin" sum of squares σ_2 satisfies

$$|\sigma_2 - \sigma| \leq \boldsymbol{\varepsilon}^2\sigma, \quad \text{and} \quad \sigma_2 \leq (1+\boldsymbol{\varepsilon}^2)\sigma. \tag{15}$$

To compute σ_2, the `SumOfSquares` function is applied to each of the two resulting bins, yielding two double-FP variables $\mathbf{U} = [U, u]$ and $\mathbf{V} = [V, v]$. Both $U + u$ and $V + v$ approximate their targets with high relative accuracies:

$$\frac{|(U+u) - \widehat{\sigma}_{\mathcal{U}}|}{\widehat{\sigma}_{\mathcal{U}}}, \; \frac{|(V+v) - \widehat{\sigma}_{\mathcal{V}}|}{\widehat{\sigma}_{\mathcal{V}}} \leq \Delta_{n-1}(3\boldsymbol{\varepsilon}^2),$$

where $\Delta_\ell(\delta) = \ell\delta/(1-\ell\delta)$. Using n instead of $n-1$ for simplicity, we have

$$\left|\gamma^k[(U+u) + \gamma^2(V+v)] - \sigma_2\right| \leq \Delta_n(3\boldsymbol{\varepsilon}^2)\sigma_2, \tag{16}$$

and

$$\gamma^k[(U+u) + \gamma^2(V+v)] \leq \left(1 + \Delta_n(3\boldsymbol{\varepsilon}^2)\right)\sigma_2. \tag{17}$$

We handle $\gamma^k[(U+u)+\gamma^2(V+v)]$ as follows. For nonzero U and V, $\beta_{\rm lo}^2 \leq U+u, V+v < \beta_{\rm hi}^2/\boldsymbol{\varepsilon}$ (as we assume that $n \leq 1/\boldsymbol{\varepsilon}$). If $U \geq \beta_{\rm lo}^2/\boldsymbol{\varepsilon}^3$,

$$\gamma^2(V+v) < \gamma^2\beta_{\rm hi}^2/\boldsymbol{\varepsilon} = \beta_{\rm lo}^2/\boldsymbol{\varepsilon} \leq \boldsymbol{\varepsilon}^2 U \leq \boldsymbol{\varepsilon}^2(1+\boldsymbol{\varepsilon})(U+u).$$

Similarly, if $V \leq \beta_{\rm lo}^2\boldsymbol{\varepsilon}^2/\gamma^2 = \beta_{\rm hi}^2\boldsymbol{\varepsilon}^2$,

$$\gamma^2(V+v) \leq \gamma^2(1+\boldsymbol{\varepsilon})V < \beta_{\rm lo}^2\boldsymbol{\varepsilon}^2(1+\boldsymbol{\varepsilon}) \leq \boldsymbol{\varepsilon}^2(1+\boldsymbol{\varepsilon})(U+u).$$

Thus, if $U \geq \beta_{\rm lo}^2/\boldsymbol{\varepsilon}^3$ or $V \leq \beta_{\rm hi}^2\boldsymbol{\varepsilon}^2$, the error by dropping the $\mathbf{V}$ term is in the order of $\boldsymbol{\varepsilon}^2$:

$$\frac{\gamma^2(V+v)}{(U+u)+\gamma^2(V+v)} \leq \boldsymbol{\varepsilon}^2(1+\boldsymbol{\varepsilon}). \tag{18}$$

If $U < \beta_{\rm lo}^2/\boldsymbol{\varepsilon}^3$ and $V > \beta_{\rm hi}^2\boldsymbol{\varepsilon}^2$, then neither $(U+u)/\gamma$ nor $\gamma(V+v)$ raises exceptions. This is because $U < \beta_{\rm lo}^2/\boldsymbol{\varepsilon}^3 \Rightarrow U/\gamma < \beta_{\rm hi}\beta_{\rm lo}/\boldsymbol{\varepsilon}^2 \approx 1/\boldsymbol{\varepsilon}$. Similarly, $V > \beta_{\rm hi}^2\boldsymbol{\varepsilon}^2 \Rightarrow \gamma V > \beta_{\rm hi}\beta_{\rm lo}\boldsymbol{\varepsilon}^2 \approx \boldsymbol{\varepsilon}^3$. These discussions are expressed in the function `SumOfSquaresBins`.

function `SumOfSquaresBins`($\mathbf{x}$) // general inputs
  Obtain bins $\mathcal{U}$, $\mathcal{V}$, and integer k as discussed
  // $\gamma^k(\widehat{\sigma}_{\mathcal{U}} + \gamma^2\widehat{\sigma}_{\mathcal{V}})$ approximates $\sum_j x_j^2$ accurately
  // $k = -2$ if $\mathcal{U}$ is $\mathcal{A}$, $k = 0$ if $\mathcal{U}$ is $\mathcal{B}$
  // Note that $k = -2$ if and only if bin $\mathcal{A}$ is nonempty
  $[U, u]$:= `SumOfSquaresP`($\mathbf{x}^{(\mathcal{U})}$);
  $[V, v]$:= `SumOfSquaresP`($\mathbf{x}^{(\mathcal{V})}$);
  if $U = 0$ // $\mathcal{A}$ and $\mathcal{B}$ are both empty
    $m := 2$, $[S, s] := [V, v]$,
    return m and $\mathbf{S} = [S, s]$.
  if $U \geq \beta_{\rm lo}^2/\boldsymbol{\varepsilon}^3$ **or** $V \leq \beta_{\rm hi}^2\boldsymbol{\varepsilon}^2$
    $m := k$, $[S, s] := [U, u]$
    return m and $\mathbf{S} = [S, s]$
  if $|v| \leq \beta_{\rm hi}^2\boldsymbol{\varepsilon}^2$, $v := 0$.
  $[U, u] := [\gamma^{-1}U, \gamma^{-1}u]$; $[V, v] := [\gamma V, \gamma v]$; $m := k+1$;
  $[S, s]$:= `SumNonNeg`($[U, u]$, $[V, v]$)
  return m and $\mathbf{S} = [S, s]$
end `SumOfSquaresBins`

THEOREM 5.2. *Let* SumOfSquaresBins($\mathbf{x}$) *return* m *and* $\mathbf{S} = [S, s]$. *Denote by* $\widehat{\sigma}$ *the scaled sums of squares* $\widehat{\sigma} = \gamma^{-m}\sigma = \gamma^{-m}\sum_j x_j^2$. *If the length* n *of* $\mathbf{x}$ *satisfies* $n+3 < ((24+\boldsymbol{\varepsilon})\boldsymbol{\varepsilon})^{-1}$, *then* $\circ(\sqrt{S}) \in \diamond(\sqrt{\widehat{\sigma}})$.

PROOF. We group the total errors incurred in computing σ as $\gamma^m\widehat{\sigma}$ into three stages. In Stage 1, the value σ, which is exactly represented in terms of γ and the three bin sums (Equation (14)), is approximated by $\sigma_2 = \gamma^k(\widehat{\sigma}_{\mathcal{U}} + \gamma^2\widehat{\sigma}_{\mathcal{V}})$. In Stage 2, the two bin sums $\widehat{\sigma}_{\mathcal{U}}$ and $\widehat{\sigma}_{\mathcal{V}}$ are approximated by the double-FP $[U, u]$ and $[V, v]$. Finally, in Stage 3, $\gamma^k((U+u)+\gamma^2(V+v))$ is approximated as $\gamma^m(S+s)$ by possibly dropping v or both V and v and the use of SumNonNeg.

Consider the Stage 3 error. There are three possible points of exit in the procedure SumOfSquaresBins. The first point of exit corresponds to a zero Stage 3 error, as there is actually only at one nonempty bin. The second point of exit corresponds to Stage 3 error bounded by $\boldsymbol{\varepsilon}^2(1+\boldsymbol{\varepsilon})$, as given by Equation (18). If the last point of exit is taken, Stage 3 error consists of one part that is due to a single application of SumNonNeg, which is bounded by $3\boldsymbol{\varepsilon}^2$ (Theorem 4.1), and one due to possibly dropping the v term, which is bounded by $\boldsymbol{\varepsilon}^2(1+\boldsymbol{\varepsilon})$ (Equation (18)). Thus we bound the error in Stage 3 conservatively by $5\boldsymbol{\varepsilon}^2$:

$$\frac{\left|\gamma^m(S+s) - \gamma^k[(U+u)+\gamma^2(V+v)]\right|}{\gamma^k[(U+u)+\gamma^2(V+v)]} \le 5\boldsymbol{\varepsilon}^2, \tag{19}$$

and

$$\gamma^m(S+s) \le (1+5\boldsymbol{\varepsilon}^2)\,\gamma^k[(U+u)+\gamma^2(V+v)]. \tag{20}$$

Stage 1 and Stage 2 errors have already been discussed in Equations (15) through (17). Putting these together,

$$\begin{aligned}
&|\gamma^m(S+s)-\sigma|/\sigma \\
&\quad\le \left|\,\gamma^m(S+s) - \gamma^k[(U+u)+\gamma^2(V+v)]\,\right|/\sigma \\
&\qquad + \left|\,\gamma^k[(U+u)+\gamma^2(V+v)] - \sigma_2\,\right|/\sigma + |\sigma_2-\sigma|/\sigma, \\
&\quad\le 5\boldsymbol{\varepsilon}^2(1+\Delta_n(3\boldsymbol{\varepsilon}^2))(1+\boldsymbol{\varepsilon}^2) + \Delta_n(3\boldsymbol{\varepsilon}^2)(1+\boldsymbol{\varepsilon}^2) + \boldsymbol{\varepsilon}^2, \\
&\quad\le \Delta_n(3\boldsymbol{\varepsilon}^2) + 7\boldsymbol{\varepsilon}^2.
\end{aligned}$$

Consequently,

$$\begin{aligned}
|(S+s)-\widehat{\sigma}| &\le (\Delta_n(3\boldsymbol{\varepsilon}^2)+7\boldsymbol{\varepsilon}^2)\widehat{\sigma}, \\
&\le (\Delta_n(3\boldsymbol{\varepsilon}^2)+9\boldsymbol{\varepsilon}^2)\widehat{\sigma}, \\
&\le \Delta_{n+3}(3\boldsymbol{\varepsilon}^2)\widehat{\sigma}.
\end{aligned} \tag{21}$$

From Theorems 4.2 and 4.3, $n+3 < ((24+3\boldsymbol{\varepsilon})\boldsymbol{\varepsilon})^{-1}$ implies that

$$|(S+s)-\widehat{\sigma}| < \boldsymbol{\varepsilon}\widehat{\sigma}/8,$$

a condition that guarantees, by Theorem 3.3, that $\circ(\sqrt{S}) \in \diamond(\sqrt{\widehat{\sigma}})$. □

AccuNrm2 is the straightforward synthesis of the previous discussions. Theorem 5.3 that follows is a formal statement that summarizes the technical results of this article.

```
function AccuNrm2(x) // general faithful l_2-norm
    (m, S) := SumOfSquaresBins(x)
    // m is an integer in the range [−2, 2] and γ^m(S + s) ≈ Σ_j x_j^2
    // By design, γ^m is an even power of 2.
    Z := sqrt(S)
    return γ^{m/2} ⊗ Z
    // Value in clip(◇(‖x‖_2)) and reports overflow/underflow faithfully (Theorem 5.3)
end AccuNrm2
```

THEOREM 5.3. *Let* $\mathbf{x}$ *be a vector of length n. If* $n < L'$ *with* $L' = ((24 + 3\boldsymbol{\varepsilon})\boldsymbol{\varepsilon})^{-1} - 3$*, then* `AccuNrm2`$(\mathbf{x}) \in \text{clip}(\diamond(\|\mathbf{x}\|_2))$ *and reports overflow and underflow faithfully.*

PROOF. This is a direct consequence of Theorems 5.1 and 5.2. □

The bound L' on the vector length n induced by 5.3 translates as follows for IEEE754 binary32 and binary64:

	Vector length bound $n < L'$
binary32	$L' = 699047$
binary64	$L' = 3.75299968947538 \cdot 10^{14}$

6. IMPLEMENTATION AND TESTING

The complete set of codes, together with testing and performance measurement auxiliary sources, is available at

http://www.christoph-lauter.org/faithfulnorm.tgz

under an open source license.

We implemented and tested our faithfully rounded, division-free l_2-norm with faithful reporting of underflow and overflow. The implementation referred to as `FaithfulNorm` closely follows the algorithmic description given in the previous sections.

We used IEEE754 binary64 as working precision and restricted ourselves to an SIMD environment, targeting in particular Intel SSE/AVX units, with or without support for the IEEE754 FMA instruction. Recent versions of SSE and all versions of AVX support IEEE754 binary64 precision. The rationale for the choice of an SIMD environment is twofold: to use an environment most similar to the existing codes to which we compare our algorithm, and to maximize our performance in a typical processor without the use of threads.

To achieve high performance on modern pipelined floating-point units, it is important to avoid branching (when possible) as well as avoid the use of expensive operations such as floating-point division. By design, our l_2-norm algorithm is division free. We are also able to make our inner-loop branching free based on three observations. First, the SSE/AVX units offer comparison instructions that return their results as masks of all ones or all zeros. Second, logical bit and a floating-point variable with all ones leave the variable unchanged while bit and with all zeros turn it into a floating-point value of zero. As a matter of course, multiplying floating-point zeros and accumulating them is innocuous. Third, discarding the $\mathcal{C}$ bin when the first $\mathcal{A}$ is found and maintaining in these registers the $\mathcal{U}$ bin from that point onward (see Section 5) just means maintaining a binary flag, which can also be implemented as a bit mask.

We shall repeat that our implementation has no memory overhead or memory access overhead: each input vector element x_j is read only once and the intermediate values (accumulators and so on) are kept in registers.

We compared the implementation of our faithfully rounded l_2-norm with implementations for other approaches with respect to both accuracy and performance. To do so, we implemented a naïve l_2-norm, called `NaiveNorm`, that plainly uses working precision for squaring the x_i and accumulating these squares, without any underflow and overflow avoidance. We further implemented the algorithm found in `netlib` [Anderson et al. 1999]; we call this implementation `NetlibNorm`. Finally, we implemented another faithfully rounded l_2-norm using the arbitrary precision library MPFR [MPFR 2015]. This implementation is simply based on an exact accumulation of the squares x_i^2 in an

Table I. Maximum Error in ulps Observed for Various Domains and Vector Lengths n, Plain SSE Implementation

	Vectors with normal results		Vectors for which results underflow		Vectors with entries around 1.0		Vectors with chosen "half-ulp" entries	
	$n = 10^3$	$n = 10^7$	$n = 10^3$	$n = 10^7$	$n = 10^3$	$n = 10^7$	$n = 10^3$	$n = 10^7$
NaiveNorm	∞	∞	$8.84 \cdot 10^{12}$	$5.46 \cdot 10^{10}$	7.73	861	250	$2.50 \cdot 10^6$
NetlibNorm	2.01	524	0.496	0.698	7.58	609	250	$2.50 \cdot 10^6$
MPFRNorm	0.494	0.481	0.490	0.498	0.468	0.497	0.0749	0.484
FaithfulNorm	0.620	0.628	0.497	0.499	0.605	0.701	0.0749	0.484

accumulator that provides enough precision: using an MPFR variable with precision $p = 2\,(e_{\max} - e_{\min} - \log_2 \varepsilon) + \lceil \log_2 n \rceil$ is just enough. We refer to it as `MPFRNorm`.

We performed testing on a 4-core Intel Core i7 at 2.67GHz with 4GB of RAM and on an 8-core Intel Xeon E3-1275 v3 at 3.50GHz with 32GB of RAM. All implementations were written in C—using built-ins for access to SIMD instructions—and compiled using gcc version 4.8 and options `-std=c99 -O3 -march=native`. Timings are given cycles per vector element, obtained using the Read-Time-Step-Counter instruction with serialization, subtracting off the measured overhead for a call to an empty function and dividing by the number of elements.

We used pseudo-random floating-point input vectors in our tests. These pseudo-random values were constructed as follows: we separately generated a uniformly distributed exponent value in range and a uniformly distributed significand for that chosen exponent value. We then constructed a floating-point value out of this exponent and significand value. When generating values for a subdomain $[a; b] \subseteq \mathbb{F}$, we performed that random-generation process for an exponent range completely covering the possible exponents of floating-point values in $[a; b]$, discarding all generated floating-point values that were outside of $[a; b]$.

We did accuracy testing with test vectors of various lengths n and input types. The testing results are summarized in Table I.

First, we considered input vectors chosen such that the final l_2-norm result is a normal floating-point number. Second, we tested the algorithms on input vectors for which the final result gradually underflows. Third, we performed testing on vectors with inputs around 1.0, that is, where underflow or overflow avoidance is not necessary. Finally, we constructed input vectors with $x_1 = 1$ and subsequent x_js are chosen to be much smaller than 1, but with the contrived property that $\circ(1 + x_j^2)$ produces a positive absolute error very close to the "half-ulp" bound of ε. This test case is admittedly artificial, but nonetheless demonstrates a near worst-case error scenario for `NaiveNorm` and `NetlibNorm`.

Testing shows that both the `NaiveNorm` and `NetlibNorm` implementations fail to provide faithfully rounded results. It demonstrates also that both our `FaithfulNorm` as well as the `MPFRNorm` algorithm do yield faithfully rounded results.

In cases when the final l_2-norm does not overflow, our algorithm `FaithfulNorm` returns a result with an error well below 1ulp. As expected, the maximum error does not vary with vector length, whereas it does for the `netlib` l_2-norm.

Accuracy testing also shows that, with random inputs, `netlib` l_2-norm can result in hundreds of ulps of error when the vector lengths n get to be 10 million or more. In deliberately constructed inputs, errors as large as about $0.25\,n$ ulp can be observed.

Turning now to performance testing, Tables II, III, IV, and V summarize our observations. We used vectors of floating-point numbers in various domains of interest. We tested for vectors for which the final l_2-norm results gradually underflow, overflow, or stay in the range of normal floating-point numbers. Measurements were done for varying vector lengths. Starting with some minimal vector length (a couple of dozen

Table II. Computation Time in Cycles Per Vector Element, Plain SSE Version on Intel Core i7

	Vectors with normal results	Vectors for which results underflow	Vectors with entries around 1.0	Vectors for which results overflow	Vectors provoking spurious underflow in `NetlibNorm`
`NaiveNorm`	47	137	3.48	46.8	128
`NetlibNorm`	156	472	19.1	156	274
`MPFRNorm`	1080	2670	818	1090	1660
`FaithfulNorm`	34.2	289	25.3	34.2	62.2

Table III. Computation Time in Cycles Per Vector Element, Plain SSE Version on Intel Xeon E3-1275

	Vectors with normal results	Vectors for which results underflow	Vectors with entries around 1.0	Vectors for which results overflow	Vectors provoking spurious underflow in `NetlibNorm`
`NaiveNorm`	4.95	4.75	4.72	4.70	4.52
`NetlibNorm`	21.9	158	12.8	21.1	21.8
`MPFRNorm`	810	1160	536	803	717
`FaithfulNorm`	21.5	87.3	21.8	21.7	20.3

Table IV. Computation Time in Cycles Per Vector Element, AVX Version w/o FMA on Intel Xeon E3-1275

	Vectors with normal results	Vectors for which results underflow	Vectors with entries around 1.0	Vectors for which results overflow	Vectors provoking spurious underflow in `NetlibNorm`
`NaiveNorm`	4.85	4.61	4.68	4.86	4.52
`NetlibNorm`	21.1	157	13.3	21.6	21.8
`MPFRNorm`	795	1250	552	765	720
`FaithfulNorm`	12	50.7	12.5	12.6	14.8

elements), vector length had no influence on computation time per element; therefore, we report only the numbers obtained for vectors of length 10^6.

These performance results speak in favor of our `FaithfulNorm` implementation. For cases when the final result is a normal floating-point number, our implementation is up to 3 times faster than the `netlib` implementation. As already explained, this is due to several factors: avoidance of spurious underflow, no use of expensive divisions, and an algorithm that is branch-free in the inner loop. In particular, the relative cost of the divisions and branches in the `netlib` implementation can be seen in the performance data: on Intel Core i7, which does not yet implement the recent AVX extensions, our algorithm is up to 4.5 times faster than `netlib`, whereas on Intel Xeon E3-1275, the same SSE codes run equally fast. However, on Intel Xeon E3-1275, AVX and FMA are available, allowing our l_2-norm to be up to 3 times faster than `netlib`. It also is worth mentioning that spurious underflow in `netlib` hurts `netlib` performance on some processors—such as the Intel Core i7—does not on others, such as the Intel Xeon E3-1285.

We shall mention, however, that our algorithm does have lower performance than the `netlib` in two cases: first, for inputs when the vector length is (very) short, typically less than a dozen elements. In this case, our algorithm has a much higher static overhead due to the elaborate computations needed in the reduction of the bins and square root. In future work we shall address this problem with a call-out to a specialized l_2-norm for very small vectors. Second, the `netlib` norm can be faster on some processors that

Table V. Computation Time in Cycles Per Vector Element, AVX Version Using FMA on Intel Xeon E3-1275

	Vectors with normal results	Vectors for which results underflow	Vectors with entries around 1.0	Vectors for which results overflow	Vectors provoking spurious underflow in `NetlibNorm`
`NaiveNorm`	4.52	4.52	4.52	4.52	4.52
`NetlibNorm`	20.5	151	12.6	20.5	22
`MPFRNorm`	722	1110	481	723	770
`FaithfulNorm`	6.94	42.3	6.94	6.94	10.4

have a faster floating-point division instruction (with respect to multiplication) and that do not suffer a performance impact due to branching or on subnormal handling. This effect can already be measured on recent Intel Xeons. But we point out that these processors come equipped with the FMA instructions, which `FaithfulNorm` can exploit. Table V shows that `FaithfulNorm` regains the speed advantage when it uses the FMA instructions on these processors appropriately.

7. CONCLUSIONS

In this article, we presented an efficient algorithm to compute the faithful rounding of the l_2-norm of a floating-point vector. While our algorithm is very accurate, it is also faster than previous algorithms, such as the one of `netlib` that gives no information about the accuracy of the result. Moreover, our algorithm avoids spurious overflow and underflow. It is also suitable for parallel implementations. We have hitherto focused our implementation on vector parallelism using SIMD instructions. Implementation and testing of our algorithm in a threaded environment as well as formulating the algorithm in terms of auto-vectorizable, auto-parallelizable code, is left to future work.

Disclaimers

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information, go to http://www.intel.com/performance.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

REFERENCES

E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (3rd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

J. L. Blue. 1978. A portable Fortran program to find the Euclidean norm of a vector. *ACM Transactions on Mathematical Software* 4, 1, 15–23.

T. J. Dekker. 1971. A floating-point technique for extending the available precision. *Numererische Mathematik* 18, 224–242.

N. J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

D. E. Knuth. 1998. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms* (3rd ed.). Addison-Wesley, Reading, MA.

X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. 2002. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software* 28, 2, 152–205.

MPFR. *MPFR (Multiple Precision Floating-Point Reliable Library)*. Retrieved August 25, 2015 from http://www.mpfr.org.

J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. 2010. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston Inc., Boston, MA.

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

S. M. Rump. 2009. Ultimately fast accurate summation. *SIAM Journal on Scientific Computing* 31, 5, 3466–3502.

S. M. Rump, T. Ogita, and S.i Oishi. 2008a. Accurate floating-point summation. I. Faithful rounding. *SIAM Journal on Scientific Computing* 31, 1, 189–224.

S. M. Rump, T. Ogita, and S. Oishi. 2008b. Accurate floating-point summation. II. Sign, *K*-fold faithful and rounding to nearest. *SIAM Journal on Scientific Computing* 31, 2, 1269–1302.

Y.-K. Zhu and W. B. Hayes. 2009. Correct rounding and a hybrid approach to exact floating-point summation. *SIAM Journal on Scientific Computing* 31, 4, 2981–3001.

Y.-K. Zhu and W. B. Hayes. 2010. Algorithm 908: Online exact summation of floating-point streams. *ACM Transactions on Mathematical Software* 37, 3.

Received December 2013; revised July 2014; accepted November 2014