in SIAM J. Sci. Comput. Volume 31, Issue 1, pp. 189-224 (2008)

ACCURATE FLOATING-POINT SUMMATION PART I: FAITHFUL ROUNDING *

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

Abstract. Given a vector of floating-point numbers with exact sum s, we present an algorithm for calculating a faithful rounding of s, i.e. the result is one of the immediate floating-point neighbors of s. If the sum s is a floating-point number, we prove that this is the result of our algorithm. The algorithm adapts to the condition number of the sum, i.e. it is fast for mildly conditioned sums with slowly increasing computing time proportional to the logarithm of the condition number. All statements are also true in the presence of underflow. The algorithm does not depend on the exponent range. Our algorithm is fast in terms of measured computing time because it allows good instruction-level parallelism, it neither requires special operations such as access to mantissa or exponent, it contains no branch in the inner loop, nor does it require some extra precision: The only operations used are standard floating-point addition, subtraction and multiplication in one working precision, for example double precision. Certain constants used in the algorithm are proved to be optimal.

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

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

1. Introduction and previous work.

We will present fast algorithms to compute high quality approximations of the sum and the dot product of vectors of floating-point numbers. Since dot products can be transformed without error into sums, we concentrate on summation algorithms.

Since sums of floating-point numbers are ubiquitous in scientific computations, there is a vast amount of literature to that, among them [2, 3, 7, 10, 13, 14, 18, 21, 22, 23, 24, 25, 31, 32, 33, 34, 35, 36, 37, 39, 40, 41, 42, 46, 47, 48, 49, 50], all aiming on some improved accuracy of the result. Higham [19] devotes an entire chapter to summation. Accurate summation or dot product algorithms have various applications in many different areas of numerical analysis. Excellent overviews can be found in [19, 32].

Most algorithms are backward stable, which means the relative error of the computed approximation is bounded by a small factor times the condition number. Many algorithms [23, 24, 25, 36, 39, 46, 49, 48] including those by Kahan, Babuška and Neumaier and others use compensated summation, i.e. the error of the individual additions is somehow corrected. Usually the relative error of the result is bounded by `eps` times the condition number of the sum, where `eps` denotes the relative rounding error unit. This is best possible by a well known rule of thumb in numerical analysis.

However, Neumaier [36] presented an algorithm where the relative error of the result is bounded by $\texttt{eps}^2$ times the condition number of the sum, an apparent contradiction to the cited rule of thumb. The key to that result are *error-free transformations*. Neumaier reinvented a method (see Algorithm 2.5) by Dekker [12] which transforms the sum $a + b$ of two floating-point numbers into a sum $x + y$, where x is the usual floating-point approximation and y comprises of the exact error. Surprisingly, x and y can be calculated

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

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

‡Department of Mathematics, Tokyo Woman's Christian University, 2–6–1 Zempukuji, Suginami-ku, Tokyo 167-8585, Japan, and Visiting Associate Professor at Waseda University, Faculty of Science and Engineering, 3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan (`ogita@lab.twcu.ac.jp`).

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

using only 3 ordinary floating-point operations if $|a| \geq |b|$. Recently such error-free transformations are used in many areas [15, 30].

This error-free transformation was generalized to vectors in so-called distillation algorithms. Prominent examples [7, 23, 24, 33, 31, 40, 3, 41, 46, 32, 50, 2, 37, 49, 48] include work by Bohlender, Priest, Anderson and XBLAS. Here a vector p_i of floating-point numbers is transformed into another vector p'_i with equal sum. Call such a process distillation. In our recent paper [37] we showed that it is possible to transform a vector p_i into a new vector p'_i such that $\mathrm{cond}\left(\sum p'_i\right)$ is basically $\texttt{eps} \cdot \mathrm{cond}\left(\sum p_i\right)$, whilst the transformation is error-free, i.e. $\sum p_i = \sum p'_i$. Repeating this process may produce an accurate approximation of the sum for arbitrary condition number.

After one distillation, the ordinary (recursive) sum of the distilled vector shares an accuracy *as if* calculated in doubled working precision. This is the quality of results given by XBLAS [32, 2] or `Sum2` in [37]. For many practical applications this is sufficient. However, the relative error of the result depends on the condition number. It does not allow, for instance, to compute the sign of a sum.

There are few methods [34, 39, 7, 40, 41, 46, 13, 14, 50] to compute an accurate approximation of a sum *independent* of the condition number, with the ultimate goal of the faithfully rounded or rounded-to-nearest exact sum. The aim of the present papers (Parts I and II) are such algorithms, and we shortly overview known approaches.

One of the first distillation algorithms by Bohlender [7] falls in that regime, it computes a rounded-to-nearest approximation of the sum. Usually only few distillations are needed for that, however, in worst case $n-1$, the length of the input vector. Others followed like Priest's [40, 41] doubly compensated summation which sorts the input data and can guarantee after three distillations a maximum relative error of `2eps`, independent of the condition number. For a good overview on distillation algorithms see [3].

Other approaches use the fact that the exponent range of floating-point numbers is limited. One of the very early algorithms by Malcolm [34] partitions the exponent range into a series of (overlapping) accumulators. Summands p_i are partitioned so that the parts can be added to some corresponding accumulator without error. The size and the number of accumulators is calculated beforehand based on the length of the input vector. In some way ARPREC [6] uses a similar method to add partial sums. Malcolm uses ideas by Wolfe [47] who's observations are presented without analysis.

Malcolm adds the accumulators in decreasing order and analyzes that the result is accurate to the last bit. Another approach is one long accumulator as popularized by Kulisch [28]. Here the exponent range is represented by an array of "adjacent" fixed point numbers, summands are split and added to the corresponding array element, and possible carries are propagated.

Zielke and Drygalla [50] follow yet another approach. They split the summands p_i relative to $\max|p_i|$ into high order and low order parts. For a small summand the high order part may be zero. The splitting point depends on the dimension and is chosen such that all high order parts can be added without error. This process is repeated until all lower parts are zero, thus receiving an array of partial sums s_j of the high order parts representing the original sum by $\sum s_j = \sum p_i$. Next the overlapping parts of the partial sums s_j are eliminated by adding them with carry in increasing order, and finally the resulting partial sums are added in decreasing order producing an accurate approximation of $\sum p_i$.

Zielke and Drygalla essentially present a Matlab-code (cf. Algorithm 3.1); they spend just 7 lines on page 29 on the description of this algorithm (in their 100-page paper [50] on the solution of linear systems of equations, written in German), and another 2 lines on a much less accurate variant. No analysis is given, underflow is excluded.

Our paper uses their idea to derive and analyze an algorithm producing a *faithfully rounded* approximation

`res` of the true sum $s := \sum p_i$. This means that there is no floating-point number between `res` and s, and provably `res` $= s$ in case the true sum s itself is a floating-point number. Such an algorithm is of fundamental interest both from a mathematical and numerical point of view, with many applications. For example, it allows accurate calculation of the residual, the key to the accurate solution of linear systems. Or it allows to compute sign(s) with rigor, a significant problem in the computation of geometrical predicates [10, 20, 45, 9, 27, 8, 14, 38], where the sign of the value of a dot product decides whether a point is exactly on a plane or on which side it is.

We improve Zielke and Drygalla's approach in several ways. First, they continue distillations until the vector of lower order parts is entirely zero. If there is only one summand small in magnitude, many unnecessary distillations are needed. We improve this by giving a criterion to decide how many distillations are necessary for a faithfully rounded result. We prove this criterion to be optimal. Second, they split the summands into higher and lower order part by some scaling and round to integer. This turns out to be slow on today's architectures. Moreover their poor scaling restricts the exponent range of the input vector severely (cf. Section 3). We derive a very simple and fast alternative. Third, we avoid the elimination of overlapping parts of the partial sums by showing that the previous higher order part can be added without error to its successor. Thus in each step only one higher order part t and a remaining vector p'_i satisfying $s = t + \sum p'_i$ are constructed. Fourth, not all partial sums need to be added, but we show that adding up the lower order parts p'_i using ordinary summation suffices to guarantee faithful rounding. The analysis of that is nontrivial. Finally, all results remain true in the presence of underflow, and the severe restriction of the exponent range is removed.

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

Our algorithms are fast. We interpret fast not only by the number of floating-point operations, but in terms of *measured computing time*. This means that special operations such as rounding to integer, access to mantissa or exponent, branches etc. are avoided. As will be seen in the computational results, special operations may slow down a computation substantially. Our algorithms use only floating-point addition, subtraction and multiplication in working precision. No extra precision is required. Mostly our algorithm to compute a faithfully rounded sum is even faster than XBLAS, although the result of XBLAS may be of much less quality.

The paper is divided into two parts; Part I is organized as follows. First we introduce our notation in Section 2 and list a number of properties. We need many careful floating-point estimations, frequently heavily relying on bit representations and the definition of the floating-point arithmetic in use. Not only that this is frequently quite tedious, such estimations are also sometimes presented in a colloquial manner and not easy to follow. To avoid this and also to ensure rigor, we found it convenient and more stringent to use inequalities. For this we developed a new machinery to characterize floating-point numbers, their bit representations and to handle delicate situations. In this section we also define faithful rounding and give a sufficient criterion for it.

In Section 3 we use this to develop an error-free transformation of a vector of floating-point numbers into an approximation of the sum and some remaining part. The magnitude of the remaining part can be estimated, so that we can derive a summation algorithm with faithful rounding in the following Section 4. Its stopping criterion is proved to be optimal. We prove faithfulness which particularly includes the exact determination of the sign. This is not only true in the presence of underflow, but the computed result is exact if it is in the underflow range. We also estimate the computing time depending on the condition number.

In Part II [44] of this paper we define and investigate K-fold faithful rounding, where the result is represented

by a vector of K floating-point numbers, develop an algorithm with directed rounding and rounding-to-nearest. Furthermore, algorithms for huge vector lengths up to almost $\texttt{eps}^{-1}$ are given, and an improved and efficient version only for sign determination. In both parts of this paper, computational results on a Pentium 4, Itanium 2 and Athlon 64 processor are presented. For all algorithms presented in Part I and II of this paper and in [37] we put a Matlab reference code on `http://www.ti3.tu-harburg.de/rump` .

As in [37] and [44], all theorems, error analysis and proofs are due to the first author of the present paper.

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

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

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

We denote by $\mathrm{fl}(\cdot)$ the result of a floating-point computation, where all operations within the parentheses are executed in working precision. If the order of execution is ambiguous and is crucial, we make it unique by using parentheses. An expression like $\mathrm{fl}\big(\sum p_i\big)$ implies inherently that summation may be performed in any order. We assume floating-point operations in rounding to nearest corresponding to the IEEE 754 arithmetic standard [1]. Then floating-point addition and subtraction satisfy [19]

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

Note that addition and subtraction is exact near underflow [16], so we need no underflow unit in (2.2). More precisely, for $a, b \in \mathbb{F}$ we have

$$\begin{array}{rcll} |a+b| \leq \texttt{eps}^{-1}\texttt{eta} & \Rightarrow & \mathrm{fl}(a+b) = a+b & \text{and} \\ \mathrm{fl}(a+b) = 0 & \Leftrightarrow & a = -b \,. & \end{array} \tag{2.3}$$

We have to distinguish between normalized and subnormal floating-point numbers. As has been noted by several authors [35, 26, 12], the error of a floating-point addition is always a floating-point number:

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

Fortunately, the error term δ can be computed using only standard floating-point operations. The following algorithm by Knuth was already given in 1969 [26]. It is a first example of an error-free transformation.

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

$$\begin{array}{l} \text{function } [x, y] = \texttt{TwoSum}(a, b) \\ \quad x = \mathrm{fl}(a+b) \\ \quad z = \mathrm{fl}(x-a) \\ \quad y = \mathrm{fl}((a-(x-z))+(b-z)) \end{array}$$

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

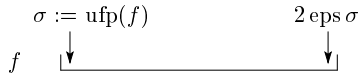

FIG. 2.1. *Normalized floating-point number: unit in the first place and unit in the last place*

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

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

This is also true in the presence of underflow. An error-free transformation for subtraction follows since $\mathbb{F} = -\mathbb{F}$.

The $\mathrm{fl}(\cdot)$ notation applies not only to operations but to real numbers as well. For $r \in \mathbb{R}$, $\mathrm{fl}(r) \in \mathbb{F}$ is r rounded to the nearest floating-point number. Following the IEEE 754 arithmetic standard tie is rounded to even. For $f_1, f_2 \in \mathbb{F}$ and $r \in \mathbb{R}$, the monotonicity of the rounding implies

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

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

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

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

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

where we set $\mathrm{ufp}(0) := 0$. This gives a convenient way to characterize the bits of a normalized floating-point number f: they range between the leading bit $\mathrm{ufp}(f)$ and the unit in the last place $2\texttt{eps} \cdot \mathrm{ufp}(f)$. The situation is depicted in Figure 2.1.

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 $\texttt{eps}\sigma\mathbb{Z}$, which can be interpreted as a set of fixed point numbers with smallest positive number $\texttt{eps}\sigma$. Of course, $\mathbb{F} \subseteq \texttt{eta}\mathbb{Z}$. Note that (2.8) is independent of a floating-point format and it applies to real numbers as well: $\mathrm{ufp}(r)$ is the value of the first nonzero bit in the binary representation of r. It follows

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

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

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

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

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

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

$$r \in \texttt{eps}\sigma\mathbb{Z},\ |r| \le \sigma \quad \text{and} \quad \texttt{eps}\sigma \ge \texttt{eta} \quad \Rightarrow \quad r \in \mathbb{F} \tag{2.14}$$

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

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

Note that (2.13) is also true for $f \in \mathbb{U}$. The assertions are clear except the last one (2.16), which is also clear after a little thinking, and a rigorous proof follows easily with our machinery. The assertion is clear for $ab \ge 0$ by using (2.13) and (2.11) because then $|\mathrm{fl}(a + b)| \ge \max(|a|, |b|)$; so without loss of generality

it suffices to show $\mathrm{fl}(a-b) \in \mathtt{eps}\sigma\mathbb{Z}$ for $a \geq b \geq 0$ and $\sigma := \mathrm{ufp}(a)$. If $\mathrm{ufp}(b) \geq \frac{1}{2}\sigma$, then (2.13) implies $a, b \in \mathtt{eps}\sigma\mathbb{Z}$ and the assertion follows by (2.15). And if $\mathrm{ufp}(b) < \frac{1}{2}\sigma$, then $b < \frac{1}{2}\sigma$, and $a \geq \sigma$ implies $a - b > \frac{1}{2}\sigma \in \mathbb{F}$. Hence (2.6) shows $\mathrm{fl}(a-b) \geq \frac{1}{2}\sigma \in \mathbb{F}$ and (2.13) implies $\mathrm{fl}(a-b) \in \mathtt{eps}\sigma\mathbb{Z}$.

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

$$\tilde{r} \neq 0 \quad \Rightarrow \quad \mathrm{ufp}(r) \leq \mathrm{ufp}(\tilde{r}) \ , \tag{2.17}$$

$$\begin{array}{rcl} \tilde{r} \in \mathbb{F}\backslash\mathbb{U} & \Rightarrow & |\tilde{r} - r| \leq \mathtt{eps} \cdot \mathrm{ufp}(r) \leq \mathtt{eps} \cdot \mathrm{ufp}(\tilde{r}) \\ \tilde{r} \in \mathbb{U} & \Rightarrow & |\tilde{r} - r| \leq \frac{1}{2}\mathtt{eta} \ . \end{array} \tag{2.18}$$

Note that a strict inequality occurs in (2.17) iff $\tilde{r}$ is a power of 2 and $|r| < |\tilde{r}|$. The assertions follow by the rounding to nearest property of $\mathrm{fl}(\cdot)$. Applying (2.18), (2.17), (2.9) and (2.3) to floating-point addition yields for $a, b \in \mathbb{F}$,

$$f = \mathrm{fl}(a+b) \quad \Rightarrow \quad f = a + b + \delta \ \text{ with } \ |\delta| \leq \mathtt{eps} \cdot \mathrm{ufp}(a+b) \leq \mathtt{eps} \cdot \mathrm{ufp}(f) \leq \mathtt{eps}|f| \ . \tag{2.19}$$

We will frequently need this refined error estimate which is up to a factor 2 better than the standard estimation (2.2). Note that (2.19) is also true in the presence of underflow because in this case $\delta = 0$, the addition is exact. The next is a refined estimation of the size and error of a sum of floating-point numbers.

$$\begin{array}{rcl} n\mathtt{eps} \leq 1, \ a_i \in \mathbb{F} \quad \text{and} \quad |a_i| \leq \sigma & \Rightarrow & |\mathrm{fl}(\sum_{i=1}^n a_i)| \leq n\sigma \quad \text{and} \\ & & |\mathrm{fl}(\sum_{i=1}^n a_i) - \sum_{i=1}^n a_i| \leq \frac{n(n-1)}{2}\mathtt{eps}\sigma \ . \end{array} \tag{2.20}$$

Estimation (2.20) is sometimes useful to avoid unnecessary quadratic terms and it is valid for any order of summation. Both inequalities follow by induction: For $\tilde{s} := \mathrm{fl}(\sum_{i \neq k} a_i)$ we have $|\tilde{s} + a_k| \leq n\sigma$. If $n\sigma$ is in the overflow range but $\mathrm{fl}(\sum a_i)$ is not, the assertions are valid anyway. Otherwise $n\mathtt{eps} \leq 1$ implies $n\sigma \in \mathbb{F}$, and (2.6) proves $|\mathrm{fl}(\tilde{s} + a_k)| \leq n\sigma$. For the second inequality in (2.20) we distinguish two cases. Firstly, if $|\tilde{s} + a_k| = n\sigma$, then $|\tilde{s}| = (n-1)\sigma$ and $|a_k| = \sigma$, so that $\mathrm{fl}(\tilde{s} + a_k) = \tilde{s} + a_k$. Secondly, if $|\tilde{s} + a_k| < n\sigma$, then $\mathrm{ufp}(\tilde{s} + a_k) \leq (n-1)\sigma$ because $\mathrm{ufp}(\cdot)$ is a power of 2. Hence (2.19) implies

$$\begin{aligned} |\mathrm{fl}(\tilde{s} + a_k) - \sum_{i=1}^n a_i| &\leq |\mathrm{fl}(\tilde{s} + a_k) - (\tilde{s} + a_k)| + |\tilde{s} - \sum_{i \neq k} a_i| \\ &\leq \mathtt{eps} \cdot \mathrm{ufp}(\tilde{s} + a_k) + \frac{1}{2}(n-1)(n-2)\mathtt{eps}\sigma \\ &\leq \frac{1}{2}n(n-1)\mathtt{eps}\sigma \ , \end{aligned}$$

We mention that the factor can be improved to a little more than $n^2/3$, but we do not need this in the following.

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

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

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

The well known result by Sterbenz [19, Theorem 2.5] says that subtraction is exact if floating-point numbers $a, b \in \mathbb{F}$ of the same sign are not too far apart. More precisely, for $a, b \geq 0$ we have

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

We mention that a proof is not difficult with our machinery. If $b \geq a$, then (2.13) implies $a, b, a-b \in 2\texttt{eps}\sigma\mathbb{Z}$ for $\sigma := \mathrm{ufp}(a)$. By assumption and (2.9), $|b-a| = b-a \leq a < 2\sigma$, and (2.21) proves this part. For $b < a$, (2.13) implies $a, b, a-b \in 2\texttt{eps}\sigma\mathbb{Z}$ for $\sigma := \mathrm{ufp}(b)$, and similarly $|b-a| = a-b \leq b < 2\sigma$ and (2.21) finish the proof.

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

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

Using the ufp concept, the predecessor and successor of a floating-point number can be characterized as follows (note that $0 \neq |f| = \mathrm{ufp}(f)$ is equivalent to f being a power of 2).

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

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

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

$$\mathrm{pred}(f) \leq f - \texttt{eps} \cdot \mathrm{ufp}(f) \leq f + \texttt{eps} \cdot \mathrm{ufp}(f) \leq \mathrm{succ}(f) \ . \tag{2.23}$$

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

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

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

PROOF. For $f \notin \mathbb{U}$ and $|f| \neq \mathrm{ufp}(f)$, use $\mathrm{ufp}(f) < |f| < 2\mathrm{ufp}(f)$, and $|f| = \mathrm{ufp}(f)$ is equivalent to $|f|$ being a power of 2. The rest is not difficult to see. □

The aim of this paper is to present a summation algorithm computing a faithfully rounded exact result of the sum. That means [12, 41, 11] 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 2.3. *A floating-point number $f \in \mathbb{F}$ is called a* faithful rounding *of a real number $r \in \mathbb{R}$ if*

$$\mathrm{pred}(f) < r < \mathrm{succ}(f) \ . \tag{2.25}$$

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

For general $r \notin \mathbb{F}$, exactly two floating-point numbers satisfy $f \in \square(r)$, so at maximum half a bit accuracy is lost compared to rounding to nearest. Conversely, for the computation of a faithful rounding of a real number r it suffices to know r up to a small error margin. In contrast, the rounded-to-nearest $\mathrm{fl}(r)$ requires ultimately to know r exactly, namely if r is the midpoint of two adjacent floating-point numbers. This requires substantial and often not necessary computational effort. Our Algorithm `NearSum` in Part II of this paper computes the rounded to nearest result. The computing time depends in this case on the exponent range of the summands rather than the condition number of the sum.

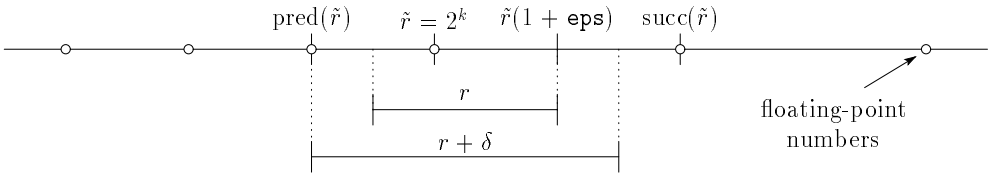

FIG. 2.2. *Faithful rounding near a power of* 2

In contrast, the computing time of Algorithm 4.5 (`AccSum`) for computing a faithfully rounded result of the sum of floating-point numbers is proportional to the logarithm of the condition number of the sum and independent of the exponent range of the summands, which is not true for Malcolm's approach [34] and the long accumulator [28].

Suppose $r + \delta$ is the exact result of a summation, composed of a (real) approximation r and an error term δ. Next we establish conditions on δ to ensure that $\mathrm{fl}(r)$ is a faithful rounding of $r + \delta$. The critical case is the change of exponent at a power of 2, as depicted in Figure 2.2.

LEMMA 2.4. *Let* $r, \delta \in \mathbb{R}$ *and* $\tilde{r} := \mathrm{fl}(r)$. *If* $\tilde{r} \notin \mathbb{U}$ *suppose* $2|\delta| < \mathtt{eps}|\tilde{r}|$, *and if* $\tilde{r} \in \mathbb{U}$ *suppose* $|\delta| < \frac{1}{2}\mathtt{eta}$. *Then* $\tilde{r} \in \square(r + \delta)$, *that means* $\tilde{r}$ *is a faithful rounding of* $r + \delta$.

PROOF. According to Definition 2.3 we have to prove $\mathrm{pred}(\tilde{r}) < r + \delta < \mathrm{succ}(\tilde{r})$. If $\tilde{r} \in \mathbb{U}$, then $|\tilde{r} - r| \le \frac{1}{2}\mathtt{eta}$ by (2.18), so Lemma 2.2 yields

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

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

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

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

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

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

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

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

$$\begin{array}{lcl} f_1 < f < f_2 & \Rightarrow & f_1 < r < f_2 \\ f_1 < r < f_2 & \Rightarrow & f_1 \le f \le f_2\ . \end{array} \tag{2.26}$$

As has been noted in (2.4), the error of a floating-point addition is always a floating-point number. Fortunately, rather than Algorithm 2.1 (`TwoSum`) which needs 6 flops, we can use in our applications the following faster algorithm due to Dekker [12], requiring only 3 flops. Again, the computation is very efficient because only standard floating-point addition and subtraction is used and no branch is needed.

ALGORITHM 2.5. *Compensated summation of two floating-point numbers.*

$$\begin{array}{l} \text{function } [x, y] = \mathtt{FastTwoSum}(a, b) \\ \quad x = \mathrm{fl}(a + b) \\ \quad q = \mathrm{fl}(x - a) \\ \quad y = \mathrm{fl}(b - q) \end{array}$$

In Dekker's original algorithm, y is computed by $y = \mathrm{fl}((a-x)+b)$, which is equivalent to the last statement in Algorithm 2.5 because $\mathbb{F} = -\mathbb{F}$ and $\mathrm{fl}(-r) = -\mathrm{fl}(r)$ for $r \in \mathbb{R}$. For floating-point arithmetic with rounding to nearest and base 2, e.g. IEEE 754 arithmetic, Dekker [12] showed in 1971 that the correction is *exact* if the input is ordered by magnitude, that is $x + y = a + b$ provided $|a| \geq |b|$. In [37] we showed that the obvious way to get rid of this assumption is suboptimal on today's computers because a branch slows down computation significantly.

Algorithm 2.5 (`FastTwoSum`) is an error-free transformation of the pair of floating-point numbers (a, b) into a pair (x, y). Algorithm 4.5 (`AccSum`) to be presented can also be viewed as an error-free transformation of a vector p into floating-point numbers τ_1, τ_2 and a vector p' such that $\sum p_i = \tau_1 + \tau_2 + \sum p'_i$, and $\texttt{res} := \mathrm{fl}(\tau_1 + (\tau_2 + \sum p'_i))$ is the faithfully rounded sum $\sum p_i$. To prove this we need to refine the analysis of Algorithm 2.5 by weakening the assumption $|a| \geq |b|$: The only assumption is that no trailing nonzero bit of the first summand a is smaller than the least significant bit of the second summand b.

LEMMA 2.6. *Let a, b be floating-point numbers with $a \in 2\texttt{eps} \cdot \mathrm{ufp}(b)\mathbb{Z}$. Let x, y be the results produced by Algorithm 2.5* (`FastTwoSum`) *applied to a, b. Then*

$$x + y = a + b \ , \quad x = \mathrm{fl}(a + b) \quad \textit{and} \quad |y| \leq \texttt{eps} \cdot \mathrm{ufp}(a + b) \leq \texttt{eps} \cdot \mathrm{ufp}(x) \ . \tag{2.27}$$

Furthermore,

$$q = \mathrm{fl}(x - a) = x - a \quad \textit{and} \quad y = \mathrm{fl}(b - q) = b - q \ , \tag{2.28}$$

that is 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 (2.13) and (2.11) implies $a \in 2\texttt{eps} \cdot \mathrm{ufp}(b)\mathbb{Z}$, the assumption of Lemma 2.6.

PROOF OF LEMMA 2.6. Let $\mathrm{fl}(a+b) = a+b+\delta$ and denote $\sigma := 2\mathrm{ufp}(b)$. Note that $a, b \in \texttt{eps}\sigma\mathbb{Z}$ and $|b| < \sigma$. If $\sigma \leq |a|$, then $|b| < |a|$ and we can use Dekker's result [12]. Otherwise, $|a + b| < 2\sigma$, so (2.19) implies $|\delta| \leq \texttt{eps}\sigma$. In fact, either $|\delta| = \texttt{eps}\sigma$ or $\delta = 0$ by (2.15). Hence $|x - a| = |b + \delta| \leq \mathrm{pred}(\sigma) + \texttt{eps}\sigma \leq \sigma$, so (2.21) yields $q = \mathrm{fl}(x - a) = x - a$, and therefore $y = \mathrm{fl}(b - q) = \mathrm{fl}(-\delta) = -\delta = b - q$, proving (2.28). Hence $x + y = x - \delta = a + b$. The estimation on $|y| = |\delta|$ follows by (2.19), and this finishes the proof. □

Lemma 2.6 may also offer possibilities for summation algorithms based on sorting: To apply `FastTwoSum` it suffices to "sort by exponent", which has complexity $\mathcal{O}(n)$.

3. Extraction of high order parts. In [50] Zielke and Drygalla presented an accurate dot product algorithm. Their Matlab source code is repeated in Algorithm 3.1. In the first six lines, the dot product $\sum \mathtt{a_i b_i}$ is transformed into the sum $\sum \mathtt{v_i}$ of length $\mathtt{2n}$ using Veltkamp's algorithm `TwoProduct` [12]. Provided no over- or underflow occurs, this transformation is error-free, i.e. $\sum_{i=1}^{\mathtt{n}} \mathtt{a_i b_i} = \sum_{i=1}^{\mathtt{2n}} \mathtt{v_i}$.

We repeat their main idea in technical terms. Let $\mathtt{2n}$ floating-point numbers $\mathtt{v_i}$ be given. First $\mathtt{emax} \in \mathbb{N}$ is computed such that $\max|\mathtt{v_i}| < 2^{\mathtt{emax}}$, and $M := \lceil \log_2 \mathtt{2n} \rceil$ such that $\mathtt{2n} \leq 2^{\mathtt{M}}$. Then, for $\texttt{eps} = 2^{-53}$, they extract in line 13 the $54 - M$ "bits" from $\mathtt{emax - 1}$ downto $\mathtt{emax - (54 - M)}$ of $\mathtt{v_i}$ into leading parts $\mathtt{g}_\nu$ and a vector of remainder parts (stored in the same vector $\mathtt{v}$), and add the $\mathtt{g}_\nu$ into $\mathtt{s}_1$. The value M should be chosen so that the sum of the $\mathtt{g}_\nu$ is exact. Then they continue by extracting the $54 - M$ "bits" from $\mathtt{emax - (54 - M) - 1}$ downto $\mathtt{emax - 2(54 - M)}$ of the remainder part and sum them into $\mathtt{s}_2$, and so forth. This process is continued until the vector of remainder parts is entirely zero or in the underflow range. Note that the (scaled) intermediate sums $\mathtt{s_j}$ may overlap. If the sum of the $\mathtt{g}_\nu$ is error-free, then after execution of line 15 it holds

$$\sum_{\nu=1}^{\mathtt{2n}} \mathtt{v}_\nu = \sum_{j=1}^{j_{max}} \mathtt{s_j} \cdot 2^{\mathtt{emax - j \cdot k}} \ , \tag{3.1}$$

where j_{max} denotes the length of the array s.

ALGORITHM 3.1. *The Matlab code by Zielke and Drygalla [50]*

```
   function y = accdot(a,b,n)
01    X = a.*b;
02    p = 2^27 + 1;
03    h = p*a; a1 = h - (h - a); a2 = a - a1;
04    h = p*b; b1 = h - (h - b); b2 = b - b1;
05    x = (((a1.*b1 - X) + a1.*b2) + b1.*a2) + b2.*a2;
06    v = [X; x];
07    ma = max(abs(X));
08    [mmax,emax] = log2(ma);             q = 2^emax;
09    k = floor(54 - log(2*n)/log(2));    p = 2^k;
10    i = 0;
11    while any(v ~= zeros(2*n,1)) & (q/p > 2^(-1023))
12      i = i + 1;
13      q = q/p;  g = fix(v/q); s(i) = sum(g);  v = v - g*q;
14    end
15    i = i + 1; s(i) = sum(v)*p/q; ue = 0;
16    for j = i : (-1) : 1
17      t = s(j) + ue; ue = floor(t/p); s(j) = t - ue*p;
18    end
19    y = ue;
20    for j = 1 : i, y = s(j) + y*p;  end
21    y = y*q/p;
```

Next the overlapping parts of the intermediate sums $\mathtt{s_j}$ are eliminated in lines 16 to 18 starting with $j = j_{max}$. Finally, the scaled $\mathtt{s_j}$ are added in line 20 starting with $j = 1$, and in line 21 a final scaling takes place.

The authors also mention a faster, though much less accurate method, by extracting only the leading $54 - M$ bits and adding the remainder terms, accepting accumulation of rounding errors. A similar method was used in "poor men's residual (`lssresidual`)" in INTLAB [43] to approximate the residual $Ax - b$ of a linear system, a purely heuristic, improved approximation.

For neither method Zielke and Drygalla give an error analysis. The accuracy of the result of their algorithm is not clear. At least the summation part (from line 7) does not necessarily produce a faithfully rounded result: For example, the vector $\mathtt{v} = [1-\mathtt{eps} \quad 1-\mathtt{eps} \quad 7\mathtt{eps} \quad 0]$ yields the result $\mathtt{y} = 2$, whereas the true sum satisfies

$$\sum_{\nu=1}^{4} \mathtt{v}_\nu = 2 + 5\mathtt{eps} > 2 + 4\mathtt{eps} = \mathrm{succ}(2) \ .$$

Note, however, that this vector v cannot be produced by lines $1 - 5$. Moreover, a poor scaling of the summands is used, severely restricting the exponent range. Consider

$$a = [\ 2^{458} \quad \mathrm{pred}(2^{-458}) \quad -\mathrm{pred}(2^{-458})\ ]^T \quad \text{and} \quad b = [\ 1 \quad 1 \quad 1\ ]^T \ ,$$

where "pred" denotes the next smaller floating-point number. Due to poor scaling, their Algorithm 3.1 computes the result `inf`. This is because with $\mathtt{emax} = 459$, $\mathtt{k} = 51$ and $j_{max} = 21$ we have $\mathtt{q/p} = 2^{\mathtt{emax}-\mathtt{j_{max}}\cdot\mathtt{k}} = 2^{-612}$, after line 15 the vector s has $\mathtt{j_{max}} = 21$ elements with the only nonzero element $\mathtt{s}_1 = 2^{50}$,

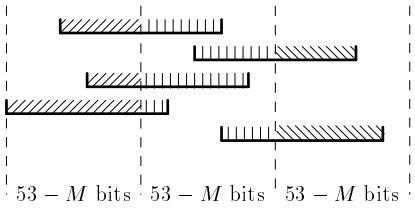

FIG. 3.1. *Extraction scheme based on [50]*

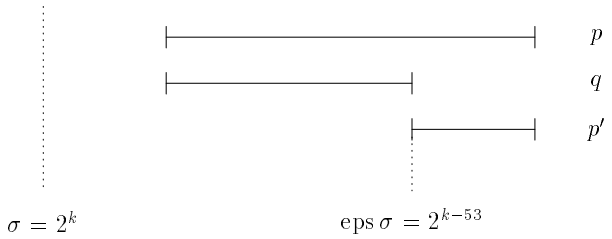

FIG. 3.2. `ExtractScalar`*: error-free transformation* $p = q + p'$

so that lines 20 and 21 compute

$$\mathtt{y} = \Big(\sum_{\mathtt{j}=1}^{\mathtt{j_{max}}} \mathtt{s_j} \cdot 2^{(\mathtt{j_{max}}-\mathtt{j})\mathtt{k}}\Big) \cdot 2^{\mathtt{emax}-\mathtt{j_{max}}\cdot\mathtt{k}} = 2^{1070} \cdot 2^{-612} = 2^{458} \ .$$

However, the intermediate result 2^{1070} causes an overflow. Note that $p \approx [7.4 \cdot 10^{137}\ 1.3 \cdot 10^{-138}\ -1.3 \cdot 10^{-138}]$ is far from over- or underflow since $\max\{f \in \mathbb{F}\} \approx 1.8 \cdot 10^{308}$.

Zielke and Drygalla's algorithm can be improved in several ways. First, the extraction can be stopped when the remainder parts are small enough, only the necessary intermediate sums $\mathtt{s_j}$ are computed. Second, the extraction in line 13 will be improved significantly, see below. Third, the elimination of the overlapping parts in lines 18 to 20 is unnecessary. Fourth, there is an easier way to compute the final result. Doing this requires an analysis. In particular the constant `k` in line 9 has to be computed correctly.

Our approach follows a similar scheme as depicted in Figure 3.1, where we carefully estimate how many bits have to be extracted to guarantee a faithful rounding of the result. We push the approach to the limits by showing that our constants are optimal.

The inner loop of the method requires to split a floating-point number $p \in \mathbb{F}$ according to Figure 3.2, which is done by scaling and chopping (`fix`) in line 13 in Algorithm 3.1. There are other possibilities, for example to round from floating-point to integer (rather than chopping), or the assignment of a floating-point number to a long integer (if supported by the hardware in use). Also direct manipulation by accessing mantissa and exponent is possible. However, all these methods slow down the extraction significantly, often by an order of magnitude and more compared to our following Algorithm 3.2 (`ExtractScalar`). We will show corresponding performance data in Section 5.

For our splitting as depicted in Figure 3.2, neither the high order part q and low order part p' need to match bitwise with the original p, nor must q and p' have the same sign; only the error-freeness of the transformation $p = q + p'$ is mandatory. This is achieved by the following fast algorithm, where σ denotes a power of 2 not less than $|p|$.

ALGORITHM 3.2. *Error-free transformation extracting high order part.*

function $[q, p'] = \texttt{ExtractScalar}(\sigma, p)$
$q = \mathrm{fl}\left((\sigma + p) - \sigma\right)$
$p' = \mathrm{fl}(p - q)$

There is an important difference to Dekker's splitting [12]. There, a 53-bit floating-point number is split into two parts relative to its exponent, and using a sign bit both the high and the low part have at most 26 significant bits in the mantissa. In `ExtractScalar` a floating-point number is split relative to σ, a fixed power of 2. The higher and the lower part of the splitting may have between 0 and 53 significant bits, depending on σ. The splitting for $p \in \mathbb{F}$ and $-p$ need not be symmetric because σ is positive. For example, $\sigma = 2^{53}$ splits $p = 1$ into $q = 0$ and $p' = 1$ because of rounding tie to even, whereas $p = -1$ is split into $q = -1$ and $p' = 0$.

The clever way of splitting[2] in Algorithm 3.2 (`ExtractScalar`) is crucial for the performance since it is in the inner loops of our algorithms. We think this method is known, at least similar ideas are around [17, 6]. However, we do not know of an analysis of Algorithm 3.2, so we develop it in the following lemma.

LEMMA 3.3. *Let q and p' be the results of Algorithm 3.2* (`ExtractScalar`) *applied to floating-point numbers σ and p. Assume $\sigma = 2^k \in \mathbb{F}$ for some $k \in \mathbb{Z}$, and assume $|p| \le 2^{-M}\sigma$ for some $0 \le M \in \mathbb{N}$. Then*

$$p = q + p' \ , \quad |p'| \le \texttt{eps}\sigma \ , \quad |q| \le 2^{-M}\sigma \quad \textit{and} \quad q \in \texttt{eps}\sigma\mathbb{Z} \ . \tag{3.2}$$

PROOF. We first note that $\texttt{ExtractScalar}(\sigma, p)$ performs exactly the same operations in the same order as $\texttt{FastTwoSum}(\sigma, p)$, so $|p| \le \sigma$ and Lemma 2.6 imply $p' = p - q$. If $|p| = \sigma$, then $p' = 0$, otherwise (2.27) implies $|p'| \le \texttt{eps} \cdot \mathrm{ufp}(\sigma + p) \le \texttt{eps}\sigma$. Furthermore, $q \in \texttt{eps}\sigma\mathbb{Z}$ follows by (2.16).

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

REMARK. As is seen from the last part of the proof, rounding tie to even is necessary to ensure $|q| \le 2^{-M}\sigma$.

Following we adapt Algorithm 3.2 (`ExtractScalar`) to the error-free transformation of an entire vector. For this case we prove that the high order parts can be summed up without error. For better readability and analysis the extracted parts are stored in a vector q_i. In a practical implementation, the vector q is not necessary but only its sum τ.

ALGORITHM 3.4. *Error-free vector transformation extracting high order part.*

function $[\tau, p'] = \texttt{ExtractVector}(\sigma, p)$
$\tau = 0$
for $i = 1 : n$
$[q_i, p'_i] = \texttt{ExtractScalar}(\sigma, p_i)$
$\tau = \mathrm{fl}(\tau + q_i)$
end for

[2]This idea was pointed out to the second author by Prof. Yasunori Ushiro.

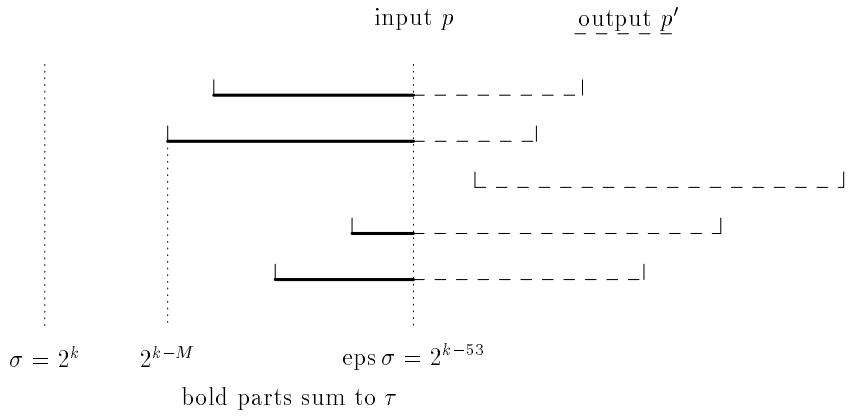

FIG. 3.3. `ExtractVector`*: error-free transformation* $\sum p_i = \tau + \sum p'_i$

Algorithm 3.4 proceeds as depicted in Figure 3.3. Note that the loop is well-suited for today's compilers optimization and instruction-level parallelism.

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

THEOREM 3.5. *Let* τ *and* p' *be the results of Algorithm 3.4* (`ExtractVector`) *applied to* $\sigma \in \mathbb{F}$ *and a vector of floating-point numbers* $p_i, 1 \le i \le n$. *Assume* $\sigma = 2^k \in \mathbb{F}$ *for some* $k \in \mathbb{Z}$, $n < 2^M$ *for some* $M \in \mathbb{N}$ *and* $|p_i| \le 2^{-M}\sigma$ *for all* i. *Then*

$$\sum_{i=1}^{n} p_i = \tau + \sum_{i=1}^{n} p'_i \;, \quad \max|p'_i| \le \mathtt{eps}\sigma \;, \quad |\tau| \le n2^{-M}\sigma < \sigma \quad \textit{and} \quad \tau \in \mathtt{eps}\sigma\mathbb{Z} \;. \tag{3.3}$$

If $2^{2M}\mathtt{eps} \le 1$, *then*

$$|\mathrm{fl}(\tau + T)| < \sigma \quad \textit{for} \quad T := \mathrm{fl}(\sum_{i=1}^{n} p'_i) \;. \tag{3.4}$$

Algorithm 3.4 (`ExtractVector`) *needs* $4n + \mathcal{O}(1)$ *flops.*

REMARK 1. The limitation to $n < 2^M$ is necessary to ensure (3.4) as is shown by a vector of length 2^M with all elements equal to $2^{-M}\sigma$.

REMARK 2. Note that for (3.3) we have no assumption on the size of M. However, for extremely large M with $2^M\mathtt{eps} \ge 1$, $|p_i| \le 2^{-M}\sigma$ implies that $p'_i = p_i$ except for $p_i = -\mathtt{eps}\sigma$, so not much is gained.

PROOF OF THEOREM 3.5. Lemma 3.3 implies $p_i = q_i + p'_i$, $|p'_i| \le \mathtt{eps}\sigma$, $|q_i| \le 2^{-M}\sigma$ and $q_i \in \mathtt{eps}\sigma\mathbb{Z}$ for all $i \in \{1, \cdots, n\}$. So $\tau \in \mathtt{eps}\sigma\mathbb{Z}$, and $\sum|q_i| \le n2^{-M}\sigma < \sigma$ and (2.21) imply $\tau = \mathrm{fl}(\sum q_i) = \sum q_i$. This proves (3.3). To show (3.4) first note that we may assume $\sigma \notin \mathbb{U}$, because otherwise all p'_i are zero and the assertion is trivial. Now (3.3) and (2.20) imply $|T| \le n\mathtt{eps}\sigma$, so Lemma 2.2 implies

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

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

Note that (3.3) requires no assumption on σ other than being a power of 2 and bounding $2^M|p_i|$; σ may well be in the underflow range.

To apply Theorem 3.5 the best (smallest) value for M is $\lceil \log_2(n+1) \rceil$. To avoid the use of the binary logarithm, this can be calculated by the following algorithm.

ALGORITHM 3.6. *Computation of* $2^{\lceil \log_2 |p| \rceil}$ *for* $p \neq 0$.

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

THEOREM 3.7. *Let* L *be the result of Algorithm 3.6* (`NextPowerTwo`) *applied to a nonzero floating-point number* p. *If no overflow occurs, then* $L = 2^{\lceil \log_2 |p| \rceil}$.

REMARK. For simplicity we skipped the obvious check for large input number p to avoid overflow in the computation of q. However, we will show that $L = 2^{\lceil \log_2 |p| \rceil}$ is satisfied in the presence of underflow. As the proof will show, rounding tie to even as in the IEEE 754 arithmetic standard is mandatory.

PROOF. First assume $|p| = 2^k$ for some $k \in \mathbb{Z}$. Then the rounding tie to even implies $\mathrm{fl}(q+p) = \mathrm{fl}(q(1+\mathtt{eps})) = q$, also for $p \in \mathbb{U}$, so that $\mathrm{fl}(|(q+p)-q|) = 0$, and $L = |p| = 2^k = 2^{\lceil \log_2 |p| \rceil}$ for the final result L. So we may assume that p is not a power of 2, and without loss of generality we assume $p > 0$. Then

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

and we have to show $L = 2\mathrm{ufp}(p)$. Define $x := \mathrm{fl}(q+p)$. By Lemma 2.6 the computation of $\mathrm{fl}(x-q)$ causes no rounding error, so that $L = \mathrm{fl}(q+p) - q$. By definition, $\mathrm{ufp}(q) = \mathtt{eps}^{-1}\mathrm{ufp}(p) < \mathtt{eps}^{-1}p = q$, so that $q \notin \mathbb{U}$ and Lemma 2.2 imply $\mathrm{succ}(q) = q + 2\mathtt{eps} \cdot \mathrm{ufp}(q)$. That means $q + \mathtt{eps} \cdot \mathrm{ufp}(q)$ is the midpoint of q and $\mathrm{succ}(q)$. Hence rounding to nearest and

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

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

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

The main part of the summation algorithm is the error-free transformation of the input vector $p^{(0)}$ into two floating-point numbers τ_1, τ_2 representing the high order part of the sum and a vector $p^{(m)}$ of low order parts. The transformation is error-free, i.e. $s := \sum p_i^{(0)} = \tau_1 + \tau_2 + \sum p_i^{(m)}$. The goal is to prove that $\mathrm{fl}\big(\tau_1 + (\tau_2 + (\sum p_i^{(m)}))\big)$ is a faithfully rounded result of s. The following formulation of the transformation algorithm is aimed on readability rather than efficiency. In particular we avoid in this first step a check for zero sum.

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

function $[\tau_1, \tau_2, p^{(m)}, \sigma] = \texttt{Transform}(p^{(0)})$
  $\mu = \max(|p_i^{(0)}|)$
  if $\mu = 0,\ \tau_1 = \tau_2 = p^{(m)} = \sigma = 0,$ return, end if
  $M = \lceil \log_2 \left(\text{length}(p^{(0)}) + 2\right) \rceil$
  $\sigma_0 = 2^{M + \lceil \log_2(\mu) \rceil}$
  $t^{(0)} = 0,\ m = 0$
  repeat
    $m = m + 1$
    $[\tau^{(m)}, p^{(m)}] = \texttt{ExtractVector}(\sigma_{m-1}, p^{(m-1)})$
    $t^{(m)} = \text{fl}(t^{(m-1)} + \tau^{(m)})$
    $\sigma_m = \text{fl}(2^M \texttt{eps} \sigma_{m-1})$
  until $|t^{(m)}| \geq \text{fl}(2^{2M} \texttt{eps} \sigma_{m-1})$ or $\sigma_{m-1} \leq \frac{1}{2} \texttt{eps}^{-1} \texttt{eta}$
  $\sigma = \sigma_{m-1}$
  $[\tau_1, \tau_2] = \texttt{FastTwoSum}(t^{(m-1)}, \tau^{(m)})$

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

REMARK 2. For clarity we also use for the moment the logarithm in the computation of M and σ_0. Later this will be replaced by Algorithm 3.6 (`NextPowerTwo`) (see the Matlab code given in the Appendix).

Before we start to prove properties of Algorithm 4.1 (`Transform`), let's interpret it. The "repeat-until"-loop extracts the vector $p^{(m-1)}$ into the sum $\tau^{(m)}$ of its leading parts and into the vector $p^{(m)}$ of remaining parts. Theorem 3.5 ensures that no rounding error occurs in the computation of the sum $\tau^{(m)}$. The main property of the algorithm is to guarantee the error-free transformation

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

for all m, in particular for the final one. For that it is mandatory that $t^{(m)} = \text{fl}(t^{(m-1)} + \tau^{(m)})$ in the "repeat-until"-loop is computed without rounding error if the loop is not yet finished because this value is used in the next loop. We first prove some properties of the algorithm and come to that again in the remarks after Lemma 4.3. As we will see, the constant $2^{2M}\texttt{eps}$ in the stopping criterion is chosen optimal.

LEMMA 4.2. *Let* $\tau_1, \tau_2, p^{(m)}, \sigma$ *be the results of Algorithm 4.1* (`Transform`) *applied to a nonzero vector of floating-point numbers* $p_i^{(0)}, 1 \leq i \leq n$. *Define* $M := \lceil \log_2(n+2) \rceil$ *and assume* $2^{2M}\texttt{eps} \leq 1$. *Furthermore, define* $\mu := \max_i |p_i^{(0)}|$ *and* $\sigma_0 = 2^{M + \lceil \log_2 \mu \rceil}$. *Denote* $s := \sum_{i=1}^{n} p_i^{(0)}$.

Then Algorithm 4.1 will stop, and

$$s = t^{(m-1)} + \tau^{(m)} + \sum_{i=1}^{n} p_i^{(m)}\ , \tag{4.2}$$

$$\max |p_i^{(m)}| \leq \texttt{eps}\sigma_{m-1}\ , \quad |\tau^{(m)}| \leq (1 - 2^{-M})\sigma_{m-1} < \sigma_{m-1} \quad \text{and} \quad t^{(m-1)}, \tau^{(m)} \in \texttt{eps}\sigma_{m-1}\mathbb{Z} \tag{4.3}$$

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

$$\tau_1 + \tau_2 = t^{(m-1)} + \tau^{(m)}\ , \quad \tau_1 = \text{fl}(\tau_1 + \tau_2) = \text{fl}(t^{(m-1)} + \tau^{(m)}) = t^{(m)} \tag{4.4}$$

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

$$\text{ufp}(\tau_1) \geq 2^{2M}\texttt{eps}\sigma_{m-1}\ . \tag{4.5}$$

Algorithm 4.1 (`Transform`*) requires* $(4m+2)n + \mathcal{O}(m)$ *flops for* m *executions of the "repeat-until"-loop.*

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

REMARK 2. In Lemma 3.4 in Part II of this paper we will show that the assertions of Lemma 4.2 remain essentially true when replacing the quantity $2^{2M}\texttt{eps}\sigma_{m-1}$ in the first inequality of the "until"-condition by $\Phi\sigma_{m-1}$, where Φ denotes a power of 2 between 1 and `eps`. The chosen factor $\Phi = 2^{2M}\texttt{eps}$ in Algorithm 4.1 (`Transform`) is the smallest choice to guarantee faithful rounding, see Remark 2 after Lemma 4.3.

However, it may not be increased beyond σ_{m-1}: By the extraction we know by (4.3) the lower bound $\texttt{eps}\sigma_{m-1}$ of the unit in the last place of $\tau^{(m)}$. So if $|t^{(m)}|$ is small enough, then no rounding error has occurred in $t^{(m)} = \mathrm{fl}(t^{(m-1)} + \tau^{(m)})$. This may be jeopardized if the "until"-condition is too weak, that is if $2^{2M}\texttt{eps}\sigma_{m-1}$ is replaced by $2\sigma_{m-1}$.

To see that, consider $p^{(0)} = \begin{bmatrix}(8\texttt{eps})^{-1} & -(8\texttt{eps})^{-1} & 8 & 1+8\texttt{eps}\end{bmatrix}$ in a floating-point format with relative rounding error unit `eps`. Then $n = 4$ so that $M = 3$, $\mu = (8\texttt{eps})^{-1}$ and $\sigma_0 = \texttt{eps}^{-1}$. One verifies that `ExtractVector` produces $p^{(1)} = \begin{bmatrix}0 & 0 & 0 & -1+8\texttt{eps}\end{bmatrix}$ and $t^{(1)} = \tau^{(1)} = 10$. In the next iteration $p^{(2)}$ is the zero vector, so that $\tau^{(2)} = \sum p_i^{(1)} = -1 + 8\texttt{eps}$ and

$$t^{(2)} = \mathrm{fl}(t^{(1)} + \tau^{(2)}) = \mathrm{fl}(10 + (-1 + 8\texttt{eps})) = 9 \ .$$

Hence

$$|t^{(2)}| \geq 8 = \sigma_1$$

implies that the "repeat-until"-loop in Algorithm 4.1 is finished. If the first inequality in the "until"-condition would be replaced by $|t^{(m)}| \geq 2\sigma_{m-1}$, which is 16 in our example, then there would be a next loop. However, $t^{(1)} + \tau^{(2)} = 9 + 8\texttt{eps}$ is not representable with the relative rounding error unit `eps`, so $t^{(2)} = \mathrm{fl}(t^{(1)} + \tau^{(2)}) \neq t^{(1)} + \tau^{(2)}$ would be used and (4.1) is not valid.

PROOF OF LEMMA 4.2. Algorithm `Transform` will stop because σ_m is decreased by a factor $2^M\texttt{eps} < 1$ in each loop. We proceed by induction to prove (4.2) and (4.3). The initialization in Algorithm 4.1 implies $\max|p_i^{(0)}| = \mu \leq 2^{\lceil \log_2(\mu)\rceil} = 2^{-M}\sigma_0$, so that the assumptions of Theorem 3.5 are satisfied for σ_0 and $p^{(0)}$ as input to `ExtractVector`. Note this is also true if $\sigma_0 \in \mathbb{U}$. This proves (4.3) for $m = 1$. Furthermore, $s = \tau^{(1)} + \sum p_i^{(1)}$, and (4.2) is also proved for $m = 1$.

Next assume the "repeat-until"-loop has been executed, denote by m the current value (immediately before the "until"-statement), and assume that (4.2) and (4.3) are true for the previous index $m-1$. The previous "until"-condition in Algorithm 4.1 implies $\sigma_{m-2} > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, so $\sigma_{m-2} \geq \texttt{eps}^{-1}\texttt{eta}$ because σ_{m-2} is a power of 2. Hence no rounding error has occurred in the previous computation of $2^{2M}\texttt{eps}\sigma_{m-2}$. So the induction hypothesis and the "until"-condition yield

$$|t^{(m-1)}| = |\mathrm{fl}(t^{(m-2)} + \tau^{(m-1)})| < 2^{2M}\texttt{eps}\sigma_{m-2} \leq \sigma_{m-2} \ , \tag{4.6}$$

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

$$t^{(m-1)} = \mathrm{fl}(t^{(m-2)} + \tau^{(m-1)}) = t^{(m-2)} + \tau^{(m-1)} \ , \tag{4.7}$$

and the induction hypothesis on (4.2) yields

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

By (4.3) we know $\max|p_i^{(m-1)}| \le \texttt{eps}\sigma_{m-2} = 2^{-M}\sigma_{m-1}$. Hence Theorem 3.5 is applicable and shows $\sum p_i^{(m-1)} = \tau^{(m)} + \sum p_i^{(m)}$, and therefore (4.2) for index m. It also shows (4.3), where $t^{(m-1)} \in \texttt{eps}\sigma_{m-1}\mathbb{Z}$ follows by $t^{(m-2)}, \tau^{(m-1)} \in \texttt{eps}\sigma_{m-2}\mathbb{Z}$, (4.7) and (2.11). We proved (4.2) and (4.3). Therefore, for the last line in Algorithm `Transform` the assumptions of Lemma 2.6 are satisfied and (4.4) follows.

If $\sigma_{m-1} > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ is satisfied for the final value of m, then $\sigma_{m-1} \ge \texttt{eps}^{-1}\texttt{eta}$. Hence $\mathrm{fl}(2^{2M}\texttt{eps}\sigma_{m-1}) = 2^{2M}\texttt{eps}\sigma_{m-1}$, and the "until"-condition and (4.4) yield $|\tau_1| = |t^{(m)}| \ge 2^{2M}\texttt{eps}\sigma_{m-1}$, which implies (4.5). The lemma is proved. □

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

LEMMA 4.3. *Let p be a nonzero vector of n floating-point numbers. Let* `res` *be computed as follows:*

$$\begin{aligned} &[\tau_1, \tau_2, p', \sigma] = \texttt{Transform}(p) \\ &\texttt{res} = \mathrm{fl}(\tau_1 + (\tau_2 + (\textstyle\sum_{i=1}^{n} p_i'))) \end{aligned} \tag{4.8}$$

Define $M := \lceil \log_2(n+2) \rceil$, and assume $2^{2M}\texttt{eps} \le 1$. Furthermore, define $\mu := \max_i |p_i|$ and $\sigma_0 = 2^{M+\lceil \log_2 \mu \rceil}$.

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

$$s = \tau_1 + \tau_2 + \sum_{i=1}^{n} p_i' \quad \text{and} \quad \max|p_i'| \le \texttt{eps}\sigma \ , \tag{4.9}$$

$$\mathrm{fl}(\tau_1 + \tau_2) = \tau_1 \ , \quad \tau_1, \tau_2 \in \texttt{eps}\sigma\mathbb{Z} \quad \text{and} \quad |\tau_2| \le \texttt{eps} \cdot \mathrm{ufp}(\tau_1) \ , \tag{4.10}$$

$$|s - \texttt{res}| < 2\texttt{eps}(1 - 2^{-M-1})\mathrm{ufp}(\texttt{res}) \ . \tag{4.11}$$

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

If `res` $= 0$*, then $s = \tau_1 = \tau_2 = 0$ and all components of the vector p' are zero.*

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

$$\mathrm{ufp}(\tau_1) \ge 2^{2M}\texttt{eps}\sigma \ . \tag{4.12}$$

The exponent $2M$ in the first inequality in the "until"-condition is optimal: If it is changed into another integer, then `res` *need not be a faithful rounding of s.*

REMARK 1. The proof will be based on Lemma 2.4, where we developed conditions on δ so that $\tilde{r} = \mathrm{fl}(r)$ is a faithful rounding of $r + \delta$. For (4.8) this means $r := \tau_1 + \mathrm{fl}(\tau_2 + (\sum p_i'))$ and $\texttt{res} = \mathrm{fl}(r)$, so that by (4.2) δ is the rounding error in the computation of $\mathrm{fl}(\tau_2 + (\sum p_i'))$. As we will see, the error in the computation of $\mathrm{fl}(\sum p_i')$, which is estimated by (2.20), is the critical one. In order not to spoil the faithful rounding, the dominant part r, which is of the order τ_1, must be large enough compared to that error. But τ_1 is the output of Algorithm 2.5 (`FastTwoSum`), so $\tau_1 = \mathrm{fl}(t^{(m-1)} + \tau^{(m)}) = t^{(m)}$, and we see that the stopping criterion of the "repeat-until"-loop in Algorithm 4.1 (`Transform`) must ensure that $|t^{(m)}|$ is large enough.

REMARK 2. To ensure that $|t^{(m)}|$ is large enough for faithful rounding, the chosen lower bound $2^{2M}\texttt{eps}\sigma_{m-1}$ in the "until"-condition is the smallest possible choice. To see that, consider the example given in Table 4.1; this also gives further insight into how Algorithm 4.1 and (4.8) in Lemma 4.3 work together. The computation is performed in IEEE 754 double precision, i.e. $\texttt{eps} = 2^{-53}$.

The input vector $p_i := p_i^{(0)}$ has 62 elements. We have $M = 6$, $\mu = 2^{47}$ and $\sigma = 2^{53}$. It follows $p_i^{(1)} = 0$ for $i = 1, 2$, and $p_i^{(1)} = p_i^{(0)} = p_i$ for $i \ge 3$. So $t^{(1)} = \tau^{(1)} = p_1 + p_2 = -2^{11} = -2^{2M-1}\texttt{eps}\sigma$. With the

TABLE 4.1
Example to show that "until"-condition $|t^{(m)}| \geq \mathrm{fl}(2^{2M-1}\mathtt{eps}\sigma_{m-1})$ may not yield a faithfully rounded result

$p_1 = 2^{47} - 2^{11}$	
$p_2 = -2^{47}$	
100000000000...00000000000000000000	$p_3 = 1$
11111111111...11111111111111111111	$p_4 = 1 - \mathtt{eps}$
11111111111...11111111111111111110	$p_{5\cdots 6} = 1 - 2\mathtt{eps}$
11111111111...11111111111111111100	$p_{7\cdots 10} = 1 - 4\mathtt{eps}$
11111111111...11111111111111111000	$p_{11\cdots 18} = 1 - 8\mathtt{eps}$
11111111111...11111111111111110000	$p_{19\cdots 34} = 1 - 16\mathtt{eps}$
11111111111...11111111111111100000	$p_{35\cdots 61} = 1 - 32\mathtt{eps}$
-1110000000000...	$p_{62} = -7 \cdot 2^{-46}$
-10000000000000000000...000000000	$\tau_1 = \tau^{(1)} = t^{(1)} = -2^{11}$
11101011111111111...111111111110010	$\tau_3 := \mathrm{fl}(\sum_{i=1}^{n} p_i^{(1)}) = \mathrm{fl}(\sum_{i=3}^{n} p_i)$
-11111000101000000000...000000000001110000	$\tau_1 + \tau_3$
-1111100010100000000000...0000000000	$\mathtt{res} = \mathrm{fl}(\tau_1 + \tau_3)$
-1111100010100000000000...00000000100000110101	$s = \sum_{i=1}^{62} p_i = -1989 - 2101\mathtt{eps}$
-10010110101	$\sum_{i=1}^{62} p_i^{(1)} - \tau_3$

original "until"-condition in Algorithm 4.1 (`Transform`), the loop would continue. If the stopping criterion is replaced by $|t^{(m)}| \geq \mathrm{fl}(2^{2M-1}\mathtt{eps}\sigma_{m-1})$, then the loop is finished. For the remainder term $\sum p_i^{(1)}$, each floating-point addition $\mathrm{fl}(\alpha_i) := \mathrm{fl}(\alpha_{i-1} + p_i^{(1)})$ where $\alpha_3 := p_3^{(1)}$ produces a maximal rounding error for $4 \leq i \leq 61$, and the final $p_{62}^{(1)}$ adds some noise to spoil the faithful rounding.

REMARK 3. To minimize the number of loops, the lower bound in the "until"-condition should be as small as possible, thus Remark 2 shows that the chosen lower bound $2^{2M}\mathtt{eps}\sigma_{m-1}$ is optimal in that sense, it cannot be decreased. But with the assumption $2^{2M}\mathtt{eps} \leq 1$ this lower bound can be as large as σ_{m-1}, thus Remark 2 after Lemma 4.2 shows that it can also not be increased, so that the chosen lower bound is the only possible power of 2 to guarantee faithful rounding.

REMARK 4. In our treatment of huge vector lengths (Part II, Lemma 3.4) we weaken the assumption $2^{2M}\mathtt{eps} \leq 1$ into $2^{M}\mathtt{eps} < 1$ which allows much larger dimensions, and we show that the factor in the "until"-condition may be decreased without jeopardizing the assertions of Lemma 4.2; however, then `res` as computed in (4.8) is no longer faithful. But this can be cured, see Algorithm 8.1 and Proposition 8.2 in Part II of this paper.

REMARK 5. By changing the first vector element p_1 in the example in Table 4.1 from $2^{47} - 2^{11}$ into $2^{47} - 2^{12}$, we can monitor the significant influence of the remainder part $\sum p_i'$: In this case (4.8) computes $\mathtt{res} = -4037$, whilst $\mathrm{fl}(\tau_1 + \tau_2) = \tau_1 = -4096$. Note that the rounded-to-nearest result is $\mathrm{pred}(-4037)$, one of the rare cases where it differs from `res`.

PROOF OF LEMMA 4.3. For the final value of m in Algorithm 4.1 (`Transform`), $p' = p^{(m)}$, $\sigma = \sigma_{m-1}$, (4.2), (4.4) and (4.3) yield (4.9). Moreover, (4.4), (4.3), (2.15) and (2.19) imply (4.10). The assertions about the exponent $2M$ have already been addressed in Remark 3.

If $\sigma \leq \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ for the final value of m in Algorithm 4.1 (`Transform`), then (4.9) implies $|p_i'| < \mathtt{eta}$, so all components of the vector p' must be zero. Hence $s = \tau_1 + \tau_2$, and $\mathtt{res} = \mathrm{fl}(\tau_1 + \tau_2) = \mathrm{fl}(s)$, which is of course a faithful rounding. The other assertions for the case $\mathtt{res} = 0$ follow as well. Moreover, (4.11) follows by (2.19).

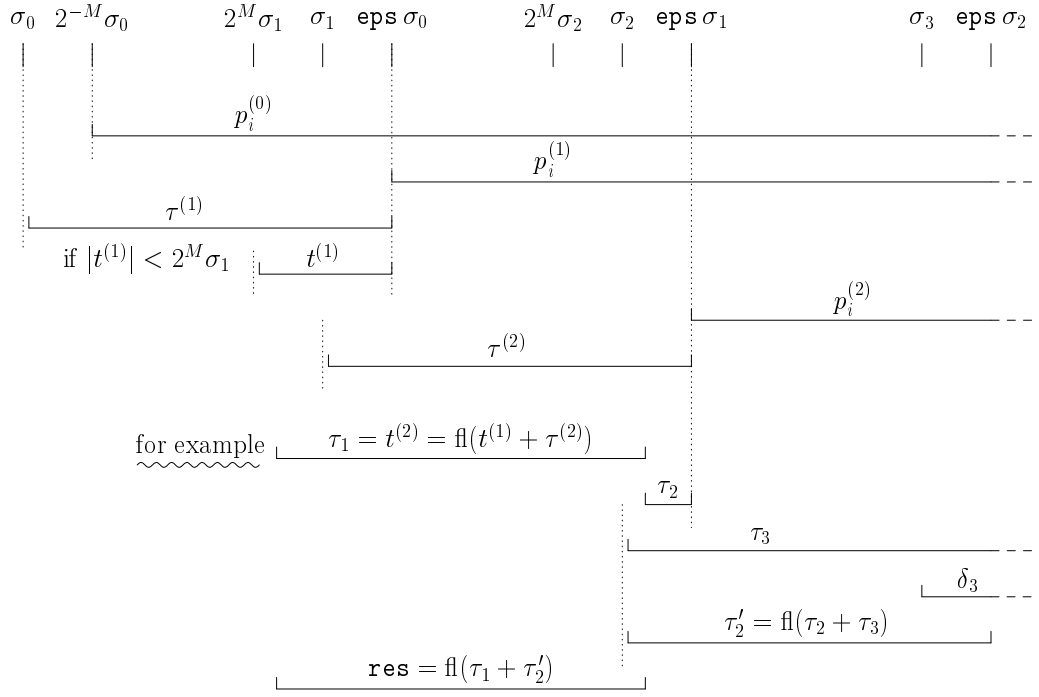

FIG. 4.1. *Outline of faithful rounding for $m = 2$.*

Henceforth assume $\sigma > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$. Then (4.5) implies (4.12), and we have to prove that res is a faithful rounding of the exact result. By assumption, $n + 2 \le 2^M$ and $2^{2M}\mathtt{eps} \le 1$. Next we abbreviate

$$\begin{array}{rcccl} \tau_3 & = & \mathrm{fl}(\sum_{i=1}^n p_i') & = & \sum_{i=1}^n p_i' - \delta_3 \ , \\ \tau_2' & = & \mathrm{fl}(\tau_2 + \tau_3) & = & \tau_2 + \tau_3 - \delta_2 \ , \\ \mathtt{res} & = & \mathrm{fl}(\tau_1 + \tau_2') & = & \tau_1 + \tau_2' - \delta_1 \ . \end{array} \tag{4.13}$$

We will use Lemma 2.4 to prove that res is a faithful rounding of s. By (4.9) and (4.13), $s = \tau_1+\tau_2+\tau_3+\delta_3 = \tau_1 + \tau_2' + \delta_2 + \delta_3$, so

$$s = r + \delta \ \text{ and } \ \mathtt{res} = \mathrm{fl}(r) \quad \text{for} \quad r := \tau_1 + \tau_2' \ \text{ and } \ \delta := \delta_2 + \delta_3 \ . \tag{4.14}$$

In Figure 4.1 we sketch the possible spread of bits of the individual variables in Algorithm 4.1 (Transform) for a final value $m = 2$. In the middle of the figure we define a possible $\tau_1 = t^{(m)}$, since the following quantities τ_2 etc. depend on that. Note this is a picture for small M, i.e. small dimension n. For larger n, up to almost $\sqrt{\mathtt{eps}^{-1}}$, the error δ_3 in the computation of $\tau_3 = \mathrm{fl}(\sum p_i^{(m)})$ can be quite significant (see Remark 5 above), though still just not too large to jeopardize faithful rounding.

By (4.9) and (2.20),

$$|\tau_3| = |\mathrm{fl}(\sum_{i=1}^n p_i')| \le n\mathtt{eps}\sigma \tag{4.15}$$

and

$$|\delta_3| \le \frac{1}{2}n(n-1)\mathtt{eps}^2\sigma \ . \tag{4.16}$$

For later use we note that (4.10) and (4.12) give

$$|\tau_2'| \le (1+\mathtt{eps})|\tau_2 + \tau_3| \le (1+\mathtt{eps})(\mathtt{eps}\cdot\mathrm{ufp}(\tau_1) + n\mathtt{eps}\sigma) < |\tau_1| \ . \tag{4.17}$$

Now (2.19) and (4.13) yield

$$|\delta_2| \le \mathtt{eps}|\tau_2+\tau_3| \le \mathtt{eps}^2(\mathrm{ufp}(\tau_1)+n\sigma)\ . \tag{4.18}$$

Furthermore, (4.13), (4.10), (4.15) and (4.18) give

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

so (2.19) and (4.13) imply

$$\begin{array}{rcl} |\mathtt{res}| & \ge & (1-\mathtt{eps})|\tau_1+\tau_2'| > (1-2\mathtt{eps})|\tau_1|-n\mathtt{eps}\sigma \\ & \ge & (1-2\mathtt{eps}-2^{-M})|\tau_1| \ge \frac{5}{8}|\tau_1|\ , \end{array} \tag{4.19}$$

a lower bound for $|\mathtt{res}|$, also in the presence of underflow. Now the definition of δ, (4.18) and (4.16) yield

$$|\delta| = |\delta_2+\delta_3| < \mathtt{eps}^2(\mathrm{ufp}(\tau_1)+n\sigma+\frac{1}{2}n(n-1)\sigma)\ . \tag{4.20}$$

By $n+2 \le 2^M$ we have $M \ge 2$, so that $2^{2M}\mathtt{eps} \le 1$ gives $2^{-M} \ge 2^M\mathtt{eps} \ge 4\mathtt{eps}$. Furthermore (4.20), (4.12) and (4.19) imply

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

Using (4.14) and (2.19) we conclude

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

and $|\mathtt{res}| < 2\mathrm{ufp}(\mathtt{res})$ proves (4.11).

If $|\tau_1| \ge \mathtt{eps}^{-1}\mathtt{eta}$, then $|\mathtt{res}| > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ by (4.19). That means $\mathtt{res} \notin \mathbb{U}$, and (4.14), (4.21) and Lemma 2.4 show that res is a faithful rounding of the sum s. This leaves us with the case $|\tau_1| < \mathtt{eps}^{-1}\mathtt{eta}$. In that case (4.18) and (4.12) yield

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

so (4.16) and (4.12) imply

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

But δ_2 and δ_3 are the errors of the sum of floating-point numbers and must be integer multiples of eta, so (4.22) and (4.23) show that both must be zero. Hence (4.14) implies that $\mathtt{res} = \mathrm{fl}(r) = \mathrm{fl}(s)$ is equal to the rounded-to-nearest exact result s.

Finally, assume $\mathtt{res} = 0$. To establish a contradiction assume $\sigma > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$. Then the "until"-condition yields $|\tau_1| = |t^{(m)}| \ge 2^{2M}\mathtt{eps}\sigma > 0$, and (4.19) implies $|\mathtt{res}| > 0$, a contradiction. Therefore $\sigma \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ and we already proved $p_i' = 0$ for all i. Hence $\tau_1 = \mathrm{fl}(\tau_1+\tau_2) = \mathtt{res} = 0$ and $\tau_2 = \mathrm{fl}(\tau_2) = \tau_1 = 0$. The lemma is proved. □

For the final version of Algorithm 4.1 (Transform) we have to take extra care about zero sums. If $s = 0$, then the extracted vector of lower order parts $p_i^{(m)}$ consists only of zeros at a certain stage and $t^{(m)}$ is zero.

Hence the preliminary version in Algorithm 4.1 of `Transform` has to continue the "repeat-until"-loop until σ_{m-1} is in the underflow range, each time processing the whole zero-vector $p_i^{(m)}$. To avoid this we may check whether the vector $p_i^{(m)}$ consists only of zeros as in the algorithm proposed by Zielke and Drygalla [50], but this is time consuming.

A simpler way is to test for $t^{(m)} = 0$. In that case $t^{(m)} = \mathrm{fl}(t^{(m-1)} + \tau^{(m)}) = 0 = t^{(m-1)} + \tau^{(m)}$ because of (2.3), so (4.2) implies $s = \sum_{i=1}^{n} p_i^{(m)}$. Hence, we can apply Algorithm `Transform` recursively to the extracted vector $p_i^{(m)}$ of lower order parts. In the recursive call, the calculation of μ needs one traversal through the vector. Mostly this just verifies $p_i^{(m)} \equiv 0$ and is necessary only once, namely if $t^{(m)} = 0$.

ALGORITHM 4.4. *Final version of Algorithm 4.1* (`Transform`) *with check for zero.*

function $[\tau_1, \tau_2, p, \sigma] =$ `Transform`(p)
  $\mu = \max(|p_i|)$
  if $\mu = 0$, $\tau_1 = \tau_2 = \sigma = 0$, return, end if
  $M =$ `NextPowerTwo`$(\mathrm{length}(p) + 2)$
  $\sigma' = 2^M$`NextPowerTwo`(μ)
  $t' = 0$
  repeat
    $t = t'$; $\sigma = \sigma'$
    $[\tau, p] =$ `ExtractVector`(σ, p)
    $t' = \mathrm{fl}(t + \tau)$
    if $t' = 0$, $[\tau_1, \tau_2, p, \sigma] =$ `Transform`(p); return; end if
    $\sigma' = \mathrm{fl}(2^M \mathtt{eps}\sigma)$
  until $|t'| \geq \mathrm{fl}(2^{2M}\mathtt{eps}\sigma)$ or $\sigma \leq \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$
  $[\tau_1, \tau_2] =$ `FastTwoSum`(t, τ)

The final version of Algorithm 4.1 (`Transform`) including this check for zero is given in Algorithm 4.4. If $t' = 0$ it might be advantageous in to eliminate zero components in the vector p. The final version of `Transform` in Algorithm 4.4 omits indices and uses Algorithm 3.6 (`NextPowerTwo`), so that only basic floating-point operations are necessary.

Otherwise, except the check for $t' = 0$, both Algorithms 4.1 and 4.4 are identical. Note, however, that the results of Algorithms 4.1 and 4.4 might be different (although always faithful) since in the recursive call of `Transform`, μ and σ are computed afresh. Nevertheless, the *assertions* of Lemmas 4.2 and 4.3 remain true for both Algorithms 4.1 and 4.4.

Hence we may safely use the same name `Transform` for both algorithms. The algorithm can still be slightly improved. For σ just before entering the underflow range one call to `ExtractVector` may be saved depending on t' (see the Matlab code given in the Appendix).

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

ALGORITHM 4.5. *Accurate summation with faithful rounding.*

function `res` $=$ `AccSum`(p)
  $[\tau_1, \tau_2, p'] =$ `Transform`(p)
  `res` $= \mathrm{fl}(\tau_1 + (\tau_2 + (\sum_{i=1}^{n} p_i')))$

PROPOSITION 4.6. *Let p be a vector of n floating-point numbers, define $M := \lceil \log_2(n+2) \rceil$ and assume $2^{2M}\mathtt{eps} \leq 1$. Let* `res` *be the result of Algorithm 4.5* (`AccSum`) *applied to p.*

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

REMARK. Algorithm 4.5 (AccSum) is identical to the piece of code we analyzed in Lemma 4.3, only the output parameter σ in Transform, which is unnecessary here, is omitted.

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

COROLLARY 4.7. *Under the assumption of Proposition 4.6 the computed result* res *of Algorithm 4.5* (AccSum) *is equal to the exact result* $s = \sum p_i$ *if* s *is a floating-point number, or if* $\mathtt{res} \in \mathbb{U}$, *i.e.*

$$s \in \mathbb{F} \quad or \quad \mathtt{res} \in \mathbb{U} \quad \Rightarrow \quad \mathtt{res} = s \ . \tag{4.24}$$

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

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

PROOF. The sum s of floating-point numbers satisfies always $s \in \mathtt{eta}\mathbb{Z}$, so the definition (2.25) of faithful rounding proves (4.24) and the corollary. □

REMARK 1. The exact determination of the sign of a sum by Algorithm 4.5 is critical in the evaluation of geometrical predicates [20, 4, 45, 9, 27, 8, 14]. Rewriting a dot product as a sum by splitting products in two parts (Dekker's and Veltkamp's [12] algorithms Split and TwoProduct, see also [37]), we can determine the exact sign of a dot product as well, which in turn decides whether a point is exactly on some plane, or on which side it is. An improved version of AccSum for sign determination also for huge vector lengths is given in Part II of the paper.

REMARK 2. We showed that the result of Algorithm 4.5 is *always* a faithful rounding of the exact sum. Computational evidence suggests that the cases, where the result of AccSum is *not* rounded to nearest, are very rare. In several billion tests we never encountered such a case.
However, we can construct examples with faithful but not rounding to nearest. Consider $p = [1 \ \mathtt{eps} \ \mathtt{eps}^2]$. Then $\mathtt{AccSum}(p)$ produces $\tau_1 = 1$, $\tau_2 = 0$ and $p' = [0 \ \mathtt{eps} \ \mathtt{eps}^2]$, and $\mathtt{res} = 1$. This is because IEEE 754 rounds tie to even, so $\mathrm{fl}(1 + \mathtt{eps}) = 1$.

Changing the strict into an "almost always" rounded to nearest offers quite a reward, namely the computational effort of Algorithm 4.5 (AccSum) depends solely on the logarithm of the condition number of the sum: only the more difficult the problem, the more computing time must be spent. The maximum number of iterations m can be estimated as follows. The condition number of summation for $\sum p_i \neq 0$ is defined [19] by

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

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

$$\mathrm{cond}\left(\sum p_i\right) = \frac{\sum |p_i|}{|\sum p_i|} \ . \tag{4.25}$$

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

THEOREM 4.8. *Assume Algorithm 4.5* (AccSum) *is applied to a vector of floating-point numbers* $p_i, 1 \leq i \leq n$, *with nonzero sum* s. *Define* $M := \lceil \log_2(n+2) \rceil$ *and assume* $2^{2M}\mathtt{eps} \leq 1$. *Then the following is true. If*

$$\mathrm{cond}\left(\sum p_i\right) \leq (1 - 2^{-M}) 2^{-2M-1} \left[2^{-M} \mathtt{eps}^{-1}\right]^m \ , \tag{4.26}$$

TABLE 4.2
Minimum treatable condition numbers by Algorithm 4.5 in IEEE 754 double precision

n	$m=1$	$m=2$	$m=3$	$m=4$	$m=5$
100	$2.1 \cdot 10^{9}$	$1.5 \cdot 10^{23}$	$1.1 \cdot 10^{37}$	$7.4 \cdot 10^{50}$	$5.2 \cdot 10^{64}$
1000	$4.2 \cdot 10^{6}$	$3.7 \cdot 10^{19}$	$3.2 \cdot 10^{32}$	$2.9 \cdot 10^{45}$	$2.5 \cdot 10^{58}$
10000	$1.0 \cdot 10^{3}$	$5.6 \cdot 10^{14}$	$3.1 \cdot 10^{26}$	$1.7 \cdot 10^{38}$	$9.4 \cdot 10^{49}$
10^5		$1.4 \cdot 10^{11}$	$9.4 \cdot 10^{21}$	$6.5 \cdot 10^{32}$	$4.5 \cdot 10^{43}$
10^6		$3.4 \cdot 10^{7}$	$2.9 \cdot 10^{17}$	$2.5 \cdot 10^{27}$	$2.1 \cdot 10^{37}$
10^7		$5.1 \cdot 10^{2}$	$2.7 \cdot 10^{11}$	$1.5 \cdot 10^{20}$	$7.9 \cdot 10^{28}$

then Algorithm 4.4 (`Transform`) *called by Algorithm 4.5* (`AccSum`) *stops after at most m executions of the "repeat-until"-loop. If the "repeat-until"-loop is executed m times and absolute value and comparison is counted as one flop, then Algorithm 4.5* (`AccSum`) *needs $(4m+3)n + \mathcal{O}(m)$ flops.*

PROOF. For the analysis we use the (except the check for zero) identical Algorithm 4.1 for `Transform`. Then $p_i^{(0)} := p_i$, and its initialization implies

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

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

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

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

Since the "repeat-until"-loop is to be executed again, we use (4.2) to conclude

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

as in (4.7) in the proof of Lemma 4.2. Denote the condition number by C. Then (4.25) and (4.27) yield $|s| = C^{-1} \sum |p_i| \ge \mu C^{-1}$. Combining this with (4.29), (4.3), (4.27), (4.26) and using $(1-2^{-M})^{-1} > 1+2^{-M}$ implies

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

a contradiction to (4.28). Up to order 1, the calculation of μ requires $2n$ flops, `ExtractVector` requires $4n$ and the computation of `res` requires n flops. The theorem is proved. □

For IEEE 754 double precision, Theorem 4.8 basically means that *at least* for condition numbers up to

$$\mathrm{cond}\left(\sum p_i\right) \lesssim 2^{m(53-M)-2M-1}$$

Algorithm 4.5 (`AccSum`) computes a faithfully rounded result in at most m executions of the "repeat-until"-loop. In Table 4.2 we show the lower bound by Theorem 4.8 for the condition number which can be treated for different values of n and m, where treatable means to produce a faithfully rounded result.

Conversely, we can use Theorem 4.8 to compute for given m and condition number $\texttt{eps}^{-1}$ the minimum length n of a vector for which a faithfully rounded result of its sum is computed. The value $\texttt{eps}^{-1}$ is the

TABLE 4.3
Minimum treatable length n for condition number $\texttt{eps}^{-1}$ *in IEEE 754 double precision*

$m = 2$	$m = 3$	$m = 4$
4094	$1.0 \cdot 10^6$	$6.7 \cdot 10^7$

TABLE 4.4
Floating-point operations needed for different dimension and condition number

n	cond	AccSum	Sum2	XBLAS
1000	10^6	$7n$	$7n$	$10n$
1000	10^{16}	$11n$	$7n$	$10n$
10^6	10^{16}	$15n$	$7n$	$10n$

condition number for which we cannot expect a single correct digit by traditional recursive summation. The value $6.7 \cdot 10^7$ in Table 4.3 corresponds to the maximum value of n satisfying $n + 2 \leq 2^M$ and $2^{2M}\texttt{eps} \leq 1$. The table shows that for condition number up to $\texttt{eps}^{-1}$ and vectors with up to a million elements never more than 3 iterations are needed. Algorithms for even larger values $n > 6.7 \cdot 10^7$ will be presented in Section 8 in Part II of this paper.

Also AccSum compares favorably to other algorithms. Consider the XBLAS summation algorithm BLAS_dsum_x [2]. Note that there are at least three implementations, the reference implementation published in [32] requiring $20n$ flops, the function ddadd in the ddfun90 package [5] by David Bailey requiring $11n$ flops, and BLAS_dsum_x taken from [2] requiring $10n$ flops. We compare against the fastest version BLAS_dsum_x taken from XBLAS requiring $10n$ flops.

An alternative to XBLAS is Algorithm 4.4 (Sum2) in [37]. The results of both algorithms are of the same quality, namely *as if* computed in quadruple precision. That means, for condition numbers up to $\texttt{eps}^{-1}$ we can expect a result accurate to the last bit. In Table 4.4 the required floating-point operations are displayed for different vector lengths and condition numbers. Note that Algorithm 4.5 (AccSum) always computes a faithfully rounded result, independent of the condition number.

In practice the computing times compare even more favorable than anticipated by Table 4.4 for our Algorithm 4.5 (AccSum) as shown in Section 5. This is because AccSum allows a better instruction-level parallelism as analyzed by Langlois [29]. The measured computing times displayed in the next section suggest that AccSum seems to be faster than the XBLAS summation algorithm by a factor 2 to 5, although being of much better quality.

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

All algorithms are tested in three different environments, namely Pentium 4, Itanium 2 and Athlon 64, see Table 5.1. We carefully choose compiler options to achieve best possible results, see Table 5.1.

We faced no problems except for Pentium 4 and the Intel Visual Fortran 9.1 compiler, where the code optimization/simplification is overdone by the compiler. A typical example is the first line $q = \mathrm{fl}\left((\sigma + p) - \sigma\right)$ in Algorithm 3.2 (ExtractScalar), which is optimized into $q = p$. This can, of course, be avoided by setting appropriate compiler options; however, this may slow down the whole computation. In this specific case the second author suggested a simple trick to overcome this by using $q = \mathrm{fl}\left(|\sigma + p| - \sigma\right)$ instead. This does not change the intended result since $|p| \leq \sigma$ is assumed in the analysis (Lemma 3.3), it avoids unintended compiler optimization, and it does not slow down the computation. For the other algorithms to be tested we had, however, to use the compile option /Op for Pentium 4. This ensures the consistency of IEEE standard 754 floating-point arithmetic. The compile options for the different algorithms are summarized in Table 5.2.

TABLE 5.1
Testing environments

	CPU, Cache sizes	Compiler, Compile options
I)	Intel Pentium 4 (2.53GHz)	Intel Visual Fortran 9.1
	L2: 512KB	`/O3 /QaxN /QxN [/Op, see Table 5.2]`
II)	Intel Itanium 2 (1.4GHz)	Intel Fortran 9.0
	L2: 256KB, L3: 3MB	`-O3`
III)	AMD Athlon 64 (2.2GHz)	GNU gfortran 4.1.1
	L2: 512KB	`-O3 -fomit-frame-pointer -march=athlon64 -funroll-loops`

TABLE 5.2
Compile options for Pentium 4, Intel Visual Fortran 9.1

Algorithm	Necessity of compile option `/Op`
`DSum`	No
`Sum2`	Yes, for `TwoSum`
`XBLAS`	Yes, for `TwoSum` and `FastTwoSum`
`Priest`	Yes, for `FastTwoSum`
`Malcolm`, `LongAccu`	Yes, for `Split`
`AccSum`	Basically, no

Our algorithms are based on extractions, the split of a floating-point number with respect to σ, a power of 2 corresponding to a certain exponent. Since this operation is in the inner loop of all our algorithms, we payed special attention to this and designed Algorithm 3.4 (`ExtractVector`) to be as fast as possible. This algorithm requires 3 floating-point operations and has no branch.

Another possibility to extract bits of a floating-point number p is proper scaling and rounding to integer as used by Zielke and Drygalla [50] (line 13 in Algorithm 3.1). Some compilers offer two possibilities of such rounding, namely chopping and rounding to the nearest integer. In the Table 5.3 the columns "`Dint`" and "`Dnint`" refer to those roundings, respectively. Another possibility of rounding a floating-point number is the assignment to a variable of type integer. One obstacle might be that an integer format with sufficient precision is not available. This approach is referred to by "`Int=Dble`".

As can be seen from Table 5.3, our Algorithm 3.2 (`ExtractScalar`), the computing time of which is normed to 1, is usually faster than the other possibilities. Our summation algorithm directly benefits from this. There is a certain drop in the ratio for large dimension which is related to the cache size, so not too much attention must be paid to the last lines of the Table 5.3 for huge dimension.

Next we tested our summation algorithm. Test examples for huge condition numbers larger than $\texttt{eps}^{-1}$ were generated by Algorithm 6.1 in [37], where a method to generate a vector whose summation is arbitrarily ill-conditioned is described. Dot products are transformed into sums by Dekker's and Veltkamp's Algorithms `Split` and `TwoProduct`, see [37].

First we compare `AccSum` with the ordinary, recursive summation `DSum`, with `Sum2` taken from [37] and the XBLAS summation algorithm `BLAS_dsum_x` from [2] (called `XBLAS` in the following tables). The latter two deliver a result *as if* calculated in approximately twice the working precision. As has been mentioned at the end of the previous section, `BLAS_dsum_x` requires $10n$ flops and is the fastest version of XBLAS summation. In the first set of examples we test sums with condition number 10^{16} for various vector lengths. This is the largest condition number for which `Sum2` and `XBLAS` produce an accurate result. Note that the comparison is not really fair since `AccSum` produces a faithfully rounded result for any condition number. We compare to recursive summation `DSum`, the time of which is normed to 1. This is only for reference; for condition

TABLE 5.3
Measured computing times for extraction, for all environments time of `ExtractScalar` *normed to 1*

CPU	Intel Pentium 4 (2.53GHz)			Intel Itanium 2 (1.4GHz)			AMD Athlon 64 (2.2GHz)		
Compiler	Intel Visual Fortran 9.1			Intel Fortran 9.0			GNU gfortran 4.1.1		
n	`Dint`	`Dnint`	`Int=Dble`	`Dint`	`Dnint`	`Int=Dble`	`Dint`	`Dnint`	`Int=Dble`
100	5.6	6.8	9.8	1.4	2.1	1.2	6.8	9.0	1.9
400	5.9	6.8	10.4	2.1	3.4	1.6	6.3	9.8	1.9
1,600	5.7	7.0	10.2	2.1	3.4	1.7	7.3	9.9	1.9
6,400	5.5	6.8	9.8	2.1	3.4	1.6	7.3	10.0	1.9
25,600	5.5	6.8	9.7	2.2	3.5	1.7	6.5	9.1	1.9
102,400	1.1	1.2	1.5	2.1	2.8	1.4	3.3	5.0	1.3
409,600	1.0	1.1	1.4	1.9	2.0	1.0	2.8	4.2	1.2
1,638,400	1.0	1.1	1.3	1.9	2.0	1.0	3.2	4.6	1.2

TABLE 5.4
Measured computing times for cond $= 10^{16}$*, for all environments time of* `DSum` *normed to 1*

CPU	Intel Pentium 4 (2.53GHz)			Intel Itanium 2 (1.4GHz)			AMD Athlon 64 (2.2GHz)		
Compiler	Intel Visual Fortran 9.1			Intel Fortran 9.0			GNU gfortran 4.1.1		
n	`Sum2`	`XBLAS`	`AccSum`	`Sum2`	`XBLAS`	`AccSum`	`Sum2`	`XBLAS`	`AccSum`
100	24.1	75.9	14.1	2.9	17.3	7.8	2.0	4.8	2.8
400	26.9	81.9	13.1	4.7	31.6	10.7	3.1	7.6	4.1
1,600	20.0	57.8	9.6	7.3	50.5	15.8	3.1	7.7	4.1
6,400	20.0	57.8	9.6	7.8	54.5	16.5	3.1	7.7	4.2
25,600	20.4	59.1	13.8	7.9	55.6	21.5	3.1	7.7	5.7
102,400	2.3	7.5	7.9	7.3	50.0	25.6	2.3	5.7	8.0
409,600	2.4	7.3	7.8	2.1	13.0	12.9	2.3	5.5	8.0
1,638,400	2.4	7.6	8.0	2.1	13.1	12.8	2.3	5.5	8.1

number 10^{16} we cannot expect `DSum` to produce a single correct digit.

The results are displayed in Table 5.4. `AccSum` achieves on the different architectures a remarkable factor of about 10, 17 or 5 compared to recursive summation. We also see that `AccSum` is significantly faster than `XBLAS`, on Pentium 4 even faster than `Sum2`. As has been mentioned earlier, this is due to a better instruction-level parallelism of `AccSum` and `Sum2` as analyzed by Langlois [29]. We also observe a certain drop in the ratio for larger dimensions, at least for Pentium 4 and Itanium 2 due to cache misses. However, this is hardly visible on the Athlon architecture.

A closer look reveals that the code produced by the GNU gfortran 4.1.1 compiler on Athlon can be significantly improved by unrolling loops. On Pentium 4 and Itanium 2 we did not observe a difference when unrolling, the compilers seem to be smart enough to take care of that. The computational results for Athlon 64 are displayed in Table 5.5, where `DSumU`, `Sum2U`, `XBLASU` refer to the unrolled versions, respectively. Note that the time for `DSumU` is normed to 1. Collecting 4 terms at a time proved to be a good choice. We observe not much difference for `Sum2`, `XBLAS` and `AccSum` when unrolling, but a significant difference for recursive summation `DSum`. Now the drop in the time-ratio due to cache misses is visible as before.

A typical application of accurate summation algorithms is the computation of the residual $A\tilde{x} - b$ for an approximate solution $\tilde{x}$ of a linear system $Ax = b$ by transforming the dot products error-free into sums. If $\tilde{x}$ is computed by some standard algorithm like Gaussian elimination, the condition number of the residual is always 10^{16}. This is true independent of the condition number of the linear system. This is the reason

TABLE 5.5
Measured computing times for cond $= 10^{16}$*, time of* DSumU *normed to 1, on AMD Athlon 64, GNU gfortran 4.1.1*

n	DSum	Sum2	Sum2U	XBLAS	XBLASU	AccSum	AccSumU
100	4.8	9.3	8.2	22.7	22.6	13.4	13.4
400	3.6	11.2	9.9	27.7	27.7	15.0	15.1
1,600	3.9	12.1	10.6	30.0	30.0	16.0	16.0
6,400	4.0	12.4	10.9	30.8	30.8	16.8	16.5
25,600	2.0	6.2	5.4	15.4	15.4	11.4	11.3
102,400	1.0	2.3	2.0	5.7	5.6	7.9	7.9
409,600	1.0	2.3	2.0	5.5	5.5	8.0	8.0
1,638,400	1.0	2.3	2.0	5.5	5.5	8.1	8.1

TABLE 5.6
Measured computing times for cond $= 10^{32}$*, for all environments time of* DSum *normed to 1*

CPU	Intel Pentium 4 (2.53GHz)			Intel Itanium 2 (1.4GHz)		
Compiler	Intel Visual Fortran 9.1			Intel Fortran 9.0		
n	Sum2	XBLAS	AccSum	Sum2	XBLAS	AccSum
100	24.1	75.9	15.8	3.0	17.5	9.6
400	26.9	81.9	15.9	5.9	39.4	16.3
1,600	20.0	57.8	14.0	7.3	50.6	20.0
6,400	20.0	57.8	14.0	7.9	54.6	24.4
25,600	20.4	59.1	15.1	7.9	63.0	25.6
102,400	2.3	7.5	13.1	5.4	36.3	33.2
409,600	2.4	7.3	11.2	2.1	13.3	19.0
1,638,400	2.4	7.6	11.1	3.9	13.1	18.7

why the previous examples are chosen with condition number 10^{16}. The performance for condition number 10^{32} is displayed in Table 5.6 for Intel Pentium 4 and Itanium 2 architectures. Again the computing time of DSum is normed to 1. The results for the AMD Athlon 64 processor for condition number 10^{32} behave, compared to the other architectures, similar to those for condition number 10^{16}.

The computing times of DSum, Sum2 and XBLAS do not depend on the condition number, so the results for these algorithms coincide with those of Table 5.4. The computing time of AccSum adapts to the condition number. For condition number 10^{32} we observe an increase in computing time for AccSum of up to 50% compared to condition number 10^{16}. Note, however, that all results by AccSum are accurate to the last bit whereas all results by Sum2 and XBLAS are completely incorrect for condition number 10^{32}.

The good performance of AccSum becomes transparent when looking at the MFlops-rate. In Figure 5.1 the MFlops are displayed for the different algorithms on Itanium 2, the left figure corresponding to the previously displayed results. For the other architectures the picture looks even more favorably for AccSum. Note that the Itanium 2 can perform 2 additions per cycle so that the peak performance is 2.8GFlops.

It is interesting as well to look at the percentage of peak performance achieved by the different algorithms. These are displayed in Tables 5.7 and 5.8 for condition numbers 10^{16} and 10^{32} for Pentium 4 and Itanium 2, respectively. The result for the Athlon 64 processor is similar to that for Itanium 2. Note that the memory bandwidth[3] of the Itanium 2 and Athlon 64 architectures is approximately 3GByte/sec, while that of Pentium 4 is around 1GByte/sec. In Table 5.7 we observe for DSum gradually decreasing performance down to around half of peak with a sharp drop for a dimension above 10^5, which is out-of-cache data. The performance of

[3]It influences the performance of the architectures for out-of-cache data.

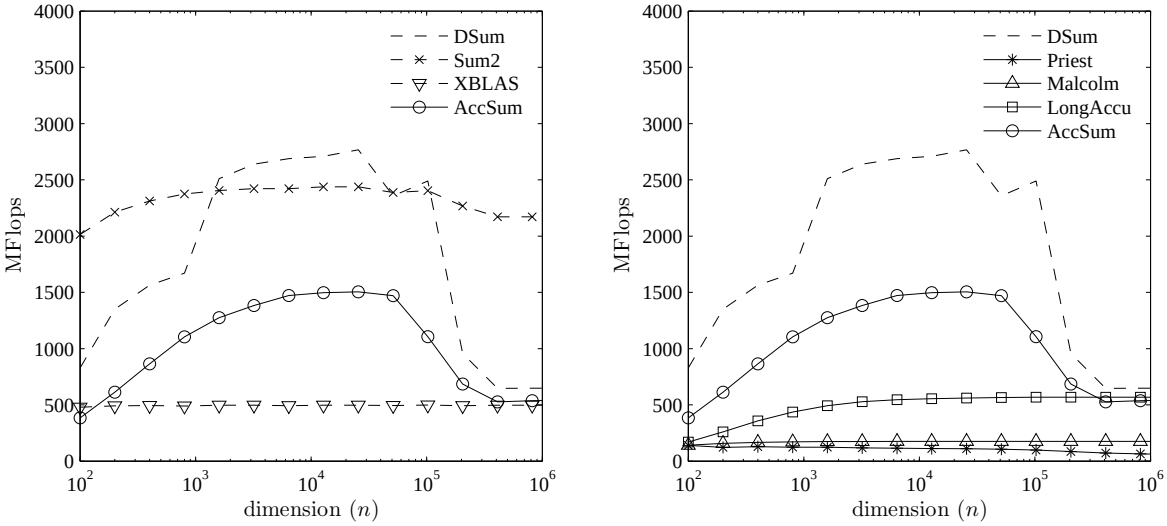

FIG. 5.1. *Measured MFlops on Itanium 2 (1.4GHz), Intel Fortran 9.0,* cond $= 10^{16}$

TABLE 5.7
Percentage of peak performance Pentium 4 (2.53 GHz) for condition numbers 10^{16} *and* 10^{32}

	cond $= 10^{16}$				cond $= 10^{32}$			
n	DSum	Sum2	XBLAS	AccSum	DSum	Sum2	XBLAS	AccSum
100	74.4 %	21.6 %	9.8 %	58.0 %	74.4 %	21.6 %	9.8 %	66.6 %
400	79.1 %	20.6 %	9.7 %	66.3 %	79.1 %	20.6 %	9.7 %	82.5 %
1,600	55.0 %	19.2 %	9.5 %	63.2 %	55.0 %	19.2 %	9.5 %	89.0 %
6,400	55.0 %	19.2 %	9.5 %	63.2 %	55.0 %	19.2 %	9.5 %	89.0 %
25,600	56.2 %	19.2 %	9.5 %	61.2 %	56.2 %	19.2 %	9.5 %	81.5 %
102,400	7.0 %	21.1 %	9.3 %	13.3 %	7.0 %	21.1 %	9.3 %	13.7 %
409,600	6.9 %	20.6 %	9.4 %	13.3 %	6.9 %	20.6 %	9.4 %	13.7 %
1,638,400	7.0 %	20.7 %	9.3 %	13.1 %	7.0 %	20.7 %	9.3 %	13.7 %

`Sum2` and `XBLAS` is constantly around 20% and below 10%, respectively, due to data dependencies. `AccSum` is much more efficient for in-cache data. For Itanium 2 we observe in Table 5.8 for `DSum` increasing performance up to peak, again with a sharp drop for out-of-cache data. For `Sum2` performance increases starting at a higher level but not reaching peak performance and with a not so sharp drop. The performance of `XBLAS` is constantly below 20% due to data dependencies, whereas `AccSum` is less efficient than `Sum2`. Remember that for condition number 10^{32} results of `Sum2` and `XBLAS` are completely incorrect whereas `AccSum` computes results accurate to the last bit.

Next we compare to competing algorithms, namely Priest's doubly compensated summation [40, 41], Malcolm's [34] and the long accumulator [28]. The comparison is also not exactly fair because Priest's algorithm produces a result accurate to 2 units in the last place, so almost faithful rounding, whereas Malcolm's and the long accumulator can easily be used to compute a rounded-to-nearest result. Note that the needed intermediate memory for the latter two approaches depend on the exponent range (in fact, is proportional to), whereas `AccSum` does not.

We first display the results for Pentium 4 and Itanium 2 in Table 5.9. Obviously `AccSum` compares favorably to its competitors. The ratio in computing time compared to `DSum` is stable and around 10 to 20 for all vector lengths. Note, however, the tremendous gain for Malcolm's algorithm with increasing dimension, a factor 17 or almost 5 from vector length 100 to 1.6 million. It seems that for huge vector lengths we basically

TABLE 5.8
Percentage of peak performance Itanium 2 (1.4 GHz) for condition numbers 10^{16} *and* 10^{32}

	cond $= 10^{16}$				cond $= 10^{32}$			
n	DSum	Sum2	XBLAS	AccSum	DSum	Sum2	XBLAS	AccSum
100	30.4 %	71.9 %	17.3 %	38.4 %	30.4 %	71.9 %	17.3 %	47.5 %
400	69.7 %	82.6 %	17.7 %	50.5 %	69.7 %	82.6 %	17.7 %	63.9 %
1,600	89.9 %	85.9 %	17.8 %	56.0 %	89.9 %	85.9 %	17.8 %	72.7 %
6,400	97.0 %	86.5 %	17.8 %	57.1 %	97.0 %	86.5 %	17.8 %	75.2 %
25,600	97.5 %	86.5 %	15.5 %	61.4 %	97.5 %	86.5 %	15.5 %	73.3 %
102,400	64.5 %	84.2 %	17.8 %	48.3 %	64.5 %	84.2 %	17.8 %	52.6 %
409,600	23.6 %	77.5 %	17.7 %	27.0 %	23.6 %	77.5 %	17.7 %	28.6 %
1,638,400	23.2 %	77.7 %	17.7 %	27.2 %	23.2 %	77.7 %	17.7 %	28.6 %

TABLE 5.9
Measured computing times for cond $= 10^{16}$*, for both environments time of* DSum *normed to 1*

CPU	Intel Pentium 4 (2.53GHz)				Intel Itanium 2 (1.4GHz)			
Compiler	Intel Visual Fortran 9.1				Intel Fortran Compiler 9.0			
n	Priest	Malcolm	LongAccu	AccSum	Priest	Malcolm	LongAccu	AccSum
100	187.1	175.9	711.2	14.1	129.7	49.0	241.5	7.8
400	311.9	148.1	761.9	13.1	317.4	43.6	448.1	10.7
1,600	305.7	97.8	540.9	9.6	609.5	50.9	717.6	15.8
6,400	345.7	96.5	540.9	9.6	797.8	49.3	767.5	16.5
25,600	407.1	98.7	554.2	13.8	957.0	49.4	790.2	21.5
102,400	93.5	11.2	67.2	7.9	1056.2	43.8	711.3	25.6
409,600	159.9	10.4	66.3	7.8	411.2	11.4	184.8	12.9
1,638,400	216.5	10.6	67.4	8.0	547.9	11.4	185.4	12.8

measure cache misses rather than performance of the algorithms. Huge vector lengths, however, may become important for matrix-vector multiplication.

Again we observe a certain drop for huge vector lengths due to cache misses. As before comparison on Athlon should be made to the unrolled version DSumU of recursive summation. The results are summarized in Table 5.10. Again observe the tremendous speedup of Malcolm's algorithm with increasing vector length. The corresponding MFlops are displayed in the right graph of Figure 5.1.

The "small" condition number 10^{16} is favorite for our Algorithm AccSum because few extractions are necessary, whereas the computing times for Priest's, Malcolm's and the long accumulator are almost independent of the condition number. We compare the algorithms for fixed vector length 1000 and huge condition numbers, where the computing time for AccSum is normed to 1. The relative computing times on the three architectures are displayed in Figure 5.2 and the left of Figure 5.3. Obviously AccSum shows a good performance on all platforms. In the right of Figure 5.3 the MFlop rate of the algorithms is displayed for Itanium 2. For reference, the MFlop rate for recursive summation DSum is shown as well. On the Itanium 2 both DSum and AccSum range not far from peak performance. For larger condition number performance increases because more operations are performed on the data. Otherwise only Malcolm's summation can reach a reasonable MFlop rate. For the other architectures the picture looks similar.

Finally that we tried to find examples where the result of Algorithm 4.5 (AccSum) is not rounded to nearest. We treated dimensions from 10 to 10^5 and condition numbers as above. In particular we used dimensions $n = 2^M - 2$. Fortunately, we have Algorithm NearSum to be presented in Part II of this paper for reference.

TABLE 5.10
Measured computing times for cond $= 10^{16}$*, time of* `DSumU` *normed to 1, AMD Athlon 64, GNU gfortran 4.1.1*

n	`DSum`	`Priest`	`Malcolm`	`LongAccu`	`AccSum`
100	4.8	106.3	66.4	269.3	13.4
400	3.6	182.5	40.6	327.0	15.0
1,600	3.9	231.4	32.8	353.1	16.0
6,400	4.0	280.7	31.2	362.2	16.8
25,600	2.0	171.1	15.4	181.2	11.4
102,400	1.0	96.7	5.6	66.0	7.9
409,600	1.0	140.9	5.5	64.3	8.0
1,638,400	1.0	190.4	5.5	64.2	8.1

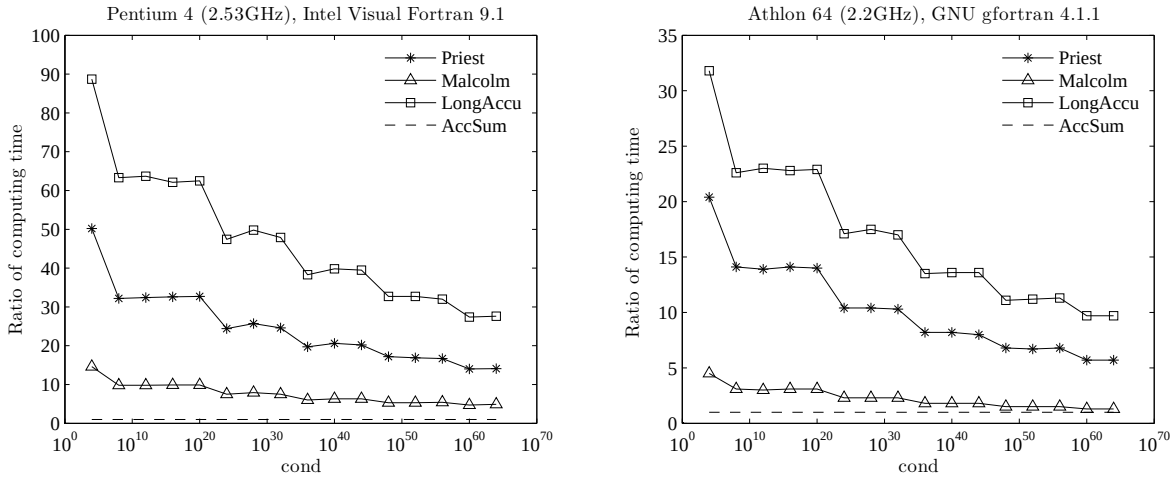

FIG. 5.2. *Measured computing times for huge condition numbers, for both environments time of* `AccSum` *normed to 1*

It is not so easy to find a long precision package delivering results always rounded to nearest; especially we observed problems with rounding tie to even. In several billion test cases we did not find one example with the result of `AccSum` not being the nearest floating-point number to the exact result. However, there are constructed examples, see Remark 5 following Lemma 4.3 and Remark 2 following Corollary 4.7.

6. Appendix. Following is executable Matlab code for Algorithm 4.5 including acceleration for zero sums and elimination of zero summands for that case (see also `http://www.ti3.tu-harburg.de/rump`). Moreover, the algorithms `ExtractVector` and `FastTwoSum` are expanded. Note that the Matlab function `nextpow2(f)` returns the smallest k such that $2^k \geq |f|$, while Algorithm 3.6 (`NextPowerTwo`) returns 2^k. Accordingly, the variable `Ms` refers to 2^M in Algorithm 4.5 (`AccSum`). Note that the check for overflow (which is easily done by scaling) and the check 2^{2M}`eps` ≤ 1 is omitted.

ALGORITHM 6.1. *Executable Matlab code for Algorithm 4.5* (`AccSum`) *including check for zero sum.*

```
function res = AccSum(p)
% For given vector p, result res is the exact sum of p_i faithfully rounded
% provided no overflow occurs. Acceleration for zero sums is included.
%
  n = length(p);                        % initialization
  mu = max(abs(p));                     % abs(p_i) <= mu
  if ( n==0 ) | ( mu==0 )               % no or only zero summands
    res = 0;
```

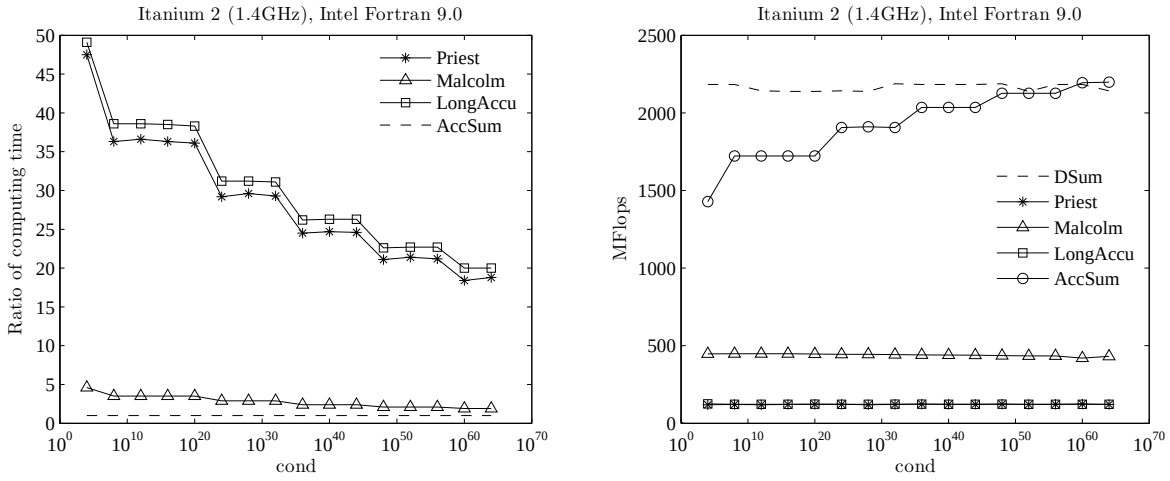

FIG. 5.3. *Measured computing times (left, time of* `AccSum` *normed to 1) and MFlops (right) on Itanium 2*

```
    return
  end
  Ms = NextPowerTwo(n+2);               % n+2 <= 2^M = Ms
  sigma = Ms*NextPowerTwo(mu);          % first extraction unit
  phi = 2^(-53)*Ms;                     % factor to decrease sigma
  factor = 2^(-53)*Ms*Ms;               % factor for sigma check
%
  t = 0;
  while 1
    q = ( sigma + p ) - sigma;          % [tau,p] = ExtractVector(sigma,p);
    tau = sum(q);                       % sum of leading terms
    p = p - q;                          % remaining terms
    tau1 = t + tau;                     % new approximation
    if ( abs(tau1)>=factor*sigma ) | ( sigma<=realmin )
      tau2 = tau - ( tau1 - t );        % [tau1,tau2] = FastTwoSum(t,tau)
      res = tau1 + ( tau2 + sum(p) );   % faithfully rounded final result
      return
    end
    t = tau1;                           % sum t+tau exact
    if t==0                             % accelerate case sum(p)=0
      res = AccSum(p);                  % sum of remainder part
      return
    end
    sigma = phi*sigma;                  % new extraction unit
  end
```

7. Summary. We presented a summation algorithm provably computing a faithfully rounded result. The algorithm uses only ordinary floating-point addition, subtraction and multiplication, no branches in the inner loops and no special operations. We showed that our summation Algorithm 4.5 (`AccSum`) is faster, sometimes much faster than other approaches. For all algorithms presented in Part I and II of this paper and in [37] we put a Matlab reference code on `http://www.ti3.tu-harburg.de/rump` .

The algorithms are based on so-called error-free transformations. We hope to see these computationally and

mathematically highly interesting operations in future computer architectures and floating-point standards.

Acknowledgement. The authors heartily wish to thank the two anonymous referees for their thorough reading and most valuable and inspiring comments. Our special thanks to Yozo Hida from Berkeley for his many thoughtful and valuable comments. Moreover we wish to thank many colleagues for their constructive comments, among them Stef Graillat, Philippe Langlois and Paul Zimmermann. The first author wishes to express his thankfulness that the paper could be written during stays at Waseda University supported by the Grant-in-Aid for Specially Promoted Research from the MEXT, Japan. The first author also wishes to thank his students of the winter term 2004/05, of the summer terms 2005 and 2006 for their patience and constructive comments. The second author would like to express his sincere thanks to Prof. Yasunori Ushiro for his stimulating discussions.

REFERENCES

[1] *ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic*, New York, 1985.
[2] *XBLAS: A Reference Implementation for Extended and Mixed Precision BLAS.* `http://crd.lbl.gov/~xiaoye/XBLAS/` .
[3] I. J. Anderson, *A distillation algorithm for floating-point summation*, SIAM J. Sci. Comput., 20 (1999), pp. 1797–1806.
[4] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. P. Preparata, and M. Yvinec, *Evaluating signs of determinants using single-precision arithmetic*, Algorithmica, 17 (1997), pp. 111–132.
[5] D. H. Bailey, *A Fortran-90 double-double precision library.* `http://crd.lbl.gov/~dhbailey/mpdist/` .
[6] D. H. Bailey, H. Yozo, X. S. Li, and B. Thompson, *ARPREC: An Arbitrary Precision Computation Package*, Tech. Report Paper LBNL-53651, Lawrence Berkeley National Laboratory, 2002.
[7] G. Bohlender, *Floating-point computation of functions with maximum accuracy*, IEEE Trans. Comput., C-26 (1977), pp. 621–632.
[8] H. Brönnimann, C. Burnikel, and S. Pion, *Interval arithmetic yields efficient dynamic filters for computational geometry*, Discrete Appl. Math., 109 (2001), pp. 25–47.
[9] H. Brönnimann and M. Yvinec, *Efficient exact evaluation of signs of determinants*, Algorithmica, 27 (2000), pp. 21–56.
[10] K. L. Clarkson, *Safe and effective determinant evaluation*, in Proceedings of the 33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, PA, IEEE Computer Society Press, 1992, pp. 387–395.
[11] M. Daumas and D. W. Matula, *Validated roundings of dot products by sticky accumulation*, IEEE Trans. Comput., 46 (1997), pp. 623–629.
[12] T. J. Dekker, *A floating-point technique for extending the available precision*, Numer. Math., 18 (1971), pp. 224–242.
[13] J. Demmel and Y. Hida, *Accurate and efficient floating point summation*, SIAM J. Sci. Comput., 25 (2003), pp. 1214–1248.
[14] J. Demmel and Y. Hida, *Fast and accurate floating point summation with application to computational geometry*, Numerical Algorithms, 37 (2004), pp. 101–112.
[15] S. Graillat, P. Langlois, and N. Louvet, *Compensated Horner Scheme*, Tech. Report RR2005-02, Laboratoire LP2A, University of Perpignan, 2005.
[16] J. R. Hauser, *Handling floating-point exceptions in numeric programs*, ACM Trans. Program. Lang. Syst., 18 (1996), pp. 139–174.
[17] C. Hecker, *Let's get to the (floating) point*, Game Developer, 2 (1996), pp. 19–24.
[18] N. J. Higham, *The accuracy of floating point summation*, SIAM J. Sci. Comput., 14 (1993), pp. 783–799.
[19] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 2nd ed., 2002.
[20] C. M. Hoffmann, *Robustness in geometric computations*, Journal of Computing and Information Science in Engineering, 1 (2001), pp. 143–156.
[21] M. Jankowski, A. Smoktunowicz, and H. Woźniakowski, *A note on floating-point summation of very many terms*, Electron. Informationsverarb. Kybernet., 19 (1983), pp. 435–440.
[22] M. Jankowski and H. Woźniakowski, *The accurate solution of certain continuous problems using only sigle precision arithmetic*, BIT, 25 (1985), pp. 635–651.
[23] W. Kahan, *A survey of error analysis*, in Proceedings of the IFIP Congress, Information Processing 71, North-Holland, Amsterdam, 1972, pp. 1214–1239.
[24] W. Kahan, *Implementation of Algorithms (lecture notes by W. S. Haugeland and D. Hough)*, Tech. Report 20, Department of Computer Science, University of California, Berkeley, CA, 1973.
[25] A. Kiełbaszyński, *Summation algorithm with corrections and some of its applications*, Math. Stos, 1 (1973), pp. 22–41.
[26] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2, Addison-Wesley, Reading, MA, 1969.
[27] S. Krishnan, M. Foskey, T. Culver, J. Keyser, and D. Manocha, *PRECISE: Efficient multiprecision evaluation of*

algebraic roots and predicates for reliable geometric computation, in Proceedings of the 17th Annual Symposium on Computational Geometry, New York, NY, 2001, ACM Press, pp. 274–283.

[28] U. Kulisch and W. L. Miranker, *Arithmetic operations in interval spaces*, Computing, Suppl. 2 (1980), pp. 51–67.

[29] P. Langlois, *Accurate Algorithms in Floating Point Arithmetic*, Invited talk at the 12th GAMM–IMACS International Symposion on Scientific Computing, Computer Arithmetic and Validated Numerics, Duisburg, 26–29 September, 2006.

[30] P. Langlois and N. Louvet, *Solving Triangular Systems More Accurately and Efficiently*, Tech. Report RR2005-02, Laboratoire LP2A, University of Perpignan, 2005.

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

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

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

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

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

[36] A. Neumaier, *Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen*, Z. Angew. Math. Mech., 54 (1974), pp. 39–51.

[37] T. Ogita, S. M. Rump, and S. Oishi, *Accurate sum and dot product*, SIAM J. Sci. Comput., 26 (2005), pp. 1955–1988.

[38] K. Ozaki, T. Ogita, S. M. Rump, and S. Oishi, *Fast and robust algorithm for geometric predicates using floating-point arithmetic*, Transactions of the Japan Society for Industrial and Applied Mathematics, 16 (2006), pp. 553–562.

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

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

[41] D. M. 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.

[42] D. R. Ross, *Reducing truncation errors using cascading accumulators*, Comm. ACM, 8 (1965), pp. 32–33.

[43] S. M. Rump, *INTLAB - INTerval LABoratory*, in Developments in Reliable Computing, T. Csendes, ed., Kluwer Academic Publishers, Dordrecht, 1999, pp. 77–104.

[44] S. M. Rump, T. Ogita, and S. Oishi, *Accurate Floating-point Summation Part II: Sign, K-fold Faithful and Rounding to Nearest*, Tech. Report 07.2, Faculty of Information and Communication Science, Hamburg University of Technology, 2007. `http://www.ti3.tu-harburg.de/rump` .

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

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

[47] J. M. Wolfe, *Reducing truncation errors by programming*, Comm. ACM, 7 (1964), pp. 355–356.

[48] Y.-K. Zhu and W. Hayes, *Fast, guaranteed-accurate sums of many floating-point numbers*, in Proceedings of the 7th Conference on Real Numbers and Computers, G. Hanrot and P. Zimmermann, eds., 2006, pp. 11–22.

[49] Y.-K. Zhu, J.-H. Yong, and G.-Q. Zheng, *A new distillation algorithm for floating-point summation*, SIAM J. Sci. Comput., 26 (2005), pp. 2066–2078.

[50] G. Zielke and V. Drygalla, *Genaue Lösung Linearer Gleichungssysteme*, GAMM Mitt. Ges. Angew. Math. Mech., 26 (2003), pp.7–107.