SIAM J. SCI. COMPUT.
Vol. 31, No. 5, pp. 3466–3502
© 2009 Society for Industrial and Applied Mathematics

ULTIMATELY FAST ACCURATE SUMMATION*

SIEGFRIED M. RUMP†

Abstract. We present two new algorithms `FastAccSum` and `FastPrecSum`, one to compute a faithful rounding of the sum of floating-point numbers and the other for a result "as if" computed in K-fold precision. Faithful rounding means the computed result either is one of the immediate floating-point neighbors of the exact result or is equal to the exact sum if this is a floating-point number. The algorithms are based on our previous algorithms `AccSum` and `PrecSum` and improve them by up to 25%. The first algorithm adapts to the condition number of the sum; i.e., the computing time is proportional to the difficulty of the problem. The second algorithm does not need extra memory, and the computing time depends only on the number of summands and K. Both algorithms are the fastest known in terms of flops. They allow good instruction-level parallelism so that they are also fast in terms of measured computing time. The algorithms require only standard floating-point addition, subtraction, and multiplication in one working precision, for example, double precision.

Key words. maximally accurate summation, faithful rounding, K-fold precision, error-free transformation, distillation, high accuracy, XBLAS, error analysis

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

DOI. 10.1137/080738490

1. Introduction.

The computation of the sum or dot product of vectors of floating-point numbers is ubiquitous in scientific computations. The dot product of two vectors of length n can be transformed (without error) into the sum of one vector of length $2n$; see section 2. Therefore a vast amount of literature is devoted to summation, among them [8, 17, 19, 21, 23, 25, 26, 35, 40, 42, 43, 44, 45].

In many applications it is important to calculate sums or dot products accurately, among them include the following.

(1) *Residual iteration.* For a system of linear or nonlinear equations $f(x) = 0$, an approximate solution $\tilde{x}$ is often improved by some Newton or quasi-Newton procedure. This requires an accurate computation of the residual $f(\tilde{x})$. For linear systems these are dot products; see also [28, 41, 31, 14, 32]. For polynomial systems the problem can be transformed into sums and dot products [15, 24, 16, 13]. For algebraic equations the problem can be transformed into a linear systems [36].

(2) *Geometrical predicates.* In computer geometry and computer graphics it is to be decided on which side of a hyperplane a point lies. The resulting determinant calculation can be transformed into the sum or dot product problem [40, 31, 11, 12].

(3) *Computer Algebra.* In many problems such as Gröbner basis computation or quantifier elimination the true sign of a function has to be determined [5, 4, 29, 10, 20]. With accurate summation or dot product routines this is sometimes possible in pure floating-point and thus faster than using multiple precision arithmetic.

*Received by the editors October 19, 2008; accepted for publication (in revised form) June 8, 2009; published electronically September 4, 2009.
http://www.siam.org/journals/sisc/31-5/73849.html
†Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße 95, 21071 Hamburg, Germany, and Visiting Professor, Faculty of Science and Engineering, Waseda University, 3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan (rump@tu-harburg.de).

(4) *Linear Programming.* The classical simplex algorithm relies on the decision of the sign of some reduced cost vector. A number of practical LP-problems are ill-conditioned [30] so that wrong decisions may lead to erroneous solutions.
(5) *Multiple Precision Arithmetic.* Core algorithms such as multiplication or division are based on dot products and may be accelerated using accurate sums and dot products [2, 9].

Denote the set of floating-point numbers by $\mathbb{F}$, the relative rounding error unit by $\texttt{eps}$, and the result of a floating-point computation by $\mathrm{fl}(\cdot)$, 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. An expression like $\mathrm{fl}\big(\sum p_i\big)$ implies inherently that the summation may be performed in any order.

Let a vector $p \in \mathbb{F}^n$ of floating-point numbers be given. The ordinary or recursive summation computes

$$\begin{aligned} &\tilde{s} = p_1 \\ &\text{for } i = 2 : n \\ &\quad \tilde{s} = \mathrm{fl}(\tilde{s} + p_i) \\ &\text{end for} \end{aligned} \tag{1.1}$$

Due to the limited precision of floating-point arithmetic, the accuracy of $\tilde{s} := \mathrm{fl}\big(\sum p_i\big)$ depends on the condition number of the sum. In fact, for a nonzero sum we can expect the relative error to satisfy

$$\left| \frac{\tilde{s} - \sum p_i}{\sum p_i} \right| \sim \texttt{eps} \cdot \mathrm{cond}\left(\sum p_i\right) = \texttt{eps} \cdot \frac{\sum |p_i|}{|\sum p_i|}. \tag{1.2}$$

In the applications mentioned above it may happen that the condition number $\mathrm{cond}\,(\sum p_i)$ is not far from $\texttt{eps}^{-1}$; for the first one, the residual calculation, it is even typical. This means that the floating-point approximation $\tilde{s}$ has almost no or no correct digit.

All algorithms for summation are aimed at some improved accuracy of the result. Higham [18] devotes an entire chapter to summation, and excellent overviews can be found in [18, 23]. Next we summarize the major ideas behind the known algorithms. For simplicity we assume the base precision to be IEEE 754 double precision corresponding to $\texttt{eps} = 2^{-53}$.

1.1. Previous approaches. One of the first algorithms by Malcolm [25] utilizes the limited exponent range of floating-point numbers. He defines an extended precision array e_i where each element corresponds to an individual exponent. The summands p_i are added into the array element corresponding to their exponent. If the extended precision has $53 + k$ bits in the mantissa, then obviously no error occurs for up to 2^k summands, and $\sum p_i = \sum e_i$. The array e_i is added starting with largest exponents. Malcolm put the entire algorithm into 19 lines of Fortran-77 code.

Instead of many small accumulators, each corresponding to one exponent, one may define one long accumulator as propagated mainly by Kulisch. The individual summands p_i are added into the long accumulator without error. To avoid unnecessary carry propagation two accumulators for positive and negative summands may be defined. The result of the sum or dot product is exactly stored in the long accumulator.

An idea in a way between these two was proposed by Demmel and Hida [8]. They collect some exponents together to one array element. The extremes are Malcolm's approach with one array element per exponent and the long accumulator with one array for all exponents.

In general, the sum of two floating-point numbers a, b is not again a floating-point number. However, the exact error y of the floating-point approximation $x := \mathrm{fl}(a+b)$ to the true sum $a+b$ can be shown to be always a floating-point number. This error can be computed by a very simple algorithm $[x, y] = \texttt{FastTwoSum}(a, b)$ [7] using only floating-point addition and subtraction (see section 3) provided $|a| \geq |b|$. Apart from overflow, the mathematical property $x + y = a + b$ is true for all $a, b \in \mathbb{F}$. I called this "error-free transformation" (EFT) in [27].

Pichat [33] and Neumaier [26] independently and apparently without knowing `FastTwoSum` use this EFT to add the p_i. This approach was called "compensated summation":

$$
\begin{array}{l}
\tilde{s} = p_1 \\
\text{for } i = 2 : n \\
\quad \text{if } |\tilde{s}| \geq |p_i| \\
\quad\quad [\tilde{s}, y_i] = \texttt{FastTwoSum}(\tilde{s}, p_i) \\
\quad \text{else} \\
\quad\quad [\tilde{s}, y_i] = \texttt{FastTwoSum}(p_i, \tilde{s}) \\
\quad \text{end if} \\
\text{end for}
\end{array}
\tag{1.3}
$$

Because `FastTwoSum` is an error-free transformation for sorted operands, (1.3) implies $\sum_{i=2}^{n} p_i = \tilde{s} + \sum_{i=2}^{n} y_i$. Pichat and Neumaier add the error terms y_i in pure floating-point and improve (1.2) into

$$
\left| \frac{\tilde{s} - \sum p_i}{\sum p_i} \right| \sim \texttt{eps} + n \cdot \texttt{eps}^2 \mathrm{cond}\left(\sum p_i \right). \tag{1.4}
$$

Priest [34, 35] first sorts the input data by absolute value and then applies a scheme similar to that in (1.3) to add the errors y_i. The result of his "doubly-compensated summation" algorithm is almost maximally accurate, independent of the condition number:

$$
\left| \frac{\tilde{s} - \sum p_i}{\sum p_i} \right| \leq 2 \cdot \texttt{eps}. \tag{1.5}
$$

However, it requires sorting by absolute value of the input data. Another algorithm by Priest [34, 35] transforms the input vector p_i recursively into a so-called nonoverlapping expansion x_i. This is a vector of floating-point numbers such that the exponents of adjacent x_i differ by at least the length of the mantissa and $\sum p_i = \sum x_i$. The costly normalization steps are relaxed by Shewchuck [40] by requiring only that the binary expansions of the x_i do not overlap. This turns out to be faster, although the length of the expansion may increase and some x_i may consist only of one bit in the mantissa.

Yet another approach by Zhu and coworkers [43, 44] first adds the positive and negative summands in the spirit of Pichat and Neumaier and treats the error vectors in a way that the final result is maximally accurate.

1.2. New and very fast methods. All methods described so far transform the input vector into another vector leaving the sum invariant. However, all methods are slowed down by access to the exponent and in particular by branches. On today's architectures there is a drastic speed-up by branch-prediction, by instruction-level parallelism, and by avoiding cache-misses. On the contrary, especially branches may slow down an algorithm significantly.

TABLE 1.1
Performance in Mflops for LAPACK Gaussian elimination with partial pivoting (DGETRF) and with total pivoting (DGETC2); a branch is counted as one FLOP.

n	DGETRF	DGETC2
500	1725	215
1000	2121	186
2000	2525	151
3000	2663	101

As an example, we display the measured Mflops of the LAPACK [1] routines for Gaussian elimination with partial pivoting DGETRF and with total pivoting DGETC2 using IMKL. We count one branch as 1 flop. The results are displayed in Table 1.1. Note that if there were no time penalty for branches, the Mflop-counts would be equal. However, we observe more than a *factor* of 25 slowdown for dimension $n = 3000$.

In [27], Knuth's branch-free algorithm `TwoSum` was used for the EFT of the sum, and it was observed that Pichat's and Neumaier's scheme (1.3) can be viewed as an *error-free vector transformation (EFVT)*:

$$\begin{array}{l} p'_1 = p_1 \\ \text{for } i = 2 : n \\ \quad [p'_i, p'_{i-1}] = \texttt{TwoSum}(p_i, p'_{i-1}) \\ \text{end for} \end{array} \tag{1.6}$$

The input vector p is transformed into the vector p' leaving the sum invariant. The result of $[x, y] = \texttt{TwoSum}(a, b)$ is identical to that of $[x, y] = \texttt{FastTwoSum}(a, b)$ but without the assumption $|a| \geq |b|$ at the cost of 3 extra floating-point operations. Nevertheless, avoiding the branch produces a much faster algorithm. The key observation in [27] is that the condition number of $\sum p'_i$ is reduced by about a factor `eps`:

$$\operatorname{cond}\left(\sum p'_i\right) \sim n \cdot \texttt{eps} \cdot \operatorname{cond}\left(\sum p_i\right). \tag{1.7}$$

Obviously, this process can be continued by EFVTs $p \to p' \to p'' \to \cdots$ reducing the condition number in each step by about a factor $n \cdot \texttt{eps}$.

In [37] we presented a different and yet faster kind of EFVT (for more details cf. section 4). Suppose a vector $p \in \mathbb{F}^n$ is given with $n < 2^M$ for $M \in \mathbb{N}$. Determine $E \in \mathbb{Z}$ so that $\max |p_i| < 2^E$. Then the chunk of bits to the powers $E+M \ldots E+M-52$ of 2 are extracted from each p_i into the leading parts q_i, and the remaining parts p'_i are the bits below $E + M - 52$ (see Figure 1.1). The extraction is error-free, i.e., $p_i = q_i + p'_i$ for all i. It follows that $q_i = 0$ or $2^{E+M-52} \leq |q_i| < 2^E$ so that there is no rounding error when adding the q_i in floating-point. Therefore

$$\sum p_i = \sum (q_i + p'_i) = \operatorname{fl}\left(\sum q_i\right) + \sum p'_i =: \tau + \sum p'_i. \tag{1.8}$$

Again the condition number reduces by about a factor $n \cdot \texttt{eps}$ as in (1.7). There are two main advantages of this approach. First, the size of the quantity $\tau := \sum q_i$ displays the condition number, i.e., the difficulty of the problem: A large τ means not much cancelation and reasonable condition number, and a small τ means heavy cancelation and a large condition number. Second, a very simple algorithm with only 3 floating-point operations and no branch was given in [37] to extract the p_i into the leading part q_i and remaining part p'_i.

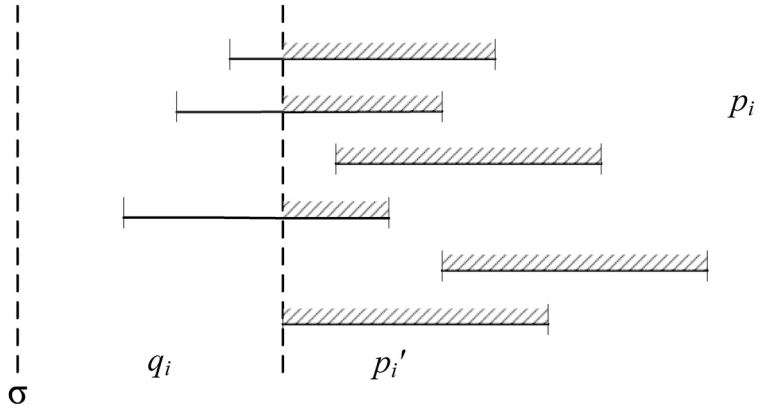

FIG. 1.1. *Extraction into leading and remaining parts by* `AccSum`.

For the ill-conditioned problem, i.e., small τ, the extraction process as in Figure 1.1 is repeated in such a way that the new leading parts q_i' and the former τ can be added in floating-point without error. It follows that

$$\sum p_i = \tau + \sum p_i' = \tau' + \sum p_i'' \ldots, \tag{1.9}$$

again an EFVT decreasing the condition number in each step by about a factor $n \cdot \texttt{eps}$. This is the Algorithm `AccSum` in [37]. The analysis is tricky and shows that the chosen constants are optimal.

This method is the fastest known summation algorithm in terms of floating-point operations and in terms of measured computing time, and there did not seem to be much room for improvement. Nevertheless, we improve its performance in this paper by up to 25% as follows. The inner loop of `AccSum` consists of $4n$ operations: $3n$ for extracting the vector p_i into the vectors q_i and p_i', and n operations for the computation of $\tau = \mathrm{fl}(\sum q_i)$. In this paper we show how to change the extraction procedure so that τ comes for free resulting in a total of only $3n$ operations. Since this is the inner loop of the algorithm, its computing time improves by up to 25%.

The paper is organized as follows. In the following section 2 we introduce the notation I developed in [37]. In contrast to other papers on this subject, a calculus is developed so that conclusions are transformed into inequalities. We think this essentially improves readability and in some sense safety. In section 3 we list some auxiliary results concerning error estimations of floating-point operations. Then we formulate our new algorithm `FastAccSum` and prove that the result is a faithful rounding of the correct sum. In the following section 5 we show that performing a fixed number of $K-1$ extractions we obtain an algorithm `FastPrecSum` producing a result "as if" computed in K-fold precision. Finally, we present some timings in section 6 and some concluding remarks.

2. Notation. We denote the set of floating-point numbers by $\mathbb{F}$, the relative rounding error unit by `eps`, and the smallest positive (unnormalized) floating-point number by `eta`. In IEEE 754 double precision we have $\texttt{eps} = 2^{-53}$ and $\texttt{eta} = 2^{-1074}$. We denote by $\mathbb{U}$ the following subset of $\mathbb{F}$ near underflow:

$$f \in \mathbb{U} \;:\Leftrightarrow\; 0 \le |f| \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}. \tag{2.1}$$

The new IEEE 754 floating-point standard demands a fused-multiply-and-add operation (FMA); that is, for given $a, b, c \in \mathbb{F}$ the result of $a \cdot b + c$ is computed in one rounding to nearest. Given two vectors $p, q \in \mathbb{F}^n$ of floating-point numbers, the products can be split into $p_i q_i = x_i + y_i$ by

$$x_i = \mathrm{fl}(p_i \cdot q_i), \quad y_i = FMA(p_i, q_i, -x_i). \tag{2.2}$$

As long as no over- or underflow occurs, $p_i q_i = x_i + y_i$ is satisfied. If the FMA-operation is not available, the same can be achieved by the algorithm `TwoProduct` [37] by Dekker and Nievergelt using traditional floating-point operations. The resulting $2n$ elements x_i, y_i can be added using our accurate summation algorithm `AccSum`, and thus the dot product problem is reduced to the summation problem. Note that the y_i's are known to be of size $\mathtt{eps}|x_i|$, so as in [27] the first extraction needs to be applied only to the x_i's.

We assume floating-point operations in rounding to nearest; i.e., the mapping $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ satisfies

$$r \in \mathbb{R} \quad \text{and} \quad f = \mathrm{fl}(r) \qquad \Rightarrow \qquad |r - f| = \min\{|r - g| : g \in \mathbb{F}\}. \tag{2.3}$$

Note that nothing is said about rounding of the tie, the midpoint between two floating-point numbers. A common rule, for example, in IEEE 754, is rounding tie to even. In any case monotonicity is preserved, that is,

$$r_1, r_2 \in \mathbb{R} \quad \text{and} \quad r_1 \le r_2 \qquad \Rightarrow \qquad \mathrm{fl}(r_1) \le \mathrm{fl}(r_2). \tag{2.4}$$

Rounding to nearest also implies fl to be a projection, i.e., $\mathrm{fl}(f) = f$ for $f \in \mathbb{F}$. An excellent introduction and many interesting details on floating-point arithmetic and current implementations can be found in the forthcoming book [3].

The standard error estimation for floating-point addition and subtraction of $a, b \in \mathbb{F}$ is

$$\begin{aligned} &\mathrm{fl}(a \circ b) = (a \circ b)(1 + \varepsilon_1) \\ &= (a \circ b)/(1 + \varepsilon_2) \quad \text{with } |\varepsilon_1|, |\varepsilon_2| \le \mathtt{eps} \text{ and } \circ \in \{+, -\}. \end{aligned} \tag{2.5}$$

Note that no error occurs in underflow. For summation this yields [18] for $p_i \in \mathbb{F}$

$$\left| \mathrm{fl}\left(\sum_{i=1}^{n} p_i \right) - \sum_{i=1}^{n} p_i \right| \le \gamma_{n-1} \sum_{i=1}^{n} |p_i|, \tag{2.6}$$

where $\gamma_k := k \cdot \mathtt{eps}/(1 - k \cdot \mathtt{eps})$. For multiplication and division, underflow has to be taken care of. The standard estimation for $a, b \in \mathbb{F}$, $\circ \in \{\cdot, /\}$, where $b \neq 0$ for division, is

$$\mathrm{fl}(a \circ b) = (a \circ b)(1 + \varepsilon) + \eta \qquad \text{with } |\varepsilon| \le \mathtt{eps}, \ \ |\eta| \le \frac{\mathtt{eta}}{2}, \ \ \varepsilon\eta = 0. \tag{2.7}$$

We frequently use

$$\mathrm{fl}(a \circ b) \notin \mathbb{U} \ \ \Rightarrow \ \ \mathrm{fl}(a \circ b) = (a \circ b)(1 + \varepsilon_1) = (a \circ b)/(1 + \varepsilon_2) \quad \text{with } |\varepsilon_1|, |\varepsilon_2| \le \mathtt{eps} \tag{2.8}$$

for $\circ \in \{\cdot, /\}$. Often these error estimations are not sufficient. In [27] I introduced the "unit in the first place"-notation (ufp) or leading bit of a real number. It is defined by

$$0 \neq r \in \mathbb{R} \quad \Rightarrow \quad \mathrm{ufp}(r) := 2^{\lfloor \log_2 |r| \rfloor}, \tag{2.9}$$

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

where $\mathrm{ufp}(0) := 0$. 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 $2\mathtt{eps} \cdot \mathrm{ufp}(f)$. The situation is depicted in Figure 2.1.

Advantages over the unit in the last place are that the concept is independent of the floating-point representation, no care is necessary in the underflow range, and the unit in the first place is defined for real numbers as well. The main advantage is that the "ufp"-concept improves the standard estimation (2.5) into (cf. (2.19) in [37])

(2.10)
$$f = \mathrm{fl}(a+b) \quad \Rightarrow \quad f = a+b+\delta \ \text{ with } \ |\delta| \le \mathtt{eps} \cdot \mathrm{ufp}(a+b) \le \mathtt{eps} \cdot \mathrm{ufp}(f) \le \mathtt{eps}|f|.$$

Note that the improvement is up to a factor 2 depending on whether $|f|$ is at the lower or upper end of the interval $[\mathrm{ufp}(f), 2\mathrm{ufp}(f))$. The "ufp"-grid improves estimations also by

$$r, r' \in \mathbb{R} \ \text{ and } \ \mathrm{ufp}(r) \le |r'| \quad \Rightarrow \quad \mathrm{ufp}(r) \le \mathrm{ufp}(r'). \tag{2.11}$$

In our analysis we will frequently view a floating-number as a scaled integer. For $\sigma = 2^k, k \in \mathbb{Z}$, we use the set $\mathtt{eps} \cdot \sigma\mathbb{Z}$, which can be interpreted as a set of fixed point numbers with smallest positive number $\mathtt{eps} \cdot \sigma$. In particular, $\mathbb{F} \subseteq \mathtt{eta}\mathbb{Z}$.

Estimations of floating-point computations are often involved and error-prone. The "ufp"-notation allows us to map most conclusions into inequalities and seems to improve the readability significantly. In the following we list some basic properties (cf. (2.9) to (2.18) in [37]): For $\sigma = 2^k, k \in \mathbb{Z}, r \in \mathbb{R}$, we have

$$0 \ne r \in \mathbb{R} \ \Rightarrow \ \mathrm{ufp}(r) \le |r| \le 2(1-\mathtt{eps}) \cdot \mathrm{ufp}(r), \tag{2.12}$$

$$\sigma' = 2^m, \ m \in \mathbb{Z}, \ \text{ and } \ \sigma' \ge \sigma \ \Rightarrow \ \mathtt{eps} \cdot \sigma'\mathbb{Z} \subseteq \mathtt{eps} \cdot \sigma\mathbb{Z}, \tag{2.13}$$

$$f \in \mathbb{F} \ \Rightarrow \ f \in 2\mathtt{eps} \cdot \mathrm{ufp}(f)\mathbb{Z}, \tag{2.14}$$

$$r \in \mathtt{eps} \cdot \sigma\mathbb{Z}, \ |r| \le \sigma, \ \text{ and } \ \mathtt{eps} \cdot \sigma \ge \mathtt{eta} \ \Rightarrow \ r \in \mathbb{F}, \tag{2.15}$$

$$a, b \in \mathbb{F}, \ a \ne 0 \Rightarrow \mathrm{fl}(a+b) \in \mathtt{eps} \cdot \mathrm{ufp}(a)\mathbb{Z}. \tag{2.16}$$

With the "ufp"-concept it is also easy to see how the standard estimation (2.6) for summation of a vector improves. We have the following (cf. (2.20) in [37]):

$$\begin{aligned} n \cdot \mathtt{eps} \le 1, \ a_i \in \mathbb{F}, \quad \text{and} \quad |a_i| \le \sigma \Rightarrow \left| \mathrm{fl}\left(\sum_{i=1}^{n} a_i \right) \right| \le n\sigma \quad \text{and} \\ \left| \mathrm{fl}\left(\sum_{i=1}^{n} a_i \right) - \sum_{i=1}^{n} a_i \right| \le \frac{n(n-1)}{2} \mathtt{eps} \cdot \sigma. \end{aligned} \tag{2.17}$$

Finally, it implies a simple criterion for the sum of two floating-point numbers to be error-free (cf. (2.21) in [37]). For $a, b \in \mathbb{F}$ and $\sigma = 2^k$, $k \in \mathbb{Z}$,

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

3. Basic facts. The small backward error in (2.6) turns into a small forward error for nonnegative summands. In fact, the standard estimation [18] can be improved for nonnegative summands a little bit as follows. Let $p_i \in \mathbb{F}$ for $1 \le i \le n$ with $n \cdot \mathtt{eps} < 1$ be given. Then

$$\widetilde{S} := \mathrm{fl}\left(\sum_{i=1}^{n} |p_i|\right) \quad \Rightarrow \quad \sum_{i=1}^{n} |p_i| = \widetilde{S}(1+\varepsilon) \quad \text{with } |\varepsilon| \le (n-1)\mathtt{eps}. \tag{3.1}$$

To see this define $\tilde{s} := \mathrm{fl}\big(\sum_{i\ne k} |p_i|\big)$ for some $1 \le k \le n$. By induction, $\sum_{i\ne k} |p_i| = \tilde{s} + \varepsilon_k$ with $|\varepsilon_k| \le (n-2)\mathtt{eps} \cdot \tilde{s} \le (n-2)\mathtt{eps} \cdot \widetilde{S}$ so that $\widetilde{S} = \mathrm{fl}(\tilde{s} + |p_k|) = \tilde{s} + |p_k| + \delta = \sum_{i=1}^{n} |p_i| - \varepsilon_k + \delta$ with $|\delta| \le \mathtt{eps} \cdot \widetilde{S}$ implies (3.1).

It follows that

$$p_i \in \mathbb{F} \quad \text{and} \quad \sum_{i=1}^{n} |p_i| \le \mathtt{eps}^{-1}\mathtt{eta} \quad \Rightarrow \quad \mathrm{fl}\left(\sum_{i=1}^{n} p_i\right) = \sum_{i=1}^{n} p_i, \tag{3.2}$$

which is a direct consequence of (2.18) with $\sigma = \mathtt{eps}^{-1}\mathtt{eta}$. Note that the neighbors of $f \in \mathbb{F}$, $|f| < \mathtt{eps}^{-1}\mathtt{eta}$ are $f \pm \mathtt{eta}$ so that f is of smallest possible distance to its floating-point neighbors.

For $p_i \in \mathbb{F}$ and $n \cdot \mathtt{eps} < 1$ we have

$$T := \mathrm{fl}\left(\left(\sum_{i=1}^{n} |p_i|\right) / (1 - n \cdot \mathtt{eps})\right) \quad \Rightarrow \quad \sum_{i=1}^{n} |p_i| \le T. \tag{3.3}$$

To see this note first that $\mathrm{fl}(n \cdot \mathtt{eps}) = n \cdot \mathtt{eps}$ because $\mathtt{eps}$ is a power of 2 and $n \cdot \mathtt{eps} < 1$, and therefore also $\mathrm{fl}(1 - n \cdot \mathtt{eps}) = 1 - n \cdot \mathtt{eps}$. We will use this a number of times in the following. Furthermore, $\mathrm{fl}(\sum |p_i|) = \sum |p_i|$ by (3.2) if $T \in \mathbb{U}$, and if $T \notin \mathbb{U}$, then the standard estimations (2.8) and (3.1) imply

$$T \ge \frac{\mathrm{fl}\left(\sum_{i=1}^{n} |p_i|\right)}{(1+\mathtt{eps})(1-n\cdot\mathtt{eps})} \ge \mathrm{fl}\left(\sum_{i=1}^{n} |p_i|\right)(1+(n-1)\mathtt{eps}) \ge \sum_{i=1}^{n} |p_i|.$$

Denote by $\mathrm{pred}(f)$ and $\mathrm{succ}(f)$ the predecessor and successor of a floating-point number f, respectively. For $0 < f \notin \mathbb{U}$, the predecessor of f is $f - \mathtt{eps} \cdot f$ or $f - 2\mathtt{eps} \cdot \mathrm{ufp}(f) < f - \mathtt{eps} \cdot f$ depending on whether f is a power of 2 or not. In any case, $\mathrm{pred}(f) \le (1 - \mathtt{eps})f$. For $0 < f \in \mathbb{U}$ we have $\mathrm{pred}(f) = f - \mathtt{eta}$ and $f \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ so that

$$0 < f \in \mathbb{F} \quad \text{implies} \quad \mathrm{pred}(f) \le (1 - \mathtt{eps}) \cdot f. \tag{3.4}$$

The aim of the paper is to compute a faithful rounding of the exact result $s = \sum p_i$ of the sum of floating-point numbers p_i. That means [7, 35, 6] that the computed result must be equal to the exact result if the latter is a floating-point number, and otherwise it must be one of the immediate floating-point neighbors of the exact result.

Definition 3.1. *A floating-point number $f \in \mathbb{F}$ is called a* faithful rounding *of a real number $r \in \mathbb{R}$ if*

$$\mathrm{pred}(f) < r < \mathrm{succ}(f). \tag{3.5}$$

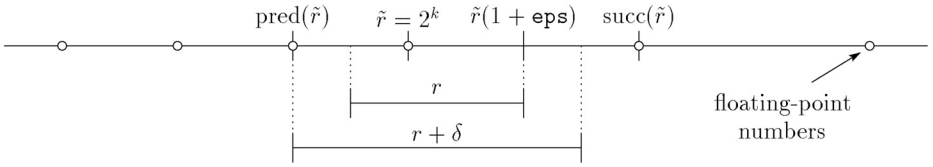

FIG. 3.1. *Faithful rounding near a power of* 2.

Note that $0 \in \mathbb{F}$ implies $\text{sign}(r) = \text{sign}(f)$ if $f \in \mathbb{F}$ is a faithful rounding of $r \in \mathbb{R}$, and in particular $f = 0 \Leftrightarrow r = 0$.

To compute a faithful rounding of a real number r it suffices to know r up to an error margin roughly of size `eps`. In contrast, the rounded-to-nearest $\text{fl}(r)$ requires us ultimately to know r exactly, namely, if r is the midpoint of two adjacent floating-point numbers. This may require substantial and often unnecessary computational effort. Furthermore, the computing time depends on the exponent range of the summands rather than on the condition number of the sum.

In the following we repeat a sufficient criterion for $f \in \mathbb{F}$ to be a faithful rounding of $r \in \mathbb{R}$ (Lemma 2.4 in [37]). The critical case is the change of exponent at a power of 2 as depicted in Figure 3.1.

LEMMA 3.2. *Let $r, \delta \in \mathbb{R}$ and $\tilde{r} := \text{fl}(r)$. 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 $\tilde{r}$ is a faithful rounding of $r + \delta$.*

To compute a faithful rounding of the exact sum independent of the condition number of the sum is an apparent contradiction to the well-accepted rule of thumb that the error of numerical computations can't be better than `eps` times the condition number. The rule of thumb is foiled by so-called *error-free transformations*, which play a key role in our algorithms.

As has been mentioned, Neumaier [26] presented an algorithm where the relative error of the result is bounded by about $n \cdot \texttt{eps}^2$ times the condition number of the sum. Neumaier reinvented a method by Dekker [7] which transforms the sum $a + b$ of two floating-point numbers without error into a sum $x + y$, where x is the usual floating-point approximation and y comprises of the exact error. Dekker's algorithm, an error-free transformation, is surprisingly simple and works as follows.

ALGORITHM 3.3 (error-free transformation of the sum of two floating-point numbers).

$$\begin{aligned} &\text{function } [x, y] = \texttt{FastTwoSum}(a, b) \\ &\quad x = \text{fl}(a + b) \\ &\quad y = \text{fl}((a - x) + b) \end{aligned}$$

The main property of Algorithm 3.3 is

$$x + y = a + b$$

for all floating-point numbers a, b with $|a| \geq |b|$. For our analysis we need the following refined analysis given in Lemma 2.6 in [37].

LEMMA 3.4. *Let a, b be floating-point numbers with $a \in 2\texttt{eps} \cdot \text{ufp}(b)\mathbb{Z}$. Let x, y be the results produced by Algorithm* 3.3 (`FastTwoSum`) *applied to a, b. Then*

$$x + y = a + b, \quad x = \text{fl}(a + b), \quad \textit{and} \quad |y| \leq \texttt{eps} \cdot \text{ufp}(a + b) \leq \texttt{eps} \cdot \text{ufp}(x). \tag{3.6}$$

Furthermore,

$$q := \mathrm{fl}(x-a) = x-a \quad \textit{and} \quad y := \mathrm{fl}(b-q) = b-q, \tag{3.7}$$

which means the floating-point subtractions $x-a$ and $b-q$ are exact.

Remark. Note that $|a| \geq |b|$ implies $\mathrm{ufp}(a) \geq \mathrm{ufp}(b)$, which in turn by (2.14) and (2.13) implies $a \in 2\mathtt{eps} \cdot \mathrm{ufp}(b)\mathbb{Z}$, the assumption of Lemma 3.4.

Finally, we need to compute the unit in the first place of a floating-point number in our new algorithms `FastAccSum` and `PrecSum`. Fortunately, there is a simple algorithm for that.[1]

ALGORITHM 3.5 (unit in the first place of a floating-point number).

function $S = \mathrm{ufp}(p)$
  $q = \mathrm{fl}(\varphi\, p)$ for $\varphi := (2\mathtt{eps})^{-1} + 1$
  $S = \mathrm{fl}(|q - (1-\mathtt{eps})q|)$

The proof of validity is by the following lemma.

LEMMA 3.6. *Let $\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta} \leq x \in \mathbb{F}$. Then if no overflow occurs,*

$$\mathrm{succ}(x) \leq \mathrm{fl}\big((1+2\mathtt{eps})x\big) \leq 2 \cdot \mathrm{ufp}(x). \tag{3.8}$$

If $x = \mathrm{ufp}(x)$, then

$$\mathrm{fl}(x - (1-\mathtt{eps})x) = \mathtt{eps} \cdot \mathrm{ufp}(x), \tag{3.9}$$

and if $x \neq \mathrm{ufp}(x)$, then

$$\mathrm{fl}(x - (1-\mathtt{eps})x) = 2\mathtt{eps} \cdot \mathrm{ufp}(x). \tag{3.10}$$

Proof. The assumption implies $x > 0$ and $\mathrm{succ}(x) = x + 2\mathtt{eps} \cdot \mathrm{ufp}(x)$ so that (2.12) yields

$$\begin{aligned}\mathrm{succ}(x) &= x + 2\mathtt{eps} \cdot \mathrm{ufp}(x) \leq (1+2\mathtt{eps})x \leq x + 2\mathtt{eps} \cdot \mathrm{pred}(2 \cdot \mathrm{ufp}(x)) \\ &= x + 4\mathtt{eps}(1-\mathtt{eps}) \cdot \mathrm{ufp}(x) \leq \mathrm{succ}(x) + (2\mathtt{eps} - 4\mathtt{eps}^2) \cdot \mathrm{ufp}(x).\end{aligned}$$

If $\mathrm{succ}(x) < 2 \cdot \mathrm{ufp}(x)$, then $(1+2\mathtt{eps})x < \mathrm{succ}(\mathrm{succ}(x)) \leq 2 \cdot \mathrm{ufp}(x)$, and for $\mathrm{succ}(x) = 2 \cdot \mathrm{ufp}(x)$ we have $(1+2\mathtt{eps})x < \frac{1}{2}\big(2 \cdot \mathrm{ufp}(x) + \mathrm{succ}(2 \cdot \mathrm{ufp}(x))\big)$. Using (2.4) and $\mathrm{fl}(f) = f$ for $f \in \mathbb{F}$ proves (3.8). For $x = \mathrm{ufp}(x)$ we have $\mathrm{pred}(x) = (1-\mathtt{eps})x$ and (3.9), and for $x \neq \mathrm{ufp}(x)$ using (2.12),

$$\mathrm{pred}(x) = x - 2\mathtt{eps} \cdot \mathrm{ufp}(x) < (1-\mathtt{eps})x < x - \mathtt{eps} \cdot \mathrm{ufp}(x) = \frac{1}{2}\big(\mathrm{pred}(x) + x\big).$$

Hence $\mathrm{fl}\big((1-\mathtt{eps})x\big) = \mathrm{pred}(x) = x - 2\mathtt{eps} \cdot \mathrm{ufp}(x)$, and the lemma follows. ∎

[1] A previous version of this algorithm used also 3 floating-point operations but among them one division. As noted by Jean-Michel Muller, this may be slower than the presented one.

Note that only rounding to nearest is required in the proof of Lemma 3.6, no matter how the tie is rounded.

LEMMA 3.7. *Let $p \in \mathbb{F}$ be given, and let S be the result computed by Algorithm* 3.5. *Then if no overflow occurs,*

$$S = \mathrm{ufp}(p).$$

Proof. The statement is correct for $p = 0$, and we may assume without loss of generality $p > 0$. Then $\bar{p} := \frac{1}{2}\mathtt{eps}^{-1}p \geq \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ satisfies $\bar{p} \in \mathbb{F}$ and the assumptions of Lemma 3.6. Hence $2\mathtt{eps} \cdot \mathrm{ufp}(\bar{p}) = \mathrm{ufp}(p)$ and

$$q = \mathrm{fl}(\varphi\, p) = \mathrm{fl}\left(\frac{1}{2}\mathtt{eps}^{-1}p + p\right) = \mathrm{fl}((1 + 2\mathtt{eps})\bar{p})$$

so that $\mathrm{succ}(\bar{p}) \leq q \leq 2 \cdot \mathrm{ufp}(\bar{p})$. If $q = \mathrm{ufp}(q)$, i.e., q is a power of 2, then $q = 2 \cdot \mathrm{ufp}(\bar{p})$ and $S = \mathtt{eps} \cdot \mathrm{ufp}(q) = 2\mathtt{eps} \cdot \mathrm{ufp}(\bar{p}) = \mathrm{ufp}(p)$, and for $q \neq \mathrm{ufp}(q)$ we have $S = 2\mathtt{eps} \cdot \mathrm{ufp}(\bar{p}) = \mathrm{ufp}(p)$. ∎

4. Algorithm `FastAccSum` with faithfully rounded result. Our new algorithm improves Algorithm `AccSum` (Algorithm 4.5 in [37]). Consider a vector $p_i \in \mathbb{F}$ of floating-point numbers, $1 \leq i \leq n$. Then a chunk of bits is extracted from each p_i resulting in a leading part q_i and remaining part p'_i (see Figure 1.1). The chunk is between two fixed exponents which depend on $\max|p_i|$ and n. The chunks are absolute, not relative to the individual p_i. The extraction is error-free, i.e., $p_i = q_i + p'_i$ for all i. Obviously, $s := \sum p_i = \sum q_i + \sum p'_i$. The extraction is surprisingly simple; it does not need access to bit patterns but is performed with only four floating-point operations per vector element.

ALGORITHM 4.1 (error-free vector extraction (Algorithm 3.4 in [37])).

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

A natural choice is $\sigma := 2^k$. Now the chunk of bits, which is determined by σ, is chosen small enough so that all q_i add without error. It follows that

$$s = \tau + \sum_{i=1}^{n} p'_i. \tag{4.1}$$

Note that the error-free vector transformation requires only $4n$ floating-point operations. By the size of τ, the sum of the leading parts, one can judge whether cancelation took place or not: If τ is small in absolute value, then cancelation occurred and the process of extraction is repeated. If τ is large enough, then adding all terms in floating-point suffices to produce a faithfully rounded result [37].

The size of chunks and the size of τ to decide to continue or not are chosen carefully so that I could show in [37] that the constants are optimal to produce a faithfully rounded result.

The sequence of operations to split p_i in Algorithm 4.1 are exactly the same as the sequence of operations in `FastTwoSum`(σ, p_i) [except that $(a-x)+b$ is replaced by $b-(x-a)$]. Therefore we know by Lemma 3.4 that $p_i = q_i + p'_i$ for any pair σ, p_i provided $\sigma \geq |p_i|$. Hence we are not restricted to the natural choice $\sigma = 2^k$ as in `AccSum`. This is the idea of the new algorithm: to start with some σ_0 and compute the next splitting parameters σ_i as follows.

ALGORITHM 4.2 (new error-free vector extraction).

```
function [σ_n, p'] = ExtractVectorNew(σ_0, p)
  for i = 1 : n
    σ_i = fl(σ_{i-1} + p_i)          % [σ_i, p'_i] = TwoSum(σ_{i-1}, p_i)
    q_i = fl(σ_i - σ_{i-1})
    p'_i = fl(p_i - q_i)
  end for
```

If $|p_i| \leq |\sigma_{i-1}|$ for all $1 \leq i \leq n$ and given σ_0, then Lemma 3.4 implies

$$q_i = \sigma_i - \sigma_{i-1}, \quad p_i = q_i + p'_i, \quad \text{and} \quad |p'_i| \leq \texttt{eps} \cdot \operatorname{ufp}(\sigma_i) \quad \text{for} \quad 1 \leq i \leq n, \tag{4.2}$$

and therefore

$$s = \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i + \sum_{i=1}^{n} p'_i = \sigma_n - \sigma_0 + \sum_{i=1}^{n} p'_i.$$

Hence the new error-free vector transformation requires only $3n$ flops. Since this is the main part of `AccSum`, the new algorithm can be up to 25% faster. In contrast to `AccSum`, not a fixed chunk corresponding to a range of exponents is extracted in `FastAccSum`, but the chunks for the p_i vary individually with the σ_i.

ALGORITHM 4.3 (Algorithm 4.4 (`Sum2`) in [27] as vector transformation).

```
function res = Sum2(p)
  for i = 2 : n
    [p_i, p_{i-1}] = TwoSum(p_i, p_{i-1})
  res = fl( ( sum_{i=1}^{n-1} p_i ) + p_n )
```

Another interpretation of Algorithm `FastAccSum` is the following. In Algorithm 4.3 (`Sum2`) we transform the vector p into a new vector p', an error-free vector transformation. Then the result `res` of `Sum2` is $\operatorname{fl}(\sum p'_i)$. In [27] it is shown that the condition number of $\sum p'_i$ is about $n \cdot \texttt{eps}$ times the condition number of $\sum p_i$. Hence the result `res` is of a quality "as if" computed in quadruple precision.

ALGORITHM 4.4 (Algorithm 4.4 (`Sum2`) in [27] with expanded `TwoSum`).

```
function res = Sum2(p)
  e = 0;  s = p_1
  for i = 2 : length(p)
    x = fl(p_i + s)                  % [x, y] = TwoSum(p_i, s)
    z = fl(x - p_i)
    y = fl((p_i - (x - z)) + (b - z))
    e = fl(e + y)                    % sum of errors
    s = x                            % ordinary floating-point sum
  end for
  res = fl(s + e)
```

In Algorithm 4.4 we rewrite `Sum2` by expanding `TwoSum`. As can be seen `Sum2` adds the vector elements p_i using `TwoSum` and adds the errors in ordinary floating-point. The error-free transformation `TwoSum` requires 6 flops, whereas `FastTwoSum` (Algorithm 3.3) requires only 3 flops. However, the transformation `TwoSum` is always error-free, whereas the input of `FastTwoSum` must be sorted.

The inner loop of our new algorithm `FastAccSum` is that of Algorithm 4.2 (`ExtractVectorNew`) and thus similar to `Sum2`, except that we start with an offset σ_0 and use `FastTwoSum`. Thus `FastAccSum` can be interpreted as choosing the offset σ_0 large enough to ensure that the operands of `FastTwoSum` are always sorted.

This process can be repeated because `ExtractVectorNew` transforms the vector p_i into an approximation $\sigma_n - \sigma_0$ and a vector p'_i of "errors." Thus the new offset σ'_0 depends only on the error-vector p'_i and not on the approximation $\sigma_n - \sigma_0$ of the original sum.

Next we choose σ_0 and other constants carefully so that $\mathrm{fl}(\sigma_n - \sigma_0) = \sigma_n - \sigma_0$ and that the final result is a faithful rounding of the correct sum. We begin with an analysis of the core of the new summation algorithm `FastAccSum` to be presented.

LEMMA 4.5. *Let $p_i, \sigma_0, T \in \mathbb{F}$ be given, and assume $\sum_{i=1}^{n} |p_i| \le T$. Assume $(4n+2)\mathtt{eps} \le 1$, and let the following code be executed:*

$$\begin{aligned}
&\sigma_0 = \mathrm{fl}\left((2T)/(1-(3n+1)\mathtt{eps})\right)\\
&[\sigma_n, p'] = \mathtt{ExtractVectorNew}(\sigma_0, p)\\
&\widetilde{M_1} = \mathrm{fl}\left(\left(\left(\tfrac{3}{2} + 4\mathtt{eps}\right)\cdot(n\cdot\mathtt{eps})\right)\cdot\sigma_0\right)\\
&\widetilde{M_2} = \mathrm{fl}((2n\cdot\mathtt{eps})\cdot\mathrm{ufp}(\sigma_0))\\
&T' = \min(\widetilde{M_1}, \widetilde{M_2})
\end{aligned}$$

Then

$$|p_k| \le \sigma_{k-1} \qquad \text{for} \quad 1 \le k \le n, \tag{4.3}$$

$$|p'_k| \le 2\mathtt{eps}\cdot\mathrm{ufp}(\sigma_0) \qquad \text{for} \quad 1 \le k \le n, \tag{4.4}$$

and

$$\sum_{i=1}^{n} p_i = \sigma_n - \sigma_0 + \sum_{i=1}^{n} p'_i. \tag{4.5}$$

Moreover,

$$|\sigma_k - \sigma_0| \le \frac{1}{2}(1-\mathtt{eps})\sigma_0 < \frac{1}{2}\sigma_0 \qquad \text{for} \quad 1 \le k \le n \tag{4.6}$$

and

$$\mathrm{fl}(\sigma_k - \sigma_0) = \sigma_k - \sigma_0 \qquad \text{for} \quad 1 \le k \le n. \tag{4.7}$$

Furthermore,

$$4T' \le \mathtt{eps}^{-1}\mathtt{eta} \quad \Rightarrow \quad \mathrm{fl}\left(\sum_{i=1}^{n} p'_i\right) = \sum_{i=1}^{n} p'_i \quad \text{and} \quad \sum_{i=1}^{n} |p'_i| \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta} \tag{4.8}$$

and

$$4T' > \mathtt{eps}^{-1}\mathtt{eta} \quad \Rightarrow \quad \sigma_0 \notin \mathbb{U} \quad \text{and} \quad \sum_{i=1}^{n} |p'_i| \le T'. \tag{4.9}$$

Remark 1. For most assertions of Lemma 4.5 a value of σ_0 near T rather than $2T$ would be sufficient. However, the larger value is necessary to ensure that $\mathrm{fl}(\sigma_0 - \sigma_n) = \sigma_0 - \sigma_n$. This in turn is necessary for the error-free splitting of the sum $\sum p_i$ into the single floating-point number and the sum $\sum p'_i$ as by (4.5).

Remark 2. Without mentioning it explicitly each time we assume in all coming proofs $n \geq 2$.

Proof of Lemma 4.5. We first note that $2T \in \mathbb{F}$ and (2.4) imply $\sigma_0 \geq 2T$. Without loss of generality we assume $T \neq 0$ and therefore $\sigma_0 > 0$. The code in Lemma 4.5 and `ExtractVectorNew` together with (2.6) imply

(4.10)
$$\sigma_k = \mathrm{fl}\left(\sigma_0 + \sum_{i=1}^{k} p_i\right) = \sigma_0 + \sum_{i=1}^{k} p_i + \varepsilon_k \quad \text{with} \quad |\varepsilon_k| \leq \gamma_k \left(\sigma_0 + \sum_{i=1}^{k} |p_i|\right) \leq \gamma_n(\sigma_0 + T).$$

If $\sigma_0 \in \mathbb{U}$, then $2T \in \mathbb{U}$ and $\sum |p_i| \leq T$ yield

$$\sigma_0 + \sum_{i=1}^{n} |p_i| \leq \frac{3}{4}\mathtt{eps}^{-1}\mathtt{eta}.$$

Hence (3.2) implies $\sigma_k = \sigma_0 + \sum_{i=1}^{k} p_i$ for $1 \leq k \leq n$ and

$$\sigma_{k-1} \geq \sigma_0 - \sum_{i=1}^{k-1} |p_i| \geq 2T - \sum_{i=1}^{k-1} |p_i| \geq |p_k|.$$

For $\sigma_0 \notin \mathbb{U}$ its definition, (2.8), and $T > 0$ imply

$$\sigma_0 \geq \frac{2T}{(1+\mathtt{eps})(1-(3n+1)\mathtt{eps})} > \frac{2T}{1-3n\cdot\mathtt{eps}}, \tag{4.11}$$

and (4.10) gives

$$\sigma_{k-1} \geq \sigma_0 - \sum_{i=1}^{k-1} |p_i| - \gamma_n(\sigma_0 + T) \geq T\left(\frac{2(1-\gamma_n)}{1-3n\cdot\mathtt{eps}} - \gamma_n\right) - \sum_{i=1}^{k-1} |p_i| \geq T - \sum_{i=1}^{k-1} |p_i| \geq |p_k|.$$

This proves (4.3). Therefore Lemma 3.4 implies $q_i = \mathrm{fl}(\sigma_i - \sigma_{i-1}) = \sigma_i - \sigma_{i-1}$ and $p_i = q_i + p'_i$ and therefore (4.5). Furthermore, by (4.10) and (4.11),

$$\left|\sum_{i=1}^{k} p_i + \varepsilon_k\right| \leq T + \gamma_n(\sigma_0 + T) < \left((1+\gamma_n)\cdot\frac{1}{2}(1-3n\cdot\mathtt{eps}) + \gamma_n\right)\sigma_0 = \frac{1}{2}\sigma_0. \tag{4.12}$$

Thus (4.10) yields $\frac{1}{2}\sigma_0 < \sigma_k < \frac{3}{2}\sigma_0$, and Sterbenz' lemma [18] implies (4.7). Again using (4.10) we obtain

$$\sum_{i=1}^{k} p_i + \varepsilon_k = \sigma_k - \sigma_0 \in \mathbb{F}$$

so that using (4.12) and (3.4) show

$$|\sigma_k - \sigma_0| = \left|\sum_{i=1}^{k} p_i + \varepsilon_k\right| \leq \mathrm{pred}\left(\frac{1}{2}\sigma_0\right) \leq \frac{1}{2}(1-\mathtt{eps})\sigma_0$$

and prove (4.6). We now have $\mathrm{ufp}(\sigma_k) \le \sigma_k < \frac{3}{2}\sigma_0 \le 2\sigma_0$ by (2.12), and therefore $\mathrm{ufp}(\sigma_k) \le \mathrm{ufp}(2\sigma_0) = 2\mathrm{ufp}(\sigma_0)$. With (4.2) we obtain

$$|p'_k| \le \mathtt{eps} \cdot \mathrm{ufp}(\sigma_k) \le \mathtt{eps} \cdot \min\left(2 \cdot \mathrm{ufp}(\sigma_0), \frac{3}{2}\sigma_0\right), \tag{4.13}$$

in particular (4.4), and therefore

$$\sum_{i=1}^{n} |p'_i| \le \min\left(2n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0), \frac{3}{2}n \cdot \mathtt{eps} \cdot \sigma_0\right). \tag{4.14}$$

In the computation of $\widetilde{M}_1$, an underflow error can occur only in the final multiplication by σ_0 so that

$$\widetilde{M}_1 \ge (1 - \mathtt{eps})\left(\frac{3}{2} + 4\mathtt{eps}\right) n \cdot \mathtt{eps} \cdot \sigma_0 - \frac{\mathtt{eta}}{2} \ge \frac{3}{2}n \cdot \mathtt{eps} \cdot \sigma_0 - \frac{\mathtt{eta}}{2}. \tag{4.15}$$

We have $2n \cdot \mathtt{eps} \in \mathbb{F}$, and since $\mathrm{ufp}(\sigma_0)$ is a power of 2, the product $\mathrm{fl}\big((2n \cdot \mathtt{eps}) \cdot \mathrm{ufp}(\sigma_0)\big)$ may produce only an underflow error. This means

$$|\widetilde{M}_2 - 2n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0)| \le \frac{\mathtt{eta}}{2}. \tag{4.16}$$

If $4T' \le \mathtt{eps}^{-1}\mathtt{eta}$, then using (4.15), (4.16), and (4.14) yields

$$\frac{1}{4}\mathtt{eps}^{-1}\mathtt{eta} \ge T' \ge \min\left(\frac{3}{2}n \cdot \mathtt{eps} \cdot \sigma_0,\ 2n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0)\right) - \frac{\mathtt{eta}}{2} \ge \sum_{i=1}^{n} |p'_i| - \frac{\mathtt{eta}}{2},$$

and (3.2) implies (4.8).

We finally suppose $4T' > \mathtt{eps}^{-1}\mathtt{eta}$. Then $2T' > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ and $2T' \in \mathbb{F}$ imply

$$2T' \ge \mathrm{succ}\left(\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}\right) = \left(\frac{1}{2}\mathtt{eps}^{-1} + 1\right)\mathtt{eta} \qquad \text{or} \qquad \frac{\mathtt{eta}}{2} \le \frac{2\mathtt{eps} \cdot T'}{1 + 2\mathtt{eps}}. \tag{4.17}$$

Furthermore,

$$2n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0) \ge \widetilde{M}_2 - \frac{\mathtt{eta}}{2} \ge T' - \frac{\mathtt{eta}}{2} > \left(\frac{\mathtt{eps}^{-1}}{4} - \frac{1}{2}\right)\mathtt{eta}$$

with (4.16), and the assumption $(4n + 2)\mathtt{eps} \le 1$ yields $2n \cdot \mathtt{eps} \le \frac{1}{2}(1 - 2\mathtt{eps})$ and

$$\sigma_0 \ge \mathrm{ufp}(\sigma_0) > \frac{2}{1 - 2\mathtt{eps}} \cdot \frac{1}{2}\mathtt{eps}^{-1}\left(\frac{1}{2} - \mathtt{eps}\right)\mathtt{eta} = \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}.$$

This proves the first part of (4.9), and it implies $\widetilde{M}_2 = 2n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0)$. We distinguish the two cases $T' = \widetilde{M}_1$ and $T' = \widetilde{M}_2$. If $T' = \widetilde{M}_2$, then (4.14) implies the second half

of (4.9). It remains the case $T' = \widetilde{M}_1$. If $\widetilde{M}_1 \notin \mathbb{U}$, then the standard estimation (2.8) yields

$$\widetilde{M}_1 \geq (1-\mathtt{eps})^2 \left(\frac{3}{2} + 4\mathtt{eps}\right)(n\cdot\mathtt{eps})\sigma_0 \geq \frac{3}{2} n\cdot\mathtt{eps}\cdot\sigma_0$$

so that (4.14) proves (4.9). Hence finally the case $T' = \widetilde{M}_1 \in \mathbb{U}$ remains. This means $\frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta} \geq T' > \frac{1}{4}\mathtt{eps}^{-1}\mathtt{eta}$. Then with the first inequality in (4.15) and with (4.17) it follows that

$$T' = \widetilde{M}_1 \geq \left(\frac{3}{2} + \frac{5}{2}\mathtt{eps} - 4\mathtt{eps}^2\right) n\cdot\mathtt{eps}\cdot\sigma_0 - \frac{2\mathtt{eps}\cdot T'}{1+2\mathtt{eps}}$$

so that

$$\begin{aligned} T' &\geq \frac{(1+2\mathtt{eps})\left(\frac{3}{2}+\frac{5}{2}\mathtt{eps}-4\mathtt{eps}^2\right)}{1+4\mathtt{eps}} n\cdot\mathtt{eps}\cdot\sigma_0 \geq \frac{\frac{3}{2}+\frac{11}{2}\mathtt{eps}}{1+4\mathtt{eps}} n\cdot\mathtt{eps}\cdot\sigma_0 \\ &= \tfrac{3}{2}\left(1 - \frac{\frac{1}{3}\mathtt{eps}}{1+4\mathtt{eps}}\right) n\cdot\mathtt{eps}\cdot\sigma_0 > \tfrac{3}{2}\left(1-\tfrac{1}{3}\mathtt{eps}\right) n\cdot\mathtt{eps}\cdot\sigma_0 \\ &= \left(1+\tfrac{1}{2}(1-\mathtt{eps})\right) n\cdot\mathtt{eps}\cdot\sigma_0. \end{aligned}$$

Hence (4.13) and the refined estimation (4.6) yield

$$\sum_{i=1}^{n} |p_i'| \leq \mathtt{eps}\sum_{i=1}^{n} \mathrm{ufp}(\sigma_i) \leq \mathtt{eps}\sum_{i=1}^{n}\left(\sigma_0 + |\sigma_i - \sigma_0|\right) \leq n\cdot\mathtt{eps}\cdot\sigma_0\left(1+\frac{1}{2}(1-\mathtt{eps})\right) \leq T'$$

and show the second part of (4.9). The lemma is proved. □

Now we state our new summation algorithm. To ease the analysis, we first give a preliminary version with superindices to identify all used quantities uniquely.

ALGORITHM 4.6 (fast summation with faithfully rounded result, preliminary version).

function $\mathtt{res} = \mathtt{FastAccSum'}(p^{(0)})$
  $n = \mathrm{length}(p^{(0)})$
  $T^{(0)} = \mathrm{fl}\left(\left(\sum_{i=1}^{n} |p_i^{(0)}|\right)/(1 - n\cdot\mathtt{eps})\right)$
  $t^{(0)} = 0,\ m = 0$
  repeat
    $m = m + 1$
    $\sigma_0^{(m)} = \mathrm{fl}\big((2T^{(m-1)})/(1-(3n+1)\mathtt{eps})\big)$
    $[\sigma_n^{(m)}, p^{(m)}] = \mathtt{ExtractVectorNew}(\sigma_0^{(m)}, p^{(m-1)})$
    $\tau^{(m)} = \mathrm{fl}(\sigma_n^{(m)} - \sigma_0^{(m)})$
    $t^{(m)} = \mathrm{fl}(t^{(m-1)} + \tau^{(m)})$
    $\Phi^{(m)} = \mathrm{fl}\left(\left((2n(n+2)\mathtt{eps})\cdot\mathrm{ufp}\left(\sigma_0^{(m)}\right)\right)/(1-5\mathtt{eps})\right)$
    $T^{(m)} = \min\left(\mathrm{fl}\left(\left(\left(\tfrac{3}{2}+4\mathtt{eps}\right)\cdot(n\cdot\mathtt{eps})\right)\cdot\sigma_0^{(m)}\right), \mathrm{fl}\left((2n\cdot\mathtt{eps})\cdot\mathrm{ufp}\left(\sigma_0^{(m)}\right)\right)\right)$
  until $|t^{(m)}| \geq \Phi^{(m)}$ or $4T^{(m)} \leq \mathtt{eps}^{-1}\mathtt{eta}$
  $[\tau_1, \tau_2] = \mathtt{FastTwoSum}(t^{(m-1)}, \tau^{(m)})$ % $\tau_1 = t^{(m)}$
  $\mathtt{res} = \mathrm{fl}\left(\tau_1 + \left(\tau_2 + \left(\sum_{i=1}^{n} p_i^{(m)}\right)\right)\right)$

The algorithm works as follows. As described before, a chunk of bits is extracted out of all p_i. In contrast to Algorithm `AccSum` (Algorithm 4.5 in [37]), the splitting is not by a constant σ but by individual σ_{k-1} for each p_k. As by (4.5) the splitting is an error-free transformation of the vector p into σ_0, σ_n, and a new vector p' so that $\sum p_i = \sigma_n - \sigma_0 + \sum p'_i$ provided $|p_k| \le \sigma_{k-1}$. Therefore σ_0 is to be chosen large enough.

Another difference to `AccSum` is that the initialization of σ_0 depends on $\sum |p_i|$ rather than on $n \cdot \max(|p_i|)$. This improvement was already used in [39]. Occasionally this may save one extraction; see section 6.

Moreover, the difference $\sigma_n - \sigma_0$ can be forced to be without rounding error if only σ_0 is large enough. The smaller σ_0, the larger in terms of bits is the chunk to be extracted. Our choice is

$$\sigma_0^{(m)} = \mathrm{fl}\big((2T^{(m-1)})/(1-(3n+1)\mathtt{eps})\big).$$

It is easy to construct examples showing that $\mathrm{fl}(\sigma_n - \sigma_0) \neq \sigma_n - \sigma_0$ may happen when replacing the constant 2 by 1.9999. So basically σ_0 is chosen optimal. This implies $\sum p_i = \tau + \sum p'_i$.

The algorithm terminates if either the extracted vectors are near underflow or $|t^{(m)}| \ge \Phi^{(m)}$. The first case is less critical; as we will see, then the computed result is even the exact sum rounded to nearest.

The main stopping criterion has to serve two tasks: First, if $|t^{(m)}|$ is large, then not much cancelation occurred and the floating-point computation of `res` should be a faithful rounding of the sum. In particular, $\Phi^{(m)}$ is chosen so that the rounding errors in $\mathrm{fl}(\sum p_i^{(m)})$ are just not too large. The larger $\Phi^{(m)}$, the better the accuracy and the less extractions are necessary.

Second, if $|t^{(m)}|$ is small, then there is too much cancelation and another extraction is necessary. If $\Phi^{(m)}$ is small enough, we will show that $t^{(m)} = \mathrm{fl}(t^{(m-1)} + \tau^{(m)})$ does not cause a rounding error so that $\sum p_i^{(m-1)} = t^{(m)} + \sum p_i^{(m)}$ in this case. In any case, $T^{(m)}$ and $\Phi^{(m)}$ should be as small as possible because the number of extractions depends directly on them.

The following example shows that $|t^{(m)}| \ge \Phi^{(m)}$ cannot be replaced by $|t^{(m)}| \ge 0.53024\Phi^{(m)}$ without jeopardizing the faithful rounding of the final result `res`. Consider the vector $p \in \mathbb{F}^{62}$ in IEEE 754 double precision with $\mathtt{eps} = 2^{-53}$ and

$$p_1 = 2^{51} - 2060,\ \ p_{61} = -(832 + 115712 \cdot \mathtt{eps})\mathtt{eps},\ \ p_{62} = -1/(4\mathtt{eps}) - 46.5,$$

and $p_i = 1 - \mathtt{eps} \cdot q_i$ for $2 \le i \le 60$, where

$$\begin{aligned}
q_{2\cdots20} &= [\ -92\ \ 1\ \ 3\ \ 2\ \ 9\ \ 4\ \ 4\ \ 10\ \ 11\ \ 8\ \ 8\ \ 8\ \ 18\ \ 22\ \ 8\ \ 8\ \ 17\ \ 21\ \ 16\ ],\\
q_{21\cdots40} &= [\ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 16\ \ 43\ \ 42\ \ 86\ \ 32\ \ 32\ \ 32\ \ 32\ ],\\
q_{41\cdots60} &= [\ 32\ \ 32\ \ 32\ \ 32\ \ 7\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 32\ \ 50\ ].
\end{aligned}$$

The true sum is $s = -2047.5 - (2058 + 115712 \cdot \mathtt{eps})\mathtt{eps}$, and the rounded to nearest sum is $\tilde{s} = -2047.5 - 2^{-42}$. After one extraction the condition $|t^{(1)}| \ge \Phi^{(1)}$ is not satisfied, so Algorithm 4.1 (`FastAccSum`) performs a second (and final) extraction. However, the condition $|t^{(1)}| \ge 0.53024\Phi^{(1)}$ is satisfied, and the computed result without the second extraction would be $\mathtt{res}^* = -2047.5 = \mathrm{succ}(\tilde{s})$. However, $s - \mathrm{pred}(\mathtt{res}^*) < 0$ and $s - \mathrm{succ}(\mathtt{res}^*) < 0$ so that $\mathtt{res}^*$ is not a faithful rounding of s by Definition 3.1.

It follows that there is not much room for improvement in the stopping criterion of Algorithm 4.1. We mention that a condition like $s - \mathrm{pred}(\mathtt{res}^*) < 0$ is easily checked with `FastAccSum` : It is true if and only if $\mathtt{FastAccSum}(\tilde{p}) < 0$, where $\tilde{p}$ is the vector p with $-\mathrm{pred}(\mathtt{res}^*)$ appended. We also mention that the construction of the example is obstructed by the fact that changing a single component p_i usually changes the behavior of the entire algorithm.

We begin the analysis of Algorithm 4.6 with some preliminary results. Henceforth the letter m is reserved for the final value of the iteration counter; for intermediate values $1 \le k \le m$ we always use the letter k.

LEMMA 4.7. *For all* $1 \le k \le m$ *in Algorithm* 4.6 *(*`FastAccSum`*′) we have*

$$\tau^{(k)} = \mathrm{fl}(\sigma_n^{(k)} - \sigma_0^{(k)}) = \sigma_n^{(k)} - \sigma_0^{(k)} \in \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(k)}\right)\mathbb{Z}. \tag{4.18}$$

Proof. Using (3.3) and recursively applying Lemma 4.5 we obtain $\mathrm{fl}(\sigma_n^{(k)} - \sigma_0^{(k)}) = \sigma_n^{(k)} - \sigma_0^{(k)}$, and (4.18) follows by (2.16). ∎

LEMMA 4.8. *If* $4T^{(k)} > \mathtt{eps}^{-1}\mathtt{eta}$ *and* $2n(n+2)\mathtt{eps} \le 1$ *and* $\mathtt{eps} \le \frac{1}{128}$*, then*

$$\sigma_0^{(k)} \notin \mathbb{U}, \qquad \Phi^{(k)} \notin \mathbb{U}, \qquad \textit{and} \qquad \sigma_0^{(k+1)} \le 5n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0)^{(k)} \le \sigma_0^{(k)}. \tag{4.19}$$

Remark. Note that (4.19) implies that $\sigma_0^{(k)}$ and therefore $\Phi^{(k)}$ decrease so that Algorithm 4.6 terminates.

Proof of Lemma 4.8. By (4.9) we know $\sigma_0^{(k)} \notin \mathbb{U}$, and therefore (2.8) and $\mathrm{ufp}(\sigma_0^{(k)})$ being a power of 2 give

$$\mathrm{fl}\left((2n \cdot \mathtt{eps}) \cdot \mathrm{ufp}\left(\sigma_0^{(k)}\right)\right) = 2n \cdot \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(k)}\right) \ge T^{(k)} > \frac{1}{4}\mathtt{eps}^{-1}\mathtt{eta}. \tag{4.20}$$

Hence $n \ge 2$ and

$$2n(n+2)\mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(k)}\right) > \mathtt{eps}^{-1}\mathtt{eta}$$

imply $\Phi^{(k)} \notin \mathbb{U}$. Furthermore, $2T^{(k)} \notin \mathbb{U}$ so that with (2.8), (4.20), $2n(n+2)\mathtt{eps} \le 1$, and $\mathtt{eps} \le \frac{1}{128}$ we obtain

$$\begin{aligned} \sigma_0^{(k+1)} &\le (1+\mathtt{eps})\frac{2T^{(k)}}{1-(3n+1)\mathtt{eps}} \le \frac{4n \cdot \mathtt{eps}}{1-(3n+2)\mathtt{eps}}\mathrm{ufp}\left(\sigma_0^{(k)}\right) \\ &\le 5n \cdot \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(k)}\right) \le \sigma_0^{(k)}, \end{aligned} \tag{4.21}$$

and this proves (4.19) and the lemma. ∎

LEMMA 4.9. *Let* $1 \le k < m$ *be given so that*

$$|t^{(k)}| < \Phi^{(k)} \quad \textit{and} \quad 4T^{(k)} > \mathtt{eps}^{-1}\mathtt{eta}. \tag{4.22}$$

Suppose $(2n^2+4n+6)\mathtt{eps} \le 1$*. Then for* $1 \le k < m$*,*

$$t^{(k)} = t^{(k-1)} + \tau^{(k)} \quad \textit{and} \quad t^{(k)} \in \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(k+1)}\right)\mathbb{Z} \tag{4.23}$$

and

$$s = \sum_{i=1}^{n} p_i^{(0)} = t^{(k)} + \sum_{i=1}^{n} p_i^{(k)} = t^{(m-1)} + \tau^{(m)} + \sum_{i=1}^{n} p_i^{(m)}. \tag{4.24}$$

Furthermore, the assumptions of Lemma 3.4 *are satisfied and* `FastTwoSum` *is applicable. In particular, this implies*

$$s = \tau_1 + \tau_2 + \sum_{i=1}^{n} p_i^{(m)} \tag{4.25}$$

and

$$|\tau_2| \le \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) \qquad \textit{and} \qquad |\tau_2| \le \mathtt{eps} \cdot |t^{(m-1)} + \tau^{(m)}|. \tag{4.26}$$

Proof. For $k = 1$ we use (4.18), (2.13), and (4.19) to see

$$t^{(1)} = \tau^{(1)} \in \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(1)}\right)\mathbb{Z} \subseteq \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(2)}\right)\mathbb{Z}.$$

Assume by induction $t^{(k-1)} \in \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0^{(k)})\mathbb{Z}$. Then $\tau^{(k)} \in \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0^{(k)})\mathbb{Z}$ by (4.18), and $\Phi^{(k)} \notin \mathbb{U}$ by (4.19) gives

$$|t^{(k)}| = |\mathrm{fl}(t^{(k-1)} + \tau^{(k)})| < \Phi^{(k)} \le \frac{2n(n+2)\mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(k)}\right)}{(1-\mathtt{eps})(1-5\mathtt{eps})}$$

$$\le \frac{2n(n+2)\mathtt{eps}}{1-6\mathtt{eps}} \mathrm{ufp}\left(\sigma_0^{(k)}\right) \le \mathrm{ufp}\left(\sigma_0^{(k)}\right)$$

so that (2.18) proves the first part of (4.23). The second part follows as before by (4.19), (2.13), and

$$t^{(k)} \in \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(k)}\right) \subseteq \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(k+1)}\right).$$

Combining (4.5), (4.18), and (4.23) yields for $1 \le k < m$,

$$s = \sigma_n^{(1)} - \sigma_0^{(1)} + \sum_{i=1}^{n} p_i^{(1)} = \tau^{(1)} + \sum_{i=1}^{n} p_i^{(1)} = t^{(1)} + \sum_{i=1}^{n} p_i^{(1)}$$

$$= t^{(1)} + \tau^{(2)} + \sum_{i=1}^{n} p_i^{(2)} = t^{(2)} + \sum_{i=1}^{n} p_i^{(2)} = \cdots$$

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

By (4.6) we know $|\tau^{(m)}| = |\sigma_n^{(m)} - \sigma_0^{(m)}| \le \frac{1}{2}\sigma_0^{(m)}$ so that $|2\tau^{(m)}| \le \sigma_0^{(m)}$. Hence (4.23) and (2.13) give

$$t^{(m-1)} \in \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right)\mathbb{Z} \subseteq 2\mathtt{eps} \cdot \mathrm{ufp}\left(\tau^{(m)}\right)\mathbb{Z} \tag{4.27}$$

so that Lemma 3.4 and therefore `FastTwoSum` are applicable. This finishes the proof. ∎

There are two possibilities in the "until"-condition of Algorithm 4.6, namely, $|t^{(m)}| \ge \Phi^{(m)}$ is true, $4T^{(m)} \le \mathtt{eps}^{-1}\mathtt{eta}$, or both. We present the proof of our main

result, namely, that the result res of Algorithm 4.6 is a faithful rounding of the sum $s := \sum_{i=1}^{n} p_i$, in two parts: First, we assume $4T^{(m)} \le \mathtt{eps}^{-1}\mathtt{eta}$. As we will see, this covers all underflow issues, and res is even the correct sum rounded to nearest in this case. Second, we treat the case $|t^{(m)}| \ge \Phi^{(m)}$.

LEMMA 4.10. *Assume* $2n(n+2)\mathtt{eps} \le 1$ *and* $\mathtt{eps} \le \frac{1}{128}$. *If* $4T^{(m)} \le \mathtt{eps}^{-1}\mathtt{eta}$ *for the final value* m *in Algorithm* 4.6, *then the result* res *of Algorithm* 4.6 *is equal to the sum* $s := \sum_{i=1}^{n} p_i$ *rounded to nearest, i.e.,* $\mathtt{res} = \mathrm{fl}(s)$.

Proof. By (4.8) in Lemma 4.5 we have

$$\tilde{S} := \mathrm{fl}\left(\sum_{i=1}^{n} p_i^{(m)}\right) = \sum_{i=1}^{n} p_i^{(m)} \quad \text{and} \quad \sum_{i=1}^{n} |p_i^{(m)}| \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}. \tag{4.28}$$

If $m = 1$, then $t^{(1)} = \tau^{(1)}$ and $\tau_2 = 0$ so that (4.25) yields

$$s = \tau_1 + \sum_{i=1}^{n} p_i^{(1)} = \tau_1 + \tilde{S},$$

and $\mathtt{res} = \mathrm{fl}(\tau_1 + \tilde{S}) = \mathrm{fl}(s)$.

Suppose $4T^{(m)} \le \mathtt{eps}^{-1}\mathtt{eta}$ and $m > 1$, which means

$$|t^{(m-1)}| < \Phi^{(m-1)} \quad \text{and} \quad 4T^{(m-1)} > \mathtt{eps}^{-1}\mathtt{eta} \tag{4.29}$$

and

$$s = t^{(m-1)} + \tau^{(m)} + \sum_{i=1}^{n} p_i^{(m)} = \tau_1 + \tau_2 + \tilde{S} \tag{4.30}$$

by Lemma 4.9. We will prove $\mathrm{fl}(\tau_2 + \tilde{S}) = \tau_2 + \tilde{S}$. Since $2T^{(m-1)} \notin \mathbb{U}$, it follows by (2.8) that we have

$$\sigma_0^{(m)} \ge \frac{2T^{(m-1)}}{1 - 3n \cdot \mathtt{eps}} > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta} \qquad \text{and} \qquad \sigma_0^{(m)} \notin \mathbb{U}. \tag{4.31}$$

Furthermore, (4.9) implies $\sigma_0^{(m-1)} \notin \mathbb{U}$. Thus

$$\mathrm{fl}\left(\left(\left(\frac{3}{2} + 4\mathtt{eps}\right) \cdot (n \cdot \mathtt{eps})\right) \cdot \sigma_0^{(m-1)}\right) \ge \frac{3}{2} n \cdot \mathtt{eps} \cdot \sigma_0^{(m-1)},$$

$$\mathrm{fl}\left(2n \cdot \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m-1)}\right)\right) = 2n \cdot \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m-1)}\right),$$

and

$$\begin{aligned} T^{(m-1)} &\ge \min\left(\frac{3}{2} n \cdot \mathtt{eps} \cdot \sigma_0^{(m-1)}, 2n \cdot \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m-1)}\right)\right) \\ &\ge \frac{3}{2} n \cdot \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m-1)}\right). \end{aligned} \tag{4.32}$$

Moreover, $\Phi^{(m-1)} \notin \mathbb{U}$ by (4.29) and (4.19), and the standard estimation (2.8), (4.32), and (4.31) yield for $n \ge 2$

$$\begin{aligned} \Phi^{(m-1)} &\le \frac{2n(n+2)\mathtt{eps}}{1 - 6\mathtt{eps}} \mathrm{ufp}\left(\sigma_0^{(m-1)}\right) \le \frac{4(n+2)}{3(1-6\mathtt{eps})} T^{(m-1)} \\ &\le \frac{2}{3}(n+2)\frac{1 - 3n \cdot \mathtt{eps}}{1 - 6\mathtt{eps}} \sigma_0^{(m)} \le \frac{2}{3}(n+2)\sigma_0^{(m)}. \end{aligned} \tag{4.33}$$

By (4.7) and (4.6) we know

$$|\tau^{(m)}| = |\sigma_n^{(m)} - \sigma_0^{(m)}| < \frac{1}{2}\sigma_0^{(m)}$$

so that (4.29) and (4.33) imply

$$\begin{aligned}|t^{(m-1)} + \tau^{(m)}| < \Phi^{(m-1)} + |\tau^{(m)}| &\le \left(\frac{2}{3}(n+2) + \frac{1}{2}\right)\sigma_0^{(m)} \\ &= \left(\frac{2}{3}n + \frac{11}{6}\right)\sigma_0^{(m)}.\end{aligned}$$

Because $\sigma_0^{(m)} \notin \mathbb{U}$ by (4.31) we have $T^{(m)} \ge n \cdot \mathtt{eps} \cdot \sigma_0^{(m)}$, and by Lemma 3.4 and $4T^{(m)} \le \mathtt{eps}^{-1}\mathtt{eta}$ the error term τ_2 of $\mathtt{FastTwoSum}(t^{(m-1)}, \tau^{(m)})$ satisfies for $n \ge 2$,

$$\begin{aligned}|\tau_2| &\le \mathtt{eps} \cdot |t^{(m-1)} + \tau^{(m)}| \\ &\le \mathtt{eps} \cdot \left(\frac{2}{3}n + \frac{11}{6}\right)(n \cdot \mathtt{eps})^{-1} T^{(m)} \\ &\le \frac{4n+11}{6n} \cdot \frac{1}{4}\mathtt{eps}^{-1}\mathtt{eta} \\ &< \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}.\end{aligned}$$

Thus $|\tau_2| + \sum |p_i^{(m)}| < \mathtt{eps}^{-1}\mathtt{eta}$ by (4.28) so that (3.2) proves

$$\mathrm{fl}\left(\tau_2 + \left(\sum_{i=1}^{n} p_i^{(m)}\right)\right) = \tau_2 + \sum_{i=1}^{n} p_i^{(m)} = \tau_2 + \tilde{S} := \tilde{T}.$$

Hence $s = \tau_1 + \tilde{T}$ by (4.30), and $\mathtt{res} = \mathrm{fl}(\tau_1 + \tilde{T}) = \mathrm{fl}(s)$. ∎

THEOREM 4.11. *Let $p_i \in \mathbb{F}$ for $1 \le i \le n$ be given, and assume $(2n^2 + 4n + 6)\mathtt{eps} \le 1$ and $\mathtt{eps} \le \frac{1}{128}$. Assume that no overflow occurs. Then the result* res *of Algorithm* 4.6 *is a faithful rounding of the sum $s := \sum_{i=1}^{n} p_i$.*

Remark 3. We note that in IEEE 754 double precision this restricts the vector length to $n \le 67,108,862$.

Remark 4. The result of `FastAccSum` is faithful but need not be rounded to nearest. In several billion test cases this never happened, although there are examples such as $p = [1\ \mathtt{eps}\ \mathtt{eps}^2]$, for which `FastAccSum` (as `AccSum`) produces $\mathtt{res} = 1$.

Remark 5. Also note that in general the intermediate results in $\mathtt{FastAccSum}(p)$ and $\mathtt{FastAccSum}(-p)$ are very different (in absolute value).

Proof of Theorem 4.11. If $4T^{(m)} \le \mathtt{eps}^{-1}\mathtt{eta}$ is satisfied for the final value m in Algorithm 4.6, then Lemma 4.10 proves that res is even the true sum rounded to nearest.

Henceforth we may assume $4T^{(m)} > \mathtt{eps}^{-1}\mathtt{eta}$. Then $\Phi^{(m)} \notin \mathbb{U}$ and $\sigma_0^{(m)} \notin \mathbb{U}$ by (4.19) so that we can use the standard estimation (2.7) for the computation of $\Phi^{(m)}$ and $\sigma_0^{(m)}$ without underflow constant.

We have $|t^{(m)}| \geq \Phi^{(m)}$ for the final value m, and we abbreviate

$$p'_i := p_i^{(m)} \quad \text{and} \quad \sigma_0 := \sigma_0^{(m)}.$$

Then (4.25) in Lemma 4.9 gives

$$s = \tau_1 + \tau_2 + \sum_{i=1}^{n} p'_i.$$

Define $\tau_3, \delta_3, \tau'_2$, and δ_2 according to

$$\begin{aligned} \tau_3 &:= \mathrm{fl}\left(\sum_{i=1}^{n} p'_i\right) = \sum_{i=1}^{n} p'_i - \delta_3, \\ \tau'_2 &:= \mathrm{fl}(\tau_2 + \tau_3) = \tau_2 + \tau_3 - \delta_2. \end{aligned} \tag{4.34}$$

Then

$$\mathtt{res} = \mathrm{fl}(\tau_1 + \tau'_2) \quad \text{and} \quad s = \tau_1 + \tau_2 + \sum p'_i = \tau_1 + \tau'_2 + \delta_2 + \delta_3$$

so that

$$\mathtt{res} = \mathrm{fl}(r) \quad \text{and} \quad s = r + \delta \quad \text{for} \quad r := \tau_1 + \tau'_2 \quad \text{and} \quad \delta := \delta_2 + \delta_3. \tag{4.35}$$

We will prove $\mathtt{res} \notin \mathbb{U}$ and $2|\delta| < \mathtt{eps} \cdot |\mathtt{res}|$ in order to apply Lemma 3.2. First, (4.4) gives

$$|p'_k| \leq 2\mathtt{eps} \cdot \mathrm{ufp}(\sigma_0)$$

so that (2.17) implies

$$|\tau_3| \leq 2n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0) \quad \text{and} \quad |\delta_3| \leq n(n-1)\mathtt{eps}^2 \cdot \mathrm{ufp}(\sigma_0). \tag{4.36}$$

Next

$$|\delta_2| \leq \mathtt{eps} \cdot |\tau_2 + \tau_3| \leq \mathtt{eps}^2 \cdot \mathrm{ufp}(\tau_1) + \mathtt{eps} \cdot |\tau_3| \tag{4.37}$$

by (2.5) and (4.26), and

$$|\tau'_2| \leq \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) + |\tau_3| + |\delta_2| \leq \mathtt{eps}(1 + \mathtt{eps}) \cdot \mathrm{ufp}(\tau_1) + (1 + \mathtt{eps})|\tau_3|. \tag{4.38}$$

Hence (4.37) and (4.36) yield

$$\mathtt{eps}^{-1}|\delta| \leq \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) + |\tau_3| + n(n-1)\mathtt{eps} \cdot \mathrm{ufp}(\sigma_0). \tag{4.39}$$

The definition of $\mathtt{res}$ and (4.38) imply

$$\begin{aligned} |\mathtt{res}| &\geq (1 - \mathtt{eps})(|\tau_1| - |\tau'_2|) \\ &\geq (1 - \mathtt{eps})\big(|\tau_1|(1 - \mathtt{eps}(1 + \mathtt{eps})) - (1 + \mathtt{eps})|\tau_3|\big) \\ &\geq (1 - 2\mathtt{eps})|\tau_1| - |\tau_3|. \end{aligned} \tag{4.40}$$

We know $\sigma_0 \notin \mathbb{U}$ and therefore $T^{(m)} \leq \mathrm{fl}(2n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0)) = 2n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0)$, and

$$|\tau_1| = |t^{(m)}| \geq \Phi^{(m)} > \frac{2n(n+2)}{1 - 4\mathtt{eps}}\mathtt{eps} \cdot \mathrm{ufp}(\sigma_0). \tag{4.41}$$

We first use (4.40), (4.41), (4.36), (4.20), and $n \geq 2$ to see

$$|\mathtt{res}| > \left((1-2\mathtt{eps})\frac{2n(n+2)}{1-4\mathtt{eps}} - 2n\right)\mathtt{eps}\cdot \mathrm{ufp}(\sigma_0) \geq 2n^2\mathtt{eps}\cdot\mathrm{ufp}(\sigma_0)$$

$$\geq nT^{(m)} > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta} \tag{4.42}$$

so that $\mathtt{res} \notin \mathbb{U}$. Then we use (4.40), (4.39), (4.36), and (4.41) to conclude

$$\begin{aligned}|\mathtt{res}| - 2\mathtt{eps}^{-1}|\delta| &\geq (1-2\mathtt{eps})|\tau_1| - |\tau_3| - 2\mathtt{eps}|\tau_1| - 2|\tau_3| - 2n(n-1)\mathtt{eps}\cdot\mathrm{ufp}(\sigma_0)\\ &\geq (1-4\mathtt{eps})|\tau_1| - (6n\cdot\mathtt{eps} + 2n(n-1)\mathtt{eps})\cdot\mathrm{ufp}(\sigma_0)\\ &= (1-4\mathtt{eps})|\tau_1| - 2n(n+2)\mathtt{eps}\cdot\mathrm{ufp}(\sigma_0) > 0.\end{aligned}$$

Lemma 3.2 finishes the proof. ∎

It is instructive to interpret Algorithm `FastAccSum` in terms of the condition number of summation. The latter is defined for $\sum p_i \neq 0$ by

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

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

$$\mathrm{cond}\left(\sum p_i\right) = \frac{\sum|p_i|}{|\sum p_i|}. \tag{4.43}$$

As by (4.24), each extraction transforms the sum $s = \sum p_i^{(0)}$ into an approximation $t^{(k)}$ and a new vector $p_i^{(k)}$ without error. With our estimations we can see that the condition number of the sum of the new vector diminishes in each step so that the problem becomes simpler and simpler.

THEOREM 4.12. *Denote by $t^{(k)}$ and $p_i^{(k)} \in \mathbb{F}$ for $1 \leq i \leq n$ the intermediate results of Algorithm* 4.6 *(*`FastAccSum`*) after the kth extraction. Assume that no underflow occurred and $|t^{(k)}| < \Phi^{(k)}$. Then*

$$\mathrm{cond}\left(t^{(k)} + \sum p_i^{(k)}\right) \ \leq\ \frac{2n+5}{1-(2n+6)\mathtt{eps}}\,(\varphi n\cdot\mathtt{eps})^k\,\mathrm{cond}\left(\sum p_i^{(0)}\right), \tag{4.44}$$

where $\varphi = \frac{3+16\mathtt{eps}}{1-(3n+2)\mathtt{eps}}$.

Remark 6. The estimation (4.44) means essentially that each extraction diminishes the condition number of the sum by a factor $3n\cdot\mathtt{eps}$. As we will see, in practice this factor is smaller.

Remark 7. Note that the assumptions and (4.24) imply $s = \sum p_i^{(0)} = t^{(k)} + \sum p_i^{(k)}$.

Proof of Theorem 4.12. The standard estimation (2.8), the definition of $T^{(0)}$, and (3.1) imply

$$T^{(0)} \leq \frac{\mathrm{fl}(\sum_{i=1}^n |p_i^{(0)}|)}{(1-n\cdot\mathtt{eps})(1-\mathtt{eps})} \leq \frac{\sum_{i=1}^n |p_i^{(0)}|}{(1-(n+1)\mathtt{eps})(1-(n-1)\mathtt{eps})}$$

so that

$$\sum_{i=1}^n |p_i^{(0)}| \geq (1-2n\cdot\mathtt{eps})T^{(0)}. \tag{4.45}$$

Furthermore, $\mathrm{fl}\left(2n\cdot\mathtt{eps}\cdot\mathrm{ufp}\left(\sigma_0^{(k)}\right)\right) = 2n\cdot\mathtt{eps}\cdot\mathrm{ufp}\left(\sigma_0^{(k)}\right) > n\cdot\mathtt{eps}\cdot\sigma_0^{(k)}$ and

$$|t^{(k)}| < \Phi^{(k)} \le \frac{2n(n+2)\mathtt{eps}}{(1-\mathtt{eps})(1-5\mathtt{eps})}\,\mathrm{ufp}\left(\sigma_0^{(k)}\right) \le \frac{2n(n+2)\mathtt{eps}}{1-6\mathtt{eps}}\,\sigma_0^{(k)} \le \frac{2(n+2)}{1-6\mathtt{eps}}\,T^{(k)}$$

so that (4.9) yields

$$|t^{(k)}| + \sum |p_i^{(k)}| \le \frac{2n+5}{1-6\mathtt{eps}}\,T^{(k)}. \tag{4.46}$$

On the other hand,

$$\begin{aligned} T^{(k)} &\le (1+\mathtt{eps})^2\left(\frac{3}{2}+4\mathtt{eps}\right) n\cdot\mathtt{eps}\cdot\sigma_0^{(k)} \le \left(\frac{3}{2}+8\mathtt{eps}\right) n\cdot\mathtt{eps}\cdot\sigma_0^{(k)} \\ &\le \frac{\frac{3}{2}+8\mathtt{eps}}{1-(3n+2)\mathtt{eps}} 2n\cdot\mathtt{eps}\cdot T^{(k-1)} \\ &= \varphi n\cdot\mathtt{eps}\cdot T^{(k-1)}. \end{aligned}$$

Therefore (4.45) and (4.46) imply

$$\begin{aligned} \frac{\mathrm{cond}\,(t^{(k)}+\sum p_i^{(k)})}{\mathrm{cond}\,(\sum p_i^{(0)})} &\le \frac{|t^{(k)}| + \sum |p_i^{(k)}|}{\sum |p_i^{(0)}|} \le \frac{(2n+5)T^{(k)}}{(1-(2n+6)\mathtt{eps})T^{(0)}} \\ &\le \frac{2n+5}{1-(2n+6)\mathtt{eps}}\,(\varphi n\cdot\mathtt{eps})^k, \end{aligned}$$

and the assertion follows. □

Our previous algorithm `AccSum` [37] follows the same principle as `FastAccSum`; i.e., the condition number of the sum decreases by some factor with each extraction. In practical examples we can verify that the factor for `FastAccSum` is smaller than for `AccSum`, namely,

$$\begin{aligned} &\frac{\mathrm{cond}\,(\sum p_i^{(k+1)})}{\mathrm{cond}\,(\sum p_i^{(k)})} \sim n\sqrt{n}\cdot\mathtt{eps} \quad \text{for } \texttt{AccSum} \quad \text{and} \\ &\frac{\mathrm{cond}\,(\sum p_i^{(k+1)})}{\mathrm{cond}\,(\sum p_i^{(k)})} \sim n\cdot\mathtt{eps} \quad \text{for } \texttt{FastAccSum}. \end{aligned} \tag{4.47}$$

In what follows we state the final version of Algorithm 4.6 (`FastAccSum`) without super-indices, expanding code of Algorithm 4.2 (`ExtractVectorNew`), Algorithm 3.5 (ufp), and Algorithm 3.3 (`FastTwoSum`) and other minor improvements. In particular, an intermediate sum may become zero, that is, $t^{(m)} = \mathrm{fl}(t^{(m-1)} + \tau^{(m)}) = 0$ in Algorithm 4.6. By (4.25) this implies $\mathtt{res} = \sum p_i^{(m)}$ so that `FastAccSum` can be applied recursively to the vector of lower order parts. Without this improvement the algorithm would continue to apply extractions to zero vectors until the stopping criterion $4T \le \mathtt{eps}^{-1}\mathtt{eta}$ is satisfied. If T is sufficiently near the underflow range, then by (3.2) there is no rounding error in the summation and the algorithm can terminate immediately.

ALGORITHM 4.13 (fast summation with faithfully rounded result, final version).

function $\mathtt{res} = \mathtt{FastAccSum}(p)$
  $n = \text{length}(p)$
  $T = \text{fl}\left(\left(\sum_{i=1}^{n} |p_i|\right)/(1 - n \cdot \mathtt{eps})\right)$ % $\sum |p_i| \le T$
  if $T \le \mathtt{eps}^{-1}\mathtt{eta}$ then $\mathtt{res} = \text{fl}\left(\sum_{i=1}^{n} p_i\right)$, return, end if % *no rounding error*
  $t' = 0$
  repeat
    $\sigma_0 = \text{fl}\left((2T)/(1 - (3n+1)\mathtt{eps})\right)$, $\sigma = \sigma_0$
    for $i = 1 : n$
      $\sigma' = \text{fl}(\sigma + p_i)$
      $q = \text{fl}(\sigma' - \sigma)$
      $p_i = \text{fl}(p_i - q)$
      $\sigma = \sigma'$
    end for % $[\sigma', p] = \mathtt{ExtractVectorNew}(\sigma_0, p)$
    $\tau = \text{fl}(\sigma' - \sigma_0)$ % *exact*
    $t = t'$, $t' = \text{fl}(t + \tau)$
    if $t' = 0$ then $\mathtt{res} = \mathtt{FastAccSum}(p)$, return, end if
    $q = \text{fl}\big((2\mathtt{eps})^{-1}\sigma_0\big)$, $u = \text{fl}\big(|q/(1 - \mathtt{eps}) - q|\big)$ % $u = \text{ufp}(\sigma_0)$
    $\Phi = \text{fl}\left(\big((2n(n+2)\mathtt{eps}) \cdot u\big)/(1 - 5\mathtt{eps})\right)$
    $T = \min\left(\text{fl}\left(\left(\left(\frac{3}{2} + 4\mathtt{eps}\right) \cdot (n \cdot \mathtt{eps})\right) \cdot \sigma_0\right), \text{fl}((2n \cdot \mathtt{eps}) \cdot u)\right)$ % $\sum |p_i| \le T$
  until $|t'| \ge \Phi$ or $4T \le \mathtt{eps}^{-1}\mathtt{eta}$
  $\tau_2 = \text{fl}((t - t') + \tau)$ % $[t', \tau_2] = \mathtt{FastTwoSum}(t, \tau)$
  $\mathtt{res} = \text{fl}\left(t' + \left(\tau_2 + \left(\sum_{i=1}^{n} p_i\right)\right)\right)$ % *faithfully rounded* $\sum p_i$

We summarize the properties of Algorithm 4.13 (FastAccSum).

PROPOSITION 4.14. *Let $p_i \in \mathbb{F}$ for $1 \le i \le n$ be given, and assume $(2n^2 + 4n + 6)\mathtt{eps} \le 1$ and $\mathtt{eps} \le \frac{1}{128}$. Assume that no overflow occurs. Then the result* res *of Algorithm* 4.13 *(*FastAccSum*) is a faithful rounding of the sum $s := \sum_{i=1}^{n} p_i$. Algorithm* 4.13 *requires $(3m + 3)n + \mathcal{O}(m)$ flops.*

The computed result res *is equal to the exact result $s = \sum p_i$ if s is a floating-point number or if* $\mathtt{res} \in \mathbb{U}$, *i.e.,*

$$s \in \mathbb{F} \quad \textit{or} \quad \mathtt{res} \in \mathbb{U} \quad \Rightarrow \quad \mathtt{res} = s. \tag{4.48}$$

In particular, $\mathtt{res} = 0$ *if and only if $s = 0$, and always*

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

The constant 2 *in the initialization $\sigma_0 = \text{fl}\left((2T)/(1 - (3n+1)\mathtt{eps})\right)$ cannot be replaced by* 1.9999, *and the stopping criterion $|t^{(m)}| \ge \Phi^{(m)}$ cannot be replaced by $|t^{(m)}| \ge 0.53024\Phi^{(m)}$ without jeopardizing the faithful rounding.*

Remark. Note that taking the absolute values in the computation of T comes without extra cost on today's architectures; nevertheless we count it as one floating-point operation.

On some architectures floating-point division is disproportionately slow. In that case one may replace the divisions in the computation of T, σ_0, and Φ by multiplications with $1 + (n+1)\mathtt{eps}$, $1 + (3n+2)\mathtt{eps}$, and $1 + 6\mathtt{eps}$, respectively.

Using Theorem 4.12 we can estimate up to which condition number a certain number of extractions is definitely sufficient. Suppose the right-hand side of (4.44) is less than 1; then formula (4.43) for the condition number implies $t^{(k)} = 0$, and the

TABLE 4.1
Minimum treatable condition number for faithful rounding with `m` *extractions.*

n	m=1	m=2	m=3	m=4
100	$1.46 \cdot 10^{11}$	$4.40 \cdot 10^{24}$	$1.32 \cdot 10^{38}$	$3.96 \cdot 10^{51}$
1000	$1.50 \cdot 10^{9}$	$4.50 \cdot 10^{21}$	$1.35 \cdot 10^{34}$	$4.05 \cdot 10^{46}$
10000	$1.50 \cdot 10^{7}$	$4.51 \cdot 10^{18}$	$1.35 \cdot 10^{30}$	$4.06 \cdot 10^{41}$
100000	$1.50 \cdot 10^{5}$	$4.51 \cdot 10^{15}$	$1.35 \cdot 10^{26}$	$4.06 \cdot 10^{36}$

TABLE 4.2
Number of floating-point operations for `m` *extractions.*

	m=1	m=2	m=3	m=4
`AccSum`	7n	11n	15n	19n
`FastAccSum`	6n	9n	12n	15n

algorithm stops with k extractions. Thus we can compute for given dimension n and number of extractions m the minimum treatable condition number.

This condition number is listed in Table 4.1. It means that for a vector of dimension n at most the displayed number m of extractions is necessary. Conversely, for sums with condition number $\texttt{eps}^{-1}$ at most 2 extractions or $8n$ flops suffice for dimensions up to about 80000. Comparing these results with Table 4.2 and Table 4.3 in [37] shows that `FastAccSum` performs better than `AccSum`. For example, the minimum treatable condition number for condition number $\texttt{eps}^{-1}$ in at most 2 extractions is only 4094 for `AccSum`.

By the bare count of floating-point operations, `AccSum` needs $(4m + 3)n$ flops, whereas `FastAccSum` needs $(3m+3)n$ flops for m extractions (see Table 4.2). Basically, both algorithms need the same number of extractions to compute a faithfully rounded result, but sometimes `FastAccSum` needs one less. As has been noted before, `AccSum` initializes σ_0 depending on $\sum |p_i|$ rather than on $n \cdot \max(|p_i|)$. As shown below, this improvement may save occasionally one extraction.

First, for fixed dimension $n = 1000$ we ran 10000 random test cases for condition numbers ranging from 10^5 to 10^{50}. Random sums with prescribed condition number are generated with Algorithm 6.1 in [27] adapted to summation. In the left picture of Figure 4.1 we display the percentage of cases where `FastAccSum` saves one extraction over `AccSum`. As can be seen, this is the case for condition numbers near 10^7, 10^{20}, 10^{33}, and 10^{46} in up to 90% of all cases. This will be reflected in the following ratios of computing time of `FastAccSum` over `AccSum`.

The reason why `FastAccSum` occasionally needs one less extraction than `AccSum` is that for fixed dimension and increasing condition number the ratio $|t^{(m)}/\Phi^{(m)}|$ decreases until a next extraction is necessary; i.e., the value of m is increased. The use of $\sum |p_i|$ in `FastAccSum` instead of $n \cdot \max(|p_i|)$ as in `AccSum` is at certain values of the condition number sufficient to save one extraction. We never encountered a case where `AccSum` needed less extractions than `FastAccSum`. Another reason is (4.47).

In the right picture of Figure 4.1 we display the same results for dimensions $n = 100, 300, 1000, 3000, 10000$, all in one picture. Changing the dimension changes the initial ratio of $|t^{(m)}/\Phi^{(m)}|$. Thus we observe a similar behavior but shifted in the condition number.

We finally mention that the inner loop of Algorithm 4.13 (`FastAccSum`) can be fastened; namely, the statement $\sigma = \sigma_0$ is suboptimal for today's architectures. To improve instruction-level parallelism the following inner loop can be used.

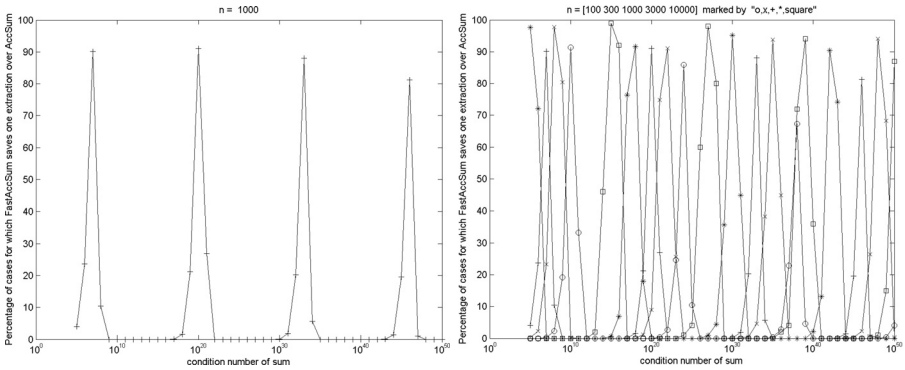

FIG. 4.1. *Percentage of cases where* `FastAccSum` *saves one extraction over* `AccSum`.

ALGORITHM 4.15 (improved inner loop for Algorithm 4.13 (`FastAccSum`)).

$\sigma = \sigma_0$
for $i = 1 : 2 : n$
  $\sigma' = \mathrm{fl}(\sigma + p_i)$
  $q = \mathrm{fl}(\sigma' - \sigma)$
  $p_i = \mathrm{fl}(p_i - q)$
  $\sigma = \mathrm{fl}(\sigma' + p_{i+1})$
  $q = \mathrm{fl}(\sigma - \sigma')$
  $p_{i+1} = \mathrm{fl}(p_{i+1} - q)$
end for $[\sigma, p] = \texttt{ExtractVectorNew}(\sigma_0, p)$

The values of σ and σ_0 are interchanged in every pair of steps. One takes care of odd n, for example, by setting $p_{n+1} = 0$ and increasing n by 1. Note that for all constants the original value of n can be used. The effect for the instruction-level parallelism is similar to Algorithm `AccSum` in [37] as analyzed by Langlois [22].

5. Algorithm `FastPrecSum` with result "as if" computed in K-fold precision. A drawback of Algorithm `FastAccSum` is that an extra vector is needed if the input vector cannot be overridden. If the dimension is large or the vector elements pop out of a computation, this may be a problem. There seems hardly a way to compute a faithfully rounded result without an extra vector if not utilizing the limitation of the exponent range like Malcolm's algorithm [25].

This changes when not insisting on a faithfully rounded result but accepting a result "as if" computed in K-fold precision. In the following we modify Algorithm `FastAccSum` to perform a fixed number of $K - 1$ extractions, similar to Algorithm `PrecSum` in [39]. This modification allows us to rearrange the computation and thus avoid memory for an extra vector.

In `FastAccSum` the input vector $p_i^{(0)}$ is transformed into the vectors $p_i^{(1)}$, $p_i^{(2)}$, etc. until the final vector $p_i^{(m)}$. One may think of this process as an array of $n \times m$ numbers. All elements $p_i^{(k)}$ are uniquely determined for $1 \le i \le n$ and $1 \le k \le m$, and all transformations are error-free. This means we may go through this scheme in any order without changing the result. In the following Algorithm 5.1 (`FastPrecSum`) we compute this array row by row rather than column by column so that only $\mathcal{O}(K)$ additional memory is needed. Since K is usually small, basically no extra memory is

needed. As we will see, the new algorithm is faster than competitors such as `PrecSum` [39], `SumXBLAS` [23], or `SumK` [27].

In contrast to Algorithm `FastAccSum`, the number of extractions in `FastPrecSum` is not determined by the difficulty, i.e., condition number of the sum, but fixed in advance. The result of Algorithm `FastPrecSum` is a faithfully rounded result except when the problem is too difficult in view of the fixed number of extractions. The analysis is similar to that of `FastAccSum`; however, special care is necessary to prove that the result is faithful for *well-conditioned* problems.

The algorithm is formulated with superscripts where needed for the analysis. Of course, as for `FastAccSum`, those are omitted in a practical implementation. In particular, $\tilde{p}_i$ needs no index, and only one variable $\tilde{p}$ is needed.

We first collect some properties. If the *ExactFlag* is set, then by Lemma 4.10 the floating-point sum of the extracted vector is equal to the true sum. Henceforth we may assume that no underflow occurs.

Suppose for all $1 \le m \le K-1$ the stopping criterion $|t^{(m)}| \ge \Phi^{(m)}$ in Algorithm `FastAccSum` is not satisfied. Then Lemma 4.9 implies $\mathrm{fl}(t + \tau^{(m)}) = t + \tau^{(m)}$ for all m and

$$s = \sum p_i = t^{(m)} + \sum_{i=1}^{n} \tilde{p}_i \tag{5.1}$$

for the final value of $t^{(m)}$. In this case the result may not be faithfully rounded, but we will prove that it is of a quality "as if" calculated in about K-fold precision.

Suppose m is the first index for which $|t^{(m)}| \ge \Phi^{(m)}$ is satisfied. Carefully comparing `FastAccSum` and `FastPrecSum` it follows that

$$s := \sum_{i=1}^{n} p_i = t^{(m-1)} + \sum_{\nu=m}^{K-1} \tau^{(\nu)} + \sum_{i=1}^{n} \tilde{p}_i. \tag{5.2}$$

ALGORITHM 5.1 (fast summation with result "as if" computed in K-fold precision).

function `res` = `FastPrecSum`(p, K)
  $n = \mathrm{length}(p)$; $ExactFlag = False$
  $T^{(0)} = \mathrm{fl}\left(\left(\sum_{i=1}^{n} |p_i|\right)/(1 - n \cdot \mathtt{eps})\right)$ % $\sum |p_i| \le T$
  for $m = 1 : K-1$
    $\sigma_0^{(m)} = \mathrm{fl}\left((2T^{(m-1)})/(1 - (3n+1)\mathtt{eps})\right)$; $\tilde{\sigma}^{(m)} = \sigma_0^{(m)}$
    $q = \mathrm{fl}\left((2\mathtt{eps})^{-1}\sigma_0^{(m)}\right)$, $u = \mathrm{fl}\left(|q/(1 - \mathtt{eps}) - q|\right)$ % $u = \mathrm{ufp}(\sigma_0)$
    $\Phi^{(m)} = \mathrm{fl}\left(\left((2n(n+2)\mathtt{eps}) \cdot u\right)/(1 - 5\mathtt{eps})\right)$
    $T^{(m)} = \min\left(\mathrm{fl}\left(\left(\left(\frac{3}{2} + 4\mathtt{eps}\right) \cdot (n \cdot \mathtt{eps})\right) \cdot \sigma_0\right), \mathrm{fl}((2n \cdot \mathtt{eps}) \cdot u)\right)$
    if $4T^{(m)} \le \mathtt{eps}^{-1}\mathtt{eta}$ then $K = m+1$; $ExactFlag = True$; end if % *exact result*
  end for
  $e = 0$
  for $i = 1 : n$
    $\tilde{p}_i = p_i$ % *index i is not necessary*
    for $m = 1 : K-1$
      $\sigma' = \mathrm{fl}(\tilde{\sigma}^{(m)} + \tilde{p}_i)$
      $q = \mathrm{fl}(\sigma' - \tilde{\sigma}^{(m)})$
      $\tilde{p}_i = \mathrm{fl}(\tilde{p}_i - q)$
      $\tilde{\sigma}^{(m)} = \sigma'$
    end for
    $e = \mathrm{fl}(e + \tilde{p}_i)$
  end for

```
if ExactFlag then res = e; return; end if                          % exact result
t = 0
for m = 1 : K − 1
  τ^(m) = fl(σ̃^(m) − σ_0^(m))                                      % exact result
  t^(m) = fl(t + τ^(m))
  if |t^(m)| ≥ Φ^(m) then                                          % surely faithful result
    τ_2 = fl((t − t^(m)) + τ^(m))                                  % [t^(m), τ_2] = FastTwoSum(t, τ^(m))
    if m = K − 1 then
      res = fl(t^(m) + (τ_2 + e))
    else                                                           % m ≤ K − 2
      τ^(m+1) = fl(σ̃^(m+1) − σ_0^(m+1))                            % exact result
      τ_3 = fl(τ_2 + τ^(m+1)); τ_4 = fl((τ_2 − τ_3) + τ^(m+1))     % [τ_3, τ_4] = FastTwoSum(τ_2, τ^(m+1))
      if m = K − 2 then
        res = fl(t^(m) + (τ_3 + (τ_4 + e)))
      else                                                         % m < K − 2
        τ^(m+2) = fl(σ̃^(m+2) − σ_0^(m+2))                          % exact result
        res = fl(t^(m) + (τ_3 + (τ_4 + τ^(m+2))))
      end if
    end if
    return
  end if
  t = t^(m)
end for
res = fl(t^(m) + e)                                                % may be no faithful rounding
```

If $m = K-1$, then the results of `FastAccSum` and `FastPrecSum` are the same and therefore faithful. This means in the following analysis we have only to show that `res` is a faithfully rounded result for $m < K-1$; i.e., the condition number of the problem would have required fewer extractions.

PROPOSITION 5.2. *Let $p_i \in \mathbb{F}$ for $1 \le i \le n$ and $K \ge 1$ be given, and assume $(2n^2+4n+6)\texttt{eps} \le 1$ and $\texttt{eps} \le \frac{1}{128}$. Assume that no overflow occurs. Then the result* `res` *of Algorithm* 5.1 *(*`FastPrecSum`*) is a faithful rounding of the sum $s := \sum_{i=1}^{n} p_i$ except when the condition $|t^{(m)}| \ge \Phi^{(m)}$ is never satisfied for all $1 \le m \le K-1$. In the latter case*

$$\left|\frac{\texttt{res} - s}{s}\right| \le (2n+3)\texttt{eps} \cdot (4n\varphi \cdot \texttt{eps})^{K-1} \cdot \operatorname{cond}\left(\sum p_i\right) \tag{5.3}$$

provided $s \ne 0$, where $\varphi = (1-(3n+2)\texttt{eps})^{-1}$. Algorithm 5.1 *requires $3Kn + \mathcal{O}(K)$ flops.*

Remark. The estimation of the relative error of `res` in (5.3) basically reads

$$\left|\frac{\texttt{res} - s}{s}\right| \lesssim \frac{1}{2}(4n \cdot \texttt{eps})^{K} \cdot \operatorname{cond}\left(\sum p_i\right),$$

so about quality "as if" calculated in K-fold precision.

Proof of Proposition 5.2. Denote by m the first index for which $|t^{(m)}| \ge \Phi^{(m)}$ is satisfied. We have only to treat the "well-conditioned" case, i.e., $m < K-1$. Also we may assume that the *ExactFlag* was not set, i.e., $4T^{(m)} > \texttt{eps}^{-1}$.

As in the algorithm, we distinguish the two cases $m = K-2$ and $m < K-2$. Define

$$T := \operatorname{fl}\left(\sum_{i=1}^{n} \tilde{p}_i\right) \quad \text{and} \quad \Delta := \sum_{i=1}^{n} \tilde{p}_i - T \quad \text{for} \quad m = K-2 \tag{5.4}$$

and

$$T := \tau^{(m+2)} \quad \text{and} \quad \Delta := \sum_{\nu=m+3}^{K-1} \tau^{(\nu)} + \sum_{i=1}^{n} \tilde{p}_i \quad \text{for} \quad m < K-2. \tag{5.5}$$

Note that $T = e$ for the case $m = K-2$. Also note that $\tau^{(m+1)}$ and $\tau^{(m+2)} = \mathrm{fl}(\tilde{\sigma}^{(m+2)} - \sigma_0^{(m+2)})$ are computed only when necessary. In any case they are computed without rounding error. By (4.27), (2.14), and Lemma 3.4

$$t^{(m-1)}, \tau^{(m)} \in 2\mathtt{eps} \cdot \mathrm{ufp}\left(\tau^{(m)}\right)\mathbb{Z} \quad \text{and} \quad \tau_2 \in 2\mathtt{eps} \cdot \mathrm{ufp}\left(\tau^{(m)}\right)\mathbb{Z} \subset 2\mathtt{eps} \cdot \mathrm{ufp}\left(\tau^{(m+1)}\right)\mathbb{Z}$$

so that Lemma 3.4 implies that `FastTwoSum` is applicable to $\tau_2, \tau^{(m+1)}$. It follows that $\tau_3 + \tau_4 = \tau_2 + \tau^{(m+1)}$. Lemma 3.4 also implies

$$|\tau_2| \le \mathtt{eps} \cdot |\tau_1| \quad \text{and} \quad |\tau_4| \le \mathtt{eps} \cdot |\tau_3|. \tag{5.6}$$

Setting $\tau_1 := t^{(m)}$ as in the analysis of `FastAccSum` and using (4.25), (4.5), and (5.2) yields

$$s = \tau_1 + \tau_3 + \tau_4 + \sum_{\nu=m+2}^{K-1} \tau^{(\nu)} + \sum_{i=1}^{n} \tilde{p}_i. \tag{5.7}$$

This is true in both cases $m = K-2$ and $m < K-2$, where for $m = K-2$ the first sum in (5.7) is empty. Combining this with (5.4) and (5.5) and using $\tau_1 = t^{(m)}$ we obtain for both cases

$$\mathtt{res} = \mathrm{fl}\big(\tau_1 + (\tau_3 + (\tau_4 + T))\big) \quad \text{and} \quad s = \tau_1 + \tau_3 + \tau_4 + T + \Delta. \tag{5.8}$$

Similar to (4.34) we define the intermediate results τ_3', τ_4' with corresponding errors δ_3, δ_4 in the computation of `res` by

$$\begin{array}{rcl} \tau_4' & := \mathrm{fl}(\tau_4 + T) & = \tau_4 + T - \delta_4, \\ \tau_3' & := \mathrm{fl}(\tau_3 + \tau_4') & = \tau_3 + \tau_4' - \delta_3, \\ \mathtt{res} & := \mathrm{fl}(\tau_1 + \tau_3') & =: \mathrm{fl}(r), \end{array} \tag{5.9}$$

where $r := \tau_1 + \tau_3'$. With (5.8) this implies

$$s = \tau_1 + \tau_3' + \delta_3 + \delta_4 + \Delta =: r + \delta \quad \text{and} \quad \mathtt{res} = \mathrm{fl}(r) \quad \text{for} \quad \delta := \delta_3 + \delta_4 + \Delta. \tag{5.10}$$

We collect some facts from the analysis of `FastAccSum`. From the proof of (4.19) in (4.21) we see

$$\sigma_0^{(k+1)} \le 4n\varphi \cdot \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(k)}\right) \quad \text{for} \quad \varphi := \frac{1}{1 - (3n+2)\mathtt{eps}} \tag{5.11}$$

so that (4.6) in Lemma 4.5 yields $|\tau^{(k)}| = |\mathrm{fl}(\tilde{\sigma}^{(k)} - \sigma_0^{(k)})| = |\sigma_n^{(k)} - \sigma_0^{(k)}| < \frac{1}{2}\sigma_0^{(k)}$ and

$$|\tau^{(m+k)}| \le \frac{1}{2}\big(4n\varphi \cdot \mathtt{eps}\big)^k \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) \tag{5.12}$$

for $k \ge 1$. Furthermore, (4.4) implies

$$|p_i^{(m+k)}| \le 2\mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m+k)}\right) \le 2\mathtt{eps}\big(4n\varphi \cdot \mathtt{eps}\big)^k \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right). \tag{5.13}$$

We will frequently use the quantity φ defined in (5.11). For later use we note that $(2n^2+4n+6)\mathtt{eps} \le 1$ and $\mathtt{eps} \le \frac{1}{128}$ imply

$$21n \cdot \mathtt{eps} < 1, \quad \varphi < \frac{6}{5}, \quad \text{and} \quad 4n\varphi \cdot \mathtt{eps} < \frac{1}{4}. \tag{5.14}$$

Next we prove

$$|\Delta| \le 100n^3\mathtt{eps}^3 \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) \quad \text{and} \quad |T| \le 12n^2\mathtt{eps}^2 \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right). \tag{5.15}$$

Here is the last place in the proof we have to distinguish the two cases $m = K-2$ and $m < K-2$. For $m = K-2$ we use $\tilde{p}_i = p_i^{(m+1)}$ and (5.13) to see $|\tilde{p}_i| \le 2n \cdot \mathtt{eps} \cdot \mathrm{ufp}(\sigma_0^{(m+1)}) \le 8n\varphi \cdot \mathtt{eps}^2 \cdot \mathrm{ufp}(\sigma_0^{(m)})$. Since the first upper bound is a power of 2, we can use (2.17) to conclude

$$|\Delta| \le 4n^3\varphi \cdot \mathtt{eps}^3 \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) \quad \text{and} \quad |T| \le 8n^2\varphi \cdot \mathtt{eps}^2 \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right).$$

For $m < K-2$ the definition in (5.5), (5.12), and (5.14) imply

$$|T| \le 8n^2\varphi^2\mathtt{eps}^2 \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) \le 12n^2\mathtt{eps}^2 \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right),$$

and (5.12), (5.13), and (5.14) give

$$\begin{aligned} |\Delta| &\le \left(\tfrac{1}{2}(4n\varphi \cdot \mathtt{eps})^3(1-4n\varphi \cdot \mathtt{eps})^{-1} + 2\mathtt{eps}(4n\varphi \cdot \mathtt{eps})^2\right) \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) \\ &\le (4n\varphi \cdot \mathtt{eps})^2\left(\tfrac{2}{3} \cdot 4n\varphi \cdot \mathtt{eps} + 2\mathtt{eps}\right) \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) \\ &\le 100n^3\mathtt{eps}^3 \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right). \end{aligned}$$

This proves (5.15). The standard floating-point estimations (2.7) and (2.8) together with (5.9) and (5.6) imply

$$|\tau_3'| \le (1+\mathtt{eps})|\tau_3+\tau_4'| \le (1+\mathtt{eps})\big(|\tau_3| + (1+\mathtt{eps})|\tau_4+T|\big) \le (1+3\mathtt{eps})\big(|\tau_3|+|T|\big) \tag{5.16}$$

so that $\tau_3 = \mathrm{fl}(\tau_2 + \tau^{(m+1)})$, (5.12), and (5.6) yield

$$|\tau_3| \le (1+\mathtt{eps})|\tau_2+\tau^{(m+1)}| \le (1+\mathtt{eps})\big(\mathtt{eps}|\tau_1| + 2n\varphi \cdot \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right)\big). \tag{5.17}$$

Furthermore, similar to the analysis of `FastAccSum`, (5.16), (5.17), and (5.15) give

$$\begin{aligned} |\mathtt{res}| &\ge (1-\mathtt{eps})\big(|\tau_1| - |\tau_3'|\big) \ge (1-\mathtt{eps})|\tau_1| - (1+2\mathtt{eps})(|\tau_3|+|T|) \\ &\ge \big(1-\mathtt{eps}-(1+4\mathtt{eps})\mathtt{eps}\big)|\tau_1| - 2n\varphi(1+4\mathtt{eps})\mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) - (1+2\mathtt{eps})|T| \\ &\ge \big(1-2\mathtt{eps}-4\mathtt{eps}^2\big)|\tau_1| - \frac{21}{10}n\varphi \cdot \mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) - 13n^2\mathtt{eps}^2 \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right). \end{aligned} \tag{5.18}$$

Since the *ExactFlag* is not set, we know by Lemma 4.8 that the computed quantities $\sigma_0^{(k)}$ and $\Phi^{(k)}$ are not in the underflow range, but we have to prove $\mathtt{res} \notin \mathbb{U}$. This follows by (5.18), (4.41), and (4.42) and

$$|\mathtt{res}| \ge \big(2n(n+2)-3n-n\big)\mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) = 2n^2\mathtt{eps} \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}.$$

Thus Lemma 3.2 proves res to be faithful if $2|\delta| < \mathtt{eps}|r|$. We use the definition (5.10) of δ, $|\tau_4| \le \mathtt{eps}|\tau_3|$, (5.17), and (5.15) to obtain

$$\begin{aligned}\mathtt{eps}^{-1}|\delta| &\le |\tau_3 + \tau_4'| + |\tau_4'| + |\Delta| \le |\tau_3| + 2|\tau_4'| + |\Delta| \\ &\le |\tau_3| + 2(1+\mathtt{eps})\big(\mathtt{eps}|\tau_3| + |T|\big) + |\Delta| \le (1+3\mathtt{eps})|\tau_3| \\ &\quad + 2(1+\mathtt{eps})|T| + |\Delta| \\ &\le (1+5\mathtt{eps})\mathtt{eps}|\tau_1| + \left(\frac{21}{10}n\varphi\cdot\mathtt{eps} + 25n^2\mathtt{eps}^2 + 100n^3\mathtt{eps}^3\right)\cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right).\end{aligned}$$

Inserting this into (5.18) and using (4.41) we finally prove

$$\begin{aligned}|\mathtt{res}| - 2\mathtt{eps}^{-1}|\delta| &\ge \big(1 - 4\mathtt{eps} - 14\mathtt{eps}^2\big)|\tau_1| - \left(\frac{63}{10}n\varphi\cdot\mathtt{eps} + 63n^2\mathtt{eps}^2 + 200n^3\mathtt{eps}^3\right) \\ &\quad \cdot \mathrm{ufp}\left(\sigma_0^{(m)}\right) \\ &\ge \left((1-\mathtt{eps})2n(n+2) - \frac{63}{10}n\varphi - 63n^2\mathtt{eps} - 200n^3\mathtt{eps}^2\right)\mathtt{eps}\cdot\mathrm{ufp}\left(\sigma_0^{(m)}\right).\end{aligned}$$

A crude estimation using (5.14) verifies

$$\begin{aligned}&(1-\mathtt{eps})2n(n+2) - \frac{63}{10}n\varphi - 63n^2\mathtt{eps} - 200n^3\mathtt{eps}^2 \\ &\quad > 2n(n+1) - 8n - 3n - 10n > 0 \quad \text{for } n \ge 10,\end{aligned}$$

and a more detailed analysis using $(2n^2+4n+6)\mathtt{eps} \le 1$ and $\mathtt{eps} \le \frac{1}{128}$ shows that this expression is positive for all suitable values of eps and $n \ge 2$. This proves res to be a faithful rounding of $s = \sum p_i$.

It remains to prove (5.3). In this case $|t^{(m)}| < \Phi^{(m)}$ for the final value of m so that (5.2) and $t^{(m)} = \mathrm{fl}(t + \tau^{(m)}) = t + \tau^{(m)}$ imply

$$s = t^{(m)} + \sum_{i=1}^{n}\tilde{p}_i, \quad \mathtt{res} = \mathrm{fl}(t^{(m)} + e), \quad \text{and} \quad e = \mathrm{fl}\left(\sum_{i=1}^{n}\tilde{p}_i\right).$$

Thus for some $|\varepsilon| \le \mathtt{eps}$,

$$|\mathtt{res} - s| = |(1+\varepsilon)(t^{(m)}+e) - s| \le \mathtt{eps}|t^{(m)}| + \left|\mathrm{fl}\left(\sum\tilde{p}_i\right) - \sum\tilde{p}_i\right| + \mathtt{eps}\left|\mathrm{fl}\left(\sum\tilde{p}_i\right)\right|.$$

By (5.13) we know $|\tilde{p}_i| \le 2\mathtt{eps}\cdot\mathrm{ufp}(\sigma_0^{(m)}) := \sigma$ so that using (2.17), the definition of $\Phi^{(m)}$, (5.11), $m = K-1$, $(2n^2+4n+6)\mathtt{eps} \le 1$, and a little computation shows

$$\begin{aligned}|\mathtt{res} - s| &< \mathtt{eps}\cdot\Phi^{(m)} + \frac{n(n-1)}{2}\mathtt{eps}\cdot\sigma + n\cdot\mathtt{eps}\cdot\sigma \\ &\le \left(\frac{2n(n+2)}{1-4\mathtt{eps}} + n(n+1)\right)\mathtt{eps}^2\cdot\mathrm{ufp}\left(\sigma_0^{(m)}\right) \\ &\le (3n^2+5n+4)\mathtt{eps}^2\cdot\mathrm{ufp}\left(\sigma_0^{(m)}\right) \\ &\le (3n^2+5n+4)\mathtt{eps}^2\big(4n\varphi\cdot\mathtt{eps}\big)^{K-2}\cdot\mathrm{ufp}\left(\sigma_0^{(1)}\right).\end{aligned}$$

To estimate the relation to the condition number $\mathrm{cond}(\sum p_i) = \sum|p_i| / |\sum p_i|$ we need a lower bound on $S := \sum_{i=1}^{n}|p_i|$ depending on $\mathrm{ufp}(\sigma_0^{(1)})$. This is obtained by

(3.1), the definition of $T^{(0)}$ and $\sigma_0^{(1)}$, standard floating-point estimations, and

$$\begin{aligned} S &\geq \mathrm{fl}\left(\sum_{i=1}^{n}|p_i|\right)(1-(n-1)\mathtt{eps}) \geq (1-n\cdot\mathtt{eps})^2 T^{(0)} \\ &\geq \frac{1}{2}\big(1-n\cdot\mathtt{eps}\big)^2\big(1-(3n+2)\mathtt{eps}\big)\cdot\sigma_0^{(1)} \\ &\geq \frac{1}{2}\big(1-(5n+2)\mathtt{eps}\big)\cdot\sigma_0^{(1)}. \end{aligned}$$

Putting things together and a little computation using $1/(1-(5n+2)\mathtt{eps}) \leq (1+3n\cdot\mathtt{eps})\varphi$ yields

$$\begin{aligned} |\mathtt{res}-s| &< (3n^2+5n+4)\mathtt{eps}^2\big(4n\varphi\cdot\mathtt{eps}\big)^{K-2}\frac{2}{1-(5n+2)\mathtt{eps}}\cdot S \\ &\leq (2n+3)\mathtt{eps}\big(4n\varphi\cdot\mathtt{eps}\big)^{K-1}\cdot S, \end{aligned}$$

and (5.3) follows. The proof is finished. ∎

As for Algorithm `FastAccSum` we can use (5.3) to compute the minimum condition number for which `FastPrecSum` computes a faithfully rounded result and compare it with other algorithms. Table 5.1 shows this condition number for $n = 100$ for Algorithm `SumK` in [27], Algorithm `PrecSum` in [39], and `FastPrecSum`. All these algorithms use error-free transformations to compute a result "as if" computed in K-fold precision.

The results are comparable, although Algorithm `PrecSum` seems to perform a little better. Note that the numbers are based on the estimations; the actual condition numbers are larger for all algorithms. However, the computing time for the three algorithms is different as well, so the comparison is not entirely fair. For m iterations, Algorithm `SumK` needs $(6m+1)n$ flops, `PrecSum` needs $(4m+3)n$ flops, and `FastPrecSum` needs $(3m+3)n$ flops. For convenience the numbers are listed in Table 5.2.

For $m = 3$, $m = 4$, and $m = 5$ extractions the three algorithms need about the same amount of floating-point operations (`FastPrecSum` needs a little less). Thus we may compare the minimum treatable condition number using this amount of floating-point operations. The results for different dimensions are listed in Table 5.3.

As can be seen, Algorithm `FastPrecSum` can compute a faithfully rounded result for significantly more ill-conditioned problems than the competitors.

TABLE 5.1
Minimum treatable condition number for faithful rounding with `m` *extractions.*

	m=1	m=2	m=3	m=4
`SumK`	$1.15\cdot10^{11}$	$5.23\cdot10^{24}$	$2.38\cdot10^{38}$	$1.08\cdot10^{52}$
`PrecSum`	$1.83\cdot10^{11}$	$1.29\cdot10^{25}$	$9.07\cdot10^{38}$	$6.39\cdot10^{52}$
`FastPrecSum`	$1.11\cdot10^{11}$	$2.50\cdot10^{24}$	$5.62\cdot10^{37}$	$1.27\cdot10^{51}$

TABLE 5.2
Number of floating-point operations for `m` *extractions.*

	m=1	m=2	m=3	m=4	m=5
`SumK`	7n	13n	19n	25n	31n
`PrecSum`	7n	11n	15n	19n	23n
`FastPrecSum`	6n	9n	12n	15n	18n

TABLE 5.3
Minimum treatable condition number for faithful rounding in about the same number of flops.

	m	flops	n=100	n=1000	n=1000
`SumK`	3	19n	$2.38 \cdot 10^{38}$	$2.29 \cdot 10^{34}$	$2.28 \cdot 10^{30}$
`PrecSum`	4	19n	$6.39 \cdot 10^{52}$	$1.95 \cdot 10^{48}$	$1.86 \cdot 10^{42}$
`FastPrecSum`	5	18n	$2.85 \cdot 10^{64}$	$2.89 \cdot 10^{58}$	$2.89 \cdot 10^{52}$

6. Timing. In this section we briefly report on some timings. We do this with great hesitation: Measuring the computing time of summation algorithms in a high-level language on today's architectures is more of a hazard than scientific research. The results are hardly predictable and often do not reflect the actual performance.

These statements sound harsh, so I give a few examples. It happens occasionally that adding statements to a code, thus supposedly increasing the computing, actually *decreases* the computing time.

The compiler optimization is also hardly predictable. Consider the code

for $i = 1 : n$
  $\sigma' = \mathrm{fl}(|\sigma + p_i|)$
  $q = \mathrm{fl}(\sigma' - \sigma)$
  $p_i = \mathrm{fl}(p_i - q)$
  $\sigma = \sigma'$
end for

as it is used in `FastAccSum`. We used Ogita's trick [37] to avoid optimization of $q = \mathrm{fl}(\sigma' - \sigma)$. Since the value of σ changes in each loop, such an optimization is not possible without additional knowledge of the data. Nevertheless, certain compilers do eliminate the evaluation of q by setting $q = p_i$ and subsequently $p_i = 0$. Obviously, there is a predicted branch which is faster than executing the 3 floating-point operations.

Declaring a variable as "volatile" is a common method to prevent compiler optimization. However, this works only sometimes. Also this slows down computation, sometimes by a factor 2 and sometimes only a few percent.

In summary we want to stress that the following numbers should be read with great hesitation. Different implementations using different compilers and/or architectures may vary the times easily by a factor 2 or more for all algorithms and in both directions.

We measured the computing time of `FastAccSum` compared to `AccSum`. We used a Pentium 4 laptop and the Intel Visual Fortran 9.1 compiler. To avoid unexpected compiler optimizations we used "improved consistency" of the floating-point arithmetic. Various comparisons in [37] seem to show that `AccSum` used to be the fastest known algorithm to compute a faithfully rounded result.

In Table 6.1 we display the ratio of computing times of `AccSum` over `FastAccSum` for various condition numbers and dimensions $n = 100, 300, 1000, 3000, 10000$. A value greater than 1 indicates that `FastAccSum` is faster by this factor than `AccSum`.

We treated 1000 random sums for each combination of condition number and dimension and display the median of the ratios. As can be seen in Table 6.1, the ratio approaches the theoretical limit of 1.33 with increasing condition number and dimension; that is, `FastAccSum` is 33% faster than `AccSum`. This is because the main loop in `AccSum` needs $4n$ operations, whereas `FastAccSum` needs $3n$ operations.

There are a few exceptions. For instance, cond $= 10^{16}$ and $n = 10,000$ show a ratio 1.88—much better than 1.33. This is a combination where `FastAccSum` often

TABLE 6.1
Ratio of computing times t(`AccSum`)$/t$(`FastAccSum`).

cond $\setminus$ n	100	300	1000	3000	10,000
10^{6}	1.09	1.18	1.30	1.35	1.33
10^{16}	1.22	1.22	1.29	1.30	1.88
10^{32}	1.33	1.27	1.45	1.25	1.38
10^{48}	1.35	1.43	1.38	1.33	1.47
10^{60}	1.25	1.33	1.29	1.27	1.40

needs one less extraction than `AccSum`. However, as mentioned, the numbers should be read with suspicion.

7. Conclusion. We presented algorithms for computing a faithful rounding of the sum of floating-point numbers and a result "as if" computed in K-fold precision. Since dot products can be transformed error-free into sums, the algorithms cover dot products as well. Along the lines of Algorithms 7.4 and 7.1 in [38] the rounded to nearest as well as rounded downwards and upwards results can be computed as well. By flop count the new methods are the fastest known algorithms for the intended task. For convenience, we put Matlab reference implementations on http://www.ti3.tu-harburg.de/rump.

Acknowledgment. The author wishes to express his thanks to the anonymous referees. Their useful comments and suggestions helped to improve the paper significantly.

REFERENCES

[1] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, AND S. D.C., *LAPACK User's Guide, Release* 2.0, 2nd ed., SIAM, Philadelphia, 1994.

[2] D. BAILEY, H. YOZO, X. LI, AND B. THOMPSON, *ARPREC: An Arbitrary Precision Computation Package*, Technical report LBNL-53651, Lawrence Berkeley National Laboratory, Berkeley, CA, 2002.

[3] J. MULLER, N. BRISEBARRE, F. DINECHIN, C. JEANNEROD, V. LEFÈVRE, G. MELQUIOND, R. REVOL, D. STEHLÉ, AND S. TORRES, *Handbook of Floating-Point Arithmetic*, Birkhäuser, Boston, 2009.

[4] B. BUCHBERGER, *Gröbner bases: An algorithmic method in polynomial ideal theory*, in Multidimensional Systems Theory, N. Bose, ed., Reidel, Dordrecht, 1985.

[5] G. COLLINS, *Quantifier elimination for real closed fields by cylindrical algebraic decomposition*, in Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern, 1975, Lecture Notes in Comput. Sci. 33, B. E. A. Caviness, ed., Springer, Berlin, 1975, pp. 134–183.

[6] M. DAUMAS AND D. MATULA, *Validated roundings of dot products by sticky accumulation*, IEEE Trans. Comput., 46 (1997), pp. 623–629.

[7] T. DEKKER, *A floating-point technique for extending the available precision*, Numer. Math., 18 (1971), pp. 224–242.

[8] J. DEMMEL AND Y. HIDA, *Accurate and efficient floating point summation*, SIAM J. Sci. Comput., 25 (2003), pp. 1214–1248.

[9] L. FOUSSE, G. HANROT, V. LEFÈVRE, P. PÉLISSIER, AND P. ZIMMERMANN, *MPFR: A multiple-precision binary floating-point library with correct rounding*, ACM Trans. Math. Software, 33 (2007), article 13.

[10] J. GRABMEIER, E. KALTOFEN, AND V. WEISPFENNIG, *Computer Algebra Handbook*, Springer, Berlin, 2003.

[11] S. GRAILLAT, *Applications of Fast and Accurate Summation in Computational Geometry*, Technical report RR2005-03, Laboratoire LP2A, University of Perpignan, Perpignan, France, 2005.

[12] S. GRAILLAT, *Applications of Fast and Accurate Summation in Computational Geometry*, technical report, University of Perpignan, Perpignan, France, 2006.

[13] S. Graillat, *Provably faithful evaluation of polynomials*, in Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, 2006.
[14] S. Graillat, *Extra Precise Iterative Refinement for Linear Systems and Accurate Validated Error Bound*, technical report, University of Perpignan, Perpignan, France, 2007.
[15] S. Graillat, P. Langlois, and N. Louvet, *Compensated Horner Scheme*, Technical report RR2005-02, Laboratoire LP2A, University of Perpignan, Perpignan, France, 2005.
[16] S. Graillat, P. Langlois, and N. Louvet, *Improving the compensated Horner scheme with a fused multiply and add*, in Proceedings of the 2006 ACM Symposium on Applied Computing, CMS-109, Dijon, France, 2006, pp. 1323–1327.
[17] N. J. Higham, *The accuracy of floating point summation*, SIAM J. Sci. Comput., 14 (1993), pp. 783–799.
[18] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, 2002.
[19] W. Kahan, *A survey of error analysis*, in Proceedings of the IFIP Congress, Ljubljana, Information Processing 71, North–Holland, Amsterdam, 1972, pp. 1214–1239.
[20] E. Kaltofen, B. Li, Z. Yang, and L. Zhi, *Exact certification of global optimality of approximate factorizations via rationalizing sums-of-squares with floating point scalars*, in Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC), Hagenberg, Austria, ACM, 2008, pp. 155–163.
[21] P. Kornerup, V. Lefèvre, N. Louvet, and J. Muller, *On the Computation of Correctly-Rounded Sums*, Technical report 2008-35, LIP, Paris, 2008.
[22] P. Langlois, *Accurate algorithms in floating point arithmetic*, lecture at the 12th GAMM-IMACS International Symposium on Scientific Computing (SCAN), Computer Arithmetic and Validated Numerics, Duisburg, IEEE, 2006.
[23] 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. Software, 28 (2002), pp. 152–205.
[24] N. Louvet, *Compensating the fused multiply and add implementation of the Horner scheme*, in Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, 2006.
[25] M. Malcolm, *On accurate floating-point summation*, Comm. ACM, 14 (1971), pp. 731–736.
[26] A. Neumaier, *Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen*, ZAMM Z. Angew. Math. Mech., 54 (1974), pp. 39–51.
[27] T. Ogita, S. M. Rump, and S. Oishi, *Accurate sum and dot product*, SIAM J. Sci. Comput., 26 (2005), pp. 1955–1988.
[28] S. Oishi and S. Rump, *Fast verification of solutions of matrix equations*, Numer. Math., 90 (2002), pp. 755–773.
[29] K. Okumura, *Classifying nonlinear circuits by Grœbner base*, in NDES'98 - Proceedings of the 6th International Specialist Workshop on Nonlinear Dynamics of Electronic Systems, Technical University of Budapest, Budapest, Hungary, 1998, pp. 267–270.
[30] F. Ordóñez and R. M. Freund, *Computational experience and the explanatory value of condition measures for linear optimization*, SIAM J. Optim., 14 (2003), pp. 307–333.
[31] K. Ozaki, T. Ogita, S. Rump, and S. Oishi, *Fast and robust algorithm for geometric predicates using floating-point arithmetic*, Trans. Japan Soc. Ind. Appl. Math., 4 (2006), pp. 553–562 (in Japanese).
[32] K. Ozaki, T. Ogita, S. Rump, and S. Oishi, *Accurate matrix multiplication by using level* 3 *BLAS operation*, in Proceedings of the 2008 International Symposium on Nonlinear Theory and its Applications, NOLTA'08, Budapest, Hungary, IEICE, 2008, pp. 508–511.
[33] M. Pichat, *Correction d'une somme en arithmetique a virgule flottante*, Numer. Math., 19 (1972), pp. 400–406.
[34] D. Priest, *Algorithms for arbitrary precision floating point arithmetic*, in Proceedings of the 10th Symposium on Computer Arithmetic, Grenoble, France, P. Kornerup and D. Matula, eds., IEEE Computer Society Press, Piscataway, NJ, 1991, pp. 132–145.
[35] D. Priest, *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*, Ph.D. thesis, Mathematics Department, University of California at Berkeley, CA, 1992.
[36] S. Rump and H. Böhm, *Least significant bit evaluation for arithmetic expressions*, Computing, 30 (1983), pp. 189–199.
[37] S. M. Rump, T. Ogita, and S. Oishi, *Accurate floating-point summation part I: faithful rounding*, SIAM J. Sci. Comput., 31 (2008), pp. 189–224.
[38] S. M. Rump, T. Ogita, and S. Oishi, *Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest*, SIAM J. Sci. Comput., 31 (2008), pp. 1269–1302.

[39] S. Rump, T. Ogita, and S. Oishi, *Fast high precision summation*, submitted.
[40] J. Shewchuk, *Adaptive precision floating-point arithmetic and fast robust geometric predicates*, Discrete Comput. Geom., 18 (1997), pp. 305–363.
[41] K. Tanabe, *Additive-form iterative refinement of LU factorization of an ill-conditioned matrix*, in Proceedings of the International Symposium on Nonlinear Theory and its Applications (NOLTA2005), Bruges, Belgium, IEICE, 2005, pp. 737–740.
[42] *XBLAS: A Reference Implementation for Extended and Mixed Precision BLAS*, http://crd.lbl.gov/∼xiaoye/XBLAS/index.html.
[43] Y. Zhu and W. Hayes, *Fast, guaranteed-accurate sums of many floating-point numbers*, in Proceedings of the RNC7 Conference on Real Numbers and Computers, Nancy, France, G. Hanrot and P. Zimmermann, eds., Loria, 2006, pp. 11–22.
[44] Y. Zhu, J. Yong, and G. Zheng, *A new distillation algorithm for floating-point summation*, SIAM J. Sci. Comput., 26 (2005), pp. 2066–2078.
[45] G. Zielke and V. Drygalla, *Genaue Lösung linearer Gleichungssysteme*, GAMM Mitt. Ges. Angew. Math. Mech., 26 (2003), pp. 7–108.