Noname manuscript No.
(will be inserted by the editor)

Fast and accurate computation of the Euclidean norm of a vector

Siegfried M. Rump

Received: date / Accepted: date

Abstract The numerical computation of the Euclidean norm of a vector is perfectly well conditioned with favorite a priori error estimates. Recently there is interest in computing a faithfully rounded approximation which means that there is no other floating-point number between the computed and the true real result. Hence the result is either the rounded to nearest result or its neighbor.

Previous publications guarantee a faithfully rounded result for large dimension, but not the rounded to nearest result. In this note we present several new and fast algorithms producing a faithfully rounded result, as well as the first algorithm to compute the rounded to nearest result. Executable MATLAB codes are included. As a by product, a fast loop-free error-free vector transformation is given. That transforms a vector such that the sum remains unchanged but the condition number of the sum multiplies with the rounding error unit.

Keywords Euclidean norm · rounding error · faithful rounding · error-free transformation

Mathematics Subject Classification (2020) 65G99, 65G50

1 Notation and introduction

We assume a precision-p binary floating-point arithmetic with rounding to nearest according to the IEEE 754 standard [10] to be given and denote the set of floating-point numbers by $\mathbb{F}$. Then $\mathbb{F}$ is symmetric, i.e., $\mathbb{F} = -\mathbb{F}$, and there is a smallest and largest positive normalized floating-point number realmin and realmax, respectively. We define $\mathcal{P} := [\text{realmin}, \text{realmax}]$ and call $\mathcal{N} := -\mathcal{P} \cup \{0\} \cup \mathcal{P}$ the normalized range.

Institute for Reliable Computing, Hamburg University of Technology, and
Visiting Professor at Waseda University, Faculty of Science and Engineering
email: rump@tuhh.de

To be more precise, for "rounding to nearest" we assume RoundTiesToEven which means that a real number being the midpoint of two adjacent floating-point numbers is rounded to the one with even mantissa. Calling that rounding function $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ it follows that $\mathrm{fl}(a \circ b)$ is the floating-point result of $a \circ b$ for $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$.

In the computation of the Euclidean norm of a vector intermediate results may be outside $\mathcal{N}$ but the final result in $\mathcal{N}$. That is taken care of by case distinctions and normalization, see [3,1,20]. Henceforth, we assume throughout this note without loss of generality that neither over- nor underflow occurs, i.e., all intermediate results are in $\mathcal{N}$.

For $\mathbf{u} := 2^{-p}$ denoting the relative rounding error unit [7] the refined error estimate [31,7,4,23]

$$x \in \mathcal{N}: \quad \max\{|x - f| : f \in \mathbb{F}\} \leq \frac{\mathbf{u}}{1+\mathbf{u}}|x| \tag{1}$$

holds true, and the same constant $\frac{\mathbf{u}}{1+\mathbf{u}}$ bounds the relative error of every floating-point operation.

Many of our results are also true for a precision-p floating-point arithmetic with general base β and $\mathbf{u} = \frac{1}{2}\beta^{1-p}$. Since we target on MATLAB implementations, we restrict our attention to binary.

Throughout this note $\|\cdot\|$ denotes the Euclidean, i.e., ℓ_2-norm. The result of a floating-point evaluation of an expression is denoted by $\mathrm{float}(\cdot)$, where parentheses are respected but otherwise any order of evaluation may be used. Hence $\mathrm{float}(a \circ b) = \mathrm{fl}(a \circ b)$ is the floating-point result of $a \circ b$ for $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$. For example, $s := \mathrm{float}(\|x\|)$ denotes a floating-point approximation of the Euclidean norm of $x \in \mathbb{F}^n$ using any order of summation. Standard error estimates [7] yield

$$|s - \|x\|| \leq \gamma_{n+1}\|x\| \qquad \text{where } \gamma_k := \frac{k\mathbf{u}}{1-k\mathbf{u}}$$

provided that $(n+1)\mathbf{u} < 1$. In [11] we proved a refined estimate without restriction on n.

Lemma 1 *Let $x \in \mathbb{F}^n$ and $s := \mathrm{float}(\sqrt{\sum_{i=1}^n x_i^2})$ computed in any order. Then*

$$|s - \|x\|| \leq \left(\frac{n}{2} + 1\right)\mathbf{u}\|x\|$$

is true without restriction on $n \in \mathbb{N}$.

The bound is basically sharp, but practical experience and probabilistic arguments [26,8,9] suggest that practically the relative error for the Euclidean norm and for summation is hardly larger than $\sqrt{n}\mathbf{u}\|x\|$.

Recently [6,20] there is interest in algorithms computing a faithful approximation of the Euclidean norm. That means that there is no other floating-point number between the computed and the true real result. Both are based on error-free transformations and some kind of double-double arithmetic [2], where the latter was already considered in [5]. The computed result is thus

equal to the rounded to nearest result or to one of its neighbors. If the true result is a floating-point number, that will be the result of the algorithms.

Both approaches [6,20] are devoted to the computation of the Euclidean norm. In [17] we introduced a novel pair arithmetic cpair and prove sufficient conditions that for a general arithmetic expression comprised of $\{+, -, \times, /, \sqrt{}\}$ the result computed using cpair is faithfully rounded. As a by-product it includes the Euclidean norm. One difference to the well-known double-double pair arithmetic [2], which is intrinsically used in [6,20], is that a final error-free transformation is omitted. That speeds up the algorithms in cpair significantly. While there is not much penalty in the accuracy of the computed result, it bears the advantage that, in contrast to [2], the higher order part is equal to the ordinary floating-point result. In that sense cpair is a floating-point arithmetic together with an error term.

In this note we will give some new algorithms for the computation of a faithfully rounding of the Euclidean norm as well as for the rounded to nearest result. All algorithms are given in executable MATLAB code [21]. We invest particular care in designing fast algorithms diminishing the interpretation overhead. In particular, we avoid loops as they may slow down the performance significantly.

This note is organized as follows. The next section recalls error-free transformations and some improvements, and mainly error estimates to ensure a faithfully rounded result. In Section 3 a vectorized error-free vector transformation is given, we recall recent sharp error estimates on summation and present the first two of our new algorithms to approximate $\|x\|$. Those are based on relative splittings and adopt methods presented in [25]. In the next section another two new algorithms are presented using absolute splitting as in [28,29], again with sufficient conditions on a faithfully rounded result. In Section 5 an algorithm is presented computing the nearest approximation of $\|x\|$ with proof of correctness. To our knowledge that is the first of its kind. The generation of ill-conditioned test examples, i.e., floating-point vectors x with $\|x\|$ very close to a switching point is addressed in Section 6. The note is closed with computational results on the computing time and the accuracy of all algorithms and a conclusion.

2 Error-free transformations and previous algorithms

Since a long time it is known [22,13,5,14] that the sum and product of two floating-point numbers can be expressed as the sum $x+y$ of two floating-point numbers, and that x and y can be computed using few pure floating-point operations. It was used implicitly by Neumaier, who wrote the remarkable paper [24] when he was a bachelor student, otherwise it was basically known to experts. The methods received wide attention when I coined the term "error-free transformations" in [25] with numerous papers following thereafter.

For this note we need only the error-free transformations for sum and product; for details of other error-free transformations see e.g. [23]. Consider

Fig. 1 Error-free sum

```
1   function [x,y] = TwoSum(a,b)      function [x,y] = FastTwoSum(a,b)
2     x = a + b;                        x = a + b;
3     z = x - a;                        y = (a-x) + b;
4     y = ( a - (x-z) ) + (b-z);
```

We display all algorithms in executable MATLAB code; later some longer algorithms appear so that we decided to add line numbers. The following is true [22,25,23,18].

Lemma 2 *Let $a, b \in \mathbb{F}$ be given and x, y be the result of Algorithm* `TwoSum` *applied to a, b. Then*

$$x + y = a + b \quad \textit{and} \quad \mathrm{fl}(x + y) = x. \tag{2}$$

If $|a| \geq |b|$, then (2) is also true for the result of Algorithm `FastTwoSum`*.*

The assumptions for Algorithm `FastTwoSum` can be weakened [23,18], but we do not need this here. One might use Algorithm `FastTwoSum` together with an "if"-statement thereby reducing the number of operations from 6 to 3, however, that is often slower [25] than applying Algorithm `TwoSum`.

The proof of correctness [23] relies on the fact that all operations from row 3 on are error-free, i.e., cannot cause a rounding error.

The key to the error-free transformation of multiplication is to split [5] both factors into a sum of two floating-point numbers such that the product of the addends does not cause a rounding error. The Algorithm `Split` for the `binary64` format can be implemented as follows.

Fig. 2 Error-free split of a floating-point number for 53-binary arithmetic

```
1   function [x,y] = Split(a)
2     y = ( pow2(27) + 1 ) * a;
3     x = y - ( y - a );
4     y = a - x;
```

In precision-p the factor in line 2 is to be replaced [5] by $2^{\lceil p/2 \rceil} + 1$. For various splitting methods and many details see [12]. For the calculation of the Euclidean norm we need only squares, so we add a specialized method for that.

Since the input is split into two parts we use, for example, the notation `Aa` to indicate that `A+a = Aa` in line 3, and similarly for `Bb` in Algorithm `TwoProduct`.

Lemma 3 *Let $a, b \in \mathbb{F}^n$ be given and $P, p \in \mathbb{F}^n$ be the results of Algorithm* `TwoProduct` *applied to a, b. Then*

$$P_i + p_i = a_i \cdot b_i \quad \textit{for all } i \in \{1, \ldots, n\}. \tag{3}$$

Fig. 3 Error-free product and square

```
1   function [P,p] = TwoProduct(Aa,Bb)
2     P = Aa.*Bb;
3     [A,a] = Split(Aa);
4     [B,b] = Split(Bb);
5     p = a.*b - (((P-A.*B)-a.*B)-A.*b);
```

```
function [P,p] = TwoSquare(Aa)
  P = sqr(Aa);
  [A,a] = Split(Aa);
  p = a.*a - ((P-A.*A)-2*a.*A);
```

In binary arithmetic the results $P, p \in \mathbb{F}^n$ of Algorithm `TwoSquare` *applied to $a \in \mathbb{F}^n$ satisfy*

$$P_i + p_i = a_i^2 \quad \textit{for all } i \in \{1, \ldots, n\}. \tag{4}$$

Furthermore, for both algorithms $|p_i| \leq \mathbf{u}|P_i|$ for all $i \in \{1, \ldots, n\}$.

Proof The first result (3) is well-known [14,23], where the proof relies on the fact all operations in lines $3-5$ do not cause a rounding error. That proves (4) as well because multiplication by 2 is error-free. The last estimate is a well-known property [23] of Algorithm `TwoProduct`. □

For given $x \in \mathbb{F}^n$, previous approaches [6,20] borrow from the double-double pair arithmetic [2] to calculate a pair (T, t) such that $T + t$ is an accurate approximation of the sum of squares $\sum_{i=1}^n x_i^2$. Another candidate for a pair arithmetic is the cpair arithmetic [17]. Both are implemented as toolboxes `dd` and `cpair` in INTLAB [27], the MATLAB toolbox for Reliable Computing.

In [6] `TwoProduct` is used to compute a pair approximation for x_i^2, in [20] FMA instructions are used. While this is part of the new floating-point standard [10] and implemented on many computers, it is not available in MATLAB[1]. Therefore some of our algorithms avoid that in this note.

Given (T, t) it remains to compute a good floating-point approximation of $\sqrt{T+t}$. In [6] just `sqrt(T)` is used ignoring the lower order part t. In [20] the algorithm of our cpair arithmetic [17] is used adapted to one output $P + p$ rather than the pair (P, p).

Fig. 4 Accurate square root of $T + t$ for a given pair (T, t)

```
1   function res = AccSqrt(T,t)
2     P = sqrt(T);
3     [H,h] = TwoSquare(P);
4     r = ( T - H ) - h;
5     r = t + r;
6     p = r / (2*P);
7     res = P + p;
```

If (T, t) are such that the correction t is below the last bit of T, i.e., $\mathrm{fl}(T+t) = T$, then the result of `AccSqrt` is almost always equal to $\sqrt{T}$, at most one bit apart. In [20, Theorem 3.6] the following error estimate is proved.

[1] There is a simulation of FMA in MATLAB's "Fixed-Point Designer"-toolbox, however, until Version 2023a there is no access to the FMA instruction provided by many processors.

Lemma 4 *Let $T, t \in \mathbb{F}$ be such that $\mathrm{fl}(T+t) = T$, and assume $\mathbf{u} \le 2^{-5}$. Let P, p be the final values in Algorithm* `AccSqrt` *when applied to the pair $[T, t]$. Then*

$$|P + p - \sqrt{T+t}| \le \frac{25}{8}\mathbf{u}^2\sqrt{T+t}.$$

The theorem estimates the error of $P+p$ rather than that of $\mathrm{fl}(P+p)$, otherwise the additional rounding error $\mathbf{u}$ would spoil the result.

Now we can display the Algorithms `normG` from [6] and `normL` from [20]. Recall that the latter used an FMA instruction to calculate P, p in line 2, that is, they use `P(i) = x(i).^2` and $p(i) = FMA(x(i), x(i), -P(i))$ inside the loop. Since the FMA instruction is not available in MATLAB, we replaced the computation in the loop by `[P,p] = TwoSquare(x)` splitting the whole vector x without loop. In that respect later time comparisons may be more fair.

Fig. 5 Algorithms by Graillat et al. [6] and Lefèvre et al. [20]

```
 1   function res = normG(x)
 2     S = 0;
 3     s = 0;
 4     for i=1:length(x)
 5       [P,p] = TwoProduct(x(i),x(i));
 6       [H,h] = TwoSum(S,P);
 7       c = s + p;
 8       d = h + c;
 9       [S,s] = FastTwoSum(H,d);
10     end
11     res = sqrt(S);
```

```
function res = normL(x)
  [P,p] = TwoSquare(x);
  [S,s] = TwoSum(P(1),P(2));
  for i=3:length(x)
    [H,h] = TwoSum(S,P(i));
    [S,s] = TwoSum(H,s+h);
  end
  sump = sum(p);
  [H,h] = TwoSum(S,sump);
  [S,s] = FastTwoSum(H,s+h);
  res = AccSqrt(S,s);
```

The summation scheme in Algorithms `normG` and `normL` is slightly different, but the main improvement is in the last line: Algorithm `normG` ignores the lower order part, whereas `normL` uses our Algorithm `AccSqrt` in Figure 4 to compute the square root approximation of the pair $S + s$. As we will see in Section 7 that produces often a nearest rounding.

An alternative is to use the double-double and the cpair toolbox directly:

Fig. 6 Algorithms using double-double and cpair arithmetic

```
1   function res = normDD(x)
2     S = sum(dd(x).*x);
3     res = AccSqrt(S.hi,S.lo);
```

```
function res = normCpair(x)
  S = sum(cpair(x).*x);
  res = AccSqrt(S.hi,S.lo);
```

The goal is to guarantee a faithfully rounded approximation to $\|x\|$ or even the rounded to nearest result. In [6] it is proved that, if computed in `binary64`, the result is a faithfully rounded approximation to $\|x\|$ if $n < (24\mathbf{u} + \mathbf{u}^2)^{-1}$, corresponding to $n \lesssim 3.7 \cdot 10^{14}$. Our cpair arithmetic proves similar conditions

for general arithmetic expressions. Applied to the Euclidean norm the the result is faithful for $n \leq (\beta\mathbf{u})^{-\frac{1}{2}}$ when using base-β arithmetic, and that corresponds to $n \lesssim 8.3 \cdot 10^7$ in `binary64`. In [20] we did not find an explicit limit for n, but the error estimates suggest that it should be a little larger than that for Algorithm `normG`.

In order to prove a faithful rounding for our algorithms to be presented we use the following criterion [17, Lemma 5.3]. That is a specialized version; the original allows for a much more general computer arithmetic.

Lemma 5 *Let $r, \delta \in \mathbb{R}$ and assume $|\delta| < \frac{\mathbf{u}}{2-\mathbf{u}}|r|$. Then $\mathrm{fl}(r)$ is a faithful rounding of $r + \delta$.*

In a typical application a pair (T, t) with $r := T + t$ is an approximation to some real quantity q. If $|r - q| < \frac{\mathbf{u}}{2-\mathbf{u}}|r|$, then $\mathrm{fl}(r)$ is a faithful rounding of q. An application is the following criterion that $\mathrm{fl}(T + t)$ is a faithful rounding of $q := \sqrt{x}$.

Lemma 6 *Let $T, t \in \mathbb{F}$ with $T + t > 0$ be given, and let $0 \leq q \in \mathbb{R}$. Assume*

$$|T + t - q^2| \leq \alpha q^2 \tag{5}$$

for some $\alpha \in \mathbb{R}$ with $\alpha < 1$. Let $r \in \mathbb{R}$ be such that

$$|r - \sqrt{T + t}| \leq \beta\sqrt{T + t} \tag{6}$$

for some $\beta < 1$. Then

$$(1 - \beta)^{-1}\left(\beta + \frac{\alpha}{2(1 - \alpha)}\right) < \frac{\mathbf{u}}{2 - \mathbf{u}} \tag{7}$$

implies that $\mathrm{fl}(r)$ is a faithful rounding of q.

Proof Note that

$$|\sqrt{x} - \sqrt{y}| = \frac{|x - y|}{\sqrt{x} + \sqrt{y}}$$

for positive $x, y \in \mathbb{R}$. We show

$$|\sqrt{T + t} - q| \leq \frac{\alpha}{2(1 - \alpha)}\sqrt{T + t}. \tag{8}$$

We distinguish two cases. First, suppose $T + t \leq q^2$. Then (5) gives

$$|\sqrt{T + t} - q| = \frac{|T + t - q^2|}{\sqrt{T + t} + q} \leq \frac{\alpha q^2}{2\sqrt{T + t}} \leq \frac{\alpha}{2(1 - \alpha)}\sqrt{T + t}.$$

Second, suppose $T + t > q^2$. Then using again $q^2 \leq \frac{T+t}{1-\alpha}$ and (5) give

$$|\sqrt{T + t} - q| \leq \frac{\alpha q^2}{2q} \leq \frac{\alpha}{2\sqrt{1 - \alpha}}\sqrt{T + t} \leq \frac{\alpha}{2(1 - \alpha)}\sqrt{T + t}$$

and proves (8). Hence (6) yields

$$|r - q| \leq |r - \sqrt{T + t}| + |\sqrt{T + t} - q| \leq \left(\beta + \frac{\alpha}{2(1 - \alpha)}\right)\sqrt{T + t},$$

and $r \geq (1 - \beta)\sqrt{T + t}$ together with Lemma 5 implies the result. □

In our applications α is very small. A sufficient criterion that `AccSqrt(T,t)` is a faithful rounding of $\sqrt{x}$ follows.

Corollary 1 *Let $T, t \in \mathbb{F}$ with $\mathrm{fl}(T+t) = T$, assume $\mathbf{u} \le 2^{-8}$ and let res be the result of Algorithm* `AccSqrt` *applied to $[T,t]$. If $0 \le x \in \mathbb{R}$ satisfies*

$$|T + t - x| \le \frac{31}{32}\mathbf{u}x,$$

then res is a faithful rounding of $\sqrt{x}$.

Proof Let $[P,p]$ be the final values in Algorithm `AccSqrt` when applied to the pair $[T,t]$, so that $res = \mathrm{fl}(P+p)$. Lemma 4 shows that (6) is true for $r := P + p$ and $\beta := \frac{25}{8}\mathbf{u}^2$. Moreover, (5) is true by assumption for $\alpha := \frac{31}{32}\mathbf{u}$ and $q := \sqrt{x}$. Hence (7) is true if, and only if,

$$d := (2-\mathbf{u})(2(1-\alpha)\beta + \alpha) - 2(1-\alpha)(1-\beta)\mathbf{u} < 0.$$

Using $\mathbf{u} \le 2^{-8}$ yields $64d = -4\mathbf{u} + 862\mathbf{u}^2 - 775\mathbf{u}^3 \le \left(-4 + 862 \cdot 2^{-8}\right)\mathbf{u} < 0$, and Lemma 6 finishes the proof. □

3 Faithfully rounding of $\|x\|$ based on relative splitting of x

The Algorithms `TwoProduct` and `TwoSquare` as in Figure 3 apply to vector input as well. As a consequence we obtain the following lemma.

Lemma 7 *For $a, b \in \mathbb{F}^n$ the output $[P,p]$ of Algorithm* `TwoProduct` *as in Figure 3 applied to a, b satisfies $\sum_{i=1}^n P_i + p_i = \sum_{i=1}^n a_i b_i$, and the output $[P,p]$ of Algorithm* `TwoSquare` *applied to a satisfies $\sum_{i=1}^n P_i + p_i = \sum_{i=1}^n a_i^2$. Furthermore, $P_i \ge 0$ and $|p_i| \le \mathbf{u}P_i$ for all $i \in \{1, \ldots, n\}$.*

Thus one way to approximate $\|x\|$ is to compute vectors $P, p \in \mathbb{F}^n$ with $P + p = \|x\|^2$ and apply some accurate summation algorithm. Both [6] and [20] follow that scheme. Note that the vectors P, p are computed based on a relative splitting of x; later we will an absolute splitting.

In [25] efficient summation algorithms are developed based on `TwoSum`. First, `q = VecSum(p)` transforms an input vector p into a vector q without changing its sum S but with the property that $q_{1\ldots n-1}$ is small in absolute value and $q_n = \mathrm{float}(\sum_{i=1}^n p_i)$. The error estimates in [25] imply that $res = \mathrm{float}(\sum_{i=1}^n q_i)$ is a very good approximation of the true sum S.

Before continuing, we need to estimate the error of ordinary floating-point summation. To that end the traditional Wilkinson-type estimate γ_{n-1} can be used. However, new and optimal bounds are available. The following sharp bound was shown in [16, Theorem 5].

Lemma 8 *For $p \in \mathbb{F}^n$ denote $S = \mathrm{float}(\sum_{i=1}^n p_i)$ for summation in any order, and denote by δ_i the errors in the $n-1$ nodes of the evaluation tree. Hence $\sum_{i=1}^n p_i = S + \sum_{i=1}^{n-1} \delta_i$. Suppose $n \le 1 + \frac{1}{2}\mathbf{u}^{-1}$. Then*

$$|S - \sum_{i=1}^{n} p_i| \le \sum_{i=1}^{n-1} |\delta_i| \le \varphi_{n-1} \sum_{i=1}^{n} |p_i| \qquad \text{with} \quad \varphi_k := \frac{k\mathbf{u}}{1+k\mathbf{u}}, \tag{9}$$

and that bound is sharp as for the input vector $p = (1, \mathbf{u}, \ldots, \mathbf{u})^T$.

The Algorithm `VecSum` is realized by a loop in [25]. In MATLAB we face some interpretation overhead, so loops should be avoided where possible. That has been done in `TwoSquare`, and next we give a new, loop-free version of `VecSum`, see Figure 7.

Fig. 7 Error-free vector transformation

```
1   function p = VecSum(p)
2     n = length(p);
3     for i=2:n
4       [p(i),p(i-1)] = TwoSum(p(i),p(i-1));
5     end
6
7
8
```

```
function [S,s] = FastVecSum(p)
  n = length(p);
  x = cumsum(p);
  a = [ 0 ; x(1:n-1) ];
  z = x - a;
  y = ( a - (x-z) ) + (p-z);
  S = x(n);
  s = y(2:n);
```

It is easily verified that Algorithms `VecSum` and `FastVecSum` produce identical results. The error analysis follows by Lemma 8.

Lemma 9 *For given $p \in \mathbb{F}^n$ let $[S, s]$ be the output of Algorithm* `FastVecSum`*. Suppose $n \le 1 + \frac{1}{2}\mathbf{u}^{-1}$. Then $s \in \mathbb{F}^{n-1}$, $\sum_{i=1}^n p_i = S + \sum_{i=1}^{n-1} s_i$ and*

$$\sum_{i=1}^{n-1} |s_i| \le \varphi_{n-1} \sum_{i=1}^{n} |p_i| \qquad \text{with} \quad \varphi_k := \frac{k\mathbf{u}}{1+k\mathbf{u}}, \tag{10}$$

and that bound is sharp as by the input vector $p = (1, \mathbf{u}, \ldots, \mathbf{u})^T$.

We mention that (10) is true [15, Theorem 2.1] without restriction on n when replacing φ_k by $\frac{k\mathbf{u}}{1+\mathbf{u}}$.

Our first algorithm is based on Algorithm `Sum2` in [25], which in turn is equivalent to Algorithm IV in [24].

Theorem 1 *Let res be the result of Algorithm* `normSum2` *applied to $x \in \mathbb{F}^n$. Suppose $n \le \sqrt{\frac{31}{32}}\mathbf{u}^{-1/2}$ and $\mathbf{u} \le 2^{-8}$. Then res is a faithful rounding of $\|x\|$.*

Proof We will prove

$$|T + t - \sum_{i=1}^{n} x_i^2| \le \frac{31}{32}\mathbf{u} \sum_{i=1}^{n} x_i^2 \tag{11}$$

Fig. 8 Algorithm normSum2

```
1   function res = normSum2(x)
2     [P,p] = TwoSquare(x(:));
3     [S,s] = FastVecSum(P);
4     [T,t] = FastTwoSum(S,sum(s)+sum(p));
5     res = AccSqrt(T,t);
```

for the scalars $[T, t]$ computed in line 4 of Algorithm `normSum2` in order to apply Corollary 1. We know

$$\sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} (P_i + p_i) \quad \text{and} \quad |p_i| \le \mathbf{u} P_i \quad \text{for all } i \in \{1, \ldots, n\} \tag{12}$$

by Lemma 7, so that Lemma 9 implies

$$\sum_{i=1}^{n} P_i = S + \sum_{i=1}^{n-1} s_i \quad \text{and} \quad \sum_{i=1}^{n-1} |s_i| \le \varphi_{n-1} \sum_{i=1}^{n} P_i.$$

Denote the floating-point sum `sum(s)` by σ_s, and correspondingly of the floating-point sum `sum(p)` by σ_p. Note that $s \in \mathbb{F}^{n-1}$ and $p \in \mathbb{F}^n$. Then Lemma 8 gives

$$|\sigma_s - \sum_{i=1}^{n-1} s_i| \le \varphi_{n-2} \sum_{i=1}^{n-1} |s_i| \le \varphi_{n-2}\varphi_{n-1} \sum_{i=1}^{n} P_i$$

and

$$|\sigma_p - \sum_{i=1}^{n} p_i| \le \varphi_{n-1} \sum_{i=1}^{n-1} |p_i| \le \varphi_{n-1}\mathbf{u} \sum_{i=1}^{n} P_i.$$

Furthermore, $T + t = S + \mathrm{fl}(\sigma_s + \sigma_p)$. Hence, using $S + \sum_{i=1}^{n-1} s_i + \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} x_i^2$,

$$\begin{aligned} |T + t - \textstyle\sum_{i=1}^{n} x_i^2| &\le |S + \sigma_s + \sigma_p - \textstyle\sum_{i=1}^{n} x_i^2| + \mathbf{u}|\sigma_s + \sigma_p| \\ &\le (\varphi_{n-2}\varphi_{n-1} + \varphi_{n-1}\mathbf{u}) \textstyle\sum_{i=1}^{n} P_i + \mathbf{u}|\sigma_s + \sigma_p|. \end{aligned}$$

Hence

$$|\sigma_s| \le (1 + \varphi_{n-2}) \sum_{i=1}^{n-1} |s_i| \le (1 + \varphi_{n-2})\, \varphi_{n-1} \sum_{i=1}^{n} P_i$$

and

$$|\sigma_p| \le (1 + \varphi_{n-1}) \sum_{i=1}^{n-1} |p_i| \le (1 + \varphi_{n-1})\, \mathbf{u} \sum_{i=1}^{n} P_i$$

and a calculation shows

$$\begin{aligned} |T + t - \textstyle\sum_{i=1}^{n-1} x_i^2| &\le \left(\varphi_{n-1}\left(\varphi_{n-2} + \mathbf{u} + \mathbf{u} + \varphi_{n-2}\mathbf{u} + \mathbf{u}^2\right) + \mathbf{u}^2\right) \textstyle\sum_{i=1}^{n} P_i \\ &= \left(\varphi_{n-1}\left((1+\mathbf{u})\varphi_{n-2} + 2\mathbf{u} + \mathbf{u}^2\right) + \mathbf{u}^2\right) \textstyle\sum_{i=1}^{n} P_i \\ &\le \left(\varphi_{n-1}\,(n + \mathbf{u})\,\mathbf{u} + \mathbf{u}^2\right) \textstyle\sum_{i=1}^{n} P_i \\ &\le (n^2 - n + 1)\mathbf{u}^2 \textstyle\sum_{i=1}^{n} P_i. \end{aligned}$$

For $n = 1$ the left hand side in (11) is zero and the result is faithful by Corollary 1. For $n \geq 2$, we use (12) to see

$$\sum_{i=1}^{n} P_i = |\sum_{i=1}^{n} x_i^2 - p_i| \leq \sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} |p_i| \leq \sum_{i=1}^{n} x_i^2 + \mathbf{u} \sum_{i=1}^{n} P_i, \qquad (13)$$

and again by Corollary 1 the result is faithful if $32(n^2 - n + 1)\mathbf{u} \leq 31(1 - \mathbf{u})$, and a computation shows that this is true because $n \leq \sqrt{\frac{31}{32}}\mathbf{u}^{-1/2}$. □

Algorithm `VecSum` is an error-free vector transformation, so as in [25] we may apply it a second time, thus further diminishing the condition number of the sum.

Fig. 9 Algorithm normSum3

```
1   function res = normSum3(x)
2     [P,p] = TwoSquare(x(:));
3     [Q,q] = FastVecSum(P);
4     [S,s] = FastVecSum([p;q]);
5     [T,t] = FastTwoSum(Q,S+sum(s));
6     res = AccSqrt(T,t);
```

Theorem 2 *Let $x \in \mathbb{F}^n$ be given and apply Algorithm normSum3 to x. Suppose $n \leq (\frac{17}{4}\mathbf{u}^2)^{-1/3}$ and $\mathbf{u} \leq 2^{-8}$. Then res is a faithful rounding of $\|x\|$.*

Proof We proceed as in the proof of Theorem 1 and show that the scalars $[T, t]$ in Algorithm `normSum3` satisfy

$$|T + t - \sum_{i=1}^{n-1} x_i^2| \leq \frac{31}{32}\mathbf{u} \sum_{i=1}^{n} x_i^2. \qquad (14)$$

The quantities in Algorithm `normSum3` are scalars Q, S, T and t as well as vectors $P, p \in \mathbb{F}^n$, $q \in \mathbb{F}^{n-1}$ and $s \in \mathbb{F}^{2n-2}$. As before $\sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n}(P_i + p_i)$ with $|p_i| \leq \mathbf{u}P_i$ for all $i \in \{1, \ldots, n\}$. Furthermore, Lemma 9 implies $\sum_{i=1}^{n} P_i = Q + \sum_{i=1}^{n-1} q_i$ and $\sum_{i=1}^{n} p_i + \sum_{i=1}^{n-1} q_i = S + \sum_{i=1}^{2n-2} s_i$ as well as $\sum_{i=1}^{n-1} |q_i| \leq \varphi_{n-1} \sum_{i=1}^{n} P_i$ and $\sum_{i=1}^{2n-2} |s_i| \leq \varphi_{2n-2} \left(\sum_{i=1}^{n} |p_i| + \sum_{i=1}^{n-1} |q_i| \right)$. Denote the floating-point sum `sum(s)` by σ_s. Then

$$|\sigma_s - \sum_{i=1}^{2n-2} s_i| \leq \varphi_{2n-3} \sum_{i=1}^{2n-2} |s_i|.$$

Furthermore, $T + t = Q + \mathrm{fl}(S + \sigma_s)$. Hence

$$\begin{aligned}
|T + t - \textstyle\sum_{i=1}^{n} x_i^2| &\le |Q + S + \sigma_s - \textstyle\sum_{i=1}^{n} x_i^2| + \mathbf{u}|S + \sigma_s| \\
&= |\sigma_s - \textstyle\sum_{i=1}^{2n-2} s_i| + \mathbf{u}|S + \textstyle\sum_{i=1}^{2n-2} s_i + \sigma_s - \textstyle\sum_{i=1}^{2n-2} s_i| \\
&\le (1+\mathbf{u})|\sigma_s - \textstyle\sum_{i=1}^{2n-2} s_i| + \mathbf{u}|\textstyle\sum_{i=1}^{n} p_i + \textstyle\sum_{i=1}^{n-1} q_i| \\
&\le \varphi_{2n-3}(1+\mathbf{u})\textstyle\sum_{i=1}^{2n-2} |s_i| + \mathbf{u}|\textstyle\sum_{i=1}^{n} p_i + \textstyle\sum_{i=1}^{n-1} q_i| \\
&\le (\varphi_{2n-3}\varphi_{2n-2}(1+\mathbf{u}) + \mathbf{u})\left(\textstyle\sum_{i=1}^{n} |p_i| + \textstyle\sum_{i=1}^{n-1} |q_i|\right) \\
&\le (\varphi_{2n-3}\varphi_{2n-2}(1+\mathbf{u}) + \mathbf{u})\,(\mathbf{u} + \varphi_{n-1})\textstyle\sum_{i=1}^{n} P_i \\
&=: \Phi \textstyle\sum_{i=1}^{n} P_i
\end{aligned}$$

and using (13) yields

$$|T + t - \sum_{i=1}^{n} x_i^2| \le (1-\mathbf{u})^{-1}\Phi \sum_{i=1}^{n} x_i^2.$$

The factor Φ is monotonically increasing in n. A direct computation for $\mathbf{u} \in \{2^{-e} : 8 \le e \le 53\}$ and the maximal value $n := \lfloor (\frac{17}{4}\mathbf{u}^2)^{-1/3} \rfloor$ verifies

$$(1-\mathbf{u})^{-1}\Phi \le \frac{31}{32}\mathbf{u}.$$

Hence (14) is true and Corollary 1 finishes the proof. □

The error of floating-point summation in Lemma 9 is sharp but, as has been mentioned after Lemma 1, highly overestimated in practice: We hardly find cases with relative error exceeding $\sqrt{n}\mathbf{u}$ - unless we looked for them. In particular it seems unlikely that the worst case bound (10) is attained for all summations in Algorithms `normSum2` or `normSum3`.

Theorems 1 and 2 prove that Algorithms `normSum2` and `normSum3` compute a faithfully rounded result if the vector length n satisfies $\frac{32}{31}n^2\mathbf{u} \le 1$ or $4n^3\mathbf{u}^2 \le 1$, respectively. These are sufficient criteria, but in practice the results are faithful for much larger n.

A rough estimate of this limit under practical assumptions, i.e., when replacing φ_k in Lemma 8 by $\sqrt{k}\mathbf{u}$, suggests a faithfully rounded result for $n \lesssim \mathbf{u}^{-1}$ for Algorithm `normSum2` and $n \lesssim \frac{1}{4}\mathbf{u}^{-4/3}$ for Algorithm `normSum3`. In other words, in practical applications it suffices to use Algorithms `normSum2` and we can always expect a faithfully rounded result.

4 Faithfully rounding of $\|x\|$ based on absolute splitting of x

The error-free transformation of $\|x\|^2$ into (P, p) with $P + p = \|x\|^2$, as used in [6] and [20] and our algorithms up to now, is based on a relative splitting of the x_i, i.e., each x_i is transformed into a sum of two floating-point numbers.

Once the vectors P, p are available, any good summation algorithm may be applied. An alternative to a relative splitting of the x_i was first proposed in [32]. A constant σ larger in absolute value than all summands is chosen. The split of the vector x into a pair of vectors (r, s) with respect to σ is such that $x = r + s$ and all bits of r reside in the same range and such that the `sum(r)` is error-free. The same principle can be applied successively.

In [32] the splitting was performed using scaling and integer rounding, and no analysis was given. In [28] we pursued that principle in Algorithm `AccSum` with an efficient implementation and thorough error analysis. Based on that Algorithm `AccDot` is presented in [28] for the accurate computation of a dot product $x^T y$. Basically, it first splits $x^T y = r + s$ as in `TwoProduct` and then applies `AccSum`. That algorithm can be used for $\|x\|$ as well.

Following we split the input vector x into vectors q, b directly such that `sum(q.*q)` is error-free. That avoids the costly splitting $\|x\|^2 = P + p$ by Algorithm `TwoSquare`. The Algorithm `normExtract` is presented in Figure 10. Note that `M`, in contrast to Algorithm `AccSum` in [28], is not a power of 2.

Fig. 10 Algorithm normExtract

```
 1   function res = normExtract(x)
 2     x = abs(x);
 3     u = pow2(-53); % u relative rounding error unit
 4     M = 4/(2-(length(x)+8)*u) * norm(x)/sqrt(u);
 5     q = ( M + x ) - M;
 6     b = x - q;            % x = q + b
 7     S = sum(q.*q);
 8     s = sum( (q+x).*b );
 9     [T,t] = FastTwoSum(S,s);
10     res = AccSqrt(T,t);
```

The bound on the dimension n as for Algorithm `normSum2` to guarantee that the approximation res is a faithful rounding of $\|x\|$ is very conservative. We present this algorithm because it is very fast and, as explained at the end of the previous section, we can expect a faithful result up to $n \lesssim 79$ million. That may be sufficient in most practical applications.

To that end we need "ufp" as introduced in [28], the unit in the first place

$$0 \neq r \in \mathbb{R} \quad \Rightarrow \quad \mathrm{ufp}(r) := 2^{\lfloor \log_2 |r| \rfloor}$$

and $\mathrm{ufp}(0) := 0$. Compared to the often used "ulp", the unit in the last place, it bears the advantage that it is independent of a floating-point format and applies to real numbers as well. The following properties are proved in [28]. For $\sigma = 2^k, k \in \mathbb{Z}, r \in \mathbb{R}$ we have

$$r \neq 0 \quad \Rightarrow \quad \mathrm{ufp}(r) \leq |r| < 2\mathrm{ufp}(r) \tag{15}$$

$$\sigma' = 2^m,\ m \in \mathbb{Z},\ \sigma' \geq \sigma \quad \Rightarrow \quad \mathbf{u}\sigma'\mathbb{Z} \subseteq \mathbf{u}\sigma\mathbb{Z} \tag{16}$$

$$f \in \mathbb{F} \quad \text{and} \quad |f| \geq \sigma \quad \Rightarrow \quad \mathrm{ufp}(f) \geq \sigma \tag{17}$$

$$f \in \mathbb{F} \quad \Rightarrow \quad f \in 2\mathbf{u} \cdot \mathrm{ufp}(f)\mathbb{Z} \tag{18}$$

$$r \in \mathbf{u}\sigma\mathbb{Z} \cap \mathcal{N},\ |r| \le \sigma \quad \Rightarrow \quad r \in \mathbb{F} \tag{19}$$

$$a, b \in \mathbb{F},\ a \ne 0 \quad \Rightarrow \quad \mathrm{fl}(a+b) \in \mathbf{u} \cdot \mathrm{ufp}(a)\mathbb{Z} \tag{20}$$

$$\tilde{r} := \mathrm{fl}(r) \in \mathbb{F} \quad \Rightarrow \quad |\tilde{r} - r| \le \mathbf{u} \cdot \mathrm{ufp}(r) \le \mathbf{u} \cdot \mathrm{ufp}(\tilde{r}). \tag{21}$$

Note that, if $b \ne 0$, $\mathrm{fl}(a+b) \in \mathbf{u} \cdot \mathrm{ufp}(b)\mathbb{Z}$ in (20) holds as well.

Theorem 3 *Let $x \in \mathbb{F}^n$ be given and apply Algorithm* `normExtract` *to x. Suppose $n \le \frac{11}{59}\mathbf{u}^{-1/3}$ and $\mathbf{u} \le 2^{-8}$. Then res is a faithful rounding of $\|x\|$.*

Proof As in the previous proofs we will show that the scalars $[T, t]$ in Algorithm `normExtract` satisfy

$$|T + t - \sum_{i=1}^{n} x_i^2| \le \frac{31}{32}\mathbf{u} \sum_{i=1}^{n} x_i^2. \tag{22}$$

Henceforth we assume $x_i \ge 0$ as justified by line 2 of Algorithm `normExtract`. Denote by $\hat{x} :=$ `norm(x)` MATLAB's floating-point approximation to $\|x\|$. Then Lemma 1 with $\beta = (\frac{n}{2} + 1)\mathbf{u}$ shows

$$(1 - \beta)\|x\| \le \hat{x} \le (1 + \beta)\|x\|. \tag{23}$$

Note that $4(n+2)\mathbf{u} \le 16n\mathbf{u} \le 1$ implies $1 - 2(n+2)\mathbf{u} \ge 1/2$, so that $\mathrm{float}(1 - 2(n+2)\mathbf{u}) = 1 - 2(n+2)\mathbf{u}$ by Sterbenz' lemma [31], and a calculation using (1) yields for all $i \in \{1, \ldots, n\}$

$$\begin{aligned} M = \mathrm{float}(\varphi\hat{x}) &\ge \tfrac{4\hat{x}}{(1+\mathbf{u})^3(2-(n+8)\mathbf{u})\sqrt{\mathbf{u}}} \ge \tfrac{4\hat{x}}{(2-(n+2)\mathbf{u})\sqrt{\mathbf{u}}} = \tfrac{2\hat{x}}{(1-\beta)\sqrt{\mathbf{u}}} \\ &\ge 2\|x\|/\sqrt{\mathbf{u}} \\ &\ge 32\|x\| \ge x_i, \end{aligned} \tag{24}$$

where $\varphi := 4/(2 - (n+8)\mathbf{u})/\sqrt{\mathbf{u}}$. Lines 5 and 6 of Algorithm `normExtract` are similar to Algorithm `FastTwoSum` in Figure 1. More precisely, the code for `FastTwoSum(M,x)` is identical to

```
N = M + x;
qs = M - N;
b = qs + x;
```

where `q` in line 5 of Algorithm `normExtract` is equal to `-qs`, and `b` in line 6 is the same. By (24), Lemma 2 for Algorithm `FastTwoSum` is applicable, so that there is no rounding error when subtracting M in line 5, i.e., $q_i = \mathrm{fl}(M + x_i) - M$. Using $\mathrm{ufp}(M + x) \le 2\mathrm{ufp}(M)$ by (24) that implies

$$x_i = q_i + b_i \quad \text{and} \quad |b_i| \le 2\mathbf{u} \cdot \mathrm{ufp}(M) \tag{25}$$

and $q_i \le x_i + \mathbf{u} \cdot \mathrm{ufp}(M + x_i)$ for all $i \in \{1, \ldots, n\}$. We distinguish three cases to show

$$q_i \le 2x_i. \tag{26}$$

First, if $2\mathbf{u}\cdot\mathrm{ufp}(M) \le x_i$, then $\mathrm{ufp}(M+x_i) \le 2\mathrm{ufp}(M)$ proves (26). Second, if $\mathbf{u}\cdot\mathrm{ufp}(M) \le x_i < 2\mathbf{u}\cdot\mathrm{ufp}(M)$, then $\mathrm{ufp}(M+x) = \mathrm{ufp}(M)$ and proves (26) as well. Third and finally, if $x_i < \mathbf{u}\cdot\mathrm{ufp}(M)$, then $\mathrm{fl}(M+x_i) = M$ and $q_i = 0$. Thus (26), (24) and (15) yield

$$\textstyle\sum_{i=1}^n q_i^2 \le 4\|x\|^2 \le \mathbf{u}M^2 < 4\mathbf{u}\cdot\mathrm{ufp}(M)^2.$$

Now (20) and (16) yield $q_i \in \mathbf{u}\cdot\mathrm{ufp}(M+x)\mathbb{Z} \subseteq 2\mathbf{u}\cdot\mathrm{ufp}(M)\mathbb{Z}$. Hence $q_i^2 \in \mathbf{u}\cdot 4\mathbf{u}\cdot\mathrm{ufp}(M)^2\mathbb{Z}$ and (19) show that the floating-point sum of all q_i^2 is error-free, i.e., $S = \sum_{i=1}^n q_i^2$. For $c_i := \mathrm{float}((q_i+x_i)b_i)$ we see by (1) that

$$|c_i - (q_i+x_i)b_i| \le ((1+\frac{\mathbf{u}}{1+\mathbf{u}})^2 - 1)|q_i+x_i||b_i| \le 2\mathbf{u}|q_i+x_i||b_i|$$

and, if $n \ge 3$,

$$|\mathrm{float}(\sum_{i=1}^n c_i) - \sum_{i=1}^n c_i| \le \frac{(n-1)\mathbf{u}}{1+(n-1)\mathbf{u}} \sum_{i=1}^n |c_i| \le (n-1)\mathbf{u}\sum_{i=1}^n |q_i+x_i||b_i|. \quad (27)$$

Moreover, using (23) and $(1+\frac{\mathbf{u}}{1+\mathbf{u}})^3 \le 1+3\mathbf{u}$,

$$M = \mathrm{float}(\varphi\hat{x}) \le \frac{4(1+3\mathbf{u})}{(2-(n+8)\mathbf{u})\sqrt{\mathbf{u}}}\left(1+(\frac{n}{2}+1)\mathbf{u}\right)\|x\| =: \gamma\|x\|. \quad (28)$$

Thus $s = \mathrm{float}(\sum_{i=1}^n c_i)$, $x_i = q_i + b_i$, (26) and (25) yield

$$\begin{aligned}
|T+t-\|x\|^2| &= |S+s-\|x\|^2| = |\textstyle\sum_{i=1}^n q_i^2 + s - \|x\|^2| \\
&= |s - \textstyle\sum_{i=1}^n (q_i+x_i)b_i| \\
&\le (n-1)\mathbf{u}\textstyle\sum_{i=1}^n |q_i+x_i||b_i| \\
&\le (n-1)\mathbf{u}\cdot 3\|x\|_1 \cdot 2\mathbf{u}M \\
&\le 6(n-1)\mathbf{u}^2\sqrt{n}\gamma\|x\|^2.
\end{aligned}$$

In order to show (22) we note that $6(n-1)\mathbf{u}^2\sqrt{n}\gamma \le \frac{31}{32}\mathbf{u}$ is equivalent to

$$384(n-1)\sqrt{n\mathbf{u}}(1+3\mathbf{u})(2+(n+2)\mathbf{u}) < 31(2-(n+8)\mathbf{u}),$$

which in turn is equivalent to $\Phi := \sum_{i=1}^3 \alpha_i\mathbf{u}^{i/2} + \alpha_5\mathbf{u}^{5/2} < 62$ where

$$\begin{aligned}
\alpha_1 &= 768(n-1)\sqrt{n} \\
\alpha_2 &= 31n + 248 \\
\alpha_3 &= (384n^2 + 2688n - 3072)\sqrt{n} \\
\alpha_5 &= (1152n^2 + 1152n - 2304)\sqrt{n}.
\end{aligned}$$

Now Φ is monotonically increasing in n, and a direct computation using the maximal value $n := \lfloor \frac{11}{59}\mathbf{u}^{-1/3} \rfloor$ shows

$$\Phi < 61.9 + 12\mathbf{u}^4 + 465\mathbf{u}^6 + 18\mathbf{u}^{10} + 93\mathbf{u}^{12}$$

and verifies (22) for $\mathbf{u} \le 2^{-8}$ and $n \ge 3$. The case $n = 2$ follows by an extra factor $\frac{1+2\mathbf{u}}{1+\mathbf{u}}$ in (27). Hence (14) is true and Corollary 1 finishes the proof. □

The reason for the severe restriction of the vector length n to guarantee a faithfully rounded result is the estimate (27). As before, a rough estimate under the practical assumption $\varphi_k \lesssim \sqrt{k}\mathbf{u}$ in Lemma 8 suggests a faithfully rounded result for $n \lesssim \frac{1}{12}\mathbf{u}^{-1/2}$ for Algorithm `normExtract`.

The limit on the dimension for guaranteed faithful rounding is improved by the following Algorithm `normExtract2` by introducing a second splitting.

Fig. 11 Algorithm normExtract2

```
 1   function res = normExtract2(x)
 2     x = abs(x);
 3     u = pow2(-53);  % u relative rounding error unit
 4     M = 4/(2-(n+8)*u) * norm(x)/sqrt(u);
 5     q = ( M + x ) - M;
 6     b = x - q;      % x = q + b
 7     N = 8/(2-(n+8)*u) * norm(b)/sqrt(u);
 8     r = ( N + b ) - N;
 9     c = b - r;      % b = r + c
10     [P,p1] = FastTwoSum( sum(2*(q+r).*c), sum(c.*c) );
11     [P,p2] = FastTwoSum( sum(r.*r) , P );
12     [P,p3] = FastTwoSum( 2*sum(q.*r) , P );
13     [P,p4] = FastTwoSum( sum(q.*q) , P );
14     [S,s] = FastTwoSum(P,p1+(p2+(p3+p4)));
15     res = AccSqrt(S,s);
```

We show this algorithm as yet another example to compute the Euclidean norm faithfully, however, we refrain from giving a complete analysis. We just mention that the main errors occur in line 10, namely, the summation of $2(q_i + r_i)c_i$ and c_i^2. The following sums of the r_i^2, the $q_i r_i$ and q_i^2 are error-free.

5 Computation of the nearest rounding of $\|x\|$

The algorithms in the previous section adapt Algorithm `AccSum` in [28] to the computation of the Euclidean norm of a vector. In [29] we explored that principle by designing Algorithm `NearSum` to compute the rounded to nearest value of the sum of floating-point numbers, and Algorithm `AccSign` to compute the sign of the sum. Several other algorithms such as storing the result in an unevaluated vector, the rounded downward and upward result, treatment of vectors of huge lengths and more.

Next we derive Algorithm `normNearest` to compute the nearest value of the Euclidean norm of a vector. To that end we first present an adapted version of the Algorithm `Transform` derived in [28]. In our adaptation we rewrote the "repeat"- into a "while"-loop and omitted the output parameter σ. Then Lemma 4.3 in [28] shows the following.

Fig. 12 Algorithm Transform

```
 1   function [tau1,tau2,p] = Transform(p)
 2     M = ceil(log2(length(p)+2));
 3     Phi = pow2(2*M);
 4     u = 2^(-53);            % u rel. rounding error unit, to be adapted
 5     sigma = pow2(ceil(log2(max(abs(p)))))/u;
 6     t = 0;
 7     while ( abs(t) < Phi*sigma )
 8       sigma = u*pow2(M)*sigma;
 9       q = ( sigma + p ) - sigma;
10       p = p - q;
11       tau = sum(q);
12       told = t;
13       t = t + tau;
14     end
15     [tau1,tau2] = FastTwoSum(told,tau);
```

Lemma 10 *Let* `tau1`, `tau2` *and* `r` *be the result of Algorithm* `Transform` *applied to* $p \in \mathbb{F}^k$, *and suppose* $k \leq \frac{1}{2}\mathbf{u}^{-1/2} - 2$. *Then*

$$\sum_{i=1}^{k} p_i = \tau_1 + \tau_2 + \sum_{i=1}^{k} r_i, \tag{29}$$

and the MATLAB statement

```
res = tau1 + (tau2 + sum(r))
```

implies that `res` *is a faithful rounding of* $\sum_{i=1}^{k} p_i$. *Moreover,*

$$\max_{1\leq i\leq n} |r_i| \leq 2^{-2M}\mathbf{u}|\tau_1| \quad \textit{and} \quad |\tau_2| \leq \mathbf{u}|\tau_1|. \tag{30}$$

When replacing the constant Φ *in line* 3 *by* $\Phi = 2^M$, *then* τ_1 *and* $\sum_{i=1}^{k} p_i$ *have the same sign under the weaker assumption* $k \leq \frac{1}{2}\mathbf{u}^{-1} - 2$.

Proof The definition of M in line 2 implies $2^M \geq k + 2 \geq 2^{M-1}$ and therefore

$$2^{2M}\mathbf{u} \leq 4(k+2)^2\mathbf{u} \leq 1.$$

Hence the the assumptions of Lemma 4.3 in [28] are satisfied, and the assertions until (30) follow. The last statement is implied by Theorem 4.2 in [29]. □

The smaller the constant Φ is, the less "while"-loops are necessary in Algorithm `Transform`. As shown in [28] and [29] the chosen constants $\Phi = 2^{2M}$ for a faithful result and $\Phi = 2^M$ for the sign are optimal.

Our Algorithm `NearSum` needs the predecessor and successor of a floating-point number. The next Algorithm `PredSucc` combines Algorithm 1 in [30].

In Theorem 2.2 in [30] it is shown that Algorithms `Pred` and `Succ` computes the predecessor and successor of a floating-point number c provided that $\mathbf{u} \leq \frac{1}{16}$

Fig. 13 Predecessor and successor of c

```
1   function [pred,succ] = PredSucc(c)
2     C = pow2(53)*c;
3     u = pow2(-53);
4     e = ( u*(1+2*u) ) * abs(C);
5     pred = (C-e)*u;
6     succ = (C+e)*u;
```

and except for a tiny range near the smallest positive normalized floating-point number. To avoid that, we scaled the input in line 2 so that, provided no overflow occurs, Algorithm `PredSucc` computes the predecessor and successor of c. Of course, proper scaling avoids overflow.

Now we can present our Algorithm `normNearest` in Figure 14 to compute the nearest value of the Euclidean norm of a vector. It borrows from Algorithm `NearSum` in [29] and is adapted to our task.

Fig. 14 Algorithm for round to nearest Euclidean norm

```
 1   function res = normNearest(x)
 2     [S,s] = TwoSquare(x(:));
 3     [tau1,tau2,p] = Transform([S;s]);
 4     f = tau1 + ( tau2 + sum(p) );  % f is faithful rounding of ||x||^2
 5     delta = Transforms([ tau1 ; tau2 ; p ; -f ]);
 6     [f2,succ] = predsucc(f);
 7     if delta>0                % hull(f,f2) bracket |x||^2
 8       f2 = succ;
 9     elseif delta=0            % fl(|x||^2) = f
10       f2 = f;
11     end
12     g1 = sqrt(f);             % fl(|x||^2) in {f,f2}
13     g2 = sqrt(f2);            % fl_near(|x||) in {g1,g2}
14     if g1==g2                 % fl_near(||x||) = g1 = g2
15       res = g1;
16     else
17       [R,r] = TwoSquare(g1);  % g1^2 = R + r
18       d = (g2-g1)/2;          % power of 2, d maybe negative
19       Delta = Transforms([ tau1 ; tau2 ; p ; -R ; -r ; -2*g1*d ; -d^2 ]);
20       if Delta<0              % |x|| < mid(g1,g2)
21         res = min(g1,g2);
22       elseif Delta>0          % ||x|| > mid(g1,g2)
23         res = max(g1,g2);
24       else                    % Delta=0, |x|| = mid(g1,g2)
25         res = g1+d;
26       end
27     end
```

Remark 1 There are obvious ways to improve Algorithm `normNearest` by utilizing information obtained in the first transformation in line 3 in the following transformations in lines 5 and possibly 19, or by integrating the call in line 5

into that of line 3. Moreover, the transformation in Algorithm 3.3 in [29] with an extra parameter ϱ computing a faithful rounding of $\varrho + \sum_{i=1}^{n} p_i$ could be used. We refrain from doing that keep the code simple.

Remark 2 Algorithm `Transform` in line 3 transforms the input vector $[S; s]$ into p. The number of "while"-loops is proportional to the condition number of the sum, i.e., how close the true is sum to the midpoint of adjacent floating-point numbers. Algorithm `Transforms` in lines 5 and possibly 19 is applied to the already transformed vector p, so that in all our examples we did not encounter more than 2 loops.

Theorem 4 *Let $x \in \mathbb{F}^n$ be given and apply Algorithm* `normNearest` *to x, where Algorithm* `Transforms` *in lines* 5 *and* 19 *is identical to* `Transform` *in Figure 12 with replacing the constant Φ in line* 3 *by $\Phi = 2^M$. Suppose $n \le \frac{1}{4}\mathbf{u}^{-1/2} - 4$. Then the computed result* `res` *is equal to the Euclidean norm of x rounded to the nearest floating-point number, i.e.,* `res` $= \mathrm{fl}(\|x\|)$.

Proof Line 2 in Algorithm `normNearest` and Lemma 3 imply $\sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} P_i + \sum_{i=1}^{n} p_i$, so that Lemma 10 shows that

$$\sum_{i=1}^{n} x_i^2 = \tau_1 + \tau_2 + \sum_{i=1}^{n} p_i \tag{31}$$

and that f computed in line 4 is a faithful rounding of $\|x\|^2$. Thus $\mathrm{pred}(f) < \|x\|^2 < \mathrm{succ}(f)$. The vector argument of `Transforms` in line 5 is equal to

$$Q := \tau_1 + \tau_2 + \sum_{i=1}^{n} p_i - f = \sum_{i=1}^{n} x_i^2 - f.$$

Lemma 10 shows that the signs of Q and the computed δ coincide, It follows that $\|x\|^2 \in (\mathrm{pred}(f), f)$ if $\delta < 0$, $\|x\|^2 \in (f, \mathrm{succ}(f))$ if $\delta > 0$, and $\|x\|^2 = f$ if $\delta = 0$. Thus lines $6 - 11$ imply that $\|x\|^2$ is in the convex union of f and f_2. Denote the pair (f, f_2) by $(s_1, s_2) \in \mathbb{F}^2$ with $s_1 \le s_2$, such that

$$s_1 < \|x\|^2 < s_2 \quad \text{or} \quad s_1 = \|x\|^2 = s_2 \tag{32}$$

and $s_2 \le \mathrm{succ}(s_1) \le (1 + 2\mathbf{u})s_1$. Set $g_i := \mathrm{fl}(\sqrt{s_i})$ for $i \in \{1, 2\}$. Then (1), $\sqrt{1 + 2\mathbf{u}} < 1 + \mathbf{u}$, the monotonicity of the rounding $\mathrm{fl}(\cdot)$ and $\mathrm{fl}((1 + \mathbf{u})x) \le \mathrm{fl}((1 + \mathbf{u})^2 \mathrm{fl}(x)) \le \mathrm{succ}(\mathrm{fl}(x))$ for $x \in \mathbb{R}$ imply

$$g_2 = \mathrm{fl}(\sqrt{s_2}) \le \mathrm{fl}(\sqrt{(1 + 2\mathbf{u})s_1}) \le \mathrm{fl}((1 + \mathbf{u})\sqrt{s_1}) \le \mathrm{succ}(\mathrm{fl}(\sqrt{s_1})) = \mathrm{succ}(g_1).$$

Hence g_1 and g_2 are equal or adjacent floating-point numbers, and (32) yields

$$g_1 = \mathrm{fl}(\sqrt{s_1}) \le \mathrm{fl}(\|x\|) \le \mathrm{fl}(\sqrt{s_2}) = g_2.$$

In other words, the nearest rounding of $\|x\|$ is in $\{g_1, g_2\}$. Thus, if $g_1 = g_2$, the nearest rounding is equal to $g_1 = g_2$ which is handled in line 15.

Otherwise, line 17 implies $g_1^2 = R + r$. Then d, which is a power of 2 because it is half the distance between g_1 and g_2, is computed in line 18 without rounding error. Thus the product $2g_1d$ is computed without error as well, and the sum of the vector argument of `Transforms` in line 19 is equal to

$$S := \tau_1 + \tau_2 + \sum_{i=1}^{n} p_i - R - r - 2g_1d - d^2 = \sum_{i=1}^{n} x_i^2 - (g_1 + d)^2.$$

Note that the length of the vector argument is $2n+6$ and the assumption on n verifies that Lemma 10 is applicable and implies that $\mathrm{sign}(\texttt{Delta}) = \mathrm{sign}(S)$. Now $g_1 + d$ is the midpoint between the adjacent floating-point numbers g_1 and g_2, and the result follows by $\|x\| \in \{g_1, g_2\}$. □

We mention that the assumption $n \le \frac{1}{4}\mathbf{u}^{-1/2} - 4$ can be lifted to $n \le \frac{1}{32}\mathbf{u}^{-1} - 64$ using the ideas in Algorithm `AccSumHugeN` in [29], but we refrain from exploring this.

We showed that the f computed in line 4 is a faithful rounding of $\|x\|^2$. As has been noted in [6] that does not imply that $\mathrm{fl}(\sqrt{f})$ is a faithful rounding of $\|x\|$, but likely `AccSqrt(f,delta)` is.

6 Generation of ill-conditioned examples

A vector p is ill-conditioned with respect to the nearest rounding of $\|p\|$ if a very small change of the input data changes the result. The closer $\|p\|$ is to a switching point, the more difficult and ill-conditioned is the computation of the nearest rounding. For positive $f \in \mathbb{F}$ its successor is $\mathrm{succ}(f) = f + 2\mathbf{u} \cdot \mathrm{ufp}(f)$, so that the switching point is $\mu = f + \mathbf{u} \cdot \mathrm{ufp}(f) =: f + \delta$. Then $\varepsilon = \frac{\delta' - \delta}{\delta}$ is the relative distance of $\|p\| = f + \delta'$ to the switching point $f + \delta$.

For given ε it is, in principle, not too difficult to generate a vector p with $\|p\|$ having a relative distance ε to a switching point. To that end a multiple precision package may be helpful. However, when doing this we observed a severe influence on the timing. The mere presence of a call to the multiple precision package, of course, outside the loop to be measured, changed the measured computing time by a factor of 2 and more. Therefore, we wrote Algorithm `GenVec`, see Figure 15. Using it ensured reliable computing times.

The challenge is to approximate the anticipated final result $\|p\|$ near a switching point s "from below": During a loop the vector norm must always stay below s. That is the principle of Algorithm `GenVec`, a nice example of our algorithms with absolute splitting used for faithful and nearest rounding.

The rationale is as follows. The output vector p is computed in K segments each of length m. The initialization in line 4 ensures that the final vector length is n. The floating-point number f in line 6 or its successor $f + 1$ is the anticipated result of the nearest rounding of the final vector p to be generated with relative distance e to the switching point $f + 0.5$. The initial vector p as

Fig. 15 Vector $p \in \mathbb{F}^n$ with relative distance ε of $\|p\|$ to a switching point

```
 1   function [p,f] = GenVec(n,e)
 2     K = ceil(2-log2(abs(e))/53);        % number of segments
 3     m = floor(n/K);                     % size of each segment
 4     p = 2^26*rand(1,n-m*K)/sqrt(n);     % initial vector
 5     [p1,p2] = TwoSquare(p);             % ||p||^2 = p1 + p2
 6     f = pow2(52)*(1.1+0.8*rand);        % ufp(f) = 2^52
 7     [F1,F2] = TwoSquare(f);             % f^2 = F1 + F2
 8     [ef1,ef2] = TwoProduct(e,f);        % e*f = ef1 + ef2
 9     S = [F1 F2 f 1/4 ef1 ef2 -p1 -p2];  % sum(S) = (f+0.5)^2 + e*f - ||p||^2
10     [tau1,tau2,q] = Transform(S);
11     sumS = tau1 + ( tau2 + sum(q) );    % faithful rounding of sum(S)
12     phi = sqrt( 1 - (abs(e)/f^2)^(1/K) );  % decay factor
13     for k=1:K
14       ps = randn(1,m);
15       ps = phi*sqrt(sumS)*ps/norm(ps);  % next segment
16       p = [ p ps ];
17       [ps1,ps2] = TwoSquare(ps);        % ps^2 = ps1 + ps2
18       S = [ S -ps1 -ps2 ];              % sum(S) = (f+0.5)^2 + e*f - ||p||^2
19       [tau1,tau2,q] = Transform(S);
20       sumS = tau1 + ( tau2 + sum(q) );  % faithful rounding of sum(S)
21     end
22     p = p(randperm(length(p)));         % random perturbation
23     if e>0
24       f = f+1;                          % f is nearest rounding of ||p||
25     end
```

in line 4 satisfies $\|p\|^2 = p_1 + p_2$ and $f - \|p\| > 0$. Lines $7-8$ yield $f^2 = F_1 + F_2$ and $e \cdot f = ef_1 + ef_2$, so that

$$\sum S_i = f^2 + f + \frac{1}{4} + e \cdot f - \|p\|^2 = (f + \frac{1}{2})^2 + e \cdot f - \|p\|^2 =: T$$

for the S in line 9. Here $\sum S_i$ denotes the mathematical sum of all elements of S. Furthermore, lines $10 - 11$ and Lemma 10 imply that `sumS` is a faithful rounding of $\sum S_i$. The φ in line 15 satisfies $\varphi \leq 1 - 4\mathbf{u}$ reasonable values of n and e, so that ps in line 14 satisfies

$$ps = \mathrm{float}(\varphi\sqrt{\texttt{sumS}}) \leq (1 + \frac{\mathbf{u}}{1+\mathbf{u}})^2(1 - 4\mathbf{u})\sqrt{\texttt{sumS}} < (1 - 2\mathbf{u})\sqrt{\texttt{sumS}}\ . \quad (33)$$

In the for-loop the element ps is appended to the vector p and $-ps^2 = -ps_1 - ps_2$ to the vector S, so that the sum $T = \sum S_i$ changes into $T - ps^2$. Since `sumS` is a faithful rounding of $\sum S_i$, (33) implies that $T - ps^2 > 0$.

At the end of every loop, `sumS` is always a faithful rounding of the sum $\sum S_i$ by lines $19 - 20$, and the construction implies that `sumS` decreases into $(1 - \varphi^2)$`sumS` in each step. The starting value of `sumS` is about f^2, and φ and K are chosen such that `sumS` $\leq |e|$ after finishing the loop.

After finishing the for-loop, `sumS` is a faithful rounding of $(f + \frac{1}{2})^2 + e \cdot f - \sum p_i^2$. Since $f \geq 2^{52}$ we conclude that $\|p\|^2$ is very close to $(f + \frac{1}{2})^2 + e \cdot f$, hence

$$\|p\| \approx \sqrt{(f + \frac{1}{2})^2 + e \cdot f} \approx f + \frac{1}{2} + \frac{e}{2}. \quad (34)$$

In the above setting $\delta = \frac{1}{2}$ and $\delta' = \frac{1+e}{2}$, so that the relative distance of $\|p\|$ is $\frac{\delta'-\delta}{\delta} = e$. The "approximations" in (34) are very accurate.

Finally, if $e < 0$, then $\|p\|$ is left of the switching point $f + \frac{1}{2}$ and f is the nearest rounding, otherwise, as computed in line 24, the nearest rounding is $f + 1$. The random perturbation in line 22 may be useful for testing the generality of algorithms.

It is clear from the code that the elements of one segment are close together, and the segments decay with the factor φ. If the number of segments K is increased, then a better distrubution of the vector elements of p is obtained, however, at the cost of increasing computing time.

7 Computational results

The following computational results are all performed using MATLAB Version 2020b on some core i7 Laptop. In all of the following examples the number of test cases is generally 1 million, but for large dimensions chosen such that the computing time stays below 1 hour.

We start with some timing comparisons of variants of MATLAB implementations. For example, an alternative to Algorithm `Split` in Figure 2 is the following.

Fig. 16 Alternative splitting

```
1    function [P,p] = Split1(Aa)
2      [f,e] = log2(Aa);
3      P = pow2(fix(2^27*f),e-27);
4      p = Aa - P;
```

The following Table 1 shows the computing time of Algorithm `Split1` divided by that for Algorithm `Split1`. We also compare `sqr(a)` vs. `a.*a`, `TwoProduct` vs. `TwoSquare` as in Figure 3 and `VecSum` vs. `FastVecSum` as in Figure 7.

Table 1 Time comparisons for different vector lengths.

Comparison	10	100	1000	10^4	10^5	10^6
`Split1 / Split`	1.24	1.25	2.38	1.79	1.67	1.65
`sqr(a) / a.*a`	0.97	0.94	0.98	0.99	1.01	0.95
`TwoProduct / TwoSquare`	1.46	1.50	1.52	1.52	1.66	1.69
`VecSum / FastVecSum`	0.37	2.41	9.19	9.91	8.89	2.91

The original Algorithm `Split` is significantly faster than the simulation by `log2` and `round`, so we use Algorithm `Split`. Similarly, Algorithm `TwoSquare`

in Figure 3 is some 50% faster than Algorithm `TwoProduct`, and the loop-free variant Algorithm `FastVecSum` in Figure 7 is much faster than Algorithm `VecSum` in [25]. We use `a.*a` because the time seems the same as for `sqr(a)`.

Before we come to timing comparisons, we give information about the accuracy of our algorithms and competitors. We start with possibilities to approximate $\|x\|$ by the built-in MATLAB routines, where the obvious candidate is `norm(x)`. We generate random testcases and display triples of numbers: The first and second is the percentage of nearest and faithful roundings, respectively, and the third the percentage where the result is not faithful.

Table 2 Percentage of rounding is nearest/faithful/none.

	10^3	10^4	10^5	10^6
for-loop	14.6/26.7/58.7	4.0/7.6/88.5	1.7/3.6/94.7	0.4/0.8/98.8
`sqrt(sum(x.*x))`	44.4/46.1/9.5	15.5/27.7/56.8	74.2/25.8/0.0	60.2/38.9/0.9
`norm(x)`	100/0/0	76.6/15.8/7.6	88.2/10.3/1.5	88.1/9.4/2.4

As can be seen in the third row of Table 2, the built-in function `norm(x)` is surprisingly accurate, more accurate than theory predicts [7–9].

We therefore perform tests on the same data using `sqrt(sum(x.*x))` and an ordinary for-loop. Still `sqrt(sum(x.*x))` is more accurate than expected, only the for-loop shows the anticipated behavior.

Details on the actual implementation of `norm` and `sum` are confidential, but the data in Table 2 suggests that some higher precision or compensating algorithms are used.

The essential difference between Algorithms `normG` [6] and `normL` [20] is the use of `AccSqrt` of [17] for the square root approximation in the last line of `normL`. To see the advantage, we use Algorithm `normGacc` which is identical to Algorithm `normG` except that

the last line `res` = `sqrt`(`S`); is changed into `res` = `AccSqrt`(`S`, `s`);

The following Table 3 shows the percentage of nearest rounding random test cases with different dimensions.

Table 3 Percentage of nearest rounding of `normG` vs. `normGacc`

Algorithm	10	100	1000	10^4	10^5	10^6	10^7
`normG`	86.3	91.6	87.1	83.8	90.1	89.0	98.1
`normGacc`	100	100	100	100	100	100	83.2

There was no case with not faithful rounding, as proved in [6], and for the improved Algorithm `normGacc` we found only for $n = 10^7$ cases where the rounding was not to nearest, in fact, some 17%.

Up to now we used random vectors produced by `randn(n,1)` for which it is not too difficult to calculate a nearest rounding of $\|x\|$. That changes when the true result is close to the midpoint between two adjacent floating-point numbers, i.e., close to a "switching point" [2].

To that end we use Algorithm `GenVec` to generate vectors x of different dimensions with relative distance ε of $\|x\|$ to a switching point. For each pair of dimension n and relative distance ε, we display the percentage of nearest roundings in Table 4.

In all test cases and for all algorithms we did not encounter an example with not faithful rounding. As already seen in Table 3, generally Algorithm `normGacc` outperforms the original `normG` in terms of accuracy. Algorithm `normG` is targeted to a faithfully rounded result S, not minimizing the error of $S+s$ versus $\|x\|^2$. Thus about half the results of Algorithm `normG` are nearest, the other half faithful but not nearest.

Algorithm `normDD` uses a general purpose double-double arithmetic, and Algorithm `normCpair` our pair arithmetic with computable error bounds. As Algorithms `normGacc` and `normL` are tailored methods but use the same principle, we expect similarly accurate results. Indeed, that can be seen in Table 4 for all test examples including very small distance to a switching point. For a relative distance ε downto about 10^{-14} the rounding is nearest. Similarly, as Algorithms `normSum2` and `normSum3` are based on similar principles, they show the same accuracy, with `normSum3` being a little bit better.

Table 4 Percentage of nearest rounding for relative distance ε of $\|x\|$ to switching point.

n	ε	DD	Cpair	G	Gacc	L	Sum2	Sum3	Extract	Extract2	Nearest
100	10^{-2}	100	100	51.6	100	100	100	100	100	100	100
	10^{-6}	100	100	50.4	100	100	100	100	100	100	100
	10^{-10}	100	100	49.4	100	100	100	100	49.9	100	100
	10^{-14}	100	100	50.4	100	100	100	100	50.9	100	100
	10^{-16}	74.2	75.9	50.2	74.1	73.8	74.1	74.4	50.0	77.0	100
	10^{-18}	41.2	42.2	50.1	43.0	41.3	40.6	44.0	50.5	54.2	100
	10^{-20}	40.8	41.6	49.9	42.6	41.0	40.4	43.4	50.0	53.9	100
10^4	10^{-2}	100	100	51.9	100	100	100	100	100	100	100
	10^{-6}	100	100	49.2	100	100	100	100	99.7	100	100
	10^{-10}	100	100	49.2	100	100	100	100	55.8	100	100
	10^{-14}	100	100	47.1	100	100	100	100	53.7	100	100
	10^{-16}	98.7	97.9	51.2	98.7	98.5	98.1	100	53.8	64.6	100
	10^{-18}	42.0	43.5	49.4	44.4	43.0	42.4	42.8	53.6	48.5	100
	10^{-20}	40.3	41.9	49.6	41.9	41.4	40.8	44.5	49.3	51.7	100
10^6	10^{-2}	100	100	56.0	100	100	100	100	100	100	100
	10^{-6}	100	100	49.0	100	100	100	100	100	100	100
	10^{-10}	100	100	43.0	100	100	100	100	75.0	100	100
	10^{-14}	100	100	54.0	100	100	100	100	77.0	100	100
	10^{-16}	98.2	98.4	53.4	98.4	99.2	99.2	100	26.3	78.8	100
	10^{-18}	40.4	38.2	48.6	42.5	41.3	40.7	42.2	27.1	49.7	100
	10^{-20}	41.5	44.7	56.1	43.9	40.7	38.2	43.1	27.6	59.3	100

Algorithms `normExtract` and `normExtract2` are based on a different principle, namely on an absolute splitting. As mentioned this avoids the costly

[2] called rounding boundary in [4]

application of Algorithm `AccSquare`. For moderate distance ε the rounding is nearest, including large vector lengths, for distance 10^{-10} and below the accuracy is similar to `normG` with roughly a 50-50 chance of nearest result. As we will see next, the little less number of nearest roundings is compensated by a much better performance.

The number of nearest cases improves a little bit with `normExtract2`, and the result of Algorithm `normNearest` is, of course, always rounded to nearest.

Next we present timing results for our algorithms and competitors for random vectors and for dimension up to $n = 10^7$. It is appropriate to use random vectors because the computing time of all algorithms except `normNearest` do not depend on the difficulty of the problem, only on the length of the input vector; times for `normNearest` for different ε are displayed separately.

It turns out that our new Algorithm `normExtract` is always the fastest. Therefore the following Table 5 shows the time ratio against `normExtract`. The timing for Algorithms `normDD` and `normCpair` is dominated by MATLAB's interpretation overhead and in particular by the use of operator overloading. Therefore comparing the computing times hardly gives information on the performance of the algorithms and is omitted.

Table 5 Timing relative to `normExtract` for random vectors of length n.

n	`normG`	`normL`	`normSum2`	`normSum3`	`normExtract2`	`normNearest`
10^2	14.7	7.2	3.2	5.7	2.0	6.8
10^3	46.7	20.3	2.6	5.1	2.0	6.0
10^4	27.2	12.2	1.3	2.9	2.1	4.6
10^5	63.4	26.4	2.0	16.7	2.1	17.4
10^6	14.0	6.5	2.3	5.5	2.0	6.1
10^7	14.0	6.5	2.4	5.5	2.0	6.8

From the operation count it may surprise that `normExtract` is so much faster than `normG` and `normL`. Now `normExtract` is based on `AccSum` in [28], and it was analyzed by Langlois [19] that it enjoys a better instruction-level parallelism than other algorithms. The same applies to `normSum2` and may explain its relatively good performance, and also to `normSum3` where we see twice the computing time of `normSum2`, as expected. That is still faster than `normG` and `normL`. There is an exception to all algorithms, namely $n = 10^5$. We think this is due to unfortunate cache management, similarly for `normNearest`.

Algorithm `Nearest` is about as fast as `normL`, for medium size dimension much faster, although it guarantees a nearest rounding of $\|x\|$.

The time in seconds for 5000 calls in dimension up to 1 million of all algorithms is shown in Figure 17. The legend on the left is ordered by performance, from the slowest `normG` downto the fastest `normExtract`. All algorithms except Algorithm `normNearest` execute the same code independent of the difficulty of the problem, hence the computing time depends almost linearly on the di-

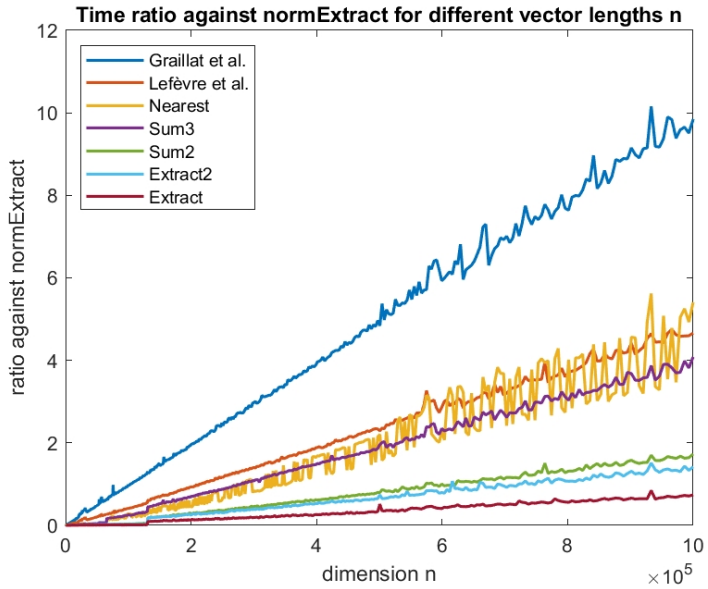

Fig. 17 Timing for 5000 calls for random vectors of dimension up to 1 million

mension. For `normNearest` we see small zig-zags depending on the number of transformations.

Finally we investigate whether the guarantee of nearest rounding causes a time penalty for Algorithm `normNearest` if $\|x\|$ is very close to a switching point. As before we generate examples with relative distance ε to a switching

Table 6 Timing of `normNearest`/`normExtract`, relative distance ε of $\|x\|$ to switching point.

n \ ε	10^{-2}	10^{-10}	10^{-14}	10^{-16}	10^{-18}	10^{-30}	10^{-50}	10^{-100}
10^2	7.9	8.1	8.2	8.4	8.4	8.8	9.3	10.5
10^4	5.8	6.1	6.2	5.9	6.0	6.4	6.7	8.5
10^6	7.3	7.7	7.9	7.5	7.7	8.5	9.2	10.5

point. The ratio of computing time of Algorithm `normNearest` to `normExtract` are displayed in Table 6; the time for the other algorithms does not change because they are independent of the condition of the problem.

There is not much performance impact on the computing time of Algorithm `normNearest` in our examples despite the guarantee of nearest rounding of $\|x\|$, even for a tiny relative distance $\varepsilon = 10^{-100}$ to a switching point.

8 Summary

We may use a general purpose pair arithmetic such as double-double [2] or [17] to calculate an accurate approximation of the Euclidean norm $\|x\|$ of a vector. To that end we presented Algorithms `normDD` and `normCpair` in Figure 6. Specialized algorithms based on a pair arithmetic have been presented in [6,20] and are displayed as Algorithms `normG` and `normL` in Figure 5.

In this note we developed Algorithms `Sum2` and `Sum3` in Section 3 based on relative splitting as algorithm in [25]. The performance is significantly improved by a vectorized version `FastVecSum` of `VecSum` in [25]. In addition, Algorithms `Extract` and `Extract2` based absolute splittings as in [28,29] are presented in Section 4.

All algorithms mentioned so far compute a faithfully rounded result of $\|x\|$, in many cases the nearest result. A first algorithm to provably compute the rounded to nearest result is presented as Algorithm `normNearest`.

The computing times of our new algorithms compare favorably to the competitors, where Algorithm `normExtract` is significantly faster than all others. Algorithm `normNearest` is also fast despite the guaranteed nearest rounding. That includes difficult cases where the true Euclidean norm $\|x\|$ has a relative distance as small as $\varepsilon = 10^{-100}$ to a switching point.

Acknowledgements The author wishes to thank an anonymous referee for thorough reading and many valuable comments.

References

1. Edward J. Anderson. Algorithm 978: Safe scaling in the level 1 BLAS. *ACM Trans. Math. Softw.*, 44(1):12:1–12:28, 2017.
2. D.H. Bailey. A Fortran-90 double-double precision library. `http://crd.lbl.gov/~dhbailey/mpdist/`.
3. James L. Blue. A portable Fortran program to find the Euclidean norm of a vector. *ACM Transactions on Mathematical Software*, 4(1):15–23, March 1978.
4. R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, New York, NY, USA, 2010.
5. T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971.
6. S. Graillat, C. Lauter, P. Tang, N. Yamanaka, and S. Oishi. Efficient calculations of faithfully rounded l_2-norms of n-vectors. *ACM TOMS*, 41(4):24:1–20, 2015.
7. N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM Publications, Philadelphia, 2nd edition, 2002.
8. N.J. Higham and T. Mary. A New Approach to Probabilistic Rounding Error Analysis. *SIAM Journal on Scientific Computing*, 41(5):A2815–A2835, 2019.
9. N.J. Higham and T. Mary. Sharper probabilistic backward error analysis for basic linear algebra kernels with random data. *SIAM J. Sci. Comput.*, 42(5):A3427–A3446, 2020.
10. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.
11. C.-P. Jeannerod and S.M. Rump. On relative errors of floating-point operations: optimal bounds and applications. *Math. Comp.*, 87:803–819, 2017.
12. 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, ARITH 2018, Amherst, MA, USA, June 25-27, 2018*, pages 53–60. IEEE, 2018.

13. W. Kahan. Further Remarks on Reducing Truncation Errors. *Communications of the ACM*, 8(1):40, 1965.
14. D.E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, 1969.
15. M. Lange and S.M. Rump. Error estimates for the summation of real numbers with application to floating-point summation. *BIT*, 57:927–941, 2017.
16. M. Lange and S.M. Rump. Sharp estimates for perturbation errors in summations. *Math.Comp.*, 88:349–368, 2019.
17. M. Lange and S.M. Rump. Faithfully rounded floating-point operations. *ACM Trans. Math. Softw.*, 46, 2020.
18. Marko Lange and Shin'ichi Oishi. A note on Dekker's FastTwoSum algorithm. *Numerische Mathematik*, 145(2):383–403, 2020.
19. P. Langlois. Accurate algorithms in floating-point arithmetic. In *Lecture at the 12th GAMM-IMACS International Symposion on Scientific Computing (SCAN), Computer Arithmetic and Validated Numerics*, Duisburg, 2006. IEEE.
20. V. Lefèvre, N. Louvet, J.-M. Muller, J. Picot, and L. Rideau. Accurate calculation of Euclidean Norms using Double-word arithmetic. working paper or preprint, December 2021, `https://hal.archives-ouvertes.fr/hal-03482567/`.
21. MATLAB. User's Guide, Version 2022b, the MathWorks Inc., 2012-2022.
22. O. Møller. Quasi double precision in floating-point arithmetic. *BIT Numerical Mathematics*, 5:37–50, 1965.
23. J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, R. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2nd edition, 2018.
24. A. Neumaier. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen. *Zeitschrift für Angew. Math. Mech. (ZAMM)*, 54:39–51, 1974.
25. T. Ogita, S.M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing (SISC)*, 26(6):1955–1988, 2005.
26. T. G. Robertazzi and S. C. Schwartz. Best "ordering" for floating-point addition. *ACM Transactions on Mathematical Software*, 14(1):101–110, March 1988.
27. S.M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. `http://www.ti3.tuhh.de/intlab`.
28. S.M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput. (SISC)*, 31(1):189–224, 2008.
29. S.M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest. *Siam J. Sci. Comput. (SISC)*, 31(2):1269–1302, 2008.
30. S.M. Rump, P. Zimmermann, S. Boldo, and G. Melquiond. Computing predecessor and successor in rounding to nearest. *BIT*, 49, 06 2009.
31. P.H. Sterbenz. *Floating-point computation*. Prentice Hall, Englewood Cliffs, NJ, 1974.
32. G. Zielke and V. Drygalla. Genaue Lösung linearer Gleichungssysteme. *GAMM Mitt. Ges. Angew. Math. Mech.*, 26:7–108, 2003.