submitted for publication, July 27, 2008

FAST HIGH PRECISION SUMMATION *

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

Abstract. Given a vector p_i of floating-point numbers with exact sum s, we present a new algorithm with the following property: Either the result is a faithful rounding of s, or otherwise the result has a relative error not larger than $\texttt{eps}^K \operatorname{cond}(\sum p_i)$ for K to be specified. The statements are also true in the presence of underflow, the computing time does not depend on the exponent range, and no extra memory is required. Our algorithm is fast in terms of measured computing time because it allows good instruction-level parallelism. A special version for $K = 2$, i.e., quadruple precision is also presented. Computational results show that this algorithm is more accurate and faster than competitors such as XBLAS.

Key words. summation, precision, accuracy, faithful rounding, error-free transformation, distillation, extra precision basic linear algebra subroutines, XBLAS, error analysis

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

1. Introduction and previous work.

We will present yet another new and fast algorithm to compute an approximation of high quality of the sum and the dot product of vectors of floating-point numbers. Since dot products can be transformed into sums without error [26], we concentrate on summation.

Sums of floating-point numbers are ubiquitous in scientific computations, and there is a vast amount of literature to that, among them [35, 1, 2, 3, 7, 8, 9, 12, 13, 14, 15, 16, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 33, 34, 36, 37, 38], all aiming on some improved accuracy of the result. Higham [10] 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 [10, 21].

We consider only algorithms using one working precision, for example, IEEE 754 double precision. One can distinguish two classes of algorithms: The first class delivers a result "as if" computed in some higher precision, for example, quadruple precision. The accuracy of the result depends on the condition number of the sum. Examples of such algorithms are XBLAS [21, 35] or `Sum2` and `SumK` in [26].

The second class of algorithms computes a result to a specified accuracy, for example a faithfully rounded result. Examples of such algorithms are using a long accumulator [23, 18], or are the newly developed algorithms in [31, 32] based on error-free transformations. Especially the latter like `AccSum` (Algorithm 4.5 in [31]) proved to be very fast, often faster than the XBLAS routines although being more accurate.

The new algorithm `AccSum` has the charming property that the computing time grows with the condition number: For "simple" problems it is fast, with mildly growing computing time for more difficult problems. However, as a drawback it requires additional memory of the size of the input vector.

*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`).

Often a result "as if" computed in quadruple precision is sufficient, for example for a residual iteration for systems of linear equations. In this paper we present a new algorithm based on `AccSum` producing a result of a quality better than when computed in quadruple precision. Basically no additional memory is required. Unlike `AccSum`, there is an upper limit for the computing time, independent of the condition number. However, for extremely ill-conditioned problems the accuracy of the result deteriorates.

The paper is organized as follows. First we introduce our new machinery introduced in [31] to analyze floating-point algorithms, namely the "unit in the first place"-notation. It allows to formulate proofs using inequalities; colloquial conclusions as often used in this realm disappear. In the following section we introduce and analyze error-free transformations. In Section 4 the new algorithm for general K is presented and analyzed, where in the following section it is shown how to avoid extra memory, comments for the special case $K = 2$, i.e., quadruple precision are added and implementation issues are discussed. The paper concludes with computational results on representative architectures and an appendix hosting parts of involved proofs.

As in [26], [31] and [32], all theorems, error analysis and proofs are due to the first author of the present paper.

2. Basic facts. Our new algorithm `PrecSum` and its analysis is based on algorithm `AccSum` (Algorithm 4.5 in [31]), which in turn uses ideas in [38]. In the sequel we repeat few basic facts from [31] to make the present paper mostly self-contained.

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 larger 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

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

Note that for $f \in \mathbb{U}$, $f \pm \texttt{eta}$ are the floating-point neighbors of f. 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 the summation may be performed in any order. We assume floating-point operations in rounding to nearest corresponding to the IEEE 754 arithmetic standard [11].

In [31] we introduced the "unit in the first place" (ufp) or leading bit of a real number by

$$0 \neq r \in \mathbb{R} \quad\Rightarrow\quad \mathrm{ufp}(r) := 2^{\lfloor \log_2 |r| \rfloor}\;, \tag{2.2}$$

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.

As in [31] 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 the set of fixed point numbers with smallest positive number $\texttt{eps}\sigma$. Of course, $\mathbb{F} \subseteq \texttt{eta}\mathbb{Z}$. These two concepts, the unit in the first place $\mathrm{ufp}(\cdot)$ together with $f \in \mathbb{F} \Rightarrow f \in 2\texttt{eps} \cdot \mathrm{ufp}(f)\mathbb{Z}$ proved to be very useful in the often delicate analysis of our algorithms. Note that (2.2) is independent of some 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

$$0 \neq r \in \mathbb{R} \quad\Rightarrow\quad \mathrm{ufp}(r) \leq |r| < 2\mathrm{ufp}(r) \tag{2.3}$$

$$a, b \in \mathbb{F} \cap \texttt{eps}\sigma\mathbb{Z} \quad\Rightarrow\quad \mathrm{fl}(a+b) \in \texttt{eps}\sigma\mathbb{Z} \tag{2.4}$$

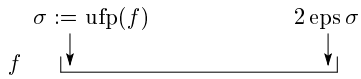

FIG. 2.1. *Normalized floating-point number: unit in the first place and unit in the last place*

$$n\mathtt{eps} \le 1,\ a_i \in \mathbb{F} \quad \text{and} \quad |a_i| \le \sigma \quad \Rightarrow \quad \begin{array}{l} |\mathrm{fl}(\sum_{i=1}^{n} a_i)| \le n\sigma \quad \text{and} \\ |\mathrm{fl}(\sum_{i=1}^{n} a_i) - (\sum_{i=1}^{n} a_i)| \le \frac{n(n-1)}{2}\mathtt{eps}\sigma \end{array} \tag{2.5}$$

$$a, b \in \mathbb{F},\ a \ne 0 \quad \Rightarrow \quad \mathrm{fl}(a+b) \in \mathtt{eps} \cdot \mathrm{ufp}(a)\mathbb{Z}\ , \tag{2.6}$$

see (2.9) through (2.20) in [31]. The fundamental error bound for floating-point addition is

$$f = \mathrm{fl}(a+b) \quad \Rightarrow \quad f = a + b + \delta \ \text{ with } \ |\delta| \le \mathtt{eps} \cdot \mathrm{ufp}(a+b) \le \mathtt{eps} \cdot \mathrm{ufp}(f) \le \mathtt{eps}|f|\ , \tag{2.7}$$

cf. (2.19) in [31]. Note that this improves the standard error bound $\mathrm{fl}(a+b) = (a+b)(1+\varepsilon)$ for $a, b \in \mathbb{F}$ and $|\varepsilon| \le \mathtt{eps}$ by up to a factor 2. Note that (2.7) is also true in the underflow range, in fact addition (and subtraction) is exact if $\mathrm{fl}(a \pm b) \in \mathbb{U}$. For $a, b \in \mathbb{F}$ and $\sigma = 2^k$, $k \in \mathbb{Z}$,

$$\begin{array}{rcl} a, b \in \mathtt{eps}\sigma\mathbb{Z} \quad \text{and} \quad |\mathrm{fl}(a+b)| < \sigma & \Rightarrow & \mathrm{fl}(a+b) = a+b \quad \text{and} \\ a, b \in \mathtt{eps}\sigma\mathbb{Z} \quad \text{and} \quad |a+b| \le \sigma & \Rightarrow & \mathrm{fl}(a+b) = a+b\ , \end{array} \tag{2.8}$$

cf. (2.21) in [31]. For later use we apply standard floating-point estimations [10] to derive the following. Let $p_i \in \mathbb{F}, 1 \le i \le n$ be given and denote $\gamma_m := m\mathtt{eps}/(1-m\mathtt{eps})$. Then $2n\mathtt{eps} < 1$ and $\mu := \mathrm{fl}\big((\sum_{i=1}^{n} |p_i|)/(1-2n\mathtt{eps})\big)$ imply $\mathrm{fl}(1-2n\mathtt{eps}) = 1-2n\mathtt{eps}$ and

$$\begin{array}{rcl} \sum_{i=1}^{n} |p_i| & \ge & \mathrm{fl}\big(\sum_{i=1}^{n} |p_i|\big)/(1+\gamma_{n-1}) \\ & = & \mu \cdot (1-2n\mathtt{eps})/(1+\gamma_n) \\ & \ge & \mu \cdot (1-3n\mathtt{eps})\ . \end{array} \tag{2.9}$$

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

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

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

LEMMA 2.1. *Let a floating-point number $0 \ne f \in \mathbb{F}$ be given. Then*

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

The result of algorithm `AccSum` (Algorithm 4.5 in [31]) is a *faithful rounding* [6, 29, 4] of the true result. Basically, a floating-point number f is a faithful rounding of a real number r if there is no other floating-point number between f and r.

DEFINITION 2.2. *A floating-point number $f \in \mathbb{F}$ is called a* faithful rounding *of a real number $r \in \mathbb{R}$ if*

$$\mathrm{pred}(f) < r < \mathrm{succ}(f)\ . \tag{2.10}$$

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

We will use the following criterion given in Lemma 2.4 in [31]).

LEMMA 2.3. *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 $r + \delta$.*

A main principle in [26, 31, 32] are *error-free transformations*: A vector p of floating-point numbers is transformed into a vector p' without changing the sum, where one element of p' is $\mathrm{fl}(\sum p_i)$. In the present paper we use the same principle.

A first step is the transformation of a pair of floating-point numbers a, b into a pair x, y with $x = \mathrm{fl}(a + b)$ and leaving the sum invariant. In [17] Knuth gave the following algorithm.

ALGORITHM 2.4. *Error-free transformation for the sum of two floating-point numbers.*

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

Knuth's algorithm satisfies

$$\forall\, a, b \in \mathbb{F}: \qquad x = \mathrm{fl}(a + b) \quad \text{and} \quad x + y = a + b. \tag{2.11}$$

This is also true in the presence of underflow. An error-free transformation for subtraction follows since $\mathbb{F} = -\mathbb{F}$. Algorithm `TwoSum` requires 6 floating-point operations. The following, faster version by Dekker [6] applies if a, b are somehow sorted.

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

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

In [31], Lemma 2.6 we analyzed the algorithm as follows.

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

$$x + y = a + b\ , \quad x = \mathrm{fl}(a + b) \quad \textit{and} \quad |y| \le \mathtt{eps} \cdot \mathrm{ufp}(a + b) \le \mathtt{eps} \cdot \mathrm{ufp}(x)\ . \tag{2.12}$$

Note that, for example, the commonly used assumption $|a| \ge |b|$ implies $a \in 2\mathtt{eps} \cdot \mathrm{ufp}(b)\mathbb{Z}$.

3. Extraction of high order parts. A key to our algorithm `AccSum` (Algorithm 4.5 in [31]) is the error-free splitting of a vector sum into high order and low order parts. The splitting is in such a way that the high order parts add without error.

The splitting is depicted in Figure 3.1. Note that neither the high order parts q_i and low order part p'_i need to match bitwise with the original p_i, nor must q_i and p'_i have the same sign; only the error-freeness of the transformation $p_i = q_i + p'_i$ is mandatory. This is achieved by the following fast algorithm [31], where σ denotes a power of 2 not less than $\max |p_i|$. For better readability 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.

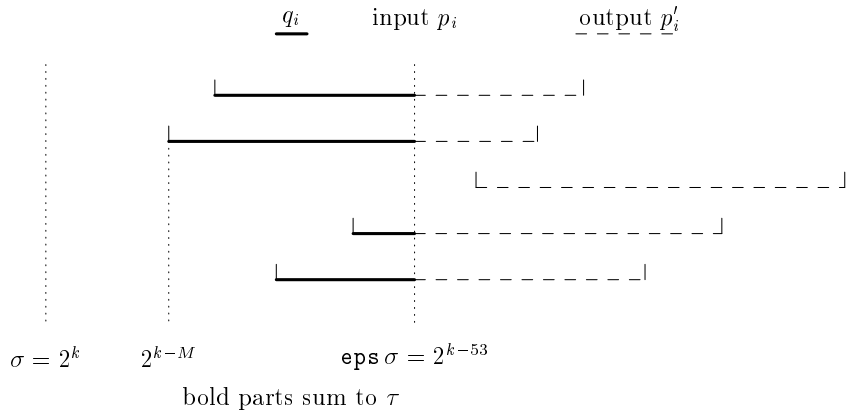

FIG. 3.1. `ExtractVector`*: error-free transformation* $\sum p_i = \tau + \sum p'_i$

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

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

Algorithm 3.1 proceeds as depicted in Figure 3.1. Note that the loop is in particular well-suited for today's compilers optimization and instruction-level parallelism.

THEOREM 3.2. *Let* τ *and* p' *be the results of Algorithm 3.1* (`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* $M \in \mathbb{N}$. *Suppose* $\max|p_i| \le 2^{-M}\sigma$ *is true, or* $\mathrm{fl}\big((\sum_{i=1}^n |p_i|)/(1 - 2n\mathtt{eps})\big) \le \sigma$ *is true. Then*

$$\sum_{i=1}^{n} p_i = \tau + \sum_{i=1}^{n} p'_i \;, \quad \max|p'_i| \le \mathtt{eps}\sigma \;, \quad |\tau| < \sigma \quad \textit{and} \quad \tau \in \mathtt{eps}\sigma\mathbb{Z} \;. \tag{3.1}$$

Algorithm 3.1 (`ExtractVector`) *needs* $4n + \mathcal{O}(1)$ *flops.*

REMARK. As will be seen in the proof, the assumption $\mathrm{fl}\big((\sum_{i=1}^n |p_i|)/(1 - 2n\mathtt{eps})\big) \le \sigma$ is used to show that the higher order parts sum without error, i.e., $\mathrm{fl}(\sum q_i) = \sum q_i$. To ensure this, $\mathrm{fl}\big(\sum_{i=1}^n |p_i|\big) \le \sigma$ is not sufficient as by the following example. Let $p \in \mathbb{F}^{10}$ with $p_{1\cdots 8} = 1 - 4\mathtt{eps} - \mathtt{eps}$, $p_9 = 16\mathtt{eps}$ and $p_{10} = -8\mathtt{eps}$. Then $\mathrm{fl}\big(\sum |p_i|\big) = 8 - 24\mathtt{eps} < 8$ implies $\sigma = 8$, $q_{1\cdots 8} = 1$, $q_9 = 16\mathtt{eps}$ and $q_{10} = -8\mathtt{eps}$. Hence $\sum q_i = 8 + 8\mathtt{eps} \notin \mathbb{F}$.

PROOF OF 3.2. For $|p_i| \le 2^{-M}\sigma$ the assertions were proved in Theorem 3.4 in [31]. It remains to prove (3.1) under the assumption $\mathrm{fl}\big((\sum_{i=1}^n |p_i|)/(1 - 2n\mathtt{eps})\big) \le \sigma$. For each $i \in \{1, \cdots, n\}$, the assumptions of Lemma 3.2 in [31] are satisfied and imply

$$p_i = q_i + p'_i \;, \quad \max|p'_i| \le \mathtt{eps}\sigma \quad \text{and} \quad q_i \in \mathtt{eps}\sigma\mathbb{Z} \;.$$

Using $1-2n\mathtt{eps} \in \mathbb{F}$ it follows

$$\begin{array}{rcl} \sum_{i=1}^n |q_i| & \leq & \sum_{i=1}^n \big(|p_i| + \mathtt{eps}\sigma\big) \\ & \leq & (1+\gamma_{n-1})\mathrm{fl}\big(\sum_{i=1}^n |p_i|\big) + n\mathtt{eps}\sigma \\ & \leq & (1+\gamma_n)\mathrm{fl}\big((\sum_{i=1}^n |p_i|)/(1-2n\mathtt{eps})\big) \cdot (1-2n\mathtt{eps}) + n\mathtt{eps}\sigma \\ & \leq & \frac{\sigma}{1-n\mathtt{eps}}(1-2n\mathtt{eps}) + n\mathtt{eps}\sigma \\ & < & \sigma \ , \end{array} \tag{3.2}$$

so that (2.8) shows $\mathrm{fl}(\sum q_i) = \sum q_i = \tau$. The lemma is proved. □

To apply Theorem 3.2 the best (smallest) value for M is $\lceil \log_2(n+1) \rceil$. It can be computed without using the binary logarithm. Algorithm 3.5 in [31] does this, but it contains a branch. The branch can be avoided as follows.

ALGORITHM 3.3. *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|)
```

THEOREM 3.4. *Let L be the result of Algorithm 3.3* (`NextPowerTwo`) *applied to a nonzero floating-point number p. If no overflow occurs, then $L = 2^{\lceil \log_2 |p| \rceil}$.*

PROOF. The assumptions imply $q \notin \mathbb{U}$. First assume $|p| = 2^k$ for some $k \in \mathbb{Z}$. Then $\mathrm{fl}(q-p) = \mathrm{fl}(q(1-\mathtt{eps})) = \mathrm{pred}(q)$, so that $\mathrm{fl}(|(q-p)-q|) = \mathtt{eps}|q| = |p|$. 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 $L = 2\mathrm{ufp}(p)$. By $q \notin \mathbb{U}$ and Lemma 2.1 we have $\mathrm{pred}(q) = q - 2\mathtt{eps} \cdot \mathrm{ufp}(q)$, so that $q - \mathtt{eps} \cdot \mathrm{ufp}(q)$ is the midpoint of $\mathrm{pred}(q)$ and q. Rounding to nearest and

$$q - \mathtt{eps} \cdot \mathrm{ufp}(q) = q - \mathrm{ufp}(p) > q - p > q - 2\mathrm{ufp}(p) = q - 2\mathtt{eps} \cdot \mathrm{ufp}(q) = \mathrm{pred}(q)$$

imply $\mathrm{fl}(q-p) = \mathrm{pred}(q)$. Hence $L = \mathrm{fl}(|\mathrm{pred}(q) - q|) = 2\mathtt{eps} \cdot \mathrm{ufp}(q) = 2\mathrm{ufp}(p)$. The theorem is proved. □

4. The general algorithm and its analysis. Let a vector $p \in \mathbb{F}^n$ be given and abbreviate $s := \sum_{i=1}^n p_i$. When computing the sum in K-fold precision, i.e., in a floating-point arithmetic $\mathrm{fl}_K(\cdot)$ with relative rounding error unit $\mathtt{eps}^K$, the standard error estimation yields [10]

$$\big|\mathrm{fl}_K\big(\sum_{i=1}^n p_i\big) - s\big| \leq \frac{(n-1)\mathtt{eps}^K}{1-(n-1)\mathtt{eps}^K} \sum_{i=1}^n |p_i| \ . \tag{4.1}$$

That means for nonzero sum the relative error of the floating-point approximation is of the order $\mathtt{eps}^K$ times the condition number $\sum |p_i| / |\sum p_i|$ of the sum [10, 31]. Note that estimation (4.1) is essentially sharp as shown by $p \in \mathbb{F}$ with $p_1 = 1$ and $p_{2\cdots n} = \mathtt{eps}^K$, because rounding tie to even implies $\mathrm{fl}_K(1 + \mathtt{eps}^K) = 1$, so that $\mathrm{fl}_K\big(\sum p_i\big) = 1$.

The aim of this paper is to derive an algorithm computing a result `res` which is a faithful rounding of the sum s, or which at least satisfies

$$\big|\mathtt{res} - s\big| \leq \mathtt{eps}^K \sum_{i=1}^n |p_i| \ .$$

That means we can expect the result to be better than when calculated in K-fold precision. For ease of analysis we specify the algorithm without overwriting variables.

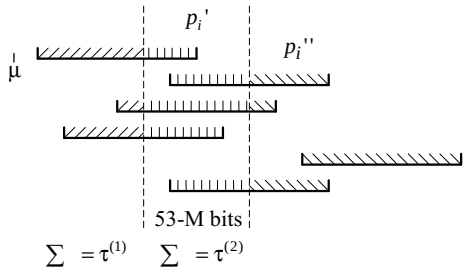

FIG. 4.1. *Extraction by Algorithm* `PrecSum`

ALGORITHM 4.1. *Preliminary version of summation algorithm.*

```
01  function res = PrecSum(p^(0), K)
02      n = length(p^(0))
03      μ = fl((Σ_{i=1}^{n} |p_i^(0)|)/(1 − 2n·eps))
04      if μ = 0, res = 0, return, end if
05      M = ⌈log_2(n + 2)⌉
06      L = ⌈(K · log_2 eps − 2)/(log_2 eps + M)⌉ − 1
07      σ_0 = NextPowerTwo(μ)
08      for k = 0 : L − 1
09          if σ_k > (1/2)·eps^{−1}·eta then σ_{k+1} = fl(2^M·eps·σ_k); else L = k; break; end if
10      end for
11      if L = 0 then res = Σ_{i=1}^{n} p_i^(0); return end if
12      for k = 1 : L
13          [τ^(k), p^(k)] = ExtractVector(σ_{k−1}, p^(k−1))
14      end for
15      π_1 = τ^(1); e_1 = 0
16      for k = 2 : L
17          [π_k, q_k] = FastTwoSum(π_{k−1}, τ^(k))
18          e_k = fl(e_{k−1} + q_k)
19      end for
20      res = fl(π_L + (e_L + (Σ_{i=1}^{n} p_i^(L))))
```

The algorithm works as follows (for simplicity we assume $\texttt{eps} = 2^{-53}$ as in IEEE 754 double precision). First μ is an upper bound of the sum of absolute values, where the factor $1/(1-2n\texttt{eps})$ takes care of rounding errors in the computation of μ. Then bits of the p_i are extracted in such a way that the unit in the last place of the high order parts is $\texttt{eps}\cdot\sigma_0$, where σ_0 is the smallest power of 2 greater or equal to μ. If p_i is smaller than $\texttt{eps}\cdot\sigma_0$, the high order part is 0. One can show that the high order parts add in floating-point without error into $\tau^{(1)}$.

Then $53-M$ bits from the low order parts p_i' are extracted into p_i'', where M is chosen in such a way that the p_i'' add into $\tau^{(2)}$ without error. This process is repeated L times. In Figure 4.1 we depict the process for $L=2$ extractions. It follows $s = \sum p_i = \tau^{(1)} + \tau^{(2)} + \sum p_i''$. The number of extractions L is computed such that rounding errors in the computation of the sum $\mathrm{fl}(\sum p_i'')$ of the elements of the finally extracted vector are small enough to either not jeopardize faithful rounding or, to guarantee a relative accuracy as if computed in K-fold precision.

PROPOSITION 4.2. *Let $p = p^{(0)}$ be a vector of n floating-point numbers, let $1 \le K \in \mathbb{N}$, define $M := \lceil \log_2(n+2) \rceil$ and assume $\texttt{eps} \le \frac{1}{1024}$ and $2^{2M}\texttt{eps} \le 1$. Assume $K \le (4\sqrt{\texttt{eps}})^{-1}$. Let* `res` *be the result of*

Algorithm 4.1 (`PrecSum`) *applied to* p. *Abbreviate* $s := \sum_{i=1}^{n} p_i$. *Then either*

$$\texttt{res} \text{ is a faithful rounding of } s \tag{4.2}$$

or, if `res` *is not a faithful rounding of* s, *then* $s \neq 0$ *and*

$$\frac{|\texttt{res} - s|}{|s|} \;<\; \frac{3 \cdot (2^M \texttt{eps})^{L+1}}{1 - 3n\texttt{eps}} \cdot \mathrm{cond}\,(\sum p_i) \;<\; \texttt{eps}^K \cdot \mathrm{cond}\,(\sum p_i) \tag{4.3}$$

using L *as computed in line* 6 *of* `PrecSum`. *Algorithm 4.1* (`PrecSum`) *needs* $(4L+3)n + \mathcal{O}(K)$ *flops.*

Remark 1. The major difference to Algorithm 4.5 (`AccSum`) in [31] is that in `PrecSum` the maximum number of extractions is estimated depending on the desired precision. An advantage is that the number of extractions and thus the maximum number of flops is known in advance (see the discussion in Section 5 for the important case $K = 2$, i.e., twice the working precision), a disadvantage is that possibly more extractions are performed than necessary.

However, this possible disadvantage seems to be more than compensated because, in contrast to `AccSum`, no extra memory of the size of the input vector is needed. We elaborate this in Section 5.

Remark 2. Note that compared to Algorithm 4.5 (`AccSum`) in [31] we changed the definition of μ from a maximum to a floating-point sum. This idea is due to the second author.

Remark 3. The code in the last for-loop (lines 16 to 19) gives the same result as Algorithm 4.4 (`Sum2`) in [26] applied to $\tau^{(1\cdots L)}$ except that the approximation π_L and the error term e_L is not added but kept separately. This is true because, as we will show, the input data implies that `FastTwoSum` and `TwoSum` yield identical results.

Remark 4. We counted the absolute value as one floating-point operation. In practice, this often comes with no extra time, in which case the flop-count can be decreased by n flops.

Remark 5. Note that Algorithm `PrecSum` definitely achieves a result "as if" computed in K-fold precision, so that for practical purposes the assumption $K \leq (4\sqrt{\texttt{eps}})^{-1}$ is artificial. In IEEE 754 single precision it limits K to 1024 or 81640 decimal digits precision, in double precision K is limited to an equivalent of more than 4 billion decimal digits precision. So the precision is much larger than the exponent range.

Remark 6. We mention that if $|\sum_{\nu=1}^{k} \tau^{(\nu)}| \geq \mathrm{fl}(2^{2M}\texttt{eps}\sigma_{k-1})$, the stopping criterion in `AccSum`, is satisfied for some $k < L$, one may replace the computation of `res` by $\texttt{res} = \mathrm{fl}\big(\sum_{\nu=1}^{k} \tau^{(\nu)}\big)$, thus skipping sum of the vector of low order parts $p_i^{(L)}$. This saves n flops, and it can be proved that the result `res` satisfies the slightly weaker condition $\mathrm{fl}(s) \in \{\mathrm{pred}(\texttt{res}), \texttt{res}, \mathrm{succ}(\texttt{res})\}$ rather than being a faithful rounding.

Proof of Proposition 4.2. If σ_0 is in the underflow range, i.e., $\sigma_0 \leq \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then the computation of μ and an estimation like in (3.2) imply $\sum_{i=1}^{n} |p_i^{(0)}| < \sigma_0$. Hence all $p_i^{(0)}$ and the sum s are in the underflow range, so that all $p_i^{(0)}$ add without rounding error. It follows $\texttt{res} = \sum_{i=1}^{n} p_i^{(0)}$. Henceforth we may assume $\sigma_0 > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, so that $K \geq 1$ implies $L \geq 1$.

The splitting constant σ_{k+1} is only computed when σ_k is not in the underflow range, so that all σ_k are positive and computed without rounding error for $0 \leq k \leq L$. For later use we note that the computation of L yields $(2^M\texttt{eps})^{L+1} \leq \texttt{eps}^K/4$, so that

$$2^{2M}\texttt{eps}^2\sigma_{L-1} = \sigma_{L+1} \leq \frac{\texttt{eps}^K}{4}\sigma_0 \;. \tag{4.4}$$

Furthermore, a straightforward computation using $2^{2M}\texttt{eps} \leq 1$ and $\texttt{eps} \leq \frac{1}{128}$ shows

$$\frac{3}{4} \leq 1 - 3n\texttt{eps} \;. \tag{4.5}$$

The computation of μ implies that the assumptions of Theorem 3.2 are satisfied for $k = 0$, so that

$$s = \sum_{i=1}^{n} p_i^{(0)} = \tau^{(1)} + \sum_{i=1}^{n} p_i^{(1)} \ , \ \ \max|p_i^{(1)}| \le \mathtt{eps}\sigma_0 = 2^{-M}\sigma_1 \ , \ \ |\tau^{(1)}| < \sigma_0 \ \text{ and } \ \tau^{(1)} \in \mathtt{eps}\sigma_0\mathbb{Z} \ . \tag{4.6}$$

By $\max|p_i^{(1)}| \le 2^{-M}\sigma_1$ the assumptions of Theorem 3.2 are satisfied for $k = 1$ as well, and repeating this argument we conclude that

$$\sum_{i=1}^{n} p_i^{(k)} = \tau^{(k+1)} + \sum_{i=1}^{n} p_i^{(k+1)} \qquad \text{for } 1 \le k < L \ , \tag{4.7}$$

and that

$$s = \sum_{\nu=1}^{k} \tau^{(\nu)} + \sum_{i=1}^{n} p_i^{(k)} \ , \ \ \max|p_i^{(k)}| \le \mathtt{eps}\sigma_{k-1} \ , \ \ |\tau^{(k)}| < \sigma_{k-1} \ \text{ and } \ \tau^{(k)} \in \mathtt{eps}\sigma_{k-1}\mathbb{Z} \tag{4.8}$$

is satisfied for $1 \le k \le L$.

In order to show that the assumptions of Lemma 2.6 for the use of `FastTwoSum` in line 17 are satisfied, we first prove

$$\pi_k \in \mathtt{eps}\sigma_{k-1}\mathbb{Z} \quad \text{ for } \quad 1 \le k \le L \tag{4.9}$$

by induction. For $k = 1$ this is true by (4.6). By the induction hypothesis $\pi_{k-1} \in \mathtt{eps}\sigma_{k-2}\mathbb{Z} \subseteq \mathtt{eps}\sigma_{k-1}\mathbb{Z}$. But also $\tau^{(k)} \in \mathtt{eps}\sigma_{k-1}\mathbb{Z}$ by (4.8), so that $\pi_k = \mathrm{fl}(\pi_{k-1} + \tau^{(k)})$ and (2.4) show (4.9). But $2\mathrm{ufp}(\tau^{(k)}) \le \sigma_{k-1}$ by (4.8), so that (4.9) implies $\pi_k \in \mathtt{eps}\sigma_{k-1}\mathbb{Z} \subseteq 2\mathtt{eps} \cdot \mathrm{ufp}(\tau^{(k)})\mathbb{Z}$, and the assumptions of Lemma 2.6 are indeed satisfied. Hence

$$\pi_k + q_k = \pi_{k-1} + \tau^{(k)} \ , \quad \pi_k = \mathrm{fl}(\pi_{k-1} + \tau^{(k)}) \quad \text{and} \quad |q_k| \le \mathtt{eps} \cdot \mathrm{ufp}(\pi_k) \tag{4.10}$$

is satisfied for $2 \le k \le L$. Combining (4.8) and (4.10) gives

$$s = \pi_L + \sum_{k=2}^{L} q_k + \sum_{i=1}^{n} p_i^{(L)} \ . \tag{4.11}$$

We distinguish four cases. First,

$$\text{assume} \quad |\pi_1| \ge 2^{2M}\mathtt{eps}\sigma_0 \qquad \text{and} \quad L = 1 \ . \tag{4.12}$$

Define

$$\tau' := \mathrm{fl}\Big(\sum_{i=1}^{n} p_i^{(1)}\Big) = \sum_{i=1}^{n} p_i^{(1)} - \delta \ .$$

Then (4.11) implies

$$s = \pi_1 + \tau' + \delta \quad \text{ and } \quad \mathtt{res} = \mathrm{fl}(\pi_1 + \tau') \ . \tag{4.13}$$

Moreover, (4.6) and (2.5) give

$$|\tau'| \le n\mathtt{eps}\sigma_0 \quad \text{ and } \quad |\delta| \le \frac{1}{2}n(n-1)\mathtt{eps}^2\sigma_0 \ . \tag{4.14}$$

If $|\pi_1| < 2\mathtt{eps}^{-1}\mathtt{eta}$, then (4.12) yields

$$|\delta| < \frac{1}{2}2^{2M}\mathtt{eps}^2\sigma_0 < \mathtt{eta} \ , \tag{4.15}$$

and $\mathbb{F} \subseteq \mathtt{eta}\mathbb{Z}$ implies $\delta = 0$, so that by (4.13) the result $\mathtt{res}$ is even the rounded-to-nearest result $\mathrm{fl}(s)$. Hence we may assume $|\pi_1| \geq 2\mathtt{eps}^{-1}\mathtt{eta}$. Then (4.13), (4.14), $n\mathtt{eps} < 2^M\mathtt{eps} \leq 2^{-M}$ and (4.12) yield

$$|\mathtt{res}| \geq (1-\mathtt{eps})|\pi_1 + \tau'| > (1-\mathtt{eps})(1-2^{-M})|\pi_1| \geq \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}\ ,$$

so that $\mathtt{res} \notin \mathbb{U}$. Hence by Lemma 2.3 we have to show $|\delta| < \frac{1}{2}\mathtt{eps}|\mathtt{res}|$ to prove that $\mathtt{res}$ is a faithful rounding. This follows by (4.14) and

$$\begin{aligned} 2\mathtt{eps}^{-1}|\delta| - |\mathtt{res}| &\leq n(n-1)\mathtt{eps}\sigma_0 - (1+\mathtt{eps})|\pi_1 + \tau'| \\ &\leq n(n-1)\mathtt{eps}\sigma_0 - (1+\mathtt{eps})(2^{2M}\mathtt{eps}\sigma_0 - n\mathtt{eps}\sigma_0) \\ &\leq (1+\mathtt{eps})n^2\mathtt{eps}\sigma_0 - (1+\mathtt{eps})2^{2M}\mathtt{eps}\sigma_0 < 0\ . \end{aligned}$$

This concludes the first case. Second,

$$\text{assume} \quad |\pi_k| < 2^{2M}\mathtt{eps}\sigma_{k-1} \quad \text{for all } k \in \{1, \cdots, L\}\ . \tag{4.16}$$

Abbreviate

$$\tau' = \mathrm{fl}\Big(\sum_{i=1}^{n} p_i^{(L)}\Big) = \sum_{i=1}^{n} p_i^{(L)} - \delta\ . \tag{4.17}$$

We first show $q_k = 0$ for $k \in \{2, \cdots, L\}$, so that by (4.11),

$$s = \pi_L + \tau' + \delta \quad \text{and} \quad \mathtt{res} = \mathrm{fl}(\pi_L + \tau')\ . \tag{4.18}$$

Indeed (4.9), (4.8) and (4.16) yield $\pi_{k-1} \in \mathtt{eps}\sigma_{k-2}\mathbb{Z} \subseteq \mathtt{eps}\sigma_{k-1}\mathbb{Z}$, $\tau^{(k)} \in \mathtt{eps}\sigma_{k-1}\mathbb{Z}$ and $|\mathrm{fl}(\pi_{k-1} + \tau^{(k)})| = |\pi_k| < \sigma_{k-1}$, so that $q_k = 0$ follows by (2.8).

By (4.8) we have $\max|p_i^{(L)}| \leq \mathtt{eps}\sigma_{L-1}$, and (4.17) and (2.5) yield

$$|\tau'| \leq n\mathtt{eps}\sigma_{L-1} \quad \text{and} \quad |\delta| \leq \frac{1}{2}n(n-1)\mathtt{eps}^2\sigma_{L-1}\ . \tag{4.19}$$

Hence there is $|\Theta| \leq \mathtt{eps}$ with $|\mathtt{res} - s| = |(1+\Theta)(\pi_L + \tau') - s| \leq \mathtt{eps}|\pi_L + \tau'| + |\delta|$, and using $|\pi_L| < 2^{2M}\mathtt{eps}\sigma_{L-1}$, (4.19) and $n + 2 \leq 2^M$ imply

$$\begin{aligned} |\mathtt{res} - s| &< 2^{2M}\mathtt{eps}^2\sigma_{L-1} + n\mathtt{eps}^2\sigma_{L-1} + \tfrac{1}{2}n(n-1)\mathtt{eps}^2\sigma_{L-1} \\ &= (2^{2M} + \tfrac{1}{2}n(n+1))\mathtt{eps}^2\sigma_{L-1} \\ &\leq \tfrac{3}{2}\cdot 2^{2M}\mathtt{eps}^2\sigma_{L-1} = 3(2^M\mathtt{eps})^{L+1}\cdot\tfrac{1}{2}\sigma_0\ . \end{aligned}$$

If $\mathtt{res}$ is not a faithful rounding of s, then $s \neq 0$ by Definition 2.2, and using $\frac{1}{2}\sigma_0 < \mu \leq \sigma_0$, (2.9), (4.4) and (4.5) proves (4.3) and concludes the second case. Third,

$$\text{assume} \quad |\pi_1| \geq 2^{2M}\mathtt{eps}\sigma_0 \qquad \text{and} \quad L > 1\ . \tag{4.20}$$

Now, in contrast to the second case, the q_k may be nonzero producing a nonzero e_L. Thus, an additional rounding error is introduced in the computation of $\mathtt{res}$. Now the estimations become involved and are moved to the Appendix.

Fourth and finally,

$$\text{assume} \quad |\pi_k| < 2^{2M}\mathtt{eps}\sigma_{k-1} \quad \text{for} \quad 1 \leq k < m \leq L \quad \text{and} \quad |\pi_m| \geq 2^{2M}\mathtt{eps}\sigma_{m-1}. \tag{4.21}$$

As in (4.18) we deduce that $q_k = 0$ for $1 \leq k < m$, so that (4.11) writes

$$s = \pi_L + \sum_{k=m}^{L} q_k + \sum_{i=1}^{n} p_i^{(L)}.$$

Hence for $m = L$ we proceed as in the first case, and for $m < L$ we proceed as in the third case. The proof is finished. □

Note that except the second case in the proof of Proposition 4.2, namely that $|\pi_k| < 2^{2M}\texttt{eps}\sigma_{k-1}$ for all $k \in \{1, \cdots, L\}$, the result `res` is always proved to be a faithful rounding of s.

Next we can determine for which condition numbers the result of Algorithm 4.1 (`PrecSum`) will definitely be a faithful rounding. Recall [10, 26] the condition number of a nonzero sum $\text{cond}\,(\sum p_i) = \sum |p_i| / |\sum p_i|$.

THEOREM 4.3. *Let p be a vector of n floating-point numbers with nonzero sum $s := \sum p_i$, let $1 \le K \in \mathbb{N}$, define $M = \lceil \log_2(n+2) \rceil$ and define L as in Algorithm 4.1 (`PrecSum`). Assume*

$$\text{cond}\,\big(\sum p_i\big) \le \frac{1 - 3n\texttt{eps}}{2(2^{2M}+n)\texttt{eps}(2^M\texttt{eps})^{L-1}}. \tag{4.22}$$

Then the result `res` of `PrecSum` is a faithful rounding of the sum s.

REMARK. Note that L is defined by K, n and `eps`, so that the right hand side in (4.22) depends only the specified K-fold precision, on the dimension n and the relative rounding error unit `eps`.

PROOF OF THEOREM 4.3. By the computation of μ in `PrecSum` and (2.9) we know

$$\sum_{i=1}^{n} |p_i| \ge \frac{1}{2}\sigma_0(1 - 3n\texttt{eps}) \ .$$

Abbreviating the right hand side in (4.22) by C it follows

$$|s| \ge \frac{1}{2C}\sigma_0(1 - 3n\texttt{eps}) \ . \tag{4.23}$$

For the purpose of establishing a contradiction, suppose (4.16) is true. Then (4.18), (4.8) and $\sigma_{L-1} = (2^M\texttt{eps})^{L-1}$ imply

$$\begin{array}{rcl} |\pi_L| - 2^{2M}\texttt{eps}\sigma_{L-1} & \ge & |s| - |\sum_{i=1}^{n} p_i^{(L)}| - 2^{2M}\texttt{eps}\sigma_{L-1} \\ & \ge & \frac{1}{2C}\sigma_0(1 - 3n\texttt{eps}) - (n + 2^{2M})\texttt{eps}(2^M\texttt{eps})^{L-1}\sigma_0 \\ & \ge & 0 \ . \end{array}$$

This contradicts (4.16), which means $|\pi_k| \ge 2^{2M}\texttt{eps}\sigma_{k-1}$ must be true for some $k \in \{1, \cdots, L\}$. The result follows. □

For example, for IEEE 754 double precision and $K = 2$ and $K = 3$ corresponding to quadruple and 6-fold precision, respectively, we display some data in Table 4.1. First, for different dimensions n, the number of extractions L is shown. Furthermore, we display the condition number C, $C\texttt{eps}^K$ and the number of flops. They mean the following. For condition number up to C the result is definitely a faithful rounding. For given K and condition number C, we may expect a result with a relative error of the order $C\texttt{eps}^K$ as displayed in columns 4 and 8 in Table 4.1. Instead, `PrecSum` produces a faithfully rounded result, i.e., with relative error not more than $\texttt{eps} = 1.1 \cdot 10^{-16}$. Therefore the result of `PrecSum` is significantly better than the one produced by computing in K-fold precision.

As can be seen by Table 4.1, for $K = 2$ corresponding to quadruple precision the number of extractions is 2 or 3. We will make use of this fact in a specialized Algorithm `QuadSum` in the next section.

For comparison we mention that XBLAS summation [21, 35] requires $10n$ flops, and `Sum2` [26] requires $7n$ flops. Both produce a result "as if" computed in quadruple precision. Moreover, `Sum3` [26] requires $13n$ flops and produces a result "as if" computed in 6-fold precision. This is the theoretical flop count; the practical, measured computing time is influenced by many factors. Detailed comparisons are given in Section 6.

TABLE 4.1
Condition numbers and flops up to which Algorithm 4.1 (`PrecSum`*) computes faithfully rounded result in double precision*

n	$K=2$				$K=3$			
	L	C	$C\mathtt{eps}^2$	flops	L	C	$C\mathtt{eps}^3$	flops
30	2	$1.2 \cdot 10^{27}$	$1.5 \cdot 10^{-5}$	$11n$	3	$3.4 \cdot 10^{41}$	$4.6 \cdot 10^{-7}$	$15n$
254	2	$2.4 \cdot 10^{24}$	$3.0 \cdot 10^{-8}$	$11n$	3	$8.5 \cdot 10^{37}$	$1.2 \cdot 10^{-10}$	$15n$
2,046	2	$4.7 \cdot 10^{21}$	$5.8 \cdot 10^{-11}$	$11n$	3	$2.1 \cdot 10^{34}$	$2.8 \cdot 10^{-14}$	$15n$
16,382	2	$9.2 \cdot 10^{18}$	$1.1 \cdot 10^{-13}$	$11n$	4	$2.8 \cdot 10^{42}$	$3.8 \cdot 10^{-6}$	$19n$
131,070	2	$1.8 \cdot 10^{16}$	$2.2 \cdot 10^{-16}$	$11n$	4	$8.5 \cdot 10^{37}$	$1.2 \cdot 10^{-10}$	$19n$
1,048,574	3	$3.0 \cdot 10^{23}$	$3.7 \cdot 10^{-9}$	$15n$	4	$2.6 \cdot 10^{33}$	$3.6 \cdot 10^{-15}$	$19n$
8,388,606	3	$7.4 \cdot 10^{19}$	$9.1 \cdot 10^{-13}$	$15n$	5	$8.5 \cdot 10^{37}$	$1.2 \cdot 10^{-10}$	$23n$
67,108,862	3	$1.8 \cdot 10^{16}$	$2.2 \cdot 10^{-16}$	$15n$	5	$3.2 \cdot 10^{32}$	$4.4 \cdot 10^{-16}$	$23n$

5. Avoiding extra memory and quadruple precision summation. There are two interesting specializations of Algorithm 4.1 (`PrecSum`). First, the case $K = 2$ corresponds to a result as if computed in (in fact, better than) twice the working precision. There are prominent competitors like XBLAS [21, 35], and our previous Algorithm `Sum2` in [26].

Second, we can fix the number of extractions to $L = 1$. This corresponds to a little less than twice the working precision. Up to a certain condition number, depending on the vector length, a faithfully rounded result is still guaranteed. A lower bound for this condition number C can be computed by Theorem 4.3 and is displayed in Table 5.1. Note that we fixed the number of extractions to $L = 1$, so that the number of floating-point operations is fixed.

TABLE 5.1
Condition numbers up to which Algorithm 4.1 (`PrecSum`*) with $L = 1$ computes faithfully rounded result*

n	L	C	flops
30	1	$4.2 \cdot 10^{12}$	$7n$
254	1	$6.8 \cdot 10^{10}$	$7n$
2,046	1	$1.0 \cdot 10^{9}$	$7n$
16,382	1	$1.6 \cdot 10^{7}$	$7n$

Algorithm 4.1 (`PrecSum`) was specified without overwriting variables to ease the analysis. Following we comment on an actual implementation and expand `ExtractVector` in the main loop in `PrecSum`. For clarity, variables are still not overwritten.

ALGORITHM 5.1. *The inner loop of Algorithm 4.1 (*`PrecSum`*).*

for $k = 1 : L$
  $\tau^{(k)} = 0$
  for $i = 1 : n$ % $[\tau^{(k)}, p^{(k)}] =$ `ExtractVector`$(\sigma_{k-1}, p^{(k-1)})$
    $q_i^{(k)} = \mathrm{fl}\left((\sigma_{k-1} + p_i^{(k)}) - \sigma_{k-1}\right)$
    $p_i^{(k+1)} = \mathrm{fl}(p_i^{(k)} - q_i^{(k)})$
    $\tau^{(k)} = \mathrm{fl}(\tau^{(k)} + q_i^{(k)})$
  end for
end for

The inner loop produces an array of $n \times L$ elements $p_i^{(k)}$, and the elements are computed columnwise. However, the computations are sufficiently independent and can be performed rowwise as well.

Therefore we can rearrange the inner loop of `PrecSum` as in Algorithm 5.2. The intermediate extracted parts are directly summed into $\tau^{(1\cdots L)}$. This summation is error-free. The final extracted parts $p_i^{(L)}$ are also summed directly into P without the need of extra storage.

ALGORITHM 5.2. *Improved inner loop of Algorithm* `PrecSum`.

```
for k = 1 : L,  τ^(k) = 0;  end for        % initialization
P = 0
for i = 1 : n
    π = p_i
    for k = 1 : L
        q = fl((σ_{k-1} + π) − σ_{k-1})    % extraction
        π = π − q                          % error-free
        τ^(k) = τ^(k) + q                  % error-free
    end for
    P = fl(P + π)                          % avoid extra vector
end for
```

In a practical application, the working precision and the desired precision K are usually fixed or at least known in advance. For example, for IEEE 754 double precision and $K = 2$, i.e., for producing a result of (better than) quadruple precision, we know by Table 4.1 that $L = 2$ or $L = 3$. That means we can expand the loop on L to improve the performance. We name that algorithm `QuadSum`.

ALGORITHM 5.3. *Inner loop of Algorithm* `QuadSum`.

```
if L = 2 then
    τ^(1) = τ^(2) = P = 0
    for i = 1 : n
        π = p_i
        q = fl((σ_0 + π) − σ_0)            % first extraction
        π = π − q                          % error-free
        τ^(1) = τ^(1) + q                  % error-free
        q = fl((σ_1 + π) − σ_1)            % second extraction
        τ^(2) = τ^(2) + q                  % error-free
        P = fl(P + (π − q))                % avoid extra vector
    end for
else
    ... similar code for L = 3 computing τ^(1), τ^(2), τ^(3), P
end if
```

Again, the variable P in Algorithm 5.3 hosts the floating-point sum $\sum p_i^{(L)}$ of the final low order parts. It is clear how to implement the entire algorithm `QuadSum`.

The main advantage of rearranging the inner loop into a rowwise computation is that no extra memory of the size of the input vector is required. Note one can expect that the achieved accuracy is much better that computation in quadruple or K-fold precision.

6. 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.

TABLE 6.1
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 6.2]`
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 6.2
Compile options for Pentium 4, Intel Visual Fortran 9.1

Algorithm	Necessity of compile option `/Op` for inner loop
`DSum`	No
`Sum2`	Yes, for `TwoSum`
`XBLAS`	Yes, for `TwoSum` and `FastTwoSum`
`PrecSum`	No
`AccSum`	No

6.1. Results on summation. We shall evaluate the performance of our high precision summation algorithms. All algorithms are tested in three different environments, namely Pentium 4, Itanium 2 and Athlon 64, see Table 6.1. We carefully choose compiler options to achieve best possible results, see Tables 6.1 and 6.2. We use a simple trick like in Algorithm 3.1 (`ExtractVector`) $|\sigma + p_i| - \sigma$ instead of $(\sigma + p_i) - \sigma$ to avoid overdoing the code optimization/simplification by the Intel compiler for the Pentium 4 environment. For details, see [31].

Test examples for huge condition numbers larger than $\texttt{eps}^{-1}$ were generated by Algorithm 6.1 in [26], where a method to generate two vectors whose dot product is arbitrarily ill-conditioned is described. Dot products are transformed into sums by Dekker's and Veltkamp's Algorithms `Split` and `TwoProduct`, see [26].

First, we compare `PrecSum` with the ordinary, recursive summation `DSum`, with `Sum2` taken from [26] and the XBLAS summation algorithm `BLAS_dsum_x` from [35] (called `XBLAS` in the following tables) in terms of measured computing time. The latter two deliver a result *as if* calculated in quasi-quadruple precision. 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. We compare to recursive summation `DSum`, the time of which is normed to 1. This is only for reference; for condition number 10^{16} we cannot expect `DSum` to produce a single correct digit. In addition, we also compare to our accurate summation algorithm `AccSum` [31]. Note that the comparison is not really fair since `AccSum` produces always a faithfully rounded result independent of the condition number. We summarize the properties of the algorithms tested:

Algorithm	`DSum`	`Sum2`	`XBLAS`	`PrecSum` $(L = 1)$	`PrecSum` $(L = 2)$	`AccSum`
Precision	`eps`	$\mathcal{O}(n^2)\texttt{eps}^2$	$\mathcal{O}(n)\texttt{eps}^2$	$\mathcal{O}(n^2)\texttt{eps}^2$	$\mathcal{O}(n^3)\texttt{eps}^3$	adaptive
flops	n	$7n$	$10n$	$7n$	$11n$	adaptive

Note that the estimated *minimum* precision is displayed; in practice it is usually better (see Figure 6.2). The results are displayed in Tables 6.3, 6.4 and 6.5. For example, `PrecSum` with $L = 1$ achieves on the different architectures a remarkable factor of about 5, 9 or 10 compared to recursive summation, and the results of `PrecSum` with $L = 2$ follow. We also see that `PrecSum` with $L = 1, 2$ is significantly faster than `XBLAS`, on Pentium 4 even faster than `Sum2`. As has been mentioned earlier, this is in particular due to a better instruction-level parallelism of `AccSum` and `Sum2` as analyzed by Langlois [19]. We also observe a

TABLE 6.3
Measured computing times for cond $= 10^{16}$*, time of* DSum *normed to 1 (CPU: Intel Pentium 4 2.53GHz, Compiler: Intel Visual Fortran 9.1)*

n	Sum2	XBLAS	PrecSum ($L=1$)	PrecSum ($L=2$)	AccSum
100	23.53	77.06	10.00	14.71	14.12
400	26.67	86.67	8.00	12.67	12.00
1,600	19.02	65.85	5.37	8.78	9.76
6,400	19.00	68.00	5.50	9.00	9.00
25,600	18.14	62.33	4.65	8.37	12.09
102,400	1.99	6.92	2.09	2.14	8.24
409,600	2.01	6.65	2.01	1.96	8.22
1,638,400	1.99	6.62	2.04	1.99	8.16

TABLE 6.4
Measured computing times for cond $= 10^{16}$*, time of* DSum *normed to 1 (CPU: Intel Itanium 2 1.4GHz, Compiler: Intel Fortran 9.0)*

n	Sum2	XBLAS	PrecSum ($L=1$)	PrecSum ($L=2$)	AccSum
100	2.53	15.19	4.37	5.65	7.69
400	5.78	38.84	8.07	10.96	13.13
1,600	7.11	49.74	9.12	12.70	15.42
6,400	7.40	51.45	9.35	13.10	15.75
25,600	7.95	56.16	9.95	13.89	21.41
102,400	7.34	50.69	9.16	12.96	25.96
409,600	2.08	12.97	3.00	3.83	12.80
1,638,400	2.07	12.98	2.97	3.83	12.76

certain drop in the ratio for larger dimensions due to cache misses. Note that in Table 6.5, DSumU, Sum2U and XBLASU refer to the unrolled versions, respectively, and the time for DSumU is normed to 1. This is because we observed a significant difference between the recursive summation DSum and its unrolled version DSumU. Collecting 4 terms at a time proved to be a good choice.

The good performance of PrecSum becomes transparent when looking at the MFlops-rate. In Figure 6.1 the MFlops are displayed for the different algorithms on Pentium 4, the figure corresponding to the previously displayed results. Note that in view of the clock rate of 2.53GHz the performances of the algorithms DSum, PrecSum and AccSum are fairly good until the cache misses occur.

Next, we compare PrecSum with SumK in terms of result accuracy. The following table displays the minimally estimated computational precision for SumK with $K = 3, 4$ and PrecSum with $L = 3, 4$:

Algorithm	SumK ($K=3$)	SumK ($K=4$)	PrecSum ($L=3$)	PrecSum ($L=4$)
Precision	$\mathcal{O}(n^3)\mathbf{eps}^3$	$\mathcal{O}(n^4)\mathbf{eps}^4$	$\mathcal{O}(n^4)\mathbf{eps}^4$	$\mathcal{O}(n^5)\mathbf{eps}^5$
flops	$13n$	$19n$	$15n$	$19n$

The flop count for PrecSum with $L = 4$ is the same as that for SumK with $K = 4$. However, the computational precision of PrecSum with $L = 4$ is much higher, so that its result accuracy is expected to be much better. To confirm it, we test sums $\sum_{i=1}^{n} p_i$ for $n = 1000$ with the condition number varying from 10 to 10^{80}. The results of relative errors for the summation algorithms are displayed in Figure 6.2. In Figure 6.2, the dashed lines represent the error bounds $n^k\mathbf{eps}^k\text{cond}\left(\sum p_i\right)$ for $k = 1, 2, \ldots, 5$ from the left to the right.

TABLE 6.5
Measured computing times for cond $= 10^{16}$*, time of* `DSumU` *normed to 1 (CPU: AMD Athlon 64 2.2GHz, Compiler: GNU gfortran 4.1.1)*

n	`Sum2U`	`XBLASU`	`PrecSum` ($L=1$)	`PrecSum` ($L=2$)	`AccSum`
100	8.71	24.09	9.39	15.24	14.29
400	10.08	28.18	9.45	16.33	15.05
1,600	10.59	30.05	9.56	16.90	16.05
6,400	10.79	30.64	9.75	16.19	16.37
25,600	5.41	15.27	4.89	8.55	10.99
102,400	2.17	5.94	2.53	3.60	8.44
409,600	2.00	5.49	2.47	3.42	7.92
1,638,400	2.00	5.51	2.44	3.44	7.95

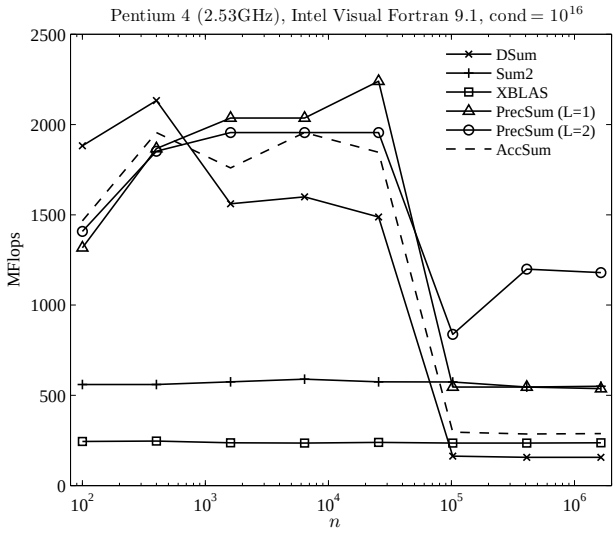

FIG. 6.1. *Measured MFlops on Pentium 4 (2.53GHz), Intel Visual Fortran 9.1,* cond $= 10^{16}$

From the results, we can see that the quality of the results obtained by `PrecSum` with $L = 4$ is several orders of magnitude better than that by `SumK` with $K = 4$, as expected. Moreover, it can also be seen that the quality of the actual results by the summation algorithms is usually much better than the worst case estimation, as mentioned before.

6.2. Results on matrix-vector product. Next, we shall apply our summation algorithm `PrecSum` to the computation of a residual $Ax - b$ with sparse $A = (a_{ij}) \in \mathbb{F}^{n\times n}$ and $x, b \in \mathbb{F}^n$, and evaluate its performance in parallel computing on a shared memory machine. We assume that CRS (Compressed Row Storage) format is used for storing a sparse matrix A, a standard row-oriented storage format. We transform products $a_{ij} \cdot x_j$ for $1 \le j \le n$ into sums by Dekker's and Veltkamp's Algorithms `Split` and `TwoProduct`. Using CRS format, no extra working memory is necessary to compute Ax (in contrast to $A^T x$). We use the following computer environment:

CPU:	Intel Dual-Core Xeon 2.80GHz × 4 processors (8 cores in total)
Compiler:	GCC 4.2.4
Compile option:	`-O3 -march=nocona -funroll-all-loops`
Parallelization:	OpenMP supported by the compiler (`-fopenmp`)

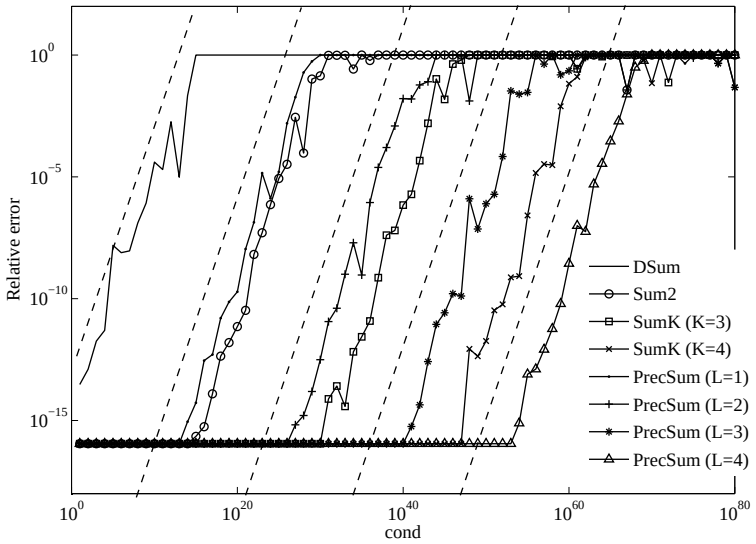

FIG. 6.2. *Relative errors of several summation algorithms,* $n = 1000$

TABLE 6.6
Ratio of computing time (1 core), elapsed time for `DDotMV` *is normed to 1.*

name	n	`nnz`(A)	`Dot2MV`	`PrecMV` $(L=1)$	`PrecMV` $(L=2)$
cryg10000	10,000	49,699	4.33	6.00	10.33
rajat26	10,605	424,587	3.23	4.50	7.27
rajat23	51,032	249,302	3.27	4.50	7.14
venkat01	62,424	1,717,792	4.30	4.84	6.56
invextr1_new	110,355	556,938	7.79	8.00	10.32
ASIC_680ks	113,076	3,805,068	2.69	3.79	6.21
Hamrle3	682,712	2,329,176	2.76	4.14	6.72
pre2	643,994	6,175,377	3.66	4.63	7.07
cage13	1,447,360	5,514,242	3.88	4.29	6.12
rajat30	1,505,785	27,130,349	3.33	4.38	6.25
cage14	4,690,002	20,316,253	3.78	4.22	6.11
cage15	5,154,859	99,199,551	3.68	4.12	5.74
Theoretical			25	25	33

We wrote a straightforward (not manually unrolled) code for the computation of $Ax-b$ using the algorithms based on `PrecSum` (called `PrecMV`). The time for the ordinary algorithm based on floating-point dot products (called `DDotMV`) is normed to 1. Moreover, we use our previous algorithm `Dot2` taken from [26] (called `Dot2MV`). The latter delivers a result *as if* calculated in quasi-quadruple precision.

As a sparse matrix A, various test matrices can be obtained from University of Florida Sparse Matrix Collection [5], among which we choose pairs of matrices of similar dimension but with significantly different number of elements. The dimensions vary from $\approx 10^4$ to about $5 \cdot 10^6$, and n-vectors x and b are randomly generated. We compare to `DDotMV`, the time of which is normed to 1.

The results are displayed in Tables 6.6 and 6.7. Without parallelization (Table 6.6), `PrecMV` with $L = 1, 2$ achieves a remarkable factor of about 4, 6 or 10 compared to the standard routine `DDotMV`. With

TABLE 6.7
Ratio of computing time (8 cores), elapsed time for `DDotMV` *is normed to 1.*

name	n	`nnz`(A)	`Dot2MV`	`PrecMV` $(L=1)$	`PrecMV` $(L=2)$
cryg10000	10,000	49,699	1.95	2.32	2.95
rajat26	10,605	424,587	1.59	2.05	2.84
rajat23	51,032	249,302	1.15	1.44	2.07
venkat01	62,424	1,717,792	1.10	1.12	1.40
invextr1_new	110,355	556,938	2.50	2.66	3.28
ASIC_680ks	113,076	3,805,068	1.06	1.18	1.65
Hamrle3	682,712	2,329,176	1.14	1.18	1.55
pre2	643,994	6,175,377	1.04	1.15	1.56
cage13	1,447,360	5,514,242	1.16	1.22	1.53
rajat30	1,505,785	27,130,349	1.61	2.06	2.42
cage14	4,690,002	20,316,253	1.17	1.17	1.50
cage15	5,154,859	99,199,551	1.19	1.21	1.47
Theoretical			25	25	33

TABLE 6.8
Average of speed-up ratios and parallel efficiencies in terms of execution time (1 core vs. 8 cores), and average of MFlops.

	`DDotMV`	`Dot2MV`	`PrecMV` $(L=1)$	`PrecMV` $(L=2)$
Speed-up ratio	1.63	4.60	5.16	5.95
Parallel efficiencies	20%	58%	65%	74%
MFlops (1 core)	277	909	738	659
MFlops (8 cores)	449	4,049	3,856	3,937

parallelization (Table 6.7), the factor even drops to a value of about 1 to 3. Note that all the routines including `DDotMV` can be very easily parallelized by OpenMP directives. We also see that `PrecMV` with $L = 1$ is slightly slower than `Dot2MV`, and `PrecMV` with $L = 2$ is about 50% slower than `Dot2MV` although of much better quality.

The high performance of `Dot2MV` and `PrecMV`, in particular with parallelization, becomes transparent when looking at the parallel efficiency. In Table 6.8, average of speed-up ratios and parallel efficiencies in terms of execution time are displayed. It can be seen that the parallel efficiencies of `Dot2MV` and `PrecMV` are much better than that of `DDotMV`.

The inefficiency of `DDotMV` is due to the following overheads concerning memory access to fetch data of a_{ij} and x_j:

- intermittent memory access according to random indices of sparse matrices,
- limited memory bandwidth of the architecture in use, especially for parallel computations on the shared memory machine.

On the other hand, in `Dot2MV` and `PrecMV`, both these drawbacks are diminished because after fetching a_{ij} and x_j more floating-point operations are performed on this data. These factors are reflected in Mflops-rate displayed in Table 6.8. As a result, the ratio of measured computing time for `Dot2MV` and `PrecMV` to `DDotMV` becomes much less than the theoretical one.

7. Appendix. We have to prove that the result `res` of Algorithm 4.1 is a faithful rounding of $s = \sum p_i^{(0)}$ under the assumptions

$$|\pi_1| \geq 2^{2M}\texttt{eps}\sigma_0, \quad \sigma_0 > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta} \quad \text{and} \quad L > 1. \tag{7.1}$$

We first prove the following lemma.

LEMMA 7.1. *Let $p \in \mathbb{F}^n$ be a vector of floating-point numbers. Assume the following code is applied to p:*

$$\begin{array}{l} \pi_1 = p_1;\ e_1 = 0 \\ \textit{for } i = 2 : n \\ \quad [\pi_i, q_i] = \texttt{TwoSum}(\pi_{i-1}, p_i) \\ \quad e_i = \mathrm{fl}(e_{i-1} + q_i) \end{array} \tag{7.2}$$

Abbreviate $s := \sum_{i=1}^n p_i$ and $S := \sum_{i=1}^n |p_i|$. Then the following is true.

$$\pi_n = \mathrm{fl}\left(\sum_{i=1}^n p_i\right) ,$$

$$e_n = \mathrm{fl}\left(\sum_{i=2}^n q_i\right) \quad \textit{and} \quad |e_n| \leq (1+\gamma_{n-1})\gamma_{n-1}S ,$$

$$\left| e_n - \sum_{i=2}^n q_i \right| \leq \gamma_{n-1}^2 S ,$$

$$s = \sum_{i=1}^n p_i = \pi_n + \sum_{i=2}^n q_i \quad \textit{and} \quad |\pi_n + e_n - s| \leq \gamma_{n-1}^2 S.$$

PROOF. By Lemma 4.2 in [27] we have

$$\sum_{i=2}^n |q_i| \leq \gamma_{n-1} S,$$

hence using

$$|e_n| \leq (1+\gamma_{n-1}) \cdot \sum_{i=2}^n |q_i| \quad \text{and} \quad |\pi_n + e_n - s| = |e_n - \sum_{i=2}^n q_i|$$

and standard error estimations the results follow. □

Now we prove the result `res` of Algorithm 4.1 to be a faithful rounding of $s := \sum_{i=1}^n p_i$ under the assumption (7.1). The code

$$\begin{array}{l} \pi_1 = \tau^{(1)};\ e_1 = 0 \\ \text{for } k = 2 : L \\ \quad [\pi_k, q_k] = \texttt{FastTwoSum}(\pi_{k-1}, \tau^{(k)}) \\ \quad e_k = \mathrm{fl}(e_{k-1} + q_k) \end{array}$$

in lines 15 to 19 to compute π_L and e_L is identical to (7.2) except that `FastTwoSum` is used. However, (4.10) shows that `FastTwoSum` and `TwoSum` produce identical results, so that the assertions of Lemma 7.1 are true. Abbreviate $\varphi := 2^M\texttt{eps}$. Then $\sigma_{k+1} = \varphi\sigma_k = \varphi^k\sigma_0$ for $0 \leq k < L$, and using Lemma 7.1 and (4.8) we obtain

$$\pi_L = \mathrm{fl}\left(\sum_{k=1}^L \tau^{(k)}\right) , \tag{7.3}$$

$$\sum_{k=1}^{L} |\tau^{(k)}| < |\tau^{(1)}| + \sum_{k=2}^{L} \sigma_{k-1} < |\tau^{(1)}| + \frac{\varphi}{1-\varphi}\sigma_0 =: S_\tau \ , \tag{7.4}$$

$$|e_L| \le (1+\gamma_{L-1})\gamma_{L-1} S_\tau \ , \tag{7.5}$$

$$\left| \pi_L + e_L - \sum_{k=1}^{L} \tau^{(k)} \right| = \left| e_L - \sum_{k=2}^{L} q_k \right| \le \gamma_{L-1}^2 S_\tau \ . \tag{7.6}$$

Furthermore,

$$\pi_L = \mathrm{fl}\left(\sum_{k=1}^{L} \tau^{(k)} \right) = \sum_{k=1}^{L} (1+\Theta_{L-1}^{(k)}) \cdot \tau^{(k)} \quad \text{with} \ \ |\Theta_{L-1}^{(k)}| \le \gamma_{L-1} \ ,$$

so that (7.4) implies

$$\begin{array}{rcl} |\pi_L| & \ge & (1-\gamma_{L-1})|\tau^{(1)}| - (1+\gamma_{L-1}) \sum_{k=2}^{L} |\tau^{(k)}| \\ & \ge & (1-\gamma_{L-1})|\tau^{(1)}| - (1+\gamma_{L-1}) \dfrac{\varphi}{1-\varphi}\sigma_0 \ . \end{array} \tag{7.7}$$

Setting $2^{-m} := \mathtt{eps}$, the definition of L in line 6 of Algorithm 4.1, $M \le m/2$, $m \ge 10$ and $K \le (4\sqrt{\mathtt{eps}})^{-1}$ imply

$$L = \left\lceil \frac{mK+2}{m-M} \right\rceil - 1 \le \left\lceil 2K + \frac{4}{m} \right\rceil - 1 \le 2K \le \frac{1}{2\sqrt{\mathtt{eps}}} \ .$$

This yields

$$\begin{array}{c} \gamma_{L-1} = \dfrac{(L-1)\mathtt{eps}}{1-(L-1)\mathtt{eps}} \le \dfrac{\frac{1}{2}\sqrt{\mathtt{eps}} - \mathtt{eps}}{1-\frac{1}{2}\sqrt{\mathtt{eps}} + \mathtt{eps}} \le \dfrac{1}{2}\sqrt{\mathtt{eps}} \dfrac{1-2\sqrt{\mathtt{eps}}}{1-\frac{1}{2}\sqrt{\mathtt{eps}} + \mathtt{eps}} \le \dfrac{1}{2}\sqrt{\mathtt{eps}} \ , \\ (1+\gamma_{L-1})\gamma_{L-1} \le \frac{1}{4}(2+\sqrt{\mathtt{eps}})\sqrt{\mathtt{eps}} \ . \end{array} \tag{7.8}$$

Abbreviate

$$\begin{array}{rcl} e_L & = & \mathrm{fl}\left(\sum_{k=2}^{L} q_k \right) = \sum_{k=2}^{L} q_k - \delta''' \ , \\ \tau'' & = & \mathrm{fl}\left(\sum_{i=1}^{n} p_i^{(L)} \right) = \sum_{i=1}^{n} p_i^{(L)} - \delta'' \ , \\ \tau' & = & \mathrm{fl}(e_L + \tau'') = e_L + \tau'' - \delta' \ . \end{array} \tag{7.9}$$

Then (7.6), (7.5) and (7.8) yield

$$\begin{array}{rcl} |\delta'''| & \le & \frac{1}{4}\mathtt{eps} S_\tau \quad \text{and} \\ |e_L| & \le & \frac{1}{4}(2+\sqrt{\mathtt{eps}})\sqrt{\mathtt{eps}} S_\tau =: f_1 \cdot S_\tau \ . \end{array} \tag{7.10}$$

By assumption $L > 1$, so that (4.8) gives

$$\max_i |p_i^{(L)}| \le \mathtt{eps}\sigma_{L-1} \le \varphi \mathtt{eps} \sigma_0,$$

and (2.5) and (7.9) yield

$$\begin{array}{rcl} |\tau''| & \le & n\varphi \ \mathtt{eps} \ \sigma_0 < \varphi^2 \sigma_0 \quad \text{and} \\ |\delta''| & \le & \frac{1}{2} n(n-1)\mathtt{eps} \cdot \varphi \ \mathtt{eps} \ \sigma_0 < \frac{1}{2}\varphi^3 \sigma_0 \ . \end{array} \tag{7.11}$$

Combining this with (7.10) shows

$$\begin{aligned} |e_L + \tau''| &< f_1 \cdot S_\tau + \varphi^2 \sigma_0 \quad \text{and} \\ |\delta'| &\le \texttt{eps} |e_L + \tau''| \,. \end{aligned} \tag{7.12}$$

We will use Lemma 2.3 to prove that `res` is a faithful rounding of $s = \sum p_i$. For that we use (4.11) and

$$\begin{aligned} s &= \pi_L + \textstyle\sum_{k=2}^{L} q_k + \sum_{i=1}^{n} p_i^{(L)} \\ &= \pi_L + e_L + \delta''' + \textstyle\sum_{i=1}^{n} p_i^{(L)} = \pi_L + e_L + \tau'' + \delta'' + \delta''' \\ &= \pi_L + \tau' + \delta' + \delta'' + \delta''', \end{aligned}$$

so that

$$s = r + \delta \quad \text{with} \quad r := \pi_L + \tau',\ \delta := \delta' + \delta'' + \delta''' \quad \text{and} \quad \texttt{res} = \mathrm{fl}(r). \tag{7.13}$$

A lower bound for `res` is given by (7.7), $|\tau'| \le (1 + \texttt{eps})|e_L + \tau''|$ and

$$\begin{aligned} |\texttt{res}| &\ge (1 - \texttt{eps})(|\pi_L| - |\tau'|) \ge (1 - \texttt{eps})(1 - \gamma_{L-1})|\tau^{(1)}| - (1 + \gamma_{L-1})\frac{\varphi}{1-\varphi}\sigma_0 - |e_L + \tau''| \\ &:= f_2 \cdot |\tau^{(1)}| - f_3 \cdot \sigma_0 - |e_L + \tau''| \,. \end{aligned} \tag{7.14}$$

If $|\pi_1| < (1 - 2^{-M-1})\texttt{eps}^{-1}\texttt{eta}$, then we claim that all floating-point operations in the computation of `res` are exact so that `res` $= s$. To prove that we first use (4.8) and (7.1) to see

$$\sum_{i=1}^{n} |p_i^{(1)}| \le n\texttt{eps}\sigma_0 < 2^M \texttt{eps}\sigma_0 \le 2^{-M}|\pi_1| < 2^{-M}\texttt{eps}^{-1}\texttt{eta}.$$

Because $2^M\texttt{eps}\sigma_0$ and $\texttt{eps}^{-1}\texttt{eta}$ are powers of 2, it follows

$$\sum_{i=1}^{n} |p_i^{(1)}| \le 2^{-M-1}\texttt{eps}^{-1}\texttt{eta},$$

and (4.6) and $\tau^{(1)} = \pi_1$ give

$$\sum_{i=1}^{n} |p_i^{(0)}| \le |\tau^{(0)}| + \sum_{i=1}^{n} |p_i^{(1)}| < \texttt{eps}^{-1}\texttt{eta}.$$

Therefore $\mathrm{ufp}(p_i^{(0)}) \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ for $1 \le i \le n$ and all operations on $\tau^{(k)}$ and $p_i^{(k)}$ are exact and `res` $= s$.

Assume $|\pi_1| \ge (1 - 2^{-m-1})\texttt{eps}^{-1}\texttt{eta}$. Then (7.14), (7.8) and $\gamma_{L-1} \le \frac{1}{64}$, (7.12), $f_1 \le \frac{64}{4096}$ and $\pi_1 = \tau^{(1)}$ yield

$$\begin{aligned} |\texttt{res}| &\ge (1 - \texttt{eps})\frac{63}{64}|\tau^{(1)}| - \frac{65}{64}\frac{\varphi}{1-\varphi}\sigma_0 - \frac{65}{4096}(|\tau^{(1)}| + \frac{\varphi}{1-\varphi}\sigma_0) - \varphi^2\sigma_0 \\ &\ge \left[\frac{59}{61}(1 - 2^{-M-1}) - \frac{33}{32}\frac{\varphi}{1-\varphi} - \varphi^2\right]\texttt{eps}^{-1}\texttt{eta} =: \Psi \cdot \texttt{eps}^{-1}\texttt{eta}. \end{aligned}$$

For $M = 2$ we have $\varphi \le 2^M\texttt{eps} \le \frac{1}{256}$ and $\frac{\varphi}{1-\varphi} \le \frac{1}{255}$, so that $\Psi > \frac{1}{2}$. And for $M \ge 3$ we have $\varphi \le 2^{-M} \le \frac{1}{8}$, $\frac{\varphi}{1-\varphi} \le \frac{1}{7}$ and $2^{-M-1} \le \frac{1}{16}$, so that again $\Psi > \frac{1}{2}$. Hence $\texttt{res} \notin \mathbb{U}$ and by Lemma 2.3 it remains to show $2|\delta| < \texttt{eps}|\texttt{res}|$ to prove `res` to be a faithful rounding of s.

We obtain by (7.12), (7.11), (7.10), (7.14), $\frac{\varphi}{1-\varphi} \le (1+2^{M+1}\mathtt{eps})\varphi$ and $2^M\sqrt{\mathtt{eps}} \le 1$,

$$\begin{aligned}
2\mathtt{eps}^{-1}|\delta| - |\mathtt{res}| &\le 2|e_L+\tau''| + 2^M\varphi^2\sigma_0 + \tfrac{1}{2}S_\tau - f_2|\tau^{(1)}| + f_3\sigma_0 + |e_L+\tau''| \\
&\le (3f_1+\tfrac{1}{2})S_\tau + (3\varphi^2 + 2^M\varphi^2 + f_3)\sigma_0 - f_2|\tau^{(1)}| \\
&\le (3f_1+\tfrac{1}{2}-f_2)|\tau^{(1)}| + ((3f_1+\tfrac{1}{2})\frac{\varphi}{1-\varphi} + 3\varphi^2 + 2^M\varphi^2 + f_3)\sigma_0 \\
&\le (-\tfrac{1}{2} + \tfrac{7}{4}\mathtt{eps} + 2\sqrt{\mathtt{eps}})|\tau^{(1)}| + ((3f_1+\tfrac{1}{2}+1+\tfrac{1}{2}\sqrt{\mathtt{eps}})\frac{\varphi}{1-\varphi} + 3\varphi^2 + 2^M\varphi^2)\sigma_0 \\
&\le (-\tfrac{1}{2} + \tfrac{7}{4}\mathtt{eps} + 2\sqrt{\mathtt{eps}})2^{2M}\sigma_0 + \big[(\tfrac{3}{2} + 3\sqrt{\mathtt{eps}})(1+2^{M+1}\mathtt{eps}) + 3\varphi + 2^M\varphi\big]\varphi\sigma_0 \\
&= \big[-2^{M-1} + \tfrac{3}{2} + (2^{M+3}+2^{2M})\mathtt{eps} + (2^{M+1}+3+3\cdot 2^{M+1}\mathtt{eps})\sqrt{\mathtt{eps}}\big]\varphi\sigma_0 \\
&\le \big[-2^{M-1} + \tfrac{3}{2} + 2^{2M}\mathtt{eps}(2^{-M+3}+1) + 2^M\sqrt{\mathtt{eps}}(2+2^{-M+2})\big]\varphi\sigma_0 \\
&=: \Delta\,.
\end{aligned}$$

By Lemma 2.3 we have to show $\Delta < 0$. This follows directly for $M = 2$ and $M = 3$, and for $M \ge 4$ we have

$$\begin{aligned}
2\mathtt{eps}^{-1}|\delta| - |\mathtt{res}| &\le \varphi\sigma_0(-2^{M-1} + \tfrac{3}{2} + 2^{-M+3} + 1 + 2 + 2^{-M+2}) \\
&\le \varphi\sigma_0(-2^{M-1} + 6) \\
&< 0
\end{aligned}$$

This proves $|\delta| < \frac{1}{2}\mathtt{eps}|\mathtt{res}|$, so that $\mathtt{res}$ is a faithful rounding of s by Lemma 2.3. The proof is finished. □

REFERENCES

[1] E. Anderson, *Robust Triangular Solvers for Use in Condition Estimation*, Cray Research, 1991.

[2] G. Bohlender, *Floating-Point Computation of Functions with Maximum Accuracy*, IEEE Trans. Comput., C-26 (1977), pp. 621–632.

[3] K. Clarkson, *Safe and Effective Determinant Evaluation*, in 33th Annual Symposium on Foundations of Computer Science (Pittsburgh, PA), IEEE Computer Society Press, 1992, pp. 387–395.

[4] M. Daumas and D. Matula, *Validated Roundings of Dot Products by Sticky Accumulation*, IEEE Trans. Comput., 46 (1997), pp. 623–629.

[5] T. Davis, *The University of Florida Sparse Matrix Collection*, 2007. submitted to ACM Trans. on Mathematical Software `http://www.cise.ufl.edu/research/sparse/matrices`.

[6] T. Dekker, *A Floating-Point Technique for Extending the Available Precision*, Numerische Mathematik, 18 (1971), pp. 224–242.

[7] J. Demmel and Y. Hida, *Accurate and efficient floating point summation*, SIAM J. Sci. Comput. (SISC), 25 (2003), pp. 1214–1248.

[8] ———, *Fast and accurate floating point summation with application to computational geometry*, Numerical Algorithms, 37 (2004), pp. 101–112.

[9] N. Higham, *The accuracy of floating point summation*, SIAM J. Sci. Comput. (SISC), 14 (1993), pp. 783–799.

[10] ———, *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadelphia, 2nd ed., 2002.

[11] *ANSI/IEEE 754-1985: IEEE Standard for Binary Floating-Point Arithmetic*, New York, 1985.

[12] M. Jankowski, A. Smoktunowicz, and H. Woźniakowski, *A note on floating-point summation of very many terms*, J. Information Processing and Cybernetics-EIK, 19 (1983), pp. 435–440.

[13] M. Jankowski and H. Woźniakowski, *The accurate solution of certain continuous problems using only single precision arithmetic*, BIT Numerical Mathematics, 25 (1985), pp. 635–651.

[14] W. Kahan, *A survey of error analysis*, in Proc. IFIP Congress, Ljubljana, vol. 71 of Information Processing, North-Holland, Amsterdam, The Netherlands, 1972, pp. 1214–1239.

[15] ———, *Implementation of Algorithms (lecture notes by W.S. Haugeland and D. Hough)*, Tech. Report 20, Department of Computer Science, University of California, Berkeley, CA, USA, 1973.

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

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

[18] U. Kulisch and W. Miranker, *Arithmetic Operations in Interval Spaces*, Computing, Suppl., 2 (1980), pp. 51–67.

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

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

[21] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, and D. Yoo, *Design, Implementation and Testing of Extended and Mixed Precision BLAS*, ACM Trans. Math. Software, 28 (2002), pp. 152–205.

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

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

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

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

[26] T. Ogita, S. Rump, and S. Oishi, *Accurate Sum and Dot Product*, SIAM Journal on Scientific Computing (SISC), 26 (2005), pp. 1955–1988.

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

[28] D. Priest, *Algorithms for Arbitrary Precision Floating Point Arithmetic*, in Proceedings of the 10th Symposium on Computer Arithmetic, P. Kornerup and D. Matula, eds., Grenoble, France, 1991, IEEE Computer Society Press, pp. 132–145.

[29] ———, *On properties of floating point arithmetics: Numerical stability and the cost of accurate computations*, PhD thesis, Mathematics Department, University of California at Berkeley, CA, USA, 1992. `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

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

[31] S. Rump, T. Ogita, and S. Oishi, *Accurate Floating-point Summation I: Faithful Rounding.* submitted for publication in SISC, 2005–2007.

[32] ———, *Accurate Floating-point Summation II: Sign, K-fold Faithful and Rounding to Nearest.* submitted for publication in SISC, 2005–2007.

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

[34] J. Wolfe, *Reducing Truncation Errors by Programming*, Comm. ACM, 7 (1964), pp. 355–356.

[35] *XBLAS: A Reference Implementation for Extended and Mixed Precision BLAS.* `http://crd.lbl.gov/~xiaoye/XBLAS/`.

[36] Y. Zhu and W. Hayes, *Fast, Guaranteed-Accurate Sums of many Floating-Point Numbers*, in Proc. of the RNC7 Conference on Real Numbers and Computers, G. Hanrot and P. Zimmermann, eds., 2006, pp. 11–22.

[37] Y. Zhu, J. Yong, and G. Zheng, *A New Distillation Algorithm for Floating-Point Summation*, SIAM J. Sci. Comput. (SISC), 26 (2005), pp. 2066–2078.

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