Technical Report 05.12, Faculty for Information- and Communication Sciences, Hamburg University of Technology, November 13, 2005

ACCURATE FLOATING-POINT SUMMATION *

SIEGFRIED M. RUMP †, TAKESHI OGITA ‡, AND SHIN'ICHI OISHI §

Abstract. Given a vector of floating-point numbers with exact sum s, we present an algorithm for calculating a faithful rounding of s into the set of floating-point numbers, i.e. one of the immediate floating-point neighbors of s. If the s is a floating-point number, we prove that this is the result of our algorithm. The algorithm adapts to the condition number of the sum, i.e. it is very fast for mildly conditioned sums with slowly increasing computing time proportional to the condition number. All statements are also true in the presence of underflow. Furthermore algorithms with K-fold accuracy are derived, where in that case the result is stored in a vector of K floating-point numbers. We also present an algorithm for rounding the sum s to the nearest floating-point number. Our algorithms are fast in terms of measured computing time because they neither require special operations such as access to mantissa or exponent, they contain no branch in the inner loop, nor do they require extra precision: The only operations used are standard floating-point addition, subtraction and multiplication in one working precision, for example double precision. Moreover, in contrast to other approaches, the algorithms are ideally suited for parallelization. We also sketch dot product algorithms with similar properties.

Key words. maximally accurate summation, faithful rounding, maximally accurate dot product, error-free transformations, fast algorithms, parallel algorithms, high precision

AMS subject classifications. 15-04, 65G99, 65-04

1. Introduction.

We will present fast algorithms to compute approximations of high quality of the sum and the dot product of vectors of floating-point numbers. In the recent paper [36] we developed such algorithms delivering a result *as if* computed in a specified *precision*. Those algorithms use only one working precision, and we could generate a result *as if* computed in K-fold precision, $K \geq 1$. The *accuracy* of the computed result, however, still depends on the condition number of the problem.

In the present paper we go one step further and present algorithms producing a result of specified *accuracy*. Again using only floating-point operations in one working precision we will show that we can compute a floating-point result `res` such that there is no other floating-point number between the true, real result and `res`.

This has been called *faithful rounding* in the literature [10, 39, 9]. We will also prove that if the true result is an exactly representable floating-number, then this will be our computed result. This implies that the sign of the result is always determined correctly, a significant problem in the computation of geometrical predicates [19, 43, 7, 27, 6, 12], where the sign of the value of a dot product decides whether a point is exactly on a plane or on which side it is.

As we will show, the computational effort of our methods is proportional to the condition number of the problem. An almost ideal situation: for simple problems the algorithm is fast, and slows down with increasing difficulty.

*This research was partially supported by Grant-in-Aid for Specially Promoted Research (No. 17002012: Establishment of Verified Numerical Computation) from the Ministry of Education, Science, Sports and Culture of Japan.

†Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße 95, Hamburg 21071, Germany, and Visiting Professor at Waseda University, Faculty of Science and Engineering, 3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan (`rump@tu-harburg.de`).

‡CREST, Japan Science and Technology Agency (JST), and Faculty of Science and Engineering, Waseda University, 3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan (`ogita@waseda.jp`).

§Department of Computer Science, Faculty of Science and Engineering, Waseda University, 3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan (`oishi@waseda.jp`).

In addition to the faithfully rounded result in working precision, we also present algorithms for a rounded to nearest result. As is known, here the computational effort cannot be proportional to the traditional condition number of the problem, but depends on the nearness of the true result to a switching point for rounding.

Moreover, algorithms for producing a faithfully rounded result in K-fold accuracy are presented. In this case the result is represented as a vector of K floating-point numbers.

Our algorithms are fast. We interpret fast not only by the number of floating-point operations, but in terms of *measured computing time*. This means that special operations such as rounding to integer, access to mantissa or exponent, branches etc. are avoided. Our algorithms use only floating-point addition, subtraction and multiplication in working precision. No extra precision is required.

Summation and dot product are most basic tasks in numerical analysis, and there are numerous algorithms for that, among them [4, 8, 11, 12, 17, 21, 22, 23, 24, 25, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 44, 45]. Higham [18] devotes an entire chapter to summation. Accurate summation or dot product algorithms have various applications in many different areas of numerical analysis. Excellent overviews can be found in [18, 31].

Many known algorithms produce a 'more accurate' result, demonstrated by examples or rigorous estimations. There are few methods producing a result as if computed in higher precision and very few methods computing a result with specified accuracy [4, 39, 11, 28]. In [28], for example, the fact is used that IEEE 754 double precision numbers can be added exactly in some superlong accumulator due to their limited exponent range. With some bit manipulation the rounded to nearest result of the sum (or dot product) can be obtained.

For more details see [36]. However, known methods require sorting of the input vector, access to mantissa/exponent, branches, some extended precision and/or other special operations. As will be shown in the computational results, this may slow down a computation substantially.

An important tool in our work are so-called error-free transformations. Here a floating-point approximation is supplemented by an exact error term. Consider the following algorithm for the product of two floating-point numbers.

ALGORITHM 1.1. *Error-free transformation of the product of two floating-point numbers.*

$$
\begin{array}{l}
\text{function } [x,y] = \texttt{TwoProduct}(a,b) \\
\quad x = \mathrm{fl}(a \cdot b) \\
\quad [a_1,a_2] = \texttt{Split}(a) \\
\quad [b_1,b_2] = \texttt{Split}(b) \\
\quad y = \mathrm{fl}(a_2 \cdot b_2 - (((x - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2))
\end{array}
$$

Here $\mathrm{fl}(\cdot)$ denotes that the expression inside the parenthesis is calculated in floating-point. If no underflow occurs, this algorithm due to G.W. Veltkamp (see [10]) satisfies for all floating-point numbers a, b

$$
x = \mathrm{fl}(a+b) \quad \text{and} \quad x + y = a \cdot b. \tag{1.1}
$$

It relies on the ingenious splitting by Dekker [10] of a 53-bit floating-point number into two 26-bit parts. This is possible using the sign bit and allows exact multiplication of the parts.

ALGORITHM 1.2. *Error-free splitting of a floating-point number into two parts.*

$$
\begin{array}{ll}
\text{function } [x,y] = \texttt{Split}(a) & \\
\quad c = \mathrm{fl}(\texttt{factor} \cdot a) & \texttt{\% factor} = 2^{27}+1 \\
\quad x = \mathrm{fl}(c - (c - a)) & \\
\quad y = \mathrm{fl}(a - x) &
\end{array}
$$

Such transformations were extensively used in our algorithms for summation and dot product [36] giving a result as if computed in a specified precision. Recently there is increasing interest in using such error-free transformations [14, 29].

By Algorithm 1.1 (`TwoProduct`) we can transform the dot product of two vectors of floating-point numbers into a sum of floating-point numbers, thus we concentrate on summation.

In the following we need many careful floating-point estimations, frequently heavily relying on bit representations and the definition of the floating-point arithmetic in use. Not only that this is frequently quite tedious, it is also sometimes presented in a colloquial manner and not easy to follow. To avoid this and also to ensure rigor, we found it convenient and more stringent to use inequalities. For this we derived some machinery to characterize floating-point numbers, their bit representations and to handle delicate situations.

The paper is organized as follows. First we introduce our notation in Section 2 and list a number of properties. Some of them may be trivial, however, we found them useful to note to get acquainted with the machinery. In this section we also define faithful rounding and give a sufficient criterion for it. In Section 3 we use this to develop an error-free transformation of a vector of floating-point numbers into an approximation of the sum and some remaining part. The magnitude of the remaining part can be estimated, so that we can introduce summation algorithms with faithful rounding in the following Section 4. We prove faithfulness which especially includes the exact determination of the sign. This is not only also true in the presence of underflow, but the computed result is exact if it is in the underflow range. We also estimate the computing time depending on the condition number. In the next Section 5 we extend the results to K-fold faithful rounding, where the result is represented as a vector of K floating-point numbers. In Section 6 we derive, based on the previous ones, a rounding to nearest algorithm. Up to now the vector lengths were restricted to about $\sqrt{\mathtt{eps}^{-1}}$; in the following Section 7 we extend the previous results to huge vector lengths up to almost $\mathtt{eps}^{-1}$. A simplified and efficient version only for sign determination is given as well. Next we sketch algorithms for dot products with similar properties, and finally we give computational results in Section 9. In the Appendix we give some proof and executable Matlab code, and a Summary concludes the paper.

As in [36], all theorems, error analysis and proofs are due to the first author.

2. Basic facts.

In this section we collect some basic facts for the analysis of our algorithms. Throughout the paper we assume that no overflow occurs, but we allow underflow. We will use only one working precision for all floating-point computations; as an example we sometimes refer to IEEE 754 double precision. This corresponds to 53 bits precision including an implicit bit for normalized numbers. However, we stress that the following analysis applies *mutatis mutandis* to a binary arithmetic with another format or exponent range by replacing the roundoff and underflow unit, for example to IEEE 754 single precision. Since we use floating-point numbers in only one working precision, we can refer to them as “the floating-point numbers”.

The set of all floating-point numbers is denoted by $\mathbb{F}$, and $\mathbb{U}$ denotes the set of unnormalized floating-point numbers together with zero and the two normalized floating-point numbers of smallest nonzero magnitude. The relative rounding error unit, the distance from 1.0 to the next smaller[1] floating-point number, is denoted by `eps`, and the underflow unit by `eta`, that is the smallest positive (unnormalized) floating-point number. For IEEE 754 double precision we have $\mathtt{eps} = 2^{-53}$ and $\mathtt{eta} = 2^{-1074}$. Then $\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ is the smallest positive normalized floating-point number and for $f \in \mathbb{F}$ we have

$$f \in \mathbb{U} \;\Leftrightarrow\; 0 \le 2\mathtt{eps}|f| \le \mathtt{eta}\ . \tag{2.1}$$

[1]Note that sometimes the distance from 1.0 to the next *larger* floating-point number is used; for example, Matlab adopts this rule.

We denote by $\mathrm{fl}(\cdot)$ the result of a floating-point computation, where all operations within the parentheses are executed in working precision. If the order of execution is ambiguous and is crucial, we make it unique by using parentheses. We assume the floating-point operations to satisfy the properties of IEEE 754 arithmetic standard in rounding to nearest [20]. Then floating-point addition and subtraction satisfy [18]

$$\mathrm{fl}(a \circ b) = (a \circ b)(1+\varepsilon) \quad \text{for } a,b \in \mathbb{F},\ \circ \in \{+,-\} \ \text{ and } \ |\varepsilon| \le \mathtt{eps}. \tag{2.2}$$

Note that addition and subtraction is exact in case of underflow [15], so we need no underflow unit in (2.2). More precisely,

$$a,b \in \mathbb{F} \quad \text{and} \quad |a+b| \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta} \qquad \text{implies} \quad \mathrm{fl}(a+b) = a+b\ . \tag{2.3}$$

We have to distinguish between normalized and unnormalized floating-point numbers since the latter lack the implicit bit. As has been noted by several authors [34, 26, 10], the error of a floating-point addition is always a floating-point number:

$$a,b \in \mathbb{F} \quad \text{implies} \quad \delta := \mathrm{fl}(a+b) - (a+b) \in \mathbb{F}\ . \tag{2.4}$$

Fortunately, the error term δ can be computed using only standard floating-point operations. The following algorithm by Knuth was already given in 1969 [26].

ALGORITHM 2.1. *Error-free transformation for the sum of two floating-point numbers.*

```
function [x, y] = TwoSum(a, b)
    x = fl(a + b)
    z = fl(x − a)
    y = fl((a − (x − z)) + (b − z))
```

Knuth's algorithm transforms any pair of floating-point numbers (a, b) into a new pair (x, y) with

$$x = \mathrm{fl}(a+b) \quad \text{and} \quad x + y = a + b. \tag{2.5}$$

This is also true in the presence of underflow. An error-free transformation for subtraction follows since $\mathbb{F} = -\mathbb{F}$. The $\mathrm{fl}(\cdot)$ notation applies not only to operations but to real numbers as well. For $r \in \mathbb{R}$, $\mathrm{fl}(r) \in \mathbb{F}$ is r rounded to the nearest floating-point number. Following the IEEE 754 arithmetic standard tie is rounded to even. For $f_1, f_2 \in \mathbb{F}$ and $r \in \mathbb{R}$, monotonicity of the rounding implies

$$f_1 \le r \le f_2 \quad \Rightarrow \quad f_1 \le \mathrm{fl}(r) \le f_2 \tag{2.6}$$

$$f_1 < \mathrm{fl}(r) < f_2 \quad \Rightarrow \quad f_1 < r < f_2\ . \tag{2.7}$$

In numerical analysis the accuracy of a result is sometimes measured by the "unit in the last place (ulp)". For the following, sometimes delicate error estimations the ulp-concept has the drawback that it depends on the floating-point format and needs extra care in the underflow range.

We found it useful to introduce the "unit in the first place" (ufp) or leading bit of a real number by

$$0 \ne r \in \mathbb{R} \quad \Rightarrow \quad \mathrm{ufp}(r) := 2^{\lfloor \log_2 |r| \rfloor}\ , \tag{2.8}$$

where we set $\mathrm{ufp}(0) := 0$. This gives a convenient way to characterize the bits of a normalized floating-point number f: they range between the leading bit $\mathrm{ufp}(f)$ and the unit in the last place $2\mathtt{eps} \cdot \mathrm{ufp}(f)$. In our analysis we will view a floating-number frequently as a scaled integer. For $\sigma = 2^k, k \in \mathbb{Z}$, we will use the set $\mathtt{eps}\sigma\mathbb{Z}$, which can be viewed as a set of fixed point numbers with smallest positive number $\mathtt{eps}\sigma$. The situation is depicted in Figure 2.1. Of course, $\mathbb{F} \subseteq \mathtt{eta}\mathbb{Z}$. Note that (2.8) is independent of some floating-

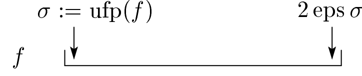

FIG. 2.1. *Normalized floating-point number: unit in the first place and unit in the last place*

point format and it applies to real numbers as well: $\mathrm{ufp}(r)$ is the value of the first nonzero bit in the binary representation of r. It follows

$$0 \neq r \in \mathbb{R} \quad \Rightarrow \quad \mathrm{ufp}(r) \leq |r| < 2\mathrm{ufp}(r) \tag{2.9}$$

$$r, r' \in \mathbb{R} \text{ and } \mathrm{ufp}(r) \leq |r'| \quad \Rightarrow \quad \mathrm{ufp}(r) \leq \mathrm{ufp}(r') \ . \tag{2.10}$$

We collect some properties. For $\sigma = 2^k, k \in \mathbb{Z}, r \in \mathbb{R}$ it follows

$$\sigma' = 2^m,\ m \in \mathbb{Z} \text{ and } \sigma' \geq \sigma \quad \Rightarrow \quad \mathtt{eps}\sigma'\mathbb{Z} \subseteq \mathtt{eps}\sigma\mathbb{Z} \tag{2.11}$$

$$f \in \mathbb{F} \quad \text{and} \quad |f| \geq \sigma \quad \Rightarrow \quad \mathrm{ufp}(f) \geq \sigma \tag{2.12}$$

$$f \in \mathbb{F} \quad \Rightarrow \quad f \in 2\mathtt{eps} \cdot \mathrm{ufp}(f)\mathbb{Z} \tag{2.13}$$

$$r \in \mathtt{eps}\sigma\mathbb{Z},\ |r| \leq \sigma \quad \text{and} \quad \mathtt{eps}\sigma \geq \mathtt{eta} \quad \Rightarrow \quad r \in \mathbb{F} \tag{2.14}$$

$$a, b \in \mathbb{F} \cap \mathtt{eps}\sigma\mathbb{Z} \quad \text{and} \quad \delta := \mathrm{fl}(a+b) - (a+b) \quad \Rightarrow \quad \mathrm{fl}(a+b),\ a+b,\ \delta \in \mathtt{eps}\sigma\mathbb{Z} \tag{2.15}$$

$$a_i \in \mathbb{F} \quad \text{and} \quad |a_i| \leq \sigma \quad \Rightarrow \quad |\mathrm{fl}(\sum_{i=1}^{n} a_i)| \leq n\sigma \tag{2.16}$$

$$a, b \in \mathbb{F} \quad \Rightarrow \quad \begin{array}{l} \mathrm{fl}(a+b) \in \mathtt{eps} \cdot \mathrm{ufp}(a)\mathbb{Z} \quad \text{and} \\ \mathrm{fl}(a+b) \in \mathtt{eps} \cdot \mathrm{ufp}(b)\mathbb{Z} \ . \end{array} \tag{2.17}$$

The assertions (2.11) to (2.15) are clear. Note that (2.13) is also true for $f \in \mathbb{U}$. Estimation (2.16) is sometimes useful to avoid unnecessary quadratic terms and it is valid for any order of summation. It follows by induction: For $s := \mathrm{fl}(\sum_{i \neq k} a_i)$ we have $|s + a_k| \leq n\sigma$, so by (2.6) and if $n\sigma \in \mathbb{F}$ also $|\mathrm{fl}(s + a_k)| \leq n\sigma$, and if $n\sigma$ is in the overflow range but $\mathrm{fl}(\sum a_i)$ is not, the estimation is true as well.

The last one (2.17) is also clear after a little thinking, and a rigorous proof follows easily with our machinery. The assertion is clear for $ab \geq 0$ by using (2.13) and (2.11) because then $|\mathrm{fl}(a+b)| \geq \max(|a|, |b|)$; so without loss of generality it suffices to show $\mathrm{fl}(a - b) \in \mathtt{eps}\sigma\mathbb{Z}$ for $a \geq b \geq 0$ and $\sigma := \mathrm{ufp}(a)$. If $\mathrm{ufp}(b) \geq \frac{1}{2}\sigma$, then (2.13) implies $a, b \in \mathtt{eps}\sigma\mathbb{Z}$ and the assertion follows by (2.15). And if $\mathrm{ufp}(b) < \frac{1}{2}\sigma$, then $b < \frac{1}{2}\sigma$, and $a \geq \sigma$ implies $a - b > \frac{1}{2}\sigma \in \mathbb{F}$. Hence (2.6) shows $\mathrm{fl}(a - b) \geq \frac{1}{2}\sigma \in \mathbb{F}$ and (2.13) implies $\mathrm{fl}(a - b) \in \mathtt{eps}\sigma\mathbb{Z}$.

For later use we collect some more properties. For $r \in \mathbb{R}$ and $\tilde{r} = \mathrm{fl}(r)$,

$$\tilde{r} \neq 0 \quad \Rightarrow \quad \mathrm{ufp}(r) \leq \mathrm{ufp}(\tilde{r}) \tag{2.18}$$

$$\begin{array}{rcl} \tilde{r} \in \mathbb{F}\backslash\mathbb{U} & \Rightarrow & |\tilde{r} - r| \leq \mathtt{eps} \cdot \mathrm{ufp}(r) \leq \mathtt{eps} \cdot \mathrm{ufp}(\tilde{r}) \\ \tilde{r} \in \mathbb{U} & \Rightarrow & |\tilde{r} - r| \leq \frac{1}{2}\mathtt{eta} \ . \end{array} \tag{2.19}$$

Note that a strict inequality occurs in (2.18) iff $\tilde{r}$ is a power of 2 and $|r| < |\tilde{r}|$. The assertions follow by the rounding to nearest property of $\mathrm{fl}(\cdot)$. Applying (2.19), (2.18), (2.9) and (2.3) to floating-point addition yields for $a, b \in \mathbb{F}$,

$$f = \mathrm{fl}(a+b) \quad \Rightarrow \quad f = a + b + \delta \ \text{ with } \ |\delta| \leq \mathtt{eps} \cdot \mathrm{ufp}(a+b) \leq \mathtt{eps} \cdot \mathrm{ufp}(f) \leq \mathtt{eps}|f| \ . \tag{2.20}$$

We will frequently need this refined version of the standard estimation (2.2). Note that (2.20) is also true in the presence of underflow because in this case $\delta = 0$, the addition is exact. The ufp concept also allows

simple sufficient conditions for the fact that a floating-point addition is exact. For $a, b \in \mathbb{F}$ and $\sigma = 2^k$, $k \in \mathbb{Z}$,

$$\begin{aligned} a, b \in \mathtt{eps}\sigma\mathbb{Z} \quad \text{and} \quad \mathrm{fl}(|a+b|) < \sigma \quad &\Rightarrow \quad \mathrm{fl}(a+b) = a+b \qquad \text{and} \\ a, b \in \mathtt{eps}\sigma\mathbb{Z} \quad \text{and} \quad |a+b| \le \sigma \quad &\Rightarrow \quad \mathrm{fl}(a+b) = a+b \; . \end{aligned} \tag{2.21}$$

We need only to prove the second part since $\mathrm{fl}(|a+b|) < \sigma$ and (2.7) imply $|a+b| < \sigma$. To see the second part we first note that $a + b \in \mathtt{eps}\sigma\mathbb{Z}$. By (2.3) the addition is exact if $|a+b| \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$, and also if $|a+b| = \sigma$. Otherwise, (2.9) and (2.12) yield $\sigma > |a+b| \ge \mathrm{ufp}(a+b) \ge \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ since $\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ is a power of 2, so $\mathtt{eps}\sigma \ge 2\mathtt{eps} \cdot \mathrm{ufp}(a+b) \ge \mathtt{eta}$ and (2.14) do the job.

The well-known result by Sterbenz [18, Theorem 2.5] says that subtraction is exact if floating-point numbers $a, b \in \mathbb{F}$ of the same sign are not too far apart. More precisely, for $a, b \ge 0$ we have

$$\frac{1}{2}a \le b \le 2a \quad \Rightarrow \quad \mathrm{fl}(b-a) = b-a. \tag{2.22}$$

As an example of a proof we mention that with our machinery this is not difficult to see. If $b \ge a$, then (2.13) implies $a, b, a-b \in 2\mathtt{eps}\sigma\mathbb{Z}$ for $\sigma := \mathrm{ufp}(a)$. By assumption and (2.9), $|b-a| = b-a \le a < 2\sigma$, and (2.21) proves this part. For $b < a$, (2.13) implies $a, b, a-b \in 2\mathtt{eps}\sigma\mathbb{Z}$ for $\sigma := \mathrm{ufp}(b)$, and similarly $|b-a| = a-b \le b < 2\sigma$ and (2.21) finish the proof.

We define the floating-point predecessor and successor of a real number r with $\min\{f : f \in \mathbb{F}\} < r < \max\{f : f \in \mathbb{F}\}$ by

$$\mathrm{pred}(r) := \max\{f \in \mathbb{F} : f < r\} \quad \& \quad \mathrm{succ}(r) := \min\{f \in \mathbb{F} : r < f\}.$$

Using the ufp concept, the predecessor and successor of a floating-point number can be characterized as follows.

Lemma 2.2. *Let a floating-point number $0 \ne f \in \mathbb{F}$ be given. Then*

$$\begin{aligned} f \notin \mathbb{U} \quad \text{and} \quad |f| \ne \mathrm{ufp}(f) \quad &\Rightarrow \quad \mathrm{pred}(f) = f - 2\mathtt{eps} \cdot \mathrm{ufp}(f) \quad \text{and} \quad f + 2\mathtt{eps} \cdot \mathrm{ufp}(f) = \mathrm{succ}(f) \; , \\ f \notin \mathbb{U} \quad \text{and} \quad f = \mathrm{ufp}(f) \quad &\Rightarrow \quad \mathrm{pred}(f) = (1-\mathtt{eps})f \quad \text{and} \quad (1+2\mathtt{eps})f = \mathrm{succ}(f) \; , \\ f \notin \mathbb{U} \quad \text{and} \quad f = -\mathrm{ufp}(f) \quad &\Rightarrow \quad \mathrm{pred}(f) = (1-2\mathtt{eps})f \quad \text{and} \quad (1+\mathtt{eps})f = \mathrm{succ}(f) \; , \\ f \in \mathbb{U} \quad &\Rightarrow \quad \mathrm{pred}(f) = f - \mathtt{eta} \quad \text{and} \quad f + \mathtt{eta} = \mathrm{succ}(f) \; . \end{aligned}$$

For any $f \in \mathbb{F}$, also in underflow,

$$\mathrm{pred}(f) \le f - \mathtt{eps} \cdot \mathrm{ufp}(f) \le f + \mathtt{eps} \cdot \mathrm{ufp}(f) \le \mathrm{succ}(f) \; . \tag{2.23}$$

For $f \notin \mathbb{U}$,

$$f - 2\mathtt{eps} \cdot \mathrm{ufp}(f) \le \mathrm{pred}(f) < \mathrm{succ}(f) \le f + 2\mathtt{eps} \cdot \mathrm{ufp}(f) \; . \tag{2.24}$$

Remark. Note that we defined $\mathbb{U}$ in (2.1) to contain $\pm\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$, the smallest normalized floating-point numbers.

Proof. For $f \notin \mathbb{U}$ and $|f| \ne \mathrm{ufp}(f)$, use $\mathrm{ufp}(f) < |f| < 2\mathrm{ufp}(f)$, and the rest is not difficult to see. □

We collect some properties when a perturbation of a floating-point number a cannot change it or, cannot go beyond the immediate neighbor of a. These will be useful in the analysis of our algorithms producing K-fold accuracy to be presented in Section 5.

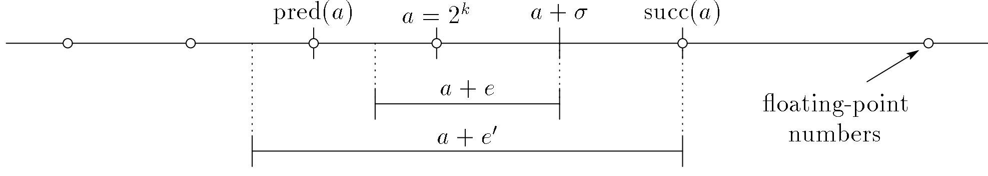

FIG. 2.2. *floating-point numbers in the neighborhood of a power of 2*

LEMMA 2.3. *Let floating-point numbers a, e, e' be given. Define $\mathcal{N}(a) := \{\mathrm{pred}(a), \mathrm{succ}(a)\}$ to be the neighbor set of a, and denote by $N(a) \in \mathbb{F}$ one of the neighbors of a, i.e. $N(a) \in \mathcal{N}(a)$. Then*

$$|e| < \frac{1}{2}|N(a) - a| \quad \Rightarrow \quad \mathrm{fl}(a+e) = a \ , \tag{2.25}$$

$$|a| \neq \mathrm{ufp}(a) \quad \textit{and} \quad |e| < \mathtt{eps} \cdot \mathrm{ufp}(a) \quad \Rightarrow \quad \mathrm{fl}(a+e) = a \ . \tag{2.26}$$

Moreover,

$$\mathrm{fl}(a+e) = a \quad \textit{and} \quad |e'| < |e| + \mathtt{eps} \cdot \mathrm{ufp}(a) \quad \Rightarrow \quad \mathrm{fl}(a+e') = a \quad \textit{or} \quad \mathrm{fl}(a+e') \in \mathcal{N}(a) \ , \tag{2.27}$$

where $e'e \geq 0$ in case $\mathrm{fl}(a+e') \neq a$.

PROOF. The first two assertions are obvious by the definition of rounding to nearest. To see (2.27), set $\sigma := \mathrm{ufp}(a)$. The critical case is that a is a power of 2, as depicted in Figure 2.2. In any case, also in the underflow range, Lemma 2.2 implies

$$\mathrm{pred}(\mathrm{pred}(a)) \leq a - 2\mathtt{eps}\sigma \leq a + 2\mathtt{eps}\sigma \leq \mathrm{succ}(\mathrm{succ}(a)) \ ,$$

and $\mathrm{fl}(a+e) = a$ yields

$$\frac{1}{2}\big(\mathrm{pred}(a) + a\big) \leq a - |e| \leq a + |e| \leq \frac{1}{2}\big(a + \mathrm{succ}(a)\big) \ .$$

Hence

$$\frac{1}{2}\big(\mathrm{pred}(\mathrm{pred}(a)) + \mathrm{pred}(a)\big) \leq a - |e| - \mathtt{eps}\sigma < a - |e'| \leq$$
$$a + |e'| < a + |e| + \mathtt{eps}\sigma \leq \frac{1}{2}\big(\mathrm{succ}(\mathrm{succ}(a)) + \mathrm{succ}(a)\big) \ .$$

The lemma follows. □

The aim of this paper is to present a summation algorithm computing a faithfully rounded exact result of the sum. That means [10, 39, 9] that the computed result is definitely one of the immediate floating-point neighbors of the exact (real) result. If the exact result is a floating-point number, then this will be the result of our algorithm.

DEFINITION 2.4. *A floating-point number $f \in \mathbb{F}$ is called a* faithful rounding *of a real number $r \in \mathbb{R}$ if*

$$\mathrm{pred}(f) < r < \mathrm{succ}(f) \ . \tag{2.28}$$

We denote this by $f \in \Box(r)$. For $r \in \mathbb{F}$ this implies $f = r$.

For general $r \notin \mathbb{F}$, two floating-point numbers satisfy $f \in \Box(r)$ for given $r \in \mathbb{R}$, so a little accuracy is lost. The rounded to nearest result, however, has the well-known drawback that if the exact result is very close to the midpoint of two adjacent floating-point numbers, its computation may require a substantial and often not necessary additional effort. Our Algorithm 6.1 (`NearSum`) computes the rounded to nearest result. The computing time depends in this case on the exponent range of the summands.

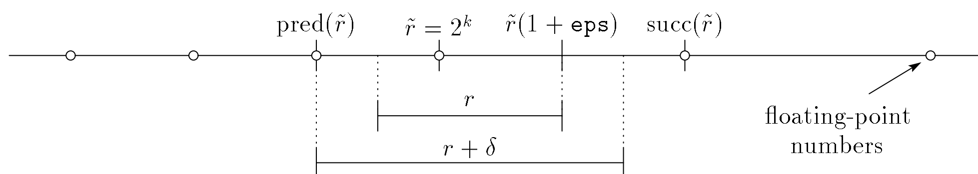

FIG. 2.3. *Faithful rounding near a power of* 2

In contrast, Algorithm 4.4 (AccSum) computes a faithfully rounded result. Its computing time is proportional to the condition number of the sum.

Suppose $r+\delta$ is the exact result of a summation, composed of a (real) approximation r and an error term δ. Next we establish conditions on δ to ensure that $\mathrm{fl}(r)$ is a faithful rounding of $r+\delta$. As in (2.27), the critical case is the change of exponent at a power of 2, depicted in Figure 2.3.

LEMMA 2.5. *Let $r, \delta \in \mathbb{R}$ and $\tilde{r} := \mathrm{fl}(r)$. If $\tilde{r} \notin \mathbb{U}$ suppose $2|\delta| < \texttt{eps}|\tilde{r}|$, and if $\tilde{r} \in \mathbb{U}$ suppose $|\delta| < \frac{1}{2}\texttt{eta}$. Then $\tilde{r} \in \Box(r+\delta)$, that means $\tilde{r}$ is a faithful rounding of $r+\delta$.*

PROOF. According to Definition 2.4 we have to prove $\mathrm{pred}(\tilde{r}) < r+\delta < \mathrm{succ}(\tilde{r})$. If $\tilde{r} \in \mathbb{U}$, then $|\tilde{r}-r| \le \frac{1}{2}\texttt{eta}$ by (2.19), so Lemma 2.2 yields

$$\mathrm{pred}(\tilde{r}) = \tilde{r} - \texttt{eta} < \tilde{r} - |\tilde{r}-r| + \delta \le r+\delta \le \tilde{r} + |\tilde{r}-r| + \delta < \tilde{r} + \texttt{eta} = \mathrm{succ}(\tilde{r})$$

and finishes this part. It remains to treat the case $\tilde{r} \notin \mathbb{U}$.

Then $\mathrm{ufp}(\tilde{r}) \le |\tilde{r}| < 2\mathrm{ufp}(\tilde{r})$ by (2.9), so $|\delta| < \texttt{eps} \cdot \mathrm{ufp}(\tilde{r})$. Suppose $r \le \tilde{r}$. Then rounding to nearest implies

$$0 \le \tilde{r} - r \le \frac{1}{2}(\tilde{r} - \mathrm{pred}(\tilde{r})) \quad \text{and} \quad |\delta| < \frac{1}{2}(\tilde{r} - \mathrm{pred}(\tilde{r})) \ ,$$

where the latter follows directly from Lemma 2.2 if $|\tilde{r}|$ is not a power of 2, and otherwise by $2|\delta| < \texttt{eps}|\tilde{r}| = \texttt{eps} \cdot \mathrm{ufp}(\tilde{r}) = \tilde{r} - \mathrm{pred}(\tilde{r})$. Hence (2.23) yields

$$\mathrm{pred}(\tilde{r}) = \tilde{r} - (\tilde{r} - \mathrm{pred}(\tilde{r})) < \tilde{r} - (\tilde{r} - r) - |\delta| \le r + \delta \le \tilde{r} + \delta < \tilde{r} + \texttt{eps} \cdot \mathrm{ufp}(\tilde{r}) \le \mathrm{succ}(\tilde{r}) \ .$$

The case $r > \tilde{r}$ follows similarly. □

A faithfully rounded result satisfies some weak ordering properties. For $f, f_1, f_2 \in \mathbb{F}$, $r \in \mathbb{R}$ and $f \in \Box(r)$, i.e. f is a faithful rounding of r, one verifies

$$\begin{array}{ccc} f_1 < f < f_2 & \Rightarrow & f_1 < r < f_2 \\ f_1 < r < f_2 & \Rightarrow & f_1 \le f \le f_2 \ . \end{array} \tag{2.29}$$

The following lemma shows that if the sum $s \in \mathbb{R}$ of some floating-point numbers is in the underflow range, a faithful rounding f of s is strong enough to imply $f = s$.

LEMMA 2.6. *Let $r \in \texttt{eta}\mathbb{Z}$ be given, and let $f \in \mathbb{F}$ be a faithful rounding of r. Then*

$$|r| \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta} \quad \Rightarrow \quad f \in \mathbb{U} \quad \Rightarrow \quad r = f \ . \tag{2.30}$$

Therefore especially $f = 0$ iff $r = 0$ and

$$\mathrm{sign}(r) = \mathrm{sign}(f) \ .$$

For $r \ne 0$ it holds

$$|r - f| < 2\texttt{eps}|r| \quad \textit{and} \quad |r - f| < 2\texttt{eps} \cdot \mathrm{ufp}(f) \ . \tag{2.31}$$

PROOF. If $|r| = \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then faithful rounding implies $f = r \in \mathbb{U}$. If $|r| < \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then $f \in \mathbb{U}$ by (2.29), so that $f \pm \texttt{eta}$ are the neighbors of f by Lemma 2.2. Hence (2.28) yields $|r - f| < \texttt{eta}$, and $f \in \mathbb{F} \subseteq \texttt{eta}\mathbb{Z}$ and $r \in \texttt{eta}\mathbb{Z}$ imply $r = f$. This proves (2.30).

To prove (2.31) we may assume $f \notin \mathbb{U}$ and without loss of generality $f > 0$. For f not being a power of 2 we have $\mathrm{ufp}(f) < |r|$, so Lemma 2.2 and (2.28) imply

$$|r - f| < 2\texttt{eps} \cdot \mathrm{ufp}(f) \le 2\texttt{eps}|r| \ .$$

This is also true for f being a power of 2 and $f \le r < \mathrm{succ}(f)$, and for $(1 - \texttt{eps})f = \mathrm{pred}(f) < r \le f$ we have

$$|r - f| < \texttt{eps} \cdot \mathrm{ufp}(f) = \texttt{eps} \cdot f < (1 - \texttt{eps})^{-1}\texttt{eps} \cdot r < 2\texttt{eps}|r| \ .$$

The lemma is proved. □

Definition 2.4 and Lemma 2.5 are formulated for general $r \in \mathbb{R}$. For our main application, the approximation of the sum $s = \sum a_i$ of floating-point numbers, $a_i \in \texttt{eta}\mathbb{Z}$ implies $s \in \texttt{eta}\mathbb{Z}$ and Lemma 2.6 is applicable.

As has been noted in (2.4), the error of a floating-point addition is always a floating-point number. Fortunately, rather than Algorithm 2.1 (TwoSum) we can use in our applications the following faster algorithm due to Dekker [10]. Again, the computation is very efficient because only standard floating-point addition and subtraction is used.

ALGORITHM 2.7. *Compensated summation of two floating-point numbers.*

```
function [x, y] = FastTwoSum(a, b)
    x = fl(a + b)
    q = fl(x − a)
    y = fl(b − q)
```

In Dekker's original algorithm, y is computed by $y = \mathrm{fl}((a-x)+b)$, which is equivalent to the last statement in Algorithm 2.7 because $\mathbb{F} = -\mathbb{F}$ and $\mathrm{fl}(-r) = -\mathrm{fl}(r)$ for $r \in \mathbb{R}$. For floating-point arithmetic with rounding to nearest and base 2, e.g. IEEE 754 arithmetic, Dekker [10] showed in 1971 that the correction is *exact* if the input is ordered by magnitude, that is

$$x + y = a + b \tag{2.32}$$

provided $|a| \ge |b|$. In [36] we showed that the obvious way to get rid of this assumption is suboptimal on today's computers because a branch slows down computation significantly (see also Section 9).

Algorithm 2.7 (FastTwoSum) is an error-free transformation of the pair of floating-point numbers (a, b) into a pair (x, y). Our first algorithm 4.4 (AccSum) to be presented can also be viewed as an error-free transformation of a vector p into floating-point numbers τ_1, τ_2 and a vector p' such that $\sum p_i = \tau_1 + \tau_2 + \sum p_i'$, and $\texttt{res} := \mathrm{fl}(\tau_1 + (\tau_2 + \sum p_i'))$ is the faithfully rounded sum $\sum p_i$. To prove this we need to refine the analysis of Algorithm 2.7 by weakening the assumption $|a| \ge |b|$. For completeness we first prove Dekker's result [10].

LEMMA 2.8. *Let a, b be floating-point numbers and $|a| \ge |b|$. Let x, y be the results produced by Algorithm 2.7 (FastTwoSum) applied to a, b. Then*

$$x + y = a + b \ , \quad x = \mathrm{fl}(a + b) \quad \textit{and} \quad |y| \le \texttt{eps} \cdot \mathrm{ufp}(a + b) \le \texttt{eps} \cdot \mathrm{ufp}(x) \ . \tag{2.33}$$

Furthermore,

$$q = \mathrm{fl}(x - a) = x - a \quad \textit{and} \quad y = \mathrm{fl}(b - q) = b - q \ , \tag{2.34}$$

```
for i = 1 : n
    q = fix(p_i/b)    % fix: Round to integer towards zero
    t_j = t_j + q
    p_i = p_i - q * b
end for
```

FIG. 3.1. *Inner loop of the summation by Zielke and Drygalla*

that is the floating-point subtractions $x - a$ and $b - q$ are exact.

PROOF. We first show $\mathrm{fl}(x-a) = x - a$. Without loss of generality we can assume $a \geq 0$. If $0 \leq b \leq a$, then $a + b \leq 2a$ and $x = \mathrm{fl}(a+b) \leq 2a$ by (2.6), so $\frac{1}{2}x \leq a \leq x$ and (2.22) imply $\mathrm{fl}(x-a) = -\mathrm{fl}(a-x) = x - a$. If $\frac{1}{2}a \leq -b \leq a$, then $-x = \mathrm{fl}(-b-a) = -b - a$ again by (2.22), and $\mathrm{fl}(x-a) = \mathrm{fl}(b) = b = x - a$. If finally $0 \leq -b < \frac{1}{2}a$, then $\frac{1}{2}a < a + b \leq a$ and $\frac{1}{2}a \leq x \leq a$ by (2.6), so $\mathrm{fl}(x-a) = x - a = q$ again by (2.22). This proves $q = x - a$.

Define $\delta := x - (a+b) = q - b$. Then $\delta \in \mathbb{F}$ by (2.4) and $y = \mathrm{fl}(b-q) = \mathrm{fl}(-\delta) = -\delta = b - q$, proving (2.34) and the first part of (2.33). The second part follows by and $|y| = |\delta|$ and (2.20). ☐

Next we weaken the assumption $|a| \geq |b|$.

LEMMA 2.9. *Let a, b be floating-point numbers with* $\mathrm{ufp}(b) \leq \mathrm{ufp}(a)$. *Let x, y be the results produced by Algorithm 2.7* (`FastTwoSum`) *applied to a, b. Then the assertions (2.33) and (2.34) of Lemma 2.8 are still valid.*

PROOF. Using Lemma 2.8 we can assume $|a| < |b|$. This implies $\mathrm{ufp}(b) \leq \mathrm{ufp}(a) \leq |a| < |b| < 2\mathrm{ufp}(b)$, so $\mathrm{ufp}(b) = \mathrm{ufp}(a) =: \sigma/2$, and $a, b \in \mathtt{eps}\sigma\mathbb{Z}$ by (2.13). Hence $|a+b| < 2\mathrm{ufp}(a) + 2\mathrm{ufp}(b) = 2\sigma$, so $\mathrm{ufp}(a+b) \leq \sigma$. Furthermore, (2.15) and (2.20) yield $\delta := x - (a+b) \in \mathtt{eps}\sigma\mathbb{Z}$ and $|\delta| \leq \mathtt{eps} \cdot \mathrm{ufp}(a+b) \leq \mathtt{eps}\sigma$. That means, the binary representation of δ has at most one nonzero bit. Furthermore, $|b| < \sigma$ and (2.23) imply $|b| \leq \mathrm{pred}(\sigma) \leq \sigma - \mathtt{eps}\sigma$, also in the presence of underflow, and therefore $|b + \delta| \leq \sigma$. So (2.21) yields $x - a = b + \delta = \mathrm{fl}(b+\delta) = \mathrm{fl}(x-a) = q$, and $y = \mathrm{fl}(b-q) = \mathrm{fl}(-\delta) = -\delta = b - q$, proving (2.34). Hence $x + y = x - \delta = a + b$, and the estimation on $|y| = |\delta|$ follows by (2.20). This finishes the proof. ☐

Lemma 2.9 may also offer possibilities for summation algorithms based on sorting: To apply `FastTwoSum` it suffices to "sort by exponent", which has complexity $\mathcal{O}(n)$.

Lemma 2.9 shows that $|a|$ may be less than $|b|$ without jeopardizing the assertions of Lemma 2.8 as long as the leading bit of a is not less than the one of b. The next lemma shows that also this assumption may be dropped as long as there are not too many trailing nonzero bits of a.

LEMMA 2.10. *Let a, b be floating-point numbers and $\sigma = 2^k$ for $k \in \mathbb{Z}$. Suppose $a, b \in \mathtt{eps}\mathbb{Z}$ and $|b| < \sigma$. Let x, y be the results produced by Algorithm 2.7* (`FastTwoSum`) *applied to a, b. Then the assertions (2.33) and (2.34) of Lemma 2.8 are still valid.*

PROOF. Let $\mathrm{fl}(a+b) = a + b + \delta$. If $|a| \geq \frac{1}{2}\sigma$, we can use Lemma 2.9. Otherwise, $|a+b| < \frac{3}{2}\sigma$, so (2.15) and (2.20) imply $|\delta| \leq \mathtt{eps}\sigma$. In fact, either $|\delta| = \mathtt{eps}\sigma$ or $\delta = 0$. Hence $|x - a| = |b + \delta| \leq \mathrm{pred}(\sigma) + \mathtt{eps}\sigma \leq \sigma$, so (2.21) yields $q = \mathrm{fl}(x-a) = x - a$. Now we can proceed as in the proof of Lemma 2.9. ☐

3. Extraction of high order parts. The main idea of the method by Zielke and Drygalla [45] to compute the sum of floating-point numbers $p_i, 1 \leq i \leq n$, is as follows. First they compute $k \in \mathbb{N}$ such that $\max |p_i| < 2^k$, and some M such that $n < 2^M$. Then, for $\mathtt{eps} = 2^{-53}$, they extract the "bits" from k downto $k - (53 - M) + 1$ of p_i into q_i and add the q_i into some t_1. By the choice of M this addition is exact. They

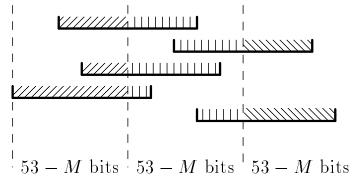

FIG. 3.2. *Zielke and Drygalla's scheme*

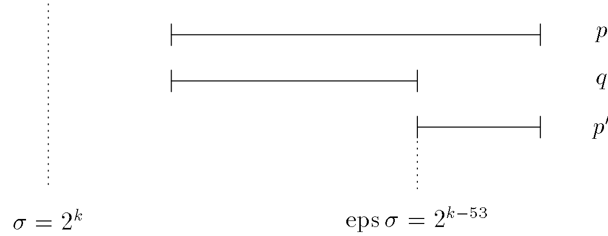

FIG. 3.3. *Extract Scalar: error-free transformation* $p = q + p'$

then continue by extracting the "bits" from $k - (53 - M)$ downto $k - 2(53 - M) + 1$ and summing them into t_2 and so forth until all bits are processed. For given M and k and assuming no underflow the inner loop looks as in Figure 3.1. Since q can be overwritten, no index is necessary. Here b is an appropriate power of 2, which is updated before entering the for-loop again. By the principle of the approach, the sum t_j of one "chunk" of $53 - M$ bits is exact. After finishing, the t_j, appropriately shifted, are added.

The method is illustrated in Figure 3.2. The authors also mention a faster, though less accurate method, to extract only the leading $53 - M$ bits and adding the rest accepting accumulating rounding errors. However, for neither method an error analysis is given.

The inner loop of the method requires to split a floating-point number $p \in \mathbb{F}$ according to Figure 3.3, which is done by chopping (fix) by Zielke and Drygalla. There are other possibilities, for example to round from floating-point to integer (rather than chopping), or the assignment of a floating-point number to a long integer (if supported by the hardware in use). Also direct manipulation by accessing mantissa and exponent is possible. However, all these methods slow down the extraction significantly, often by an order of magnitude and more compared to our following Algorithm 3.1 (`ExtractScalar`). We will show corresponding performance data in Section 9.

Our approach follows a similar scheme as in Figure 3.2. However, we carefully estimate how many bits of the result have to be extracted to guarantee a faithful rounding of the result. For the splitting as depicted in Figure 3.3, neither the high order part q and low order part p' need to match bitwise with the original p, nor must q and p' have the same sign; only the error-freeness of the transformation $p = q + p'$ is mandatory. This is achieved by the following fast algorithm, where σ denotes a power of 2 not less than $|p|$.

ALGORITHM 3.1. *Error-free transformation extracting high order part.*

$$\begin{aligned}&\text{function } [q, p'] = \texttt{ExtractScalar}(\sigma, p)\\ &\quad q = \text{fl}\left((\sigma + p) - \sigma\right)\\ &\quad p' = \text{fl}(p - q)\end{aligned}$$

There is an important difference to the splitting in Algorithm 1.2 (`Split`). There, a floating-point number is split into two parts relative to its exponent, and both the high and the low part has at most 26 significant

bits. In `ExtractScalar` a floating-point number is split relative to σ, a fixed power of 2. The higher and the lower part of the splitting may have between 0 and 53 significant bits, depending on σ.

This clever way of splitting in Algorithm 3.1 (`ExtractScalar`) is due to the second author. The performance of splitting is crucial since it is in the inner loops of our algorithms.

We think this method is known, but the only reference we found for this is [16]. However, we do not know of any analysis of Algorithm 3.1, so we develop it in the following lemma.

LEMMA 3.2. *Let q and p' be the results of Algorithm 3.1* (`ExtractScalar`) *applied to floating-point numbers σ and p. Assume $\sigma = 2^k \in \mathbb{F}$ for some $k \in \mathbb{Z}$, assume $|p| \le 2^{-M}\sigma$ for some $0 \le M \in \mathbb{N}$. Then*

$$p = q + p' \ , \quad |p'| \le \mathtt{eps}\sigma \ , \quad |q| \le 2^{-M}\sigma \quad and \quad q \in \mathtt{eps}\sigma\mathbb{Z} \ . \tag{3.1}$$

PROOF. We first note that $\mathtt{ExtractScalar}(\sigma, p)$ performs exactly the same operations in the same order as $[x, y] = \mathtt{FastTwoSum}(\sigma, p)$, so $|p| \le \sigma$ and Lemma 2.8 imply $p' = p - q$. If $|p| = \sigma$, then $p' = 0$, otherwise (2.33) implies $|p'| \le \mathtt{eps} \cdot \mathrm{ufp}(\sigma + p) \le \mathtt{eps}\sigma$. Furthermore, $q \in \mathtt{eps}\sigma\mathbb{Z}$ follows by (2.17).

Finally we prove $|q| \le 2^{-M}\sigma$. First, suppose $\sigma + \mathrm{sign}(p)2^{-M}\sigma$ is a floating-point number. Then $\sigma + p$ is in the interval with floating-point endpoints σ and $\sigma + \mathrm{sign}(p)2^{-M}\sigma$, so (2.6) implies that $x := \mathrm{fl}(\sigma + p)$ is in that interval and $|q| = |x - \sigma| \le 2^{-M}\sigma$ follows. Second, suppose $\sigma + \mathrm{sign}(p)2^{-M}\sigma$ is not a floating-point number. Then $\mathrm{fl}(\sigma + \mathrm{sign}(p)2^{-M}\sigma) = \sigma$ because $\mathrm{sign}(p)2^{-M}\sigma$ is a power of 2 and rounding to nearest is tie to even, so monotonicity of the rounding implies $\mathrm{fl}(\sigma + p) = \sigma$ and $q = 0$. ☐

Following we adapt Algorithm 3.1 (`ExtractScalar`) to the error-free transformation of an entire vector. For this case we prove that the high order parts can be summed up without error. For better readability and analysis the extracted parts are stored in a vector q_i. In a practical implementation, the vector q is not necessary but only its sum.

ALGORITHM 3.3. *Error-free vector transformation extracting high order part.*

```
function [τ, p'] = ExtractVector(σ, p)
    τ = 0
    for i = 1 : n
        [q_i, p'_i] = ExtractScalar(σ, p_i)
        τ = fl(τ + q_i)
    end for
```

Algorithm 3.3 proceeds as depicted in Figure 3.4. Note that the loop can be executed in parallel.

Note again that the low order parts, which are collected in p', neither need to be bitwise identical to those of p nor do they need to have the same sign. The important property we are after is that the transformation is performed without error, i.e. $\sum p_i = \tau + \sum p'_i$, and that $|p'_i|$ stays below $\mathtt{eps}\sigma$. The validity of the algorithm is demonstrated by the following theorem.

THEOREM 3.4. *Let τ and p' be the results of Algorithm 3.3* (`ExtractVector`) *applied to $\sigma \in \mathbb{F}$ and a vector of floating-point numbers $p_i, 1 \le i \le n$. Assume $\sigma = 2^k \in \mathbb{F}$ for some $k \in \mathbb{Z}$, $n + 2 \le 2^M$ for some $M \in \mathbb{N}$ and $|p_i| \le 2^{-M}\sigma$ for all i. Then*

$$\sum_{i=1}^{n} p_i = \tau + \sum_{i=1}^{n} p'_i \ , \quad \max |p'_i| \le \mathtt{eps}\sigma \ , \quad |\tau| < \sigma \quad and \quad \tau \in \mathtt{eps}\sigma\mathbb{Z} \ . \tag{3.2}$$

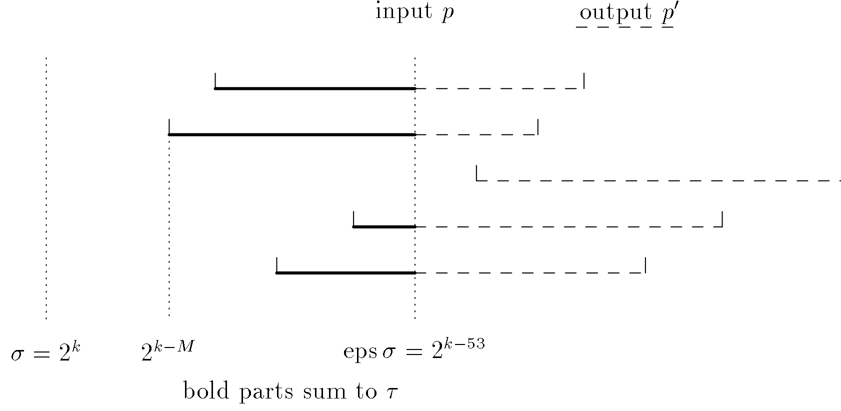

FIG. 3.4. *Extract vector: error-free transformation* $\sum p_i = \tau + \sum p'_i$

If $2^{2M-1}\mathtt{eps} \le 1$ *and* $\sigma \notin \mathbb{U}$, *then*

$$|\mathrm{fl}(\tau + T)| < \sigma \quad \textit{for} \quad T := \mathrm{fl}(\sum_{i=1}^{n} p'_i) \ . \tag{3.3}$$

REMARK. Note that for (3.2) we have no assumption on the size of M. However, for extremely large M with $2^M\mathtt{eps} \ge 1$, $|p_i| \le 2^{-M}\sigma$ implies that the extracted parts summed up into τ consist of at most one bit, so not much is gained.

PROOF OF THEOREM 3.4. Lemma 3.2 implies $p_i = q_i + p'_i$, $|p'_i| \le \mathtt{eps}\sigma$, $|q_i| \le 2^{-M}\sigma$ and $q_i \in \mathtt{eps}\sigma\mathbb{Z}$ for all $i \in \{1, \cdots, n\}$. So $\tau \in \mathtt{eps}\sigma\mathbb{Z}$, and $\sum |q_i| \le n2^{-M}\sigma < \sigma$ and (2.21) imply $\tau = \mathrm{fl}(\sum q_i) = \sum q_i$ and $|\tau| \le n2^{-M}\sigma < \sigma$. This proves (3.2). By (3.2) and (2.16) it follows $|T| \le n\mathtt{eps}\sigma$, so Lemma 2.2 implies

$$\begin{aligned} |\tau + T| &\le (n2^{-M} + n\mathtt{eps})\sigma \le (2^M - 2)(2^{-M} + \mathtt{eps})\sigma \\ &= (1 - 2^{-M+1}(1 - 2^{2M-1}\mathtt{eps}) - 2\mathtt{eps})\sigma < (1 - \mathtt{eps})\sigma \\ &= \mathrm{pred}(\sigma) \end{aligned}$$

because $\sigma \notin \mathbb{U}$, so that (2.6) implies $|\mathrm{fl}(\tau + T)| = \mathrm{fl}(|\tau + T|) \le \mathrm{pred}(\sigma) < \sigma$. □

Note that (3.2) requires no assumption on σ other than being a power or 2 and bounding $2^M|p_i|$; σ may well be in the underflow range.

To apply Theorem 3.4 the best (smallest) values for M and σ are $\lceil \log_2(n+2) \rceil$ and $2^{M+\lceil \log_2(\max |p_i|) \rceil}$, respectively. To avoid the use of standard functions, these can be calculated by the following algorithm.

ALGORITHM 3.5. *Computation of* $2^{\lceil \log_2 |p| \rceil}$ *for* $p \ne 0$.

```
function L = NextPowerTwo(p)
    q = eps^{-1} p
    L = fl(|(q + p) - q|)
    if L = 0
        L = |p|
    end if
```

THEOREM 3.6. *Let L be the result of Algorithm 3.5* (NextPowerTwo) *applied to a nonzero floating-point number p. If no overflow occurs, then* $L = 2^{\lceil \log_2 |p| \rceil}$.

REMARK. For simplicity we skipped the obvious check for large input number p to avoid overflow in the computation of q. However, we will show that $L = 2^{\lceil \log_2 |p| \rceil}$ is satisfied in the presence of underflow.

PROOF. First assume $|p| = 2^k$ for some $k \in \mathbb{Z}$. Then the rounding tie to even implies $\mathrm{fl}(q+p) = \mathrm{fl}(q(1+\mathtt{eps})) = q$, so that $\mathrm{fl}(|(q+p)-q|) = 0$, and $L = |p| = 2^k = 2^{\lceil \log_2 |p| \rceil}$ for the final result L.

So we may assume that p is not a power of 2, and without loss of generality we assume $p > 0$. Then

$$\mathrm{ufp}(p) < p < 2\mathrm{ufp}(p) \ ,$$

and we have to show $L = 2\mathrm{ufp}(p)$. Define $x := \mathrm{fl}(q+p)$. By Lemma 2.8 the computation of $\mathrm{fl}(x-q)$ causes no rounding error, so that $L = \mathrm{fl}(q+p) - q$. By definition, $\mathrm{ufp}(q) = \mathtt{eps}^{-1}\mathrm{ufp}(p) < \mathtt{eps}^{-1}p = q$, so that $q \notin \mathbb{U}$ and Lemma 2.2 imply $\mathrm{succ}(q) = q + 2\mathtt{eps} \cdot \mathrm{ufp}(q)$. That means $q + \mathtt{eps} \cdot \mathrm{ufp}(q)$ is the midpoint of q and $\mathrm{succ}(q)$. Hence rounding to nearest and

$$q + \mathtt{eps} \cdot \mathrm{ufp}(q) < q + \mathtt{eps} \cdot q = q + p < q + 2\mathrm{ufp}(p) = \mathrm{succ}(q)$$

implies $\mathrm{fl}(q+p) = \mathrm{succ}(q)$, so that $L = \mathrm{fl}(q+p) - q = 2\mathtt{eps} \cdot \mathrm{ufp}(q) = 2\mathrm{ufp}(p)$. The theorem is proved. □

4. Algorithms and analysis. To ease analysis, we formulate our summation algorithms with superscripts to variables to identify the different stages. Of course in the actual implementation especially vectors are overwritten.

We first transform an input vector plus offset into two leading and a remaining part. The following formulation is aimed on readability rather than efficiency. In this first step we especially avoid a check for zero sum.

ALGORITHM 4.1. *Preliminary version of transformation of vector $p^{(0)}$ plus offset ϱ.*

```
function [τ_1, τ_2, p^(m), σ] = Transform(p^(0), ϱ)
    μ = max(|p_i^(0)|)
    if μ = 0,  τ_1 = ϱ,  τ_2 = p^(m) = σ = 0,  return,  end if
    M = ⌈ log_2 (length(p^(0)) + 2) ⌉
    σ_0 = 2^(M+⌈log_2(μ)⌉)
    t^(0) = ϱ,  m = 0
    repeat
        m = m + 1
        [τ^(m), p^(m)] = ExtractVector(σ_{m-1}, p^(m-1))
        t^(m) = fl(t^(m-1) + τ^(m))
        σ_m = fl(2^M eps σ_{m-1})
    until  σ_{m-1} ≤ (1/2) eps^-1 eta   or   |t^(m)| ≥ fl(2^(2M+1) eps σ_{m-1})
    σ = σ_{m-1}
    [τ_1, τ_2] = FastTwoSum(t^(m-1), τ^(m))
```

REMARK 1. The output parameter σ is not necessary in the following applications of `Transform` but added for clarity in the forthcoming proofs.

REMARK 2. For clarity we also use for the moment the logarithm in the computation of M and σ_0. Later this will be replaced by Algorithm 3.5 (`NextPowerTwo`) (see the Matlab code given in the Appendix).

LEMMA 4.2. *Let $\tau_1, \tau_2, p^{(m)}, \sigma$ be the results of Algorithm 4.1 (`Transform`) applied to a nonzero vector of floating-point numbers $p_i^{(0)}, 1 \le i \le n$, and $\varrho \in \mathbb{F}$. Assume that $2(n+2)^2\mathtt{eps} \le 1$ and $\varrho \in \mathtt{eps}\sigma_0\mathbb{Z}$ is satisfied for $M := \lceil \log_2(n+2) \rceil$, $\mu := \max_i |p_i|$ and $\sigma_0 = 2^{M + \lceil \log_2 \mu \rceil}$. Denote $s := \sum_{i=1}^n p_i^{(0)}$.*

Then Algorithm 4.1 will stop, and

$$s + \varrho = t^{(m-1)} + \tau^{(m)} + \sum_{i=1}^{n} p_i^{(m)} \tag{4.1}$$

$$\max |p_i^{(m)}| \le \texttt{eps}\sigma_{m-1} \ , \quad |\tau^{(m)}| < \sigma_{m-1} \quad \text{and} \quad t^{(m-1)}, \tau^{(m)} \in \texttt{eps}\sigma_{m-1}\mathbb{Z} \tag{4.2}$$

is true for all m between 1 and its final value. Moreover,

$$\tau_1 + \tau_2 = t^{(m-1)} + \tau^{(m)} \ , \quad \tau_1 = \mathrm{fl}(\tau_1 + \tau_2) = \mathrm{fl}(t^{(m-1)} + \tau^{(m)}) = t^{(m)} \tag{4.3}$$

is satisfied for the final value of m. If $\sigma_{m-1} > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ is satisfied for the final value of m, then

$$\mathrm{ufp}(\tau_1) \ge 2^{2M+1}\texttt{eps}\sigma_{m-1} \ . \tag{4.4}$$

Remark. Note that the computation of σ_m may be afflicted with a rounding error if σ_{m-1} is in the underflow range $\mathbb{U}$. However, we will see that this cannot do any harm. The computation of $\sigma = \sigma_{m-1}$ can never be afflicted with rounding errors.

Proof of Lemma 4.2. Algorithm `Transform` will stop because σ_m is decreased in each loop. We proceed by induction to prove (4.1) and (4.2). Theorem 3.6 and the initialization in Algorithm 4.1 imply $M = \lceil \log_2(n+2) \rceil$ and $\max |p_i^{(0)}| = \mu \le 2^{\lceil \log_2(\mu) \rceil} = 2^{-M}\sigma_0$, so that the assumptions of Theorem 3.4 are satisfied for σ_0 and $p^{(0)}$ as input to `ExtractVector`. Note this is also true if $\sigma_0 \in \mathbb{U}$. This yields (4.2) for $m = 1$, where $t^{(0)} = \varrho \in \texttt{eps}\sigma_0\mathbb{Z}$ is assured by the assumptions of the lemma. Furthermore, $s = \tau^{(1)} + \sum p_i^{(1)}$, and (4.1) is also proved for $m = 1$.

Now assume the repeat-until loop has been executed, denote by m the current value (immediately before the "until"-statement), and assume that (4.1) and (4.2) are true for the previous index $m-1$. The previous "until"-condition in Algorithm 4.1 implies

$$\sigma_{m-2} \ge \texttt{eps}^{-1}\texttt{eta}$$

because σ_{m-2} is a power of 2, hence no rounding error has occurred in the previous computation of $2^{2M+1}\texttt{eps}\sigma_{m-2}$. So the induction hypothesis and the "until"-condition yield

$$|t^{(m-1)}| = |\mathrm{fl}(t^{(m-2)} + \tau^{(m-1)})| < 2^{2M+1}\texttt{eps}\sigma_{m-2} \le \sigma_{m-2} \ , \tag{4.5}$$

the latter by the assumptions on n and M. By induction hypothesis, $t^{(m-2)}, \tau^{(m-1)} \in \texttt{eps}\sigma_{m-2}\mathbb{Z}$, so that (2.21) implies

$$t^{(m-1)} = \mathrm{fl}(t^{(m-2)} + \tau^{(m-1)}) = t^{(m-2)} + \tau^{(m-1)} \ , \tag{4.6}$$

and the induction hypothesis on (4.1) yields

$$s + \varrho = t^{(m-2)} + \tau^{(m-1)} + \sum_{i=1}^{n} p_i^{(m-1)} = t^{(m-1)} + \sum_{i=1}^{n} p_i^{(m-1)} \ .$$

By (4.2) we know $\max |p_i^{(m-1)}| \le \texttt{eps}\sigma_{m-2} = 2^{-M}\sigma_{m-1}$. Hence Theorem 3.4 is applicable and shows $\sum p_i^{(m-1)} = \tau^{(m)} + \sum p_i^{(m)}$, and therefore (4.1) for index m. It also shows (4.2), where $t^{(m-1)} \subseteq \texttt{eps}\sigma_{m-1}\mathbb{Z}$ follows by $t^{(m-2)}, \tau^{(m-1)} \in \texttt{eps}\sigma_{m-2}\mathbb{Z}$, (4.6) and (2.11). We proved (4.1) and (4.2). Therefore, for the last line in Algorithm `Transform` the assumptions of Lemma 2.10 are satisfied and (4.3) follows.

If $\sigma_{m-1} > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ is satisfied for the final value of m, then $\sigma_{m-1} \ge \texttt{eps}^{-1}\texttt{eta}$ because σ_{m-1} is a power of 2. Therefore, $\mathrm{fl}(2^{2M+1}\texttt{eps}\sigma_{m-1}) = 2^{2M+1}\texttt{eps}\sigma_{m-1}$, and the "until"-condition and (4.3) yield $|\tau_1| = |t^{(m)}| \ge 2^{2M+1}\texttt{eps}\sigma_{m-1}$, which implies (4.4). The lemma is proved. □

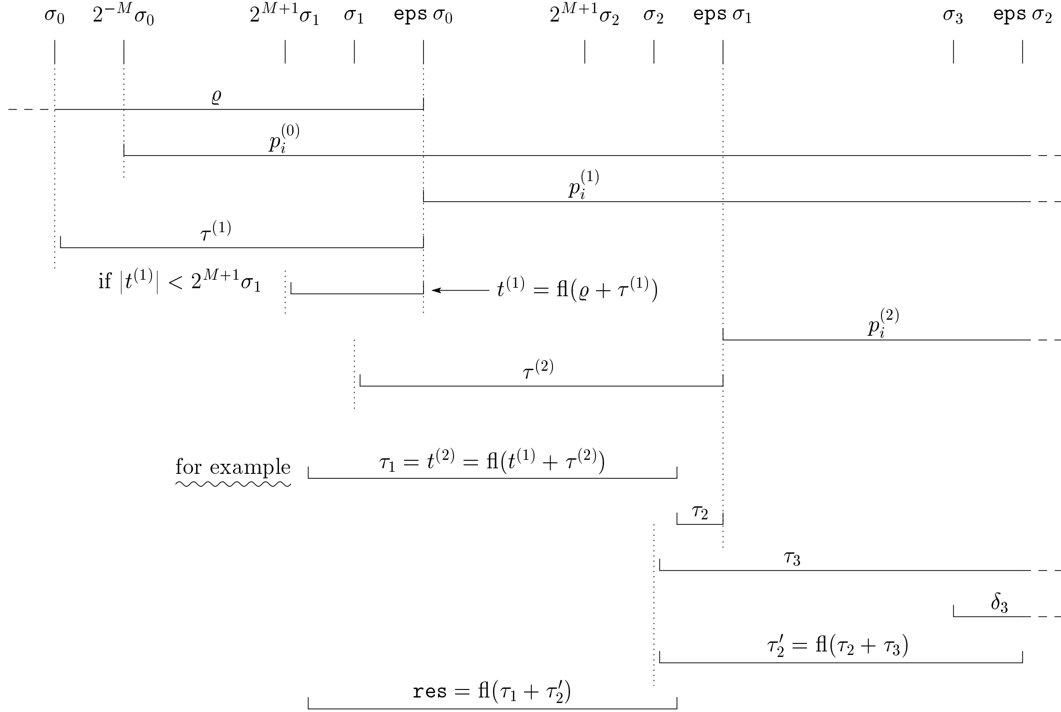

FIG. 4.1. *Outline of faithful rounding for $m = 2$.*

The proof of Lemma 4.2 makes also clear why the assumption $\varrho \in \mathtt{eps}\sigma_0\mathbb{Z}$ is necessary. If Algorithm `Transform` does not stop for $m = 1$, then we showed that no rounding error occurs in the computation of $t^{(1)} = \mathrm{fl}(t^{(0)} + \tau^{(1)})$. Without the assumption on $\varrho = t^{(0)}$ this need not be true since ϱ could be less than $\mathtt{eps}|\tau^{(1)}|$.

The case $s = 0$ is far from being treated optimal. In this case the preliminary version of Algorithm 4.1 *always* iterates until $\sigma_{m-1} \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$, and each time a vector p is extracted which may long consist only of zero components. The case $s = 0$ is not that rare, for example when checking geometrical predicates. We will improve on that later.

Next we will show how to compute a faithfully rounded result. In Figure 4.1 we sketch the possible spread of bits of the individual variables in Algorithm 4.1 (`Transform`) for a final value $m = 2$. In the middle of the figure we define a possible $\tau_1 = t^{(m)}$, since the following quantities τ_2 etc. depend on that. Note this is a picture for small M, i.e. small dimension n. For larger n, up to almost $\sqrt{\frac{1}{2}\mathtt{eps}^{-1}}$, the error δ_3 in the computation of $\tau_3 = \mathrm{fl}(\sum p_i^{(m)})$ can be quite significant, though still just not too large to jeopardize faithful rounding.

LEMMA 4.3. *Let p be a nonzero vector of n floating-point numbers and $\varrho \in \mathbb{F}$. Let* `res` *be computed as follows:*

$$[\tau_1, \tau_2, p', \sigma] = \mathtt{Transform}(p, \varrho)$$
$$\mathtt{res} = \mathrm{fl}(\tau_1 + (\tau_2 + (\textstyle\sum_{i=1}^n p_i')))$$

Assume that $2(n+2)^2\mathtt{eps} \le 1$, and assume $\varrho \in \mathtt{eps}\sigma_0\mathbb{Z}$ is satisfied for $M := \lceil \log_2(n+2) \rceil$, $\mu := \max_i |p_i|$ and $\sigma_0 = 2^{M+\lceil \log_2 \mu \rceil}$.

Then `res` *is a faithful rounding of $\sum_{i=1}^n p_i + \varrho =: s + \varrho$. Moreover,*

$$s + \varrho = \tau_1 + \tau_2 + \sum_{i=1}^n p_i' \quad \text{and} \quad \max |p_i'| \le \mathtt{eps}\sigma \;, \tag{4.7}$$

$$\text{fl}(\tau_1+\tau_2)=\tau_1\ ,\quad \tau_1,\tau_2\in \mathtt{eps}\sigma\mathbb{Z}\quad\text{and}\quad |\tau_2|\le \mathtt{eps}\cdot\text{ufp}(\tau_1)\ , \tag{4.8}$$

$$|s+\varrho-\mathtt{res}|\le 2\mathtt{eps}(1-2^{-M-1})\text{ufp}(\mathtt{res})\ . \tag{4.9}$$

If $\sigma\le\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$*, then all components of the vector* p' *are zero and* $s+\varrho=\tau_1+\tau_2$*.*

If $\mathtt{res}=0$*, then* $s+\varrho=\tau_1=\tau_2=0$ *and all components of the vector* p' *are zero.*

If $\sigma>\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$*, then*

$$\text{ufp}(\tau_1)\ge 2^{2M+1}\mathtt{eps}\sigma\ . \tag{4.10}$$

REMARK 1. A standard application is the case $\varrho=0$ which will be our Algorithm 4.4 (AccSum). In this case, of course, also $\varrho\in\mathtt{eps}\sigma_0\mathbb{Z}$. We stated this piece of code with offset ϱ since we will use it for the calculation of a non-overlapping sequence of numbers representing K-fold accuracy (Algorithm 5.6).

REMARK 2. Besides the main result that res is a faithful rounding, Lemma 4.3 also formulates some properties of the results of Algorithm 4.1 (Transform) independent of the (internal) final value of m.

PROOF OF LEMMA 4.3. For the final value of m in Algorithm 4.1 (Transform), (4.3), (4.2), (2.15) and (2.20) imply (4.8). Moreover, $p'=p^{(m)}$, $\sigma=\sigma_{m-1}$, (4.1), (4.3) and (4.2) yield (4.7).

If $\sigma\le\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ for the final value of m in Algorithm 4.1 (Transform), then (4.7) implies $|p'_i|<\mathtt{eta}$, so all components of the vector p' must be zero. Hence $s+\varrho=\tau_1+\tau_2$, and $\mathtt{res}=\text{fl}(\tau_1+\tau_2)=\text{fl}(s+\varrho)$, which is of course a faithful rounding, and (4.9) follows by (2.20).

Henceforth assume $\sigma>\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$. Then (4.4) implies (4.10), and we have to prove that res is a faithful rounding of the exact result. Our assumptions imply

$$n+2\le 2^M\quad\text{and}\quad 2^{2M+1}\mathtt{eps}\le 1\ . \tag{4.11}$$

Next we abbreviate

$$\begin{array}{rclcl}\tau_3&=&\text{fl}(\sum_{i=1}^n p'_i)&=&\sum_{i=1}^n p'_i-\delta_3\ ,\\ \tau'_2&=&\text{fl}(\tau_2+\tau_3)&=&\tau_2+\tau_3-\delta_2\ ,\\ \mathtt{res}&=&\text{fl}(\tau_1+\tau'_2)&=&\tau_1+\tau'_2-\delta_1\ .\end{array} \tag{4.12}$$

We will use Lemma 2.5 to prove that res is a faithful rounding of s. By (4.7) and (4.12), $s+\varrho=\tau_1+\tau_2+\tau_3+\delta_3=\tau_1+\tau'_2+\delta_2+\delta_3$, so

$$s+\varrho=r+\delta\ \text{ and }\ \mathtt{res}=\text{fl}(r)\quad\text{for}\quad r:=\tau_1+\tau'_2\ \text{ and }\ \delta:=\delta_2+\delta_3\ . \tag{4.13}$$

By (4.7) and (2.16),

$$|\tau_3|=|\text{fl}(\sum_{i=1}^n p'_i)|\le n\mathtt{eps}\sigma\ . \tag{4.14}$$

The standard estimation of floating-point sums [18] gives $|\delta_3|\le\gamma_{n-1}\sum_{i=1}^n|p'_i|$ independent of the order of summation, where $\gamma_k:=k\mathtt{eps}/(1-k\mathtt{eps})$. Note that this is also satisfied in the presence of underflow since summation is exact in that case. A little computation using $n^2\mathtt{eps}<2^{2M}\mathtt{eps}<1$ shows $\gamma_{n-1}<n\mathtt{eps}$, and (4.7) gives

$$|\delta_3|<n^2\mathtt{eps}^2\sigma\ . \tag{4.15}$$

For later use we note that (4.14), (4.11) and (4.10) imply

$$|\tau_3|<2^M\mathtt{eps}\sigma<2^{-M-1}|\tau_1|\ . \tag{4.16}$$

Next (2.20), (4.12), (4.8) and (4.14) yield

$$|\delta_2| \le \mathtt{eps}|\tau_2+\tau_3| \le \mathtt{eps}^2(\mathrm{ufp}(\tau_1)+n\sigma) \tag{4.17}$$

and (4.12), (4.8), (4.14) and (4.17) give

$$\begin{aligned} |\tau_1+\tau_2'| &\ge |\tau_1+\tau_2|-|\tau_2-\tau_2'| = |\tau_1+\tau_2|-|\tau_3-\delta_2| \\ &\ge (1-\mathtt{eps}-\mathtt{eps}^2)|\tau_1|-(1+\mathtt{eps})n\mathtt{eps}\sigma\ , \end{aligned} \tag{4.18}$$

so (2.20) and (4.12) imply

$$|\mathtt{res}| \ge (1-\mathtt{eps})|\tau_1+\tau_2'| > (1-2\mathtt{eps})|\tau_1|-n\mathtt{eps}\sigma\ , \tag{4.19}$$

a lower bound for $|\mathtt{res}|$, also in the presence of underflow.

Now (4.13), (4.17) and (4.15) yield

$$|\delta| = |\delta_2+\delta_3| < \mathtt{eps}^2(\mathrm{ufp}(\tau_1)+n\sigma+n^2\sigma)\ . \tag{4.20}$$

Furthermore (4.20), (4.11), (4.10) and (4.19) imply

$$\begin{aligned} 2\mathtt{eps}^{-1}|\delta| &< 2\mathtt{eps}\cdot\mathrm{ufp}(\tau_1)+n(3+2n)\mathtt{eps}\sigma-n\mathtt{eps}\sigma \\ &< 2\mathtt{eps}\cdot\mathrm{ufp}(\tau_1)+(2^M-2)2^{M+1}\mathtt{eps}\sigma-n\mathtt{eps}\sigma \\ &= 2\mathtt{eps}\cdot\mathrm{ufp}(\tau_1)+(1-2^{-M+1})2^{2M+1}\mathtt{eps}\sigma-n\mathtt{eps}\sigma \\ &\le 2\mathtt{eps}\cdot\mathrm{ufp}(\tau_1)+(1-4\mathtt{eps}-2^{-M})\mathrm{ufp}(\tau_1)-n\mathtt{eps}\sigma \\ &= (1-2\mathtt{eps}-2^{-M})\mathrm{ufp}(\tau_1)-n\mathtt{eps}\sigma \\ &\le (1-2\mathtt{eps})(1-2^{-M})\mathrm{ufp}(\tau_1)-(1-2^{-M})n\mathtt{eps}\sigma \\ &< (1-2^{-M})|\mathtt{res}| < |\mathtt{res}|\ . \end{aligned} \tag{4.21}$$

Using (4.13) and (2.20) we conclude

$$|s+\varrho-\mathtt{res}| \le |\mathrm{fl}(r)-r|+|\delta| \le \mathtt{eps}\cdot\mathrm{ufp}(\mathtt{res})+\frac{1}{2}\mathtt{eps}(1-2^{-M})|\mathtt{res}|\ ,$$

and $|\mathtt{res}| < 2\mathrm{ufp}(\mathtt{res})$ proves (4.9).

If $|\tau_1| \ge \mathtt{eps}^{-1}\mathtt{eta}$, then (4.19), (4.11) and (4.10) yield

$$|\mathtt{res}| > (1-2\mathtt{eps}-2^{-M-1})|\tau_1| > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}\ .$$

That means $\mathtt{res} \notin \mathbb{U}$, and (4.13), (4.21) and Lemma 2.5 show that res is a faithful rounding of the sum $s+\varrho$. This leaves us with the case $|\tau_1| < \mathtt{eps}^{-1}\mathtt{eta}$. In that case (4.17), (4.8) and (4.16) yield

$$|\delta_2| \le \mathtt{eps}|\tau_2+\tau_3| \le \mathtt{eps}^2|\tau_1|+2^{-M-1}\mathtt{eps}|\tau_1| < \frac{1}{2}\mathtt{eps}|\tau_1| < \mathtt{eta}\ , \tag{4.22}$$

and (4.15), (4.10) and (4.11) imply

$$|\delta_3| < n^2\mathtt{eps}\cdot 2^{-2M-1}|\tau_1| < \frac{1}{2}\mathtt{eps}|\tau_1| < \mathtt{eta}\ . \tag{4.23}$$

But δ_2 and δ_3 are the errors of the single floating-point additions to compute τ_2' and τ_3, respectively, and by (2.4) they are floating-point numbers, so by (4.22) and (4.23) both must be zero. Hence (4.13) implies that $\mathtt{res} = \mathrm{fl}(r) = \mathrm{fl}(s+\varrho)$ is a faithful rounding of the exact result $s+\varrho$.

Finally, let $\mathtt{res} = 0$. To establish a contradiction assume $\sigma > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$. The "until"-condition yields $|\tau_1| = |t^{(m)}| \ge 2^{2M+1}\mathtt{eps}\sigma > 0$, and (4.19) and (4.16) imply

$$|\mathtt{res}| > (1-2\mathtt{eps})|\tau_1|-n\mathtt{eps}\sigma > \frac{1}{2}|\tau_1| > 0\ ,$$

a contradiction. Therefore $\sigma \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ and we already proved $p'_i = 0$ for all i. Hence $\tau_1 = \mathrm{fl}(\tau_1 + \tau_2) = \mathrm{fl}(s + \varrho) = 0$ and $\tau_2 = 0$ because $\mathtt{res} = 0$ is a faithful rounding of $s + \varrho$. The lemma is proved. □

A special application of Algorithm 4.1 (`Transform`) with $\varrho \ne 0$ may be the following. Suppose it is known that one component, p_1 say, of a vector p is much larger in magnitude than the others. Then the call $\mathtt{Transform}(p(2:n), p_1)$, in Matlab notation, may reduce the number of loops since then σ_0 only depends on the smaller components $p_2, \cdots, p_n$ and not on p_1. We will use this in Section 5, where we will present Algorithm 5.6 (`AccSumK`) to compute a faithfully rounded result of K-fold accuracy.

We are now in the position to state our first algorithm for computing a faithfully rounded result of the sum of a vector of floating-point numbers.

Algorithm 4.4. *Accurate summation with faithful rounding.*

$$
\begin{array}{l}
\text{function } \mathtt{res} = \mathtt{AccSum}(p) \\
\quad [\tau_1, \tau_2, p'] = \mathtt{Transform}(p, 0) \\
\quad \mathtt{res} = \mathrm{fl}(\tau_1 + (\tau_2 + (\sum_{i=1}^{n} p'_i)))
\end{array}
$$

Proposition 4.5. *Let p be a vector of n floating-point numbers and assume $2(n+2)^2\mathtt{eps} \le 1$. Let* `res` *be the result of Algorithm 4.4 (*`AccSum`*) applied to p. Then* `res` *is a faithful rounding of $s := \sum_{i=1}^{n} p_i$.*

Remark. Algorithm 4.4 (`AccSum`) is identical to the piece of code we analyzed in Lemma 4.3, only the offset ϱ is set to zero and the output parameter σ in `Transform`, which is unnecessary here, is omitted.

Proof. For zero input vector p, Algorithm 4.1 (`Transform`) implies $\tau_1 = \tau_2 = p'_i = \mathtt{res} = 0$ for all i. For nonzero input vector p, the assumptions of Lemma 4.3 are satisfied, and the assertion follows. □

Lemma 2.6 implies the following corollary.

Corollary 4.6. *Under the assumption of Proposition 4.5 the computed result* `res` *of Algorithm 4.4 (*`AccSum`*) is equal to the exact result $s = \sum p_i$ if s is a floating-point number, or if $\mathtt{res} \in \mathbb{U}$. Moreover,*

$$|s| \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta} \quad \Rightarrow \quad \mathtt{res} \in \mathbb{U} \quad \Rightarrow \quad \mathtt{res} = s \ .$$

Therefore especially $\mathtt{res} = 0$ if and only if $s = 0$, and

$$\mathrm{sign}(\mathtt{res}) = \mathrm{sign}(s) \ .$$

Remark. The exact determination of the sign of a sum by Algorithm 4.4 is critical in the evaluation of geometrical predicates [19, 2, 43, 7, 27, 6, 12]. Rewriting a dot product as a sum by Algorithm 1.1 (`TwoProduct`), we can determine the exact sign of a dot product as well, which in turn decides whether a point is exactly on or on which side of a plane it is.

We showed that the result of Algorithm 4.4 is *always* a faithful rounding of the exact sum. Computational evidence suggests that the cases, where the result of `AccSum` is not rounded to nearest, are very rare. In several billion tests we never encountered such a case. This behavior can be explained as follows.

By (4.12) the error $s + \varrho - \mathtt{res}$ computes to $\delta_1 + \delta_2 + \delta_3$, comprising of the rounding δ_1 of `res` plus the additional error term $\delta = \delta_2 + \delta_3$. So the question is, when is δ large enough to change the rounding of `res` so that it is not to nearest (but still faithful). From (4.20) we know that $|\delta|$ is not larger than $n^2\mathtt{eps}^2|\tau_1|$ plus some very small term, and only for huge values of n we may expect δ to have an impact on the rounding of `res`.

For example, for $n = 1000$ that means that the true value of δ_1 must be as close as $10^6\texttt{eps} \sim 10^{-10}$ to the switching point for rounding. One may expect that this happens once in some 10 billion cases.

However, we can construct examples with faithful but not rounding to nearest. Consider $p = [1\ \texttt{eps}\ \texttt{eps}^2]$. Then $\texttt{AccSum}(p)$ produces $\tau_1 = 1$, $\tau_2 = 0$ and $p' = [0\ \texttt{eps}\ \texttt{eps}^2]$, and $\texttt{res} = 1$. This is because IEEE 754 rounds tie to even, so $\text{fl}(1 + \texttt{eps}) = 1$.

Changing the strict into an "almost always" rounded to nearest offers quite a reward, namely the computational effort of Algorithm 4.4 (AccSum) depends solely on the condition number: only the more difficult the problem, the more time must be spent. The maximum number of iterations m can be estimated as follows. The condition number of summation for $\sum p_i \neq 0$ is defined [18] by

$$\text{cond}\left(\sum p_i\right) := \lim_{\varepsilon \to 0} \sup \left\{ \left| \frac{\sum \widetilde{p}_i - \sum p_i}{\varepsilon \sum p_i} \right| : \ |\widetilde{p}| \leq \varepsilon |p| \right\},$$

where absolute value and comparison of vectors is to be understood componentwise. Obviously

$$\text{cond}\left(\sum p_i\right) = \frac{\sum |p_i|}{|\sum p_i|} \ . \tag{4.24}$$

The following theorem estimates the maximal number m of iterations needed in AccSum depending on the number of elements n and the condition number.

THEOREM 4.7. *Assume Algorithm 4.4* (AccSum) *is applied to a vector of floating-point numbers* $p_i, 1 \leq i \leq n$, *with nonzero sum* s *and* $2(n+2)^2\texttt{eps} \leq 1$. *Then the following is true. If*

$$\text{cond}\left(\sum p_i\right) \leq 2^{-2M-2}(1 - 2^{-M}) \left[2^{-M}\texttt{eps}^{-1}\right]^m \ , \tag{4.25}$$

*then Algorithm 4.1 (*Transform*) called by Algorithm 4.4* (AccSum) *stops after at most* m *executions of the "repeat-until"-loop. If the "repeat-until"-loop in Algorithm 4.1 (*Transform*) has been executed at least* $m \geq 2$ *times, then*

$$\text{cond}\left(\sum p_i\right) > 2^{-2M-2}(1 - 2^{-M}) \left[2^{-M}\texttt{eps}^{-1}\right]^{m-1} \ . \tag{4.26}$$

If the repeat-until loop is executed m *times and absolute value and comparison is counted as one flop, then Algorithm 4.4 (*AccSum*) needs* $(4m+3)n + \mathcal{O}(1)$ *flops.*

PROOF. For the called Algorithm 4.1 (Transform) we have $p_i^{(0)} := p_i$, and its initialization implies

$$n + 2 \leq 2^M, \ \ \max_i |p_i| = \mu \ , \ \ \frac{1}{2}\sigma_0 < 2^M\mu \leq \sigma_0 \quad \text{and} \quad \sigma_k = \varphi^k \sigma_0 \tag{4.27}$$

for $\varphi := 2^M\texttt{eps}$ and $0 \leq k \leq m$.

To establish a contradiction assume Algorithm 4.1 is not finished after m executions of the "repeat-until"-loop. Then the "until"-condition, which is not satisfied for the value m, implies $\sigma_{m-1} > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ so that $\text{fl}(2^{2M+1}\texttt{eps}\sigma_{m-1}) = 2^{2M+1}\texttt{eps}\sigma_{m-1}$, and

$$|t^{(m)}| < 2^{2M+1}\texttt{eps}\sigma_{m-1} = 2^{M+1}\sigma_m \ . \tag{4.28}$$

Since the "repeat-until"-loop is to be executed again, we conclude

$$s = t^{(m)} + \sum p_i^{(m)} \tag{4.29}$$

as in (4.6) in the proof of Lemma 4.2. Denote the condition number by C. Then (4.24) and (4.27) yield $|s| = C^{-1} \sum |p_i| \geq \mu C^{-1}$. Combining this with (4.29), (4.2), (4.27), (4.25) and using $(1 - 2^{-M})^{-1} > 1 + 2^{-M-1}$

Table 4.1
Minimum treatable condition numbers by Algorithm 4.4

n	$m=1$	$m=2$	$m=3$	$m=4$	$m=5$
100	$1.1 \cdot 10^{9}$	$7.5 \cdot 10^{22}$	$5.3 \cdot 10^{36}$	$3.7 \cdot 10^{50}$	$2.6 \cdot 10^{64}$
1000	$2.1 \cdot 10^{6}$	$1.8 \cdot 10^{19}$	$1.6 \cdot 10^{32}$	$1.4 \cdot 10^{45}$	$1.2 \cdot 10^{58}$
10000	$5.1 \cdot 10^{2}$	$2.8 \cdot 10^{14}$	$1.5 \cdot 10^{26}$	$8.5 \cdot 10^{37}$	$4.6 \cdot 10^{49}$
10^5		$6.8 \cdot 10^{10}$	$4.7 \cdot 10^{21}$	$3.2 \cdot 10^{32}$	$2.2 \cdot 10^{43}$
10^6		$1.6 \cdot 10^{7}$	$1.4 \cdot 10^{17}$	$1.2 \cdot 10^{27}$	$1.0 \cdot 10^{37}$
10^7		$2.5 \cdot 10^{2}$	$1.2 \cdot 10^{11}$	$7.3 \cdot 10^{19}$	$3.9 \cdot 10^{28}$

Table 4.2
Minimum treatable length n for condition number $\texttt{eps}^{-1}$

$m=2$	$m=3$	$m=4$
4096	$1.0 \cdot 10^{6}$	$6.7 \cdot 10^{7}$

implies

$$\begin{aligned} |t^{(m)}| &\geq |s| - |\sum p_i^{(m)}| \geq \mu C^{-1} - 2^M \texttt{eps}\sigma_{m-1} \\ &= \mu C^{-1} - \varphi^m \sigma_0 > (2^{-M-1}C^{-1} - \varphi^m)\sigma_0 \\ &\geq \left(2^{-M-1}2^{2M+2}(1+2^{-M-1}) - 1\right)\varphi^m\sigma_0 = 2^{M+1}\varphi^m\sigma_0 \\ &= 2^{M+1}\sigma_m \ , \end{aligned}$$

a contradiction to (4.28). This proves our first assertion.

Next assume the "repeat-until"-loop in Algorithm 4.1 (`Transform`) has been executed at least $m \geq 2$ times. Then the "until"-condition was not satisfied for $m-1$, so that $|t^{(m-1)}| < 2^{2M+1}\texttt{eps}\sigma_{m-2} = 2^{M+1}\sigma_{m-1}$. Hence (4.1), (4.2), (4.27) and $1 + 2^{-M-1} + \frac{1}{2}\texttt{eps} < (1-2^{-M})^{-1}$ imply

$$|s| = |t^{(m-1)} + \tau^{(m)} + \sum_{i=1}^{n} p_i^{(m)}| < \left(2^{M+1} + 1 + 2^M\texttt{eps}\right)\sigma_{m-1} < 2^{M+1}(1-2^{-M})^{-1}\sigma_{m-1} \ .$$

Hence (4.24) and (4.27) yield

$$C \geq \frac{\mu}{|s|} > \frac{2^{-M-1}\sigma_0}{2^{M+1}(1-2^{-M})^{-1}\sigma_{m-1}} = 2^{-2M-2}(1-2^{-M})\varphi^{-m+1} \ .$$

This proves the second assertion. Up to order 1, the calculation of μ requires $2n$ flops, `ExtractVector` requires $4n$ and the computation of `res` requires n flops. □

For IEEE 754 double precision, Theorem 4.7 basically means that at least for condition numbers up to

$$\mathrm{cond}\left(\sum p_i\right) \lesssim 2^{m(53-M)-2M-2}$$

Algorithm 4.4 (`AccSum`) computes a faithfully rounded result in at most m executions of the repeat-until loop. In Table 4.1 we show the lower bound by Theorem 4.7 for the condition number which can be treated for different values of n and m, where treatable means to produce a faithfully rounded result.

Conversely, we can use Theorem 4.7 to compute for given m and condition number $\texttt{eps}^{-1}$ the minimum length n of a vector for which a faithfully rounded result of its sum is computed. The value $\texttt{eps}^{-1}$ is the condition number for which we cannot expect a single correct digit by traditional recursive summation.

The value $6.7 \cdot 10^7$ in Table 4.2 corresponds to the maximum value of n satisfying $2(n+2)^2\texttt{eps} \leq 1$. This means for condition number up to $\texttt{eps}^{-1}$ never more than $m = 4$ iterations are needed. Note that the lower

Table 4.3
Floating-point operations needed for different dimension and condition number

	AccSum	Sum2	XBLAS
$n = 1000$, cond $= 10^6$	$7n$	$7n$	$10n$
$n = 1000$, cond $= 10^{16}$	$10n$	$7n$	$10n$
$n = 10^6$, cond $= 10^{16}$	$13n$	$7n$	$10n$

and upper bound for the condition number in Theorem 4.7 differ by a factor $2^M \mathtt{eps}$, just the shift for σ in `Transform` . Algorithms for even larger values $n > 6.7 \cdot 10^7$ will be presented in Section 7.

Our computing times compare favorably to other algorithms. Consider Algorithms 4.4 (`Sum2`) in [36] and the XBLAS summation algorithm `BLAS_dsum_x` in [31]. The results of both algorithms are of the same quality, namely *as if* computed in 2-fold precision. That means, for condition numbers up to $\mathtt{eps}^{-1}$ we can expect a result accurate to the last bit. In Table 4.3 the required floating point operations are displayed for different vector lengths and condition numbers. Note that Algorithm 4.4 (`AccSum`) always computes a faithfully rounded result, independent of the condition number.

There is another big advantage of our Algorithm 4.4 (`AccSum`) compared to `Sum2` and XBLAS summation. Both `Sum2` and XBLAS summation pass through the input vector using the error-free transformation `TwoSum` (Algorithm 2.1), so that step $i + 1$ of the transformation depends on step i. Hence parallelization of the algorithms is difficult. In contrast, the inner loop of `AccSum` can be perfectly executed in parallel (see also the Matlab code Algorithm 10.1 in the Appendix).

Next we take extra care about zero sums. The algorithm for provably computing a rounding to nearest result of the sum must be postponed to Section 6 because we need results from the following Section 5 on algorithms with K-fold accuracy.

We can significantly improve Algorithm 4.1 (`Transform`) by checking for $t^{(m)} = 0$. In that case $\mathrm{fl}(t^{(m-1)} + \tau^{(m)}) = 0 = t^{(m-1)} + \tau^{(m)}$ because addition in the underflow range is exact, so (4.1) implies $s + \varrho = \sum_{i=1}^{n} p_i^{(m)}$. Hence, we can apply Algorithm 4.1 recursively to $p_i^{(m)}$.

This has two advantages. First, if $s = 0$, then $p_i^{(m)}$ consists only of zeros at a certain stage. With checking for $t^{(m)} = 0$ this is determined immediately in the recursive call of `Transform` by $\mu = 0$. Without this check, the "while"-loop would continue until σ_{m-1} is decreased into the underflow range, each time processing the whole vector $p^{(m-1)}$ which may long be zero. Second, the input vector $p_i^{(0)}$ might consist of some components large in absolute value which cancel exactly to zero, and some remaining small components. Then the latter form the new input vector $p_i^{(m)}$, and the recursive call skips the gap in one step.

The following modification of Algorithm 4.1 (`Transform`) includes this check for zero. If $t^{(m)} = 0$ it may be advantageous to eliminate zero components in the vector $p_i^{(m)}$. This is done in the executable Matlab code given in the Appendix.

ALGORITHM 4.8. *Modified Algorithm 4.1 (*`Transform`*) with check for zero.*

function $[\tau_1, \tau_2, p, \sigma, M] = \texttt{Transform}(p, \varrho)$
  $\mu = \max(|p_i|)$
  if $\mu = 0$, $\tau_1 = \varrho$, $\tau_2 = 0$, return, end if
  $M = \left\lceil \log_2 \left(\text{length}(p^{(0)}) + 2\right)\right\rceil$
  $\sigma' = 2^{M+\log_2(\mu)}$
  $t' = \varrho$
  repeat
    $t = t'$; $\sigma = \sigma'$
    $[\tau, p] = \texttt{ExtractVector}(\sigma, p)$
    $t' = \text{fl}(t + \tau)$
    if $t' = 0$, $[\tau_1, \tau_2, p] = \texttt{Transform}(p, 0)$; return; end if
    $\sigma' = \text{fl}(2^M \texttt{eps}\sigma)$
  until $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ or $|t'| \ge \text{fl}(2^{2M+1}\texttt{eps}\sigma)$
  $[\tau_1, \tau_2] = \texttt{FastTwoSum}(t, \tau)$

Compared to Algorithm 4.1, the modified Algorithm 4.8 omits indices and adds M as output, the latter being used in Section 7 on huge n. Note that for clarity we still use the logarithm rather than Algorithm 3.5 (`NextPowerTwo`), but replace this in the Matlab code given in the Appendix.

Otherwise, both algorithms are identical except the check for $t' = 0$. Therefore all assertions on Algorithm 4.1 given in Lemma 4.2, 4.3, etc. are true for Algorithm 4.8 as well. Hence we may safely use the same name `Transform` for both algorithms. Note that the results might in fact be different because the shift σ may change, however, the *properties* such as the faithful rounding of the result etc. remain unchanged.

The algorithm can still be slightly improved. For σ just before entering the underflow range one call to `ExtractVector` may be saved depending on t'.

Next we prepare for the computation of a result with faithfully rounded K-fold accuracy, stored in a result vector of length K. The aim is to assure that the result vector is a non-overlapping sequence of numbers.

5. K-fold accuracy. Sometimes the precision of the given floating-point format may not be sufficient. One possibility for improvement is the use of multiple precision arithmetic such as [3, 5, 13, 1]. Frequently such packages support a special long precision data type.

An alternative may be to put together a long number by pieces of floating-point numbers, for example XBLAS uses this to simulate quadruple precision [31]. For more than two pieces this technique was used, for example, in [41] to compute highly accurate inclusions of the solution of systems of linear equations using a so-called long accumulator, which was developed in the 1970's in Karlsruhe for accurate computation of dot products. To gain as much accuracy as possible, the pieces should be non-overlapping, i.e. their bit representations should have no bit in common.

DEFINITION 5.1. *Two floating-point numbers f_1, f_2 with $|f_1| \ge |f_2|$ are called non-overlapping if*

$$\text{ufp}(f_2) < 2\texttt{eps} \cdot \text{ufp}(f_1) \ . \tag{5.1}$$

A sequence of $k \ge 1$ floating-point numbers $f_1, \cdots, f_k$ is called non-overlapping, if $|f_1| \ge |f_2| \ge \cdots \ge |f_{k'}| \ge 0 = f_{k'+1} = \cdots = f_k$ and f_i, f_{i+1} is non-overlapping for $1 \le i < k'$. This includes a sequence of zeros.

Non-overlapping sequences have been used for example by Priest [38, 39] in the realm of geometrical predicates. Shewchuck [44] observed that enforcing non-overlapping may require substantial effort. Therefore he

defined and used weak non-overlapping, which means that bits of f_2 may overlap zero bits "at the end" of f_1.

Though more efficiently computable than Priest's, weakly non-overlapping sequences may comprise (and do) of f_i consisting only of one single bit, the leading one. Thus we cannot expect a K-fold accuracy for a sequence of length K. This changes for (strongly) non-overlapping sequences as demonstrated in the following lemma.

LEMMA 5.2. *Let $f_1, \cdots, f_k$ be a non-overlapping sequence of floating-point numbers. Then*

$$\mathrm{ufp}(f_k) \le \mathtt{eps}^{k-1} \cdot \mathrm{ufp}(f_1) \ . \tag{5.2}$$

Moreover,

$$\mathrm{ufp}(\sum_{i=1}^{k} f_i) \le \mathrm{ufp}(f_1) \ . \tag{5.3}$$

REMARK. Note that Lemma 5.2 is applicable to any subsequence $f_{i_1}, \cdots, f_{i_m}$ with $i_1 > \cdots > i_m$ because it is non-overlapping as well.

PROOF. Let f, g be a non-overlapping sequence. Then Definition 5.1 implies

$$\mathrm{ufp}(g) \le \mathtt{eps} \cdot \mathrm{ufp}(f) \ , \tag{5.4}$$

and (5.2) follows. To prove (5.3) we first show

$$|f| + 2\mathrm{ufp}(g) \le 2\mathrm{ufp}(f) \tag{5.5}$$

If $f \in \mathbb{U}$, then (2.1) and (5.2) imply $\mathrm{ufp}(g) \le \frac{1}{2}\mathtt{eta}$ and therefore $g = 0$. If $f \notin \mathbb{U}$, then $|f| \le \mathrm{pred}(2\mathrm{ufp}(f)) = 2(1 - \mathtt{eps})\mathrm{ufp}(f)$, and (5.4) implies (5.5). Without loss of generality we can omit remainder zeros in the sequence $f_1, \cdots, f_k$ and assume $f_k \ne 0$. Then

$$|\sum_{i=1}^{k} f_i| < \sum_{i=1}^{k-1} |f_i| + 2\mathrm{ufp}(f_k) \le \sum_{i=1}^{k-2} |f_i| + 2\mathrm{ufp}(f_{k-1}) \le \cdots \le 2\mathrm{ufp}(f_1) \ ,$$

and (5.3) follows. □

We are aiming on non-overlapping sequences in the sense of Definition 5.1. Note that this is a little more than a faithfully rounded result in K-fold precision because there may be "gaps" between adjacent members of a sequence.

Suppose $f_1 \in \mathbb{F}$ is a faithful rounding of $r \in \mathbb{R}$, and let $f_2 \in \mathbb{F}$ be a faithful rounding of $r - f_1$. Only from the definition of faithful rounding this is not enough to ensure non-overlapping of f_1 and f_2. Consider $r = 1 - \mathtt{eps}^3$. Then

$$f_1 := 1 - \mathtt{eps} = \mathrm{pred}(1) \in \square(r) \quad \text{and} \quad f_2 := \mathtt{eps} \in \square(\mathtt{eps} - \mathtt{eps}^3) \ ,$$

but f_2 is the last bit of f_1. Fortunately, our Algorithm 4.4 (`AccSum`) produces a faithfully rounded result good enough to guarantee non-overlapping.

LEMMA 5.3. *Let the assumptions of Lemma 4.3 be valid so that some* $\mathtt{res} \in \mathbb{F}$ *has been computed with the code given over there and* $\mathtt{res} \in \square(s + \varrho)$. *Furthermore, suppose* $f_2 \in \square(s + \varrho - \mathtt{res})$ *for some* $f_2 \in \mathbb{F}$. *Then* $\mathtt{res}, f_2$ *are non-overlapping.*

PROOF. If $\mathtt{res} \in \mathbb{U}$, then Lemma 2.6 implies $\mathtt{res} = s + \varrho$ and $f_2 = 0$. Therefore, we may assume $\mathtt{res} \notin \mathbb{U}$, so that $f_2 \in \square(s + \varrho - \mathtt{res})$, (5.1), (2.31) and (4.9) give

$$\begin{aligned}|f_2| &\le |s + \varrho - \mathtt{res}| + |s + \varrho - \mathtt{res} - f_2| \le |s + \varrho - \mathtt{res}| + 2\mathtt{eps} \cdot \mathrm{ufp}(s + \varrho - \mathtt{res}) \\ &\le (1 + 4\mathtt{eps})|s + \varrho - \mathtt{res}| \le 2\mathtt{eps}(1 + 4\mathtt{eps})(1 - 2^{-M-1})\mathrm{ufp}(\mathtt{res}) \\ &< 2\mathtt{eps} \cdot \mathrm{ufp}(\mathtt{res}) ,\end{aligned}$$

proving by (5.1) that $\mathtt{res}$ and f_2 are non-overlapping. □

When applying the code given in Lemma 4.3 to a vector p and $\varrho \in \mathbb{F}$, it calculates a faithful rounding $\mathtt{res}$ of $s + \varrho$, and also extracts p into some vector p'. But we have no *equation* relating s, ϱ, $\mathtt{res}$ and p', but only an *estimation* following by $\mathtt{res} \in \square(s + \varrho)$. The equation we know is $s + \varrho = \tau_1 + \tau_2 + \sum p'_i$ as by (4.7).

However, in order to continue applying Lemma 5.3 to produce a non-overlapping sequence, we need a faithful rounding of $s + \varrho - \mathtt{res}$. Computing this by applying the code given in Lemma 4.3 to ϱ and the vector p appended by $-\mathtt{res}$ would extract the entire vector p again and be highly ineffective. Fortunately, the following Algorithm 5.4 (TransformK) gives an effective way to compute a single floating-point number R relating s, ϱ, $\mathtt{res}$ and p' by an equation. It paves the way to compute a non-overlapping sequence of K numbers $f_1, \cdots, f_K$ approximating s, thus establishing an approximation of K-fold accuracy.

ALGORITHM 5.4. *Error-free vector transformation including faithful rounding.*

$$\begin{array}{l}\text{function } [\mathtt{res}, R, p'] = \mathtt{TransformK}(p, \varrho) \\ \quad [\tau_1, \tau_2, p', \sigma] = \mathtt{Transform}(p, \varrho) \\ \quad \mathtt{res} = \mathrm{fl}(\tau_1 + (\tau_2 + (\sum_{i=1}^{n} p'_i))) \\ \quad R = \mathrm{fl}(\tau_2 - (\mathtt{res} - \tau_1))\end{array}$$

LEMMA 5.5. *Let p be a nonzero vector of n floating-point numbers and let $\varrho \in \mathbb{F}$. Abbreviate $s := \sum p_i$. Assume that $2(n+2)^2\mathtt{eps} \le 1$, and assume $\varrho \in \mathtt{eps}\sigma_0\mathbb{Z}$ is satisfied for $M := \lceil \log_2(n+2) \rceil$, $\mu := \max_i |p_i|$ and $\sigma_0 = 2^{M + \lceil \log_2 \mu \rceil}$. Let $\mathtt{res}$, R and p' be the results of Algorithm 5.4 (*TransformK*) applied to p and ϱ. Then $\mathtt{res}$ is a faithful rounding of $s + \varrho$ and*

$$s + \varrho - \mathtt{res} = R + \sum_{i=1}^{n} p'_i \ . \tag{5.6}$$

If $\mathtt{res} = 0$, then $\mathtt{res} = s + \varrho = 0$ and R and all p'_i are zero. If the vector p' is nonzero, then

$$R \in \mathtt{eps}\sigma'\mathbb{Z} \tag{5.7}$$

is satisfied for $\mu' := \max_i |p'_i|$ and $\sigma' := 2^{M + \lceil \log_2 \mu' \rceil}$.

REMARK 1. The main part of the proof of Lemma 5.5 will be to show that no rounding error can occur in the computation of R, that is $R = \tau_2 - (\mathtt{res} - \tau_1)$. This proof is quite involved and moved to the Appendix.

REMARK 2. Algorithm TransformK replaces the pair τ_1, τ_2 without error into the pair $\mathtt{res}, R$. Therefore it is suitable for cascading, see Algorithm 5.6 (AccSumK).

REMARK 3. As is Matlab convention, an output parameter, in this case M for TransformK, has been omitted.

PROOF OF LEMMA 5.5. Algorithm 5.4 (TransformK) uses the same piece of code as the one in Lemma 4.3 for the computation of $\tau_1, \tau_2, p', \sigma$ and $\mathtt{res}$, so the assertions of Lemma 4.3 are valid. If $\mathtt{res} = 0$, then $s + \varrho = 0$ since $\mathtt{res}$ is a faithful rounding of $s + \varrho$, and all p'_i are zero. Moreover, $0 = s + \varrho = \tau_1 + \tau_2$ and $\tau_1 = \mathrm{fl}(\tau_1 + \tau_2) = 0 = \tau_2$, so $R = 0$ as well. It remains to prove (5.6) and (5.7).

First assume $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$. Then $p'_i = 0$ for all i by Lemma 4.3, and (4.7) and (4.8) imply $s + \varrho = \tau_1 + \tau_2$ and $\texttt{res} = \text{fl}(\tau_1 + \tau_2) = \tau_1$, so $R = \tau_2$ and (5.6) follows. The case $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ is finished.

Henceforth we can assume $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$. We can use the analysis of Algorithm 4.1 (`Transform`) and the notation in the proof of Lemma 4.3, especially (4.12). Then by (2.20), (4.8) and (4.16) we have

$$|\tau'_2| = |\text{fl}(\tau_2 + \tau_3)| \le (1 + \texttt{eps})|\tau_2 + \tau_3| < (1 + \texttt{eps})(\texttt{eps} + 2^{-M-1})|\tau_1| < |\tau_1| \ , \tag{5.8}$$

so $\texttt{res} = \text{fl}(\tau_1 + \tau'_2)$ and Lemma 2.8 imply

$$\Delta := \texttt{res} - \tau_1 = \text{fl}(\texttt{res} - \tau_1) \in \mathbb{F} \ . \tag{5.9}$$

Next we have to show

$$R = \text{fl}(\tau_2 - \Delta) = \tau_2 - \Delta \tag{5.10}$$

provided $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$. This proof is quite involved and left to the Appendix. Now (5.9) and (5.10) yield $R = \tau_1 + \tau_2 - \texttt{res}$, and (4.7) gives

$$s + \varrho - \texttt{res} = \tau_1 + \tau_2 + \sum_{i=1}^{n} p'_i - \texttt{res} = R + \sum_{i=1}^{n} p'_i \ .$$

This proves (5.6). To see (5.7) note that (2.17), (4.10) and (2.11) imply $\Delta = \texttt{res} - \tau_1 \in \texttt{eps} \cdot \text{ufp}(\tau_1) \subseteq 2^{2M+1}\texttt{eps}^2\sigma\mathbb{Z}$. Furthermore, (4.8) and (4.11) yield $\tau_2 \in \texttt{eps}\sigma\mathbb{Z} \subseteq 2^{2M+1}\texttt{eps}^2\sigma\mathbb{Z}$, and (2.15) gives

$$R = \tau_2 - \Delta \in 2^{2M+1}\texttt{eps}^2\sigma\mathbb{Z} \ .$$

But (4.7) and the definition of σ' imply $\texttt{eps}\sigma' \le 2^M\texttt{eps}^2\sigma$, and (5.7) follows by (2.11). □

Next we can formulate an algorithm to compute a result of K-fold accuracy, stored in a non-overlapping result vector `Res` of length K.

Algorithm 5.6. *Accurate summation with faithful rounding and result of K-fold accuracy.*

```
function Res = AccSumK(p, K)
    p^(0) = p,  R_0 = 0
    for k = 1 : K
        [Res_k, R_k, p^(k)] = TransformK(p^(k-1), R_{k-1})
        if Res_k ∈ U, return; end if
    end for
```

Proposition 5.7. *Let p be a vector of n floating-point numbers, abbreviate $s := \sum p_i$ and assume $2(n+2)^2\texttt{eps} \le 1$. Let $1 \le K \in \mathbb{N}$ be given, and let `Res` be the vector of results of Algorithm 5.6 (`AccSumK`) applied to p and K. Denote the final value of k by K'. Then*

$$s = \sum_{\nu=1}^{k} \texttt{Res}_\nu + R_k + \sum_{i=1}^{n} p_i^{(k)} \quad \textit{for } 1 \le k \le K' \tag{5.11}$$

and

$$\texttt{Res}_k \quad \textit{is a faithful rounding of} \quad s - \sum_{\nu=1}^{k-1} \texttt{Res}_\nu \quad \textit{for } 1 \le k \le K' \ , \tag{5.12}$$

so that

$$\text{pred}(\texttt{Res}_k) < s - \sum_{\nu=1}^{k-1} \texttt{Res}_\nu < \text{succ}(\texttt{Res}_k) \quad \textit{for } 1 \le k \le K' \ . \tag{5.13}$$

Moreover, $\texttt{Res}_1, \cdots, \texttt{Res}_{K'}$ *is a non-overlapping sequence of floating-point numbers, i.e. no bits in the binary representation of* $\texttt{Res}_{k-1}$ *and* $\texttt{Res}_k$ *overlap for* $k = 2, \cdots, K'$.

Abbreviate $\hat{\texttt{Res}} := \sum_{k=1}^{K'} \texttt{Res}_k$. *Then* $s \in \mathbb{U}$ *implies* $K' = 1$ *and* $\hat{\texttt{Res}} = s$, *and* $\texttt{Res}_{K'} \in \mathbb{U}$ *implies* $\hat{\texttt{Res}} = s$. *If* $\texttt{Res}_{K'} \notin \mathbb{U}$, *then* $K' = K$ *and the following error estimates are satisfied:*

$$|s - \hat{\texttt{Res}}| < \frac{2}{1 - \texttt{eps}} \texttt{eps}^K |s| \tag{5.14}$$

and

$$|s - \hat{\texttt{Res}}| < 2\texttt{eps}^K \text{ufp}(\texttt{Res}_1) \le 2\texttt{eps}^K |\texttt{Res}_1| \ . \tag{5.15}$$

Remark 1. In fact, the sum $\sum \texttt{Res}_k$ is frequently of better accuracy than K-fold since zero bits in the binary representation of s "between" $\texttt{Res}_{k-1}$ and $\texttt{Res}_k$ are not stored.

Remark 2. For $k \ge 3$, say, it may be advantageous to eliminate zero summands in the vectors $p^{(k)}$.

Remark 3. Also note that the limited exponent range poses no problem to achieve K-fold accuracy. Since the exact result s is a sum of floating-point numbers, it follows $s \in \texttt{eta}\mathbb{Z}$, and also $s - \sum f_k \in \texttt{eta}\mathbb{Z}$ for arbitrary floating-point numbers f_k. So for a certain K the exact sum s is stored in $\sum \texttt{Res}_k$.

Proof of Proposition 5.7. For zero input vector p the assertions are evident; henceforth we assume p to be nonzero.

For $k = 0$ the assumptions of Lemma 5.5 are satisfied, so $\texttt{Res}_1$ is a faithful rounding of $s = \sum p_i^{(0)}$. Moreover, by (5.7) the assumptions of Lemma 5.5 are satisfied for $k \ge 1$ as well, which proves (5.12) and (5.13). By construction and Lemma 5.3, the sequence $\texttt{Res}_1, \cdots, \texttt{Res}_K$ is non-overlapping.

Abbreviate $s_k := \sum p_i^{(k)}$. Then Lemma 5.5 and (5.6) imply

$$\begin{aligned} s = s_0 &= \texttt{Res}_1 + R_1 + s_1 \\ &= \texttt{Res}_1 + \texttt{Res}_2 + R_2 + s_2 \\ &= \cdots \\ &= \textstyle\sum_{k=1}^{K'} \texttt{Res}_k + R_{K'} + s_{K'} \ , \end{aligned}$$

which is (5.11). If $s \in \mathbb{U}$, then (5.12) and (2.30) imply $\hat{\texttt{Res}} = \texttt{Res}_1 = s$. If Algorithm 5.6 (`AccSumK`) ends with $\texttt{Res}_{K'} \in \mathbb{U}$, then (2.30) yields $\texttt{Res}_{K'} = R_{K'-1} + s_{K'-1}$, so $s = \sum_{k=1}^{K'} \texttt{Res}_k = \hat{\texttt{Res}}$.

Assume $\texttt{Res}_{K'} \notin \mathbb{U}$. Then of course $K' = K$, and (5.12), (2.31) and (5.2) give

$$\text{ufp}(s - \sum_{k=1}^{K} \texttt{Res}_k) < 2\texttt{eps} \cdot \text{ufp}(\texttt{Res}_k) \le 2\texttt{eps}^K \text{ufp}(\texttt{Res}_1) \ , \tag{5.16}$$

proving (5.15). If $\text{ufp}(\texttt{Res}_1) \le \text{ufp}(s)$, then (5.14) follows as well. Otherwise, $\texttt{Res}_1 \in \square(s)$ implies $\text{pred}(\texttt{Res}_1) < s < \text{succ}(\texttt{Res}_1)$, so that $\text{ufp}(s) < \text{ufp}(\texttt{Res}_1)$ is only possible for $|\texttt{Res}_1| = \text{ufp}(\texttt{Res}_1)$. In that case

$$(1 - \texttt{eps})|\texttt{Res}_1| = \text{pred}(\texttt{Res}_1) < s < |\texttt{Res}_1| \ ,$$

so that $\text{ufp}(\texttt{Res}_1) = |\texttt{Res}_1| < (1 - \texttt{eps})^{-1}|s|$. The proof is finished. $\square$

Now it becomes clear why so much effort was spent (see the Appendix) to prove (5.6) and (5.7): It is the key to cascade `TransformK` in Algorithm `AccSumK`. Especially for large gaps in the input vector, R_{k-1} in `AccSumK` may be large compared to $p^{(k-1)}$. In that case many case distinctions are necessary in the proof in the Appendix to assure that no rounding error can occur in the computation of R_k in `TransformK`.

6. Rounding to nearest. Now we have also the tools to design an algorithm computing the rounded to nearest result of the sum of a vector of floating-point numbers in the sense of IEEE 754.

ALGORITHM 6.1. *Accurate summation with rounding to nearest.*

function $\texttt{resN} = \texttt{NearSum}(p)$
  $[\texttt{res}, R, p'] = \texttt{TransformK}(p, 0)$
  $[\delta', R', p''] = \texttt{TransformK}(p', R)$
  if $\delta' = 0$, $\texttt{resN} = \texttt{res}$, return, end if
  $\texttt{res}' = \mathrm{fl}(\texttt{res} + \mathrm{sign}(\delta')\texttt{eps}|\texttt{res}|)$
  if $\texttt{res}' = \texttt{res}$
    $\mu = \mathrm{sign}(\delta')\texttt{eps}|\texttt{res}|$; $\texttt{res}' = \mathrm{fl}(\texttt{res} + 2\mathrm{sign}(\delta')\texttt{eps}|\texttt{res}|)$
  else
    $\mu = \mathrm{fl}((\texttt{res}' - \texttt{res})/2)$
  end if
  if $|\delta'| < |\mu|$, $\texttt{resN} = \texttt{res}$, return, end if
  if $|\delta'| > |\mu|$, $\texttt{resN} = \texttt{res}'$, return, end if
  $\delta'' = \texttt{TransformK}(p'', R')$
  if $\delta'' = 0$, $\texttt{resN} = \mathrm{fl}(\texttt{res} + \mu)$, return, end if
  if $\mathrm{sign}(\delta'') = \mathrm{sign}(\mu)$, $\texttt{resN} = \texttt{res}'$, return, end if
  $\texttt{resN} = \texttt{res}$

REMARK. Following Matlab convention the second and third output argument are omitted in the third call of `TransformK` because they are not needed.

The proof that `resN` is the rounded to nearest result of s is not difficult, though a little lengthy due to various case distinctions.

THEOREM 6.2. *Let p be a vector of n floating-point numbers and assume $2(n+2)^2\texttt{eps} \le 1$. Let* `resN` *be the result of Algorithm 6.1 (*`NearSum`*). Then* `resN` *is the rounded to nearest exact sum $s := \sum_{i=1}^n p_i$ in the sense of IEEE 754.*

PROOF. The assumptions of Lemma 5.5 are satisfied, so

$$s = \texttt{res} + R + \sum_{i=1}^n p_i' = \texttt{res} + \delta' + R' + \sum_{i=1}^n p_i'' \quad \text{and} \quad \delta' \in \Box(s - \texttt{res}) \ , \tag{6.1}$$

and Lemma 2.6 implies

$$\mathrm{sign}(\delta') = \mathrm{sign}(s - \texttt{res}) \ . \tag{6.2}$$

We will determine the "switching point" M for rounding to nearest determined by `res` and the sign of δ'. Then we have to check whether the true sum s is less than or greater than or equal to M.

If $\delta' = 0$, the final value of `resN` is the correct value $\texttt{res} = s$. Moreover, if $\texttt{res} \in \mathbb{U}$, then Lemma 2.6 implies $\texttt{res} = s$ and $\delta' = 0$, so that $\texttt{resN} = \texttt{res} = s$.

Henceforth we may assume $\delta' \neq 0$ and $\texttt{res} \notin \mathbb{U}$. Then (6.2) yields

$$\begin{array}{lllll} \delta' < 0 & \Rightarrow & N(\texttt{res}) := \mathrm{pred}(\texttt{res}) < & s & < \texttt{res} \\ \delta' > 0 & \Rightarrow & \texttt{res} < & s & < \mathrm{succ}(\texttt{res}) =: N(\texttt{res}) \ . \end{array} \tag{6.3}$$

That means the "switching point" deciding the rounding to nearest of s is $M := \frac{1}{2}\big(\texttt{res} + N(\texttt{res})\big)$, with the neighbor $N(\texttt{res})$ defined by (6.3) depending on the sign of δ'. We claim that

$$M = \texttt{res} + \mu \quad \text{and} \quad \texttt{res}' = N(\texttt{res}) \ . \tag{6.4}$$

To prove (6.4) first assume $|\mathtt{res}| \neq \mathrm{ufp}(\mathtt{res})$. Then $|N(\mathtt{res}) - \mathtt{res}| = 2\mathtt{eps} \cdot \mathrm{ufp}(\mathtt{res})$ since $\mathtt{res} \notin \mathbb{U}$, and

$$|M - \mathtt{res}| = |\frac{1}{2}\big(N(\mathtt{res}) - \mathtt{res}\big)| = \mathtt{eps} \cdot \mathrm{ufp}(\mathtt{res}) < \mathtt{eps}|\mathtt{res}| < 2\mathtt{eps} \cdot \mathrm{ufp}(\mathtt{res}) = |N(\mathtt{res}) - \mathtt{res}| \ ,$$

i.e. $\mathtt{res} + \mathrm{sign}(\delta')\mathtt{eps}|\mathtt{res}|$ is in the open interval with endpoints M, which is the midpoint between $\mathtt{res}$ and $N(\mathtt{res})$, and $N(\mathtt{res})$. Hence $\mathtt{res}' = N(\mathtt{res}) \neq \mathtt{res}$. Furthermore, $\mathtt{res} \notin \mathbb{U}$ implies $(\mathtt{res}' - \mathtt{res})/2 = \big(N(\mathtt{res}) - \mathtt{res}\big)/2 \in \mathbb{F}$, so that $\mu = M - \mathtt{res}$.

Next assume $|\mathtt{res}| = \mathrm{ufp}(\mathtt{res})$, so that $|\mathtt{res}| \geq \mathtt{eps}^{-1}\mathtt{eta}$ since $\mathtt{res} \notin \mathbb{U}$, and

$$\begin{aligned} |N(\mathtt{res}) - \mathtt{res}| &= 2\mathrm{ufp}(\mathtt{res}) \quad \text{for} \quad |N(\mathtt{res})| = \mathrm{succ}(|\mathtt{res}|) \ , \\ |N(\mathtt{res}) - \mathtt{res}| &= \mathrm{ufp}(\mathtt{res}) \quad \text{for} \quad |N(\mathtt{res})| = \mathrm{pred}(|\mathtt{res}|) \end{aligned}$$

In the first case, $\mathtt{res} + \mathrm{sign}(\delta')\mathtt{eps}|\mathtt{res}|$ is equal to the midpoint M between $\mathtt{res}$ and $N(\mathtt{res})$, so rounding of tie to even implies $\mathrm{fl}(\mathtt{res} + \mathrm{sign}(\delta')\mathtt{eps}|\mathtt{res}|) = \mathtt{res}$. Therefore $\mu = M - \mathtt{res}$ and $\mathtt{res}'$ is corrected to $N(\mathtt{res})$.

In the second case, $\mathtt{res} + \mathrm{sign}(\delta')\mathtt{eps}|\mathtt{res}|$ is equal to $N(\mathtt{res})$, the floating-point number adjacent to $\mathtt{res}$ towards zero. Therefore $\mathtt{res}' = N(\mathtt{res})$. If $|\mathtt{res}| \geq 2\mathtt{eps}^{-1}\mathtt{eta}$, then $|\mathtt{res}' - \mathtt{res}| \geq 2\mathtt{eta}$ so that $(\mathtt{res}' - \mathtt{res})/2 \in \mathbb{F}$, $\mu = (\mathtt{res}' - \mathtt{res})/2$ and (6.4). But $|\mathtt{res}| = \mathtt{eps}^{-1}\mathtt{eta}$ is not possible because in this case s cannot be strictly between $N(\mathtt{res})$ and $\mathtt{res}$ since $|N(\mathtt{res}) - \mathtt{res}| = \mathtt{eta}$ and, as a sum of floating-point numbers, $s \in \mathtt{eta}\mathbb{Z}$. This proves (6.4).

Of course, $\mathrm{sign}(\mu) = \mathrm{sign}(\delta')$. It follows

$$\begin{array}{lllll} & |s - \mathtt{res}| < |\mu| & \Rightarrow & \mathrm{fl}(s) = \mathtt{res} & \text{and} \\ (6.5) & |s - \mathtt{res}| > |\mu| & \Rightarrow & \mathrm{fl}(s) = \mathtt{res}' & \text{and} \\ & |s - \mathtt{res}| = |\mu| & \Rightarrow & \mathrm{fl}(s) = \mathrm{fl}(\mathtt{res} + \mu) \ . & \end{array}$$

By Lemma 5.5, δ' is a faithful rounding of $s - \mathtt{res}$, so $\mu \in \mathbb{F}$ and (2.29) imply that $|\delta'| \lessgtr |\mu| \Rightarrow |s - \mathtt{res}| \lessgtr |\mu|$, and these cases are handled correctly in Algorithm `NearSum` . In case $|\delta'| = |\mu|$, which means $\delta' = \mu$, we need more work to decide the sign of $s - \mathtt{res} - \mu$. By Lemma 2.6 and (6.1), δ'' is a faithful rounding of $s - \mathtt{res} - \mu = R' + \sum_{i=1}^{n} p_i''$. So $\delta' = \mu$ and Lemma 2.6 imply

$$\begin{array}{rcll} \mathrm{sign}(\mu) & = & \mathrm{sign}(s - \mathtt{res}) & \text{and} \\ \mathrm{sign}(\delta'') & = & \mathrm{sign}(s - \mathtt{res} - \mu) \ . & \end{array}$$

Hence Algorithm `NearSum` handles the cases correctly according to (6.5). □

In contrast to Algorithm 4.4 (`AccSum`), the computing time of Algorithm 6.1 (`NearSum`) depends on the maximum exponent E of the vector entries: The maximum number of iterations in `Transform` can only be bounded by some $m \lesssim (E - \log_2(\mathtt{eps}^{-1}\mathtt{eta}))/(\log_2(\mathtt{eps}^{-1} - M)$, which is about $m \lesssim (E + 1024)/(53 - M)$ for IEEE 754 double precision. This is independent of the condition number of the sum. However, it seems unlikely that this maximum is achieved in other than constructed examples. It can only be achieved if there are vector elements $|p_i|$ near the overflow range, the binary representations of the other vector elements completely cover the whole exponent range from 1023 downto -1024 without gap, and the exact sum cancels completely such that the exact result is exactly the midpoint between two adjacent floating-point numbers.

7. Vectors of huge length. In IEEE 754 double precision our summation algorithm computes a faithful rounding of the exact result for up to about $6.7 \cdot 10^7$ summands. This should suffice for most practical purposes. In IEEE 754 single precision with $\mathtt{eps} = 2^{-24}$, the number of summands is restricted to $n = 2894$, which may be an obstacle for the application of our algorithms. This can be improved by rewriting of Algorithm 4.8 (`Transform`).

We first observe that the extraction in Transform works for large n as well if we only change the "until"-condition on $|t^{(m)}|$.

LEMMA 7.1. *Assume the "until"-condition in Algorithm 4.8 (*Transform*) is changed into*

$$\text{until} \quad \sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta} \quad \text{or} \quad |t'| \ge \sigma \tag{7.1}$$

and call that algorithm TransformP*. Let $\tau_1, \tau_2, p', \sigma$ be the results of Algorithm* TransformP *applied to a nonzero vector of floating-point numbers $p_i, 1 \le i \le n$, and $\varrho \in \mathbb{F}$. Assume that $2^M\texttt{eps} \le \frac{1}{2}$ and $\varrho \in \texttt{eps}\sigma_0\mathbb{Z}$ is satisfied for $M := \lceil \log_2(n+2) \rceil$, $\mu := \max_i |p_i|$ and $\sigma_0 = 2^{M+\lceil \log_2 \mu \rceil}$. Denote $s := \sum_{i=1}^n p_i$.*

Then Algorithm TransformP *will stop, and*

$$s + \varrho = \tau_1 + \tau_2 + \sum_{i=1}^n p'_i \ , \tag{7.2}$$

$$\max |p'_i| \le \texttt{eps}\sigma \ , \quad \tau_1 = \text{fl}(\tau_1 + \tau_2) \quad \text{and} \quad \tau_1, \tau_2 \in \texttt{eps}\sigma\mathbb{Z} \ . \tag{7.3}$$

If $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then the vector p' is identically zero. If $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then

$$|\tau_1| \ge \sigma \ . \tag{7.4}$$

REMARK. Note that the computation of σ' may be afflicted with a rounding error if σ is in the underflow range $\mathbb{U}$. However, in this case the algorithm stops as in the original version and σ' is not used any more.

PROOF OF LEMMA 7.1. Carefully going through the proof of correctness of Algorithm 4.1 (Transform) in Lemma 4.2, we see that the assumptions of Theorem 3.4 are satisfied, and that the only places in the proof of (4.1), (4.2) and (4.3), where the stronger assumption $2(n+2)^2\texttt{eps} \le 1$ is used, are firstly $2^M\texttt{eps} < 1$ so that the algorithms stops, and secondly the deduction of $|t^{(m-1)}| < \sigma_{m-2}$ in (4.5). However, the latter was in the induction argument, assuming that the corresponding "until"-condition was not satisfied. The changed "until"-condition (7.1) implies $|t^{(m-1)}| < \sigma_{m-2}$ directly.

Algorithm 4.8 (Transform) is identical to Algorithm 4.1 (Transform) despite check for zero sum and omitting indices, so as before the assertions are not affected. The remaining assertions are merely a transcription. □

So far, so good. However, by (7.2) and (7.3), the error of τ_1 to the true result $s + \varrho$ may be as large as $\sum p'_i$, which is bounded by $n\texttt{eps}\sigma$. For huge n this may come close to σ, the lower bound for τ_1. Nevertheless, it suffices to determine the sign of s.

THEOREM 7.2. *Assume the "until"-condition in Algorithm 4.8 (*Transform*) is changed into (7.1) and call that algorithm* TransformP*. Let $\tau_1, \tau_2, p', \sigma$ be the results of Algorithm* TransformP *applied to a nonzero vector of floating-point numbers $p_i, 1 \le i \le n$, and $\varrho \in \mathbb{F}$. Assume that $2^M\texttt{eps} \le \frac{1}{2}$ and $\varrho \in \texttt{eps}\sigma_0\mathbb{Z}$ is satisfied for $M := \lceil \log_2(n+2) \rceil$, $\mu := \max_i |p_i|$ and $\sigma_0 = 2^{M+\lceil \log_2 \mu \rceil}$. Denote $s := \sum_{i=1}^n p_i$.*

Then Algorithm TransformP *will stop, and*

$$\text{sign}(s + \varrho) = \text{sign}(\texttt{res}) \ . \tag{7.5}$$

PROOF. The assumptions of Lemma 7.1 are satisfied. Hence $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ implies $\tau_1 = \text{fl}(\tau_1+\tau_2) = \text{fl}(s+\varrho)$ and $\text{sign}(s+\varrho) = \text{sign}(\tau_1) = \text{sign}(\texttt{res})$. If $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then $|\tau_1| \ge \sigma \ge \texttt{eps}^{-1}\texttt{eta}$. Hence (7.3) gives

$$|\tau_2| + |\sum_{i=1}^n p'_i| \le \texttt{eps}|\tau_1| + 2^M\texttt{eps}\sigma \le (\texttt{eps} + \frac{1}{2})|\tau_1| \ .$$

and (7.2) finishes the proof. □

This simplified version of Algorithm 4.4 (AccSum) works in single precision for dimensions up to $8.3 \cdot 10^6$, and in double precision for dimensions up to $4.5 \cdot 10^{15}$.

Next we improve the approximation by continuing to extract p_i' until the error term is small enough compared to τ_1. This is done in the following Algorithm 7.3 (AccSumHugeN) to achieve faithful rounding.

ALGORITHM 7.3. *Accurate summation with faithful rounding for huge n.*

```
function res = AccSumHugeN(p)
   [τ1, τ2, q^(0), σ, M] = TransformP(p, 0)
   if σ ≤ (1/2)eps^(-1)eta, res = τ1, return, end if
   k = 0;  σ0 = fl(2^M eps σ)
   repeat
      k = k + 1
      [τ^(k), q^(k)] = ExtractVector(σ_(k-1), q^(k-1))
      σ_k = fl(2^M eps σ_(k-1))
   until  σ_(k-1) ≤ (1/2)eps^(-1)eta   or   σ_(k-1) < fl(4eps|τ1|)
   τ~^(1) = fl(τ^(1) + (τ^(2) + ··· + (τ^(k) + (τ2 + (Σ_{i=1}^n q_i^(k)))) ···))
   res = fl(τ1 + τ~^(1))
```

PROPOSITION 7.4. *Assume the "until"-condition in Algorithm 4.8 (*Transform*) is changed into (7.1) and call that algorithm* TransformP. *Let p be a vector of n floating-point numbers, assume $8(n+2)\mathtt{eps} \le 1$ and $\mathtt{eps} \le 1/128$. Let* res *be the result of Algorithm 7.3 (*AccSumHugeN*) applied to p. Then* res *is a faithful rounding of $s := \sum_{i=1}^n p_i$.*

REMARK 1. For IEEE 754 single precision with $\mathtt{eps} = 2^{-24}$ this limits the vector length to a little over 2 million rather than $n = 2894$ for Algorithm AccSum. For IEEE 754 double precision the vector length is now limited to about $1.1 \cdot 10^{15}$.

REMARK 2. Note that the code given in the Appendix uses Algorithm 3.5 (NextPowerTwo) to compute 2^M rather than M, so that AccSumHugeN is to be adapted accordingly.

PROOF OF PROPOSITION 7.4. Algorithm TransformP is called with $\varrho = 0$. So $2^M\mathtt{eps} \le \frac{1}{2}$ and Lemma 7.1 imply

$$s = \tau_1 + \tau_2 + \sum_{i=1}^n q_i^{(0)} \ , \tag{7.6}$$

$$\max |q_i^{(0)}| \le \mathtt{eps}\sigma, \tag{7.7}$$

$$\tau_1 = \mathrm{fl}(\tau_1 + \tau_2) \quad \text{and} \quad |\tau_2| \le \mathtt{eps}|\tau_1| \ . \tag{7.8}$$

If σ computed by Algorithm TransformP is in the underflow range, then $q^{(0)}$ the zero vector by Lemma 7.1, and $\mathrm{fl}(s) = \mathtt{res}$ by (7.6).

Henceforth we may assume $\sigma > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$, so that (7.4) implies

$$|\tau_1| \ge \sigma \ge \mathtt{eps}^{-1}\mathtt{eta} \ . \tag{7.9}$$

Therefore no rounding error occurs in the second part of the "until"-condition, i.e. $\mathrm{fl}(4\mathtt{eps}|\tau_1|) = 4\mathtt{eps}|\tau_1|$.

By definition of σ_0 and (7.7), $\max_i |q_i^{(0)}| \le 2^{-M}\sigma_0$, so the assumptions of Theorem 3.4 are satisfied for the first call of ExtractVector for $k = 1$. Therefore,

$$s = \tau_1 + \tau_2 + \tau^{(1)} + \sum_{i=1}^n q_i^{(1)} \ , \quad |\tau^{(1)}| < \sigma_0 \quad \text{and} \quad \max_i |q_i^{(1)}| \le \mathtt{eps}\sigma_0 = 2^{-M}\sigma_1 \ .$$

If $\sigma_0 \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$, then the loop finishes and a possible rounding error in the computation of σ_1 does no harm since σ_1 is not used. If $\sigma_0 > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$, then σ_1 is computed without rounding error and can safely be used.

Again, the assumptions of Theorem 3.4 are satisfied for the next call of `ExtractVector`, and repeating this argument shows

$$s = \tau_1 + \tau_2 + \sum_{\nu=1}^{k} \tau^{(\nu)} + \sum_{i=1}^{n} q_i^{(k)} \ , \quad |\tau^{(k)}| < \sigma_{k-1} \quad \text{and} \quad \max_i |q_i^{(k)}| \le 2^{-M}\sigma_k \tag{7.10}$$

for k between 1 and its final value K. The same argument as before applies to possible rounding errors in the computation of σ_K. Until further notice assume $\sigma_{K-1} > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$, so that the "until"-condition yields

$$\sigma_{K-1} < 4\mathtt{eps}|\tau_1| \ . \tag{7.11}$$

Denote

$$\tau^{(K+1)} := \tau_2 + \Big(\sum_{i=1}^{n} q_i^{(K)}\Big) \quad \text{and} \quad \tilde\tau^{(K+1)} := \mathrm{fl}(\tau^{(K+1)}) = \tau^{(K+1)} - \delta_{K+1}, \tag{7.12}$$

$$\tilde\tau^{(k)} := \mathrm{fl}\big(\tau^{(k)} + \tilde\tau^{(k+1)}\big) = \tau^{(k)} + \tilde\tau^{(k+1)} - \delta_k \quad \text{for } 1 \le k \le K \ . \tag{7.13}$$

This implies $\tilde\tau^{(1)} = \tau^{(1)} + \tilde\tau^{(2)} - \delta_1 = \tau^{(1)} + \tau^{(2)} + \tilde\tau^{(3)} - \delta_2 - \delta_1 = \cdots$, so that

$$\tilde\tau^{(1)} = \sum_{k=1}^{K+1} \tau^{(k)} - \sum_{k=1}^{K+1} \delta_k = \sum_{k=1}^{K} \tau^{(k)} + \tau_2 + \Big(\sum_{i=1}^{n} q_i^{(K)}\Big) - \sum_{k=1}^{K+1} \delta_k \ . \tag{7.14}$$

Abbreviate $\varphi := 2^M\mathtt{eps}$. The assumptions imply

$$\varphi \le \frac{1}{8} \ , \quad \mathtt{eps} \le \frac{1}{128} \ , \quad M \ge 2 \quad \text{and} \quad \sigma_k = \varphi^{k+1}\sigma \ \text{ for } \ 0 \le k \le K \ . \tag{7.15}$$

Then (7.10), (7.8), the standard estimation for floating-point sums [18, Lemma 8.4] and (7.11) yield

$$|\delta_{K+1}| \le \gamma_n\Big(|\tau_2| + \sum_{i=1}^{n} |q_i^{(K)}|\Big) \le \frac{1}{7}(|\tau_2| + \sigma_K) \le \frac{3}{14}\mathtt{eps}|\tau_1| \ . \tag{7.16}$$

Next we need an upper bound on $|\tilde\tau^{(k)}|$. We use an induction argument to show

$$|\tilde\tau^{(k)}| \le \Big(\sum_{\nu=0}^{K+1-k} (1+\mathtt{eps})^{\nu+1}\varphi^\nu\Big)\sigma_{k-1} + (1+\mathtt{eps})^{K+2-k}|\tau_2| \quad \text{for } 1 \le k \le K+1 \ . \tag{7.17}$$

For $k = K+1$ this follows by

$$\begin{aligned} |\tilde\tau^{(K+1)}| &\le |\mathrm{fl}(\tau_2 + \Big(\sum q_i^{(K)}\Big))| \le (1+\mathtt{eps})\big(|\tau_2| + |\mathrm{fl}(\sum q_i^{(K)})|\big) \le (1+\mathtt{eps})\big(|\tau_2| + n2^{-M}\sigma_K\big) \\ &\le (1+\mathtt{eps})\big(|\tau_2| + \sigma_K\big) \end{aligned}$$

using (7.12), (2.16) and (7.10), and the induction step uses

$$|\tilde\tau^{(k)}| \le (1+\mathtt{eps})\big(|\tau^{(k)}| + |\tilde\tau^{(k+1)}|\big)$$

by (7.13), and $|\tau^{(k)}| < \sigma_{k-1}$ by (7.10). Hence (7.15) gives

$$|\tilde\tau^{(k)}| \le \frac{1+\mathtt{eps}}{1-(1+\mathtt{eps})\varphi}\varphi^k\sigma + (1+\mathtt{eps})^{K+2-k}|\tau_2| \quad \text{for } 1 \le k \le K+1 \ ,$$

and by (7.13) and (7.15) it follows

$$\mathtt{eps}^{-1}|\delta_k| \le |\tilde\tau^{(k)}| \le \frac{7}{6}\varphi^k\sigma + (1+\mathtt{eps})^{K+2-k}|\tau_2| \quad \text{for } 1 \le k \le K \ . \tag{7.18}$$

Next we need an upper bound on K. Denote $\texttt{eps} = 2^{-m}$, then (7.15) gives $m \ge 7$ and $M - m \le -3$. For $K > 2$, the "until"-condition and (7.9) imply $\varphi^{K-1}\sigma = \sigma_{K-2} \ge 4\texttt{eps}|\tau_1| \ge 4\texttt{eps}\sigma$, so that $(K-1)(M-m) \ge 2-m$ and

$$K \le 1 + \frac{m-2}{m-M} \le \frac{m+1}{3} \ . \tag{7.19}$$

This is of course also true for $K \le 2$. For later use we use $m \ge 7$ to show

$$(1+\texttt{eps})^2 \frac{(1+\texttt{eps})^K - 1}{\texttt{eps}} \le \big(\frac{129}{128}\big)^2 \big((1+2^{-m})^{\frac{m+1}{3}} - 1\big)\texttt{eps}^{-1} < \frac{1}{28}\texttt{eps}^{-1} \ . \tag{7.20}$$

Moreover, $m \ge 7$ implies $(1+\texttt{eps})^{K+1} \le (1+2^{-m})^{1+\frac{m+1}{3}} \le 4/3$, so that the definition of $\texttt{res}$, (7.18), (7.8), (7.15) and (7.9) yield

$$\begin{aligned} |\texttt{res}| &\ge (1-\texttt{eps})\big(|\tau_1| - |\tilde{\tau}^{(1)}|\big) \\ &\ge (1-\texttt{eps})\big(|\tau_1| - \tfrac{7}{6}\varphi\sigma - (1+\texttt{eps})^{K+1}\texttt{eps}|\tau_1|\big) \\ &\ge (1-\texttt{eps})\big(\tfrac{95}{96}|\tau_1| - \tfrac{7}{48}\sigma\big) \\ &> \tfrac{1}{2}\sigma \ge \tfrac{1}{2}\texttt{eps}^{-1}\texttt{eta} \ , \end{aligned} \tag{7.21}$$

proving that $\texttt{res} \notin \mathbb{U}$. Moreover, (7.15) and (7.21) yield

$$|\texttt{res}| > \frac{5}{6}|\tau_1| \ . \tag{7.22}$$

Now set $r := \tau_1 + \tilde{\tau}^{(1)}$, so that $\texttt{res} = \text{fl}(r)$. Then (7.10) and (7.14) imply

$$\delta := s - r = \sum_{k=1}^{K+1} \delta_k \ .$$

We will show $2|\delta| < \texttt{eps}|\texttt{res}|$ and apply Lemma 2.5 to demonstrate that $\texttt{res}$ is a faithful rounding of the true result s. With (7.18) and using (7.15) it holds

$$\texttt{eps}^{-1}|\delta| \le \frac{7}{6}\frac{\varphi}{1-\varphi}\sigma + (1+\texttt{eps})^2\frac{(1+\texttt{eps})^K - 1}{\texttt{eps}}|\tau_2| + \texttt{eps}^{-1}|\delta_{K+1}| \ . \tag{7.23}$$

Using (7.15), (7.8), (7.16), (7.20), $\sigma \le |\tau_1|$ and (7.22), a little computation shows

$$2\texttt{eps}^{-1}|\delta| < 2\left(\frac{1}{6}\sigma + \frac{1}{4}|\tau_1|\right) \le \frac{5}{6}|\tau_1| < |\texttt{res}| \ . \tag{7.24}$$

By (7.21) we know $\texttt{res} \notin \mathbb{U}$, so Lemma 2.5 proves that $\texttt{res}$ is a faithful rounding of s. This closes the case $\sigma_{K-1} > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$.

Suppose $\sigma_{K-1} \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then (7.10) implies $q_i^{(K)} = 0$ for all i and the second part of the "until"-condition, that is (7.11) need not be true. But both are only needed to derive the upper bound of δ_{K+1}, and (7.12) implies $\delta_{K+1} = 0$. Especially (7.21) is still valid, so $\texttt{res} \notin \mathbb{U}$, and (7.24) and Lemma 2.5 finish the proof. □

With (7.19) we can also estimate the treatable vector length n in Algorithm 4.8 (`Transform`) depending on the number of extra extractions. The results are summarized in Table 7.1.

As before an algorithm with K-fold faithful rounding for input vectors of huge length may be developed as well.

Table 7.1
Maximum treatable dimension n depending on K

K	single precision	double precision
3	4094	$9.5{\cdot}10^{7}$
4	65634	$4.3{\cdot}10^{10}$
5	$2.6{\cdot}10^{5}$	$9.2{\cdot}10^{11}$
6	$6.0{\cdot}10^{5}$	$5.8{\cdot}10^{12}$
7	$1.0{\cdot}10^{6}$	$2.0{\cdot}10^{13}$

Table 9.1
Ratio of computing time for Dekker's algorithm 2.1 with branch to Knuth's algorithm 2.1

n	CPU: Pentium IV 2.53GHz, Compiler: Intel Fortran 8.1	CPU: Pentium M 1GHz, Compiler: Intel Fortran 8.1
100	1.662	6.571
1,000	2.247	8.429
10,000	2.532	8.700
100,000	2.538	2.313
1,000,000	2.462	1.644

8. Dot products. We briefly mention how to compute a faithful rounding of the value of a dot product of floating-point vectors, as well the rounded to nearest result. We already mentioned that for given $a, b \in \mathbb{F}$, Algorithm 1.1 (`TwoProduct`) together with Algorithm 1.2 (`Split`) compute $x, y \in \mathbb{F}$ with

$$x + y = a \cdot b \ ,$$

provided no overflow and no underflow occurred. It might happen that an over- or underflow cancels, so that the exact result of a dot product in within the floating-point range. Although this occurs most likely only in constructed examples, it is easy to take care of.

For given vectors $u, v \in \mathbb{F}^n$, first split the index set $\{1, \cdots, n\}$ into large, small and medium, i.e. those indices where $u_i \cdot v_i$ is near or in the overflow range, near or in the underflow range, and neither of those, respectively. This can of course be decided without causing an exception.

Then first handle the "large" indices with properly scaled input data, ignoring the rest. Using our estimations it can be decided whether the final result can possibly be influenced by the medium range. If this is not so, we are finished, possibly reporting overflow.

If not, treat the medium range the same way. Again it can be decided whether the "small" indices need to be treated. If this is the case, we proceed the same way with the scaled input data.

9. Computational results. In the following we give some computational results on different architectures and using different compilers. All programming and measurement was done by the second author.

In Section 2 we mentioned the severe slow down by branches due to lack of optimization. For example, Knuth's Algorithm 2.1 (`TwoSum`) may be replaced by Dekker's Algorithm 2.7 equipped with a branch $|a| \geq |b|$. We measured the elapsed time for both approaches and list the ratio of computing time in Table 9.1. For example, the ratio 1.662 means that Dekker's algorithm with 3 floating-point operations and one branch requires 66% more computing time than Knuth's algorithm with 6 floating operations. So Table 9.1 demonstrates how branches can ruin computing times. Fortunately we have ensured in all our error-free transformations of the sum of two floating-point numbers that Dekker's Algorithm 2.7 can be used *without* branch.

Table 9.2
Ratio of computing times for extraction (CPU: Pentium IV 2.53GHz, Compiler: Intel Fortran 8.1)

n	ExtractScalar	Dint	Dnint	Int=Dble	M/E
100	1.0	5.11	7.00	9.78	48.44
1,000	1.0	5.25	8.63	10.50	54.63
10,000	1.0	4.67	8.11	9.56	48.33
100,000	1.0	0.91	1.01	1.02	4.78

Table 9.3
Ratio of computing times for extraction (CPU: Pentium M 1GHz, Compiler: Intel Fortran 8.1)

n	ExtractScalar	Dint	Dnint	Int=Dble	M/E
100	1.0	4.45	6.00	17.94	31.97
1,000	1.0	3.79	5.03	14.84	26.26
10,000	1.0	3.22	4.24	12.30	21.65
100,000	1.0	1.39	1.75	4.82	8.43

All our algorithms are based on extractions, the split of a floating-point number with respect to σ, a power of 2 corresponding to a certain exponent. Since this operation is in the inner loop of all our algorithms, we payed special attention to this and designed Algorithm 3.1 (`ExtractScalar`) to be as fast as possible. This algorithm requires 3 floating-point operations and has no branch.

Another possibility to extract bits of a floating-point number p is proper scaling and rounding to integer. Some compilers offer two possibilities of such rounding, namely chopping and rounding to the nearest integer. In the Tables 9.2 and 9.3 the columns "Dint" and "Dnint" refer to those roundings, respectively. Another possibility of rounding a floating-point number is the assignment to a variable of type integer. One obstacle might be that an integer format with sufficient precision is not available. This approach is referred to by "Int=Dble". Finally, the desired bits may be extracted by access to mantissa and exponent and some bit manipulation. This is referred to by "M/E".

As can be seen from the Tables 9.2 to 9.3 our Algorithm 3.1 (`ExtractScalar`) is much faster than the other possibilities. Our algorithms directly benefit from this. There is a certain drop in the ratio for large dimension which is related to the cache size. We programmed all tests straightforwardly without blocking, so not too much attention must be paid to the last lines of the Tables 9.2 and 9.3 for huge dimension.

We designed our algorithms to take advantage of compiler optimization. However, sometimes we have to make sure that the latter is not overdone. The first line $q = \mathrm{fl}\left((\sigma + p) - \sigma\right)$ in Algorithm 3.1 (`ExtractScalar`), for example, may be erroneously optimized into $q = p$. This can, of course, be avoided by setting appropriate compiler options; however, this may slow down the whole computation. In this specific case the second author suggested a simple trick to overcome this by using $q = \mathrm{fl}\left(|\sigma + p| - \sigma\right)$ instead. This does not change the intended result since $|p| \le \sigma$ is assumed in the analysis (Lemma 3.2), it avoids unintended compiler optimization and it does not slow down the computation.

Next we tested our summation algorithms. For huge condition numbers larger than $\mathtt{eps}^{-1}$ we used our Algorithm 6.1 in [36] and residuals for the matrices from [42], where methods to generate arbitrarily ill-conditioned matrices exactly representable in floating-point are described. Dot products are transformed into sums by means of Algorithms 1.2 (`Split`) and 1.1 (`TwoProduct`).

We display results for dimension $n = 1000$. For other dimensions the results are similar, and again we observe a drop in the factors for huge dimensions due to limited cache size. In the following Tables 9.4 and 9.5 the column "DSum" refers to the ordinary recursive summation, the computing time of which is normed to 1.

TABLE 9.4
Ratio of computing times for summation ($n = 1000$, CPU: Pentium IV 2.53GHz, Compiler: Intel Visual Fortran 8.1)

cond	Dsum	AccSum	Dint	Dnint	Int=Dble	Sum2	Sum3	XBLAS
1e8	1.0	**7.00**	21.80	28.20	36.20	**18.20**	**32.60**	**25.80**
1e16	1.0	**7.02**	21.83	28.20	36.20	**18.20**	**32.60**	**25.80**
1e32	1.0	**9.60**	37.00	44.80	62.60	18.20	**32.60**	25.80
1e64	1.0	14.17	58.17	77.67	101.33	18.20	32.60	25.80
1e128	1.0	29.00	124.20	161.00	215.40	18.20	32.60	25.80

TABLE 9.5
Ratio of computing times for summation ($n = 1000$, CPU: Pentium M 1GHz, Compiler: Intel Visual Fortran 8.1)

cond	Dsum	AccSum	Dint	Dnint	Int=Dble	Sum2	Sum3	XBLAS
1e8	1.0	**8.38**	23.75	29.81	75.56	**8.59**	**15.41**	**14.35**
1e16	1.0	**8.40**	23.78	29.82	75.56	**8.59**	**15.41**	**14.35**
1e32	1.0	**11.44**	38.06	47.31	122.06	8.59	**15.41**	14.35
1e64	1.0	20.50	72.69	98.50	240.62	8.59	15.41	14.35
1e128	1.0	33.65	119.47	160.18	404.54	8.59	15.41	14.35

Column “AccSum” shows the computing time for our Algorithm 4.4 (`AccSum`), and the following columns “Dint”, “Dnint” and “Int=Dble” refer to `AccSum` with only the extraction changed as indicated.

Furthermore, columns “Sum2” and “Sum3” display the timing for our Algorithms 4.4 (`Sum2`) and Algorithm 4.8 (`SumK`) for $K = 3$ out of [36]. These algorithms produce a result of quality *as if* computed in 2-fold or 3-fold precision, respectively. That means, for condition numbers up to $\texttt{eps}^{-1}$ or $\texttt{eps}^{-2}$, respectively, we can expect a result accurate to the last bit. Finally, column “XBLAS” displays the computing time for the corresponding algorithm `BLAS_dsum_x` in XBLAS [31], the result of which is of the same quality as that of `Sum2`, namely *as if* computed in 2-fold precision.

For condition numbers up to $\texttt{eps}^{-1}$ we achieve a remarkable factor of about 7 or 8 compared to recursive summation. This is achieved by straightforward Fortran code without further optimization and without blocking.

For larger condition numbers gradually more extractions are needed, and correspondingly our Algorithm 4.4 (`AccSum`) requires more computing time. The timing is compared with ordinary (recursive) summation, the time of which is normed to 1.0, but the results of recursive summation are meaningless for condition numbers beyond $\texttt{eps}^{-1}$. Note that the factors between the different options for extraction are similar to the ones obtained in Tables 9.2 and 9.3.

Similarly, the results of algorithm Sum2 and the results of XBLAS summation are meaningless for condition numbers beyond $\texttt{eps}^{-2}$, and the results of algorithm Sum3 are meaningless for condition numbers beyond $\texttt{eps}^{-3}$.

As we showed, algorithm AccSum always delivers a faithfully rounded result, i.e. accurate to the last bit. In those rows of Tables 9.4 and 9.5, where we can also expect algorithms Sum2, Sum3 and XBLAS to compute results accurate to the last bit, we displayed the corresponding computing times in **bold**. So the timings displayed in bold in one row refer to algorithms the results of which are of comparable accuracy.

As we see, our Algorithm 4.4 (`AccSum`) is always faster, sometimes much faster than the other approaches. The striking advantage of AccSum is that it guarantees a result *accurate* to the last bit rather than a result as if computed in some specified *precision*. Moreover, it only requires more computing time if necessary because of ill condition, and, in contrast to the other approaches, AccSum is perfectly suited for parallelization.

The computational results for dot products are very similar to the ones for summation and are therefore omitted.

Finally that we tried to find examples where the result of Algorithm 4.4 (`AccSum`) is not rounded to nearest. We treated dimensions from 10 to 10^5 and condition numbers as above. We especially used dimensions $n = 2^M - 2$. Fortunately, we have Algorithm 6.1 (`NearSum`) for reference. It is not so easy to find a long precision package delivering results always rounded to nearest; especially we observed problems with rounding tie to even. In several billion test cases we did not find one example with the result of `AccSum` not being the nearest floating-point number to the exact result.

10. Appendix. Following we give the proof of (5.10), that is

$$\tilde{R} := \tau_2 - \Delta \in \mathbb{F} \tag{10.1}$$

under the assumptions of Lemma 5.5 and for $\sigma > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$. This implies $R = \mathrm{fl}(\tau_2 - \Delta) = \tau_2 - \Delta$. Since σ is a power of 2, we have

$$\sigma \geq \mathtt{eps}^{-1}\mathtt{eta}\ . \tag{10.2}$$

We already know

$$\Delta = \mathtt{res} - \tau_1 \in \mathbb{F}\ . \tag{10.3}$$

We first prove some facts, namely

$$|\mathtt{res}| \geq \frac{3}{4}|\tau_1|\ , \tag{10.4}$$

$$\frac{1}{2}\mathtt{eps}^{-1}\sigma \leq |\tau_1| \quad \Rightarrow \quad \mathtt{res} \in \{\mathrm{pred}(\tau_1), \tau_1, \mathrm{succ}(\tau_1)\}\ , \tag{10.5}$$

$$\frac{1}{2}\mathtt{eps}^{-1}\sigma \leq |\tau_1| \quad \text{and} \quad \mathtt{res} \neq \tau_1 \quad \Rightarrow \quad \mathrm{sign}(\tau_2)\mathrm{sign}(\Delta) \geq 0\ . \tag{10.6}$$

Using (4.19), (4.11) and (4.10) it follows

$$|\mathtt{res}| \geq (1 - 2\mathtt{eps})|\tau_1| - n\mathtt{eps}\sigma \geq (1 - 2\mathtt{eps})|\tau_1| - 2^{-M-1}\mathrm{ufp}(\tau_1) \geq \frac{3}{4}|\tau_1|$$

and prove (10.4). Furthermore, (5.8), (4.14), (4.11), (4.8) and the assumption $\frac{1}{2}\mathtt{eps}^{-1}\sigma \leq |\tau_1|$ imply

$$\begin{aligned} |\tau_2'| &\leq (1 + \mathtt{eps})|\tau_2 + \tau_3| \leq |\tau_2| + \mathtt{eps}|\tau_2| + 2^M\mathtt{eps}\sigma \\ &\leq |\tau_2| + \left(\mathtt{eps}^2 + 2^{M+2}\mathtt{eps}^2\right)\mathrm{ufp}(\tau_1) \\ &< |\tau_2| + \mathtt{eps} \cdot \mathrm{ufp}(\tau_1)\ , \end{aligned} \tag{10.7}$$

so that $\tau_1 = \mathrm{fl}(\tau_1 + \tau_2)$, $\mathtt{res} = \mathrm{fl}(\tau_1 + \tau_2')$ and (2.27) imply (10.5). If $\mathtt{res} \neq \tau_1$, then $\mathrm{sign}(\tau_2) = \mathrm{sign}(\tau_2') = \mathrm{sign}(\mathtt{res} - \tau_1) = \mathrm{sign}(\Delta)$ by (2.27) and the monotonicity of rounding, so (10.6).

Proof of (5.10). We can use the analysis of Algorithm 4.1 (`Transform`) and the notation in the proof of Lemma 4.3, especially (4.12). We distinguish several cases.

First, assume $|\tau_1| < \sigma$. Then (4.8) yields $\tau_2 \in \mathtt{eps}\sigma\mathbb{Z}$ and $|\tau_2| \leq \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) < \mathtt{eps}\sigma$, so $\tau_2 = 0$ and $\tilde{R} = -\Delta \in \mathbb{F}$.

Second, assume $\sigma \leq |\tau_1| < \frac{3}{4}\mathtt{eps}^{-1}\sigma$. Then $\mathtt{res} = \mathrm{fl}(\tau_1 + \tau_2')$, (2.17) and (2.11) yield $\mathtt{res} \in \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) \subseteq \mathtt{eps}\sigma\mathbb{Z}$, so $\mathtt{res}, \tau_1, \tau_2 \in \mathtt{eps}\sigma\mathbb{Z}$ from (4.8) and $\Delta, \tilde{R} \in \mathtt{eps}\sigma\mathbb{Z}$. Moreover, (4.12), (2.20), (4.14) and (4.11) imply

$$\begin{aligned} |\tilde{R}| = |\tau_2 - \Delta| &= |\tau_2 - \tau_2' + \delta_1| \leq |\tau_2 - \tau_2'| + \mathtt{eps}|\tau_1 + \tau_2'| \\ &\leq |\tau_3| + \mathtt{eps}|\tau_2 + \tau_3| + \mathtt{eps}|\tau_1| + \mathtt{eps}(1 + \mathtt{eps})|\tau_2 + \tau_3| \\ &\leq \mathtt{eps}|\tau_1| + \mathtt{eps}(2 + \mathtt{eps})|\tau_2| + (1 + \mathtt{eps})^2|\tau_3| \\ &\leq \mathtt{eps}(1 + 3\mathtt{eps})|\tau_1| + 2^M\mathtt{eps}\sigma \\ &< \sigma\ , \end{aligned} \tag{10.8}$$

and (2.14) and (10.2) prove $\tilde{R} \in \mathbb{F}$.

Third, assume $\frac{3}{4}\texttt{eps}^{-1}\sigma \le |\tau_1| < \texttt{eps}^{-1}\sigma$. Then (10.5) implies $\texttt{res} \in \{\text{pred}(\tau_1), \tau_1, \text{succ}(\tau_1)\}$. If $\texttt{res} = \tau_1$, then $\tilde{R} = \tau_2 \in \mathbb{F}$. If $\texttt{res} \ne \tau_1$, the assumption on τ_1 implies $\text{ufp}(\tau_1) = \frac{1}{2}\texttt{eps}^{-1}\sigma$ and

$$|\Delta| = |\texttt{res} - \tau_1| = 2\texttt{eps} \cdot \text{ufp}(\tau_1) = \sigma$$

by Lemma 2.2. But $|\tau_2| \le \texttt{eps} \cdot \text{ufp}(\tau_1) = \frac{1}{2}\sigma$ by (4.8) and $\text{sign}(\tau_2)\text{sign}(\Delta) \ge 0$ by (10.6) implies

$$\frac{1}{2}\sigma \le |\tau_2 - \Delta| = |\tilde{R}| \le \sigma \quad \text{and} \quad \tilde{R} \in \texttt{eps}\sigma\mathbb{Z} \ ,$$

so (2.14) and (10.3) prove $\tilde{R} \in \mathbb{F}$ also for that case.

Fourth, assume $\texttt{eps}^{-1}\sigma \le |\tau_1| < 2\texttt{eps}^{-1}\sigma$. Then (10.4) yields

$$|\texttt{res}| \ge \frac{3}{4}|\tau_1| > \frac{1}{2}\texttt{eps}^{-1}\sigma \ ,$$

so the distance of $\texttt{res}$ to its neighbors is at least $2\texttt{eps} \cdot \text{ufp}(\texttt{res}) \ge \sigma$. Using $s + \varrho = \texttt{res} + \delta$ as in (4.13) and (4.21) implies

$$|\varrho - \texttt{res}| \le |\delta| + |s| < \frac{1}{2}\texttt{eps} \cdot \text{ufp}(\tau_1) + 2^{-M}\sigma = \big(\frac{1}{2} + 2^{-M}\big)\sigma < \sigma \ ,$$

so $\varrho, \texttt{res} \in \mathbb{F}$ implies $\varrho = \texttt{res}$. With (4.7) this yields

$$\begin{aligned} |\texttt{res} - \tau_1| &\le |\delta| + |\tau_2| + |\textstyle\sum p_i'| < \big(\frac{1}{2}\texttt{eps} + \texttt{eps}\big)\text{ufp}(\tau_1) + 2^M\texttt{eps}\sigma \\ &= \big(1 + \frac{1}{2} + 2^M\texttt{eps}\big)\sigma < 2\sigma = 2\texttt{eps} \cdot \text{ufp}(\tau_1) \ . \end{aligned}$$

If $\texttt{res} = \tau_1$, then $\tilde{R} = \tau_2 \in \mathbb{F}$. Otherwise, if $\texttt{res} \ne \tau_1$ and as by Lemma 2.2, the only possibility that the distance of the floating-point number τ_1 to another floating-point number $\texttt{res}$ is strictly less than $2\texttt{eps}\cdot\text{ufp}(\tau_1)$ is $|\tau_1| = \text{ufp}(\tau_1) = \texttt{eps}^{-1}\sigma$ and

$$|\Delta| = |\texttt{res} - \tau_1| = \texttt{eps} \cdot \text{ufp}(\tau_1) = \sigma \ .$$

Moreover, (10.5) implies $\text{sign}(\tau_2)\text{sign}(\Delta) \ge 0$, so that with $\tilde{R} = \tau_2 - \Delta$ with $|\tau_2| \le \texttt{eps} \cdot \text{ufp}(\tau_1) = \sigma$ we have

$$0 \le |\tau_2 - \Delta| = |\tilde{R}| \le \sigma \quad \text{and} \quad \tilde{R} \in \texttt{eps}\sigma\mathbb{Z} \ ,$$

and again (2.14) and (10.3) prove $\tilde{R} \in \mathbb{F}$ also for that case.

Fifth and last, assume $|\tau_1| \ge 2\texttt{eps}^{-1}\sigma$. By (4.3) and (4.2) we have for the final value m in Algorithm 4.1 ($\texttt{Transform}$)

$$|t^{(m-1)}| = |\tau_1 + \tau_2 - \tau^{(m)}| \ge (1 - \texttt{eps})|\tau_1| - \sigma > \texttt{eps}^{-1}\sigma \ .$$

Suppose $m \ge 2$, then the "until"-condition in $\texttt{Transform}$ implies

$$|t^{(m-1)}| < 2^{2M+1}\texttt{eps}\sigma_{m-2} = 2^{M+1}\sigma_{m-1} = 2^{M+1}\sigma < \texttt{eps}^{-1}\sigma \ ,$$

a contradiction. Thus $m = 1$, $t^{(m-1)} = t^{(0)} = \varrho$, $\tau_1 + \tau_2 = \varrho + \tau^{(m)}$ and $\texttt{res} = \text{fl}(\varrho + \tau^{(m)})$. But (10.4) yields

$$|\texttt{res}| \ge \frac{3}{4}|\tau_1| > \texttt{eps}^{-1}\sigma \ ,$$

and the distance of $\texttt{res}$ to its neighbors is at least $2\texttt{eps} \cdot \text{ufp}(\texttt{res}) \ge 2\sigma$. Hence $|\tau^{(m)}| < \sigma_{m-1} = \sigma$ implies

$$\varrho = \text{fl}(\varrho + \tau^{(m)}) = \text{fl}(\tau_1 + \tau_2) = \tau_1 \ ,$$

it follows $\Delta = 0$ and $\tilde{R} = \tau_2 \in \mathbb{F}$ and also this case is finished.

All together it shows $R = \mathrm{fl}(\tau_2 - \Delta) = \mathrm{fl}(\tilde{R}) = \tilde{R} = \tau_2 - \Delta$. This closes the proof of (5.10). □

Following is executable Matlab code for Algorithm 4.4 including acceleration for zero sums and elimination of zero summands for that case. Moreover, the algorithms `ExtractVector` and `FastTwoSum` are expanded. Note that the Matlab function `nextpow2(f)` returns the smallest k such that $2^k \geq |f|$, while Algorithm 3.5 (`NextPowerTwo`) returns 2^k. Accordingly, the variable `Ms` refers to 2^M in Algorithm 4.4 (`AccSum`).

ALGORITHM 10.1. *Executable Matlab code for Algorithm 4.4* (`AccSum`) *including check for zero sum.*

```
function res = AccSum(p)
% For given vector p, result res is the exact sum of p_i faithfully rounded.
% Acceleration for zero sums is included.
%
  n = length(p);                      % initialization
  mu = max(abs(p));                   % abs(p_i) <= mu
  if ( n==0 ) | ( mu==0 )             % no or only zero summands
    res = 0;
    return
  end
  Ms = NextPowerTwo(n+2);             % n+2 <= 2^M
  sigma = Ms*NextPowerTwo(mu);        % first extraction unit
  phi = 2^(-53)*Ms;                   % factor to decrease sigma
  factor = 2^(-52)*Ms*Ms;             % factor for sigma check
%
  t = 0;
  while 1
    q = ( sigma + p ) - sigma;        % [tau,p] = ExtractVector(sigma,p);
    tau = sum(q);                     % sum of leading terms
    p = p - q;                        % remaining terms
    tau1 = t + tau;                   % new approximation
    if ( abs(tau1)>=factor*sigma ) | ( sigma<=realmin )
      tau2 = tau - ( tau1 - t );      % [tau1,tau2] = FastTwoSum(t,tau)
      res = tau1 + ( tau2 + sum(p) ); % faithfully rounded final result
      return
    end
    t = tau1;                         % sum t+tau exact
    if t==0                           % accelerate case sum(p)=0
      res = AccSum(p(p~=0));          % recursive call, zeros eliminated
      return
    end
    sigma = phi*sigma;                % new extraction unit
  end
```

11. Summary. We presented algorithms for summation with provably faithfully rounded result and provably rounded to nearest result in working precision. Furthermore, an algorithm was presented for summation with faithfully rounded result in K-fold accuracy. The paper contains as well the ingredients to compute a rounded to nearest result in K-fold accuracy. All our algorithms use only ordinary floating-point addition, subtraction and multiplication, no branches in the inner loops and no special operations. Similar

algorithms for dot products are sketched.

We showed that our summation Algorithm 4.4 (`AccSum`) is always faster, sometimes much faster than other approaches. The striking advantage of AccSum is that it guarantees a result *accurate* to the last bit rather than a result as if computed in some specified *precision*. Moreover, in contrast to other approaches, AccSum is perfectly suited for parallelization.

The algorithms are based on so-called error-free transformations. We hope to see these computationally and mathematically highly interesting operations in future computer architectures and floating-point standards.

Acknowledgement. The first author wishes to express his thankfulness that the paper could be written during stays at Waseda University supported by the Grant-in-Aid for Specially Promoted Research from the Mext, Japan. The first author also wishes to thank his students of the winter term 2004/05 and the summer term 2005 for their patience and constructive comments. The second author would like to express his sincere thanks to Dr. Y. Ushiro for his stimulating discussions.

REFERENCES

[1] *MPFR: A C library for multiple-precision floating-point computations with exact rounding.* available at `http://www.mpfr.org`.
[2] F. Avnaim, J. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec, *Evaluating signs of determinants using single-precision arithmetic*, Algorithmica, 17 (1997), pp. 111–132.
[3] D. Bailey, *A Fortran-90 based multiprecision system*, ACM Trans. Math. Software, 21 (1995), pp. 379–387.
[4] G. Bohlender, *Floating-point computation of functions with maximum accuracy*, IEEE Trans. Comput., C-26 (1977), pp. 621–632.
[5] R. Brent, *A Fortran Multiple-Precision Arithmetic Package*, tech. report, Dept. of Computer Science, Australian National University, Canberra, 1975.
[6] H. Brönnimann, C. Burnikel, and S. Pion, *Interval arithmetic yields efficient dynamic filters for computational geometry*, Discrete Applied Mathematics, 109 (2001), pp. 25–47.
[7] H. Brönnimann and M. Yvinec, *Efficient exact evaluation of signs of determinants*, Algorithmica, 27 (2000), pp. 21–56.
[8] K. Clarkson, *Safe and effective determinant evaluation*, in Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (Pittsburgh, PA, IEEE Computer Society Press, 1992, pp. 387–395.
[9] M. Daumas and D. Matula, *Validated roundings of dot products by sticky accumulation*, IEEE Trans. Computers, 46 (1997), pp. 623–629.
[10] T. J. Dekker, *A floating-point technique for extending the available precision*, Numer. Math., 18 (1971), pp. 224–242.
[11] J. Demmel and Y. Hida, *Accurate and Efficient Floating Point Summation*, SIAM J. Sci. Comput., 25 (2003), pp. 1214–1248.
[12] J. Demmel and Y. Hida, *Fast and accurate floating point summation with application to computational geometry*, Numerical Algorithms, 37 (2004), pp. 101–112.
[13] *GNU multiple precision arithmetic library (GMP), version 4.1.2.*, 2003. Code and documentation available at `http://swox.com/gmp`.
[14] S. Graillat, P. Langlois, and N. Louvet, *Compensated Horner Scheme*, tech. report, Laboratoire LP2A, University of Perpignan, 2005.
[15] J. Hauser, *Handling floating-point exceptions in numeric programs*, ACM Trans. Program. Lang. Syst., 18 (1996), pp. 139–174.
[16] C. Hecker, *Let's get to the (floating) point*, Game Developer, 2 (1996), pp. 19–24.
[17] N. Higham, *The accuracy of floating point summation*, SIAM J. Sci. Comput., 14 (1993), pp. 783–799.
[18] ———, *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadelphia, 2nd ed., 2002.
[19] C. Hoffmann, *Robustness in geometric computations*, JCISE, 1 (2001), pp. 143–156.
[20] ANSI/IEEE, *IEEE Standard for Binary Floating Point Arithmetic*, IEEE, New York, Std 754–1985 ed., 1985.
[21] M. Jankowski, A. Smoktunowicz, and Woźniakowski, *A note on floating-point summation of very many terms*, J. Information Processing and Cybernetics-EIK, 19 (1983), pp. 435–440.
[22] M. Jankowski and Woźniakowski, *The accurate solution of certain continuous problems using only sigle precision arithmetic*, BIT, 25 (1985), pp. 635–651.
[23] W. Kahan, *A survey of error analysis*, in Proceedings of the IFIP Congress, Information Processing 71, North-Holland, Amsterdam, The Netherlands, 1972, pp. 1214–1239.
[24] ———, *Implementation of Algorithms (Lecture Notes by W.S. Haugeland and D. Hough)*, Tech. Report 20, Department of Computer Science, University of California, Berkeley, CA, 1973.

[25] A. Kiełbasziński, *Summation algorithm with corrections and some of its applications*, Math. Stos, 1 (1973), pp. 22–41.

[26] D. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2, Addison Wesley, Reading, Massachusetts, 1969.

[27] S. Krishnan, M. Foskey, T. Culver, J. Keyser, and D. Manocha, *PRECISE: efficient multiprecision evaluation of algebraic roots and predicates for reliable geometric computation*, in SCG '01: Proceedings of the 17th annual symposium on Computational geometry, New York, NY, USA, 2001, ACM Press, pp. 274–283.

[28] U. Kulisch and W. L. Miranker, *The arithmetic of the digital computer: A new approach*, SIAM Review, 28 (1986), pp. 1–40.

[29] P. Langlois and N. Louvet, *Solving Trinagular Systems More Accurately and Efficiently*, tech. report, Laboratoire LP2A, University of Perpignan, 2005.

[30] H. Leuprecht and W. Oberaigner, *Parallel algorithms for the rounding exact summation of floating point numbers*, Computing, 28 (1982), pp. 89–104.

[31] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, and D. Yoo, *Design, implementation and testing of extended and mixed precision BLAS*, ACM Trans. Math. Softw., 28 (2002), pp. 152–205.

[32] S. Linnainmaa, *Software for doubled-precision floating point computations*, ACM Trans. Math. Soft., 7 (1981), pp. 272–283.

[33] M. Malcolm, *On accurate floating-point summation*, Comm. ACM, 14 (1971), pp. 731–736.

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

[35] A. Neumaier, *Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen*, ZAMM, 54 (1974), pp. 39–51.

[36] T. Ogita, S. Rump, and S. Oishi, *Accurate Sum and Dot Product*, SIAM J. Sci. Comput., 26 (2005), pp. 1955–1988.

[37] M. Pichat, *Correction d'une somme en arithmétique à virgule flottante*, Numer. Math., 19 (1972), pp. 400–406.

[38] D. Priest, *Algorithms for arbitrary precision floating point arithmetic*, in Preceedings of the 10th Symposium on Comuter Arithmetic, P. Kornerup and D. Matula, eds., Grenoble, France, 1991, IEEE Computer Society Press, pp. 132–145.

[39] ———, *On properties of floating point arithmetics: Numerical stability and the cost of accurate computations*, PhD thesis, Mathematics Department, University of California at Berkeley, CA, 1992.

[40] D. Ross, *Reducing truncation errors using cascading accumulators*, Communication of ACM, 8 (1965), pp. 32–33.

[41] S. Rump, *Kleine Fehlerschranken bei Matrixproblemen*, PhD thesis, Universität Karlsruhe, 1980.

[42] ———, *A class of arbitrarily ill-conditioned floating-point matrices*, SIAM J. Mat. Anal. Appl., 12 (1991), pp. 645–653.

[43] S. Schirra, *Precision and robustness in geometric computations*, in Algorithmic Foundations of Geographic Information Systems, M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, eds., Lecture Notes in Computer Science 1340, Springer Verlag, Berlin, 1997, pp. 255–287.

[44] J. Shewchuk, *Adaptive precision floating-point arithmetic and fast robust geometric predicates*, Discrete Comput. Geom., 18 (1997), pp. 305–363.

[45] G. Zielke and V. Drygalla, *Genaue Lösung linearer Gleichungssysteme*, GAMM Mitt. Ges. Angew. Math. Mech., 26 (2003), pp. 7–108.