submitted for publication in SISC, November 13, 2005, revised April 13, 2007 and April 15, 2008, accepted August 21, 2008; appeared SIAM J. Sci. Comput. Vol. 31, Issue 2, pp. 1269-1302 (2008)

ACCURATE FLOATING-POINT SUMMATION PART II: SIGN, K-FOLD FAITHFUL AND ROUNDING TO NEAREST [*]

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

Abstract. In this Part II of this paper we first refine the analysis of error-free vector transformations presented in Part I. Based on that we present an algorithm for calculating the rounded-to-nearest result of $s := \sum p_i$ for a given vector of floating-point numbers p_i, as well as algorithms for directed rounding. A special algorithm for computing the sign of s is given, also working for huge dimensions. Assume a floating-point working precision with relative rounding error unit $\texttt{eps}$. We define and investigate a K-fold faithful rounding of a real number r. Basically the result is stored in a vector $\texttt{Res}_\nu$ of K non-overlapping floating-point numbers such that $\sum \texttt{Res}_\nu$ approximates r with relative accuracy $\texttt{eps}^K$, and replacing $\texttt{Res}_K$ by its floating-point neighbors in $\sum \texttt{Res}_\nu$ forms a lower and upper bound for r. For a given vector of floating-point numbers with exact sum s, we present an algorithm for calculating a K-fold faithful rounding of s using solely the working precision. Furthermore, an algorithm for calculating a faithfully rounded result of the sum of a vector of huge dimension is presented. Our algorithms are fast in terms of measured computing time because they allow good instruction-level parallelism, they neither require special operations such as access to mantissa or exponent, they contain no branch in the inner loop, nor do they 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 algorithms are proved to be optimal.

Key words. maximally accurate summation, faithful rounding, rounding to nearest, directed rounding, K-fold accuracy, sign, error-free transformations, distillation, high accuracy, XBLAS, error analysis

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

1. Introduction, notation and basic facts.

We will present fast algorithms to compute approximations of high quality of the sum of a vector p_i of floating-point numbers. Since sums of floating-point numbers are ubiquitous in scientific computations, there is a vast amount of literature to that; excellent surveys can be found in [9, 14].

In Part I [22] of this paper we gave a fast algorithm to compute a faithfully rounded result of $\sum p_i$. Our methods are based on *error-free transformations*. For example, Knuth [11] gave an algorithm (cf. Part I, Algorithm 2.1) transforming the sum $a + b$ of two floating-point numbers into a sum $x + y$, where x is the usual floating-point approximation of the sum and y comprises of the exact error. Surprisingly, x and y can be calculated using only 6 ordinary floating-point operations. Such error-free transformations receive interest in many areas [1, 8, 13, 14, 17, 18, 19, 20, 24, 25, 26, 27].

More background, an overview of existing methods and more details are given in Part I [22] of this paper.

This Part II of our paper extends the results of Part I [22] in various ways, and it is organized as follows. For the often delicate estimations we developed a framework for the analysis in Part I. Those techniques and main results are summarized in Section 2. Next we redefine the error-free vector transformation in Algorithm

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

3.3 (`Transform`) and refine the analysis of Part I. This is the key to all algorithms in Part II. We especially allow for huge vector lengths up to about $\texttt{eps}^{-1}$, and give as a first example in Section 4 an algorithm to compute the sign of $\sum p_i$. We show that the constant in the stopping criterion is best possible.

In Part I we developed an algorithm to compute a faithful rounding `res` of the sum s of a vector of floating-point numbers. This means that there is no floating-point number between `res` and s. Especially, if s is itself a floating-point number or is in the underflow range, then $\texttt{res} = s$. In the following Section 5 of this paper we define and investigate K-fold faithful rounding. Suppose a floating-point working precision with relative rounding error unit `eps` to be given. Then K-fold faithful rounding of $s := \sum p_i$ means basically that a vector $\texttt{Res}_\nu$ of K floating-point numbers is computed such that $\sum \texttt{Res}_\nu$ is an approximation of relative accuracy $\texttt{eps}^K$, and $\texttt{Res}_K$ is a faithful rounding of $s - \sum_{\nu=1}^{K-1} \texttt{Res}_\nu$. This implies that replacing $\texttt{Res}_K$ by its two floating-point neighbors in $\sum \texttt{Res}_\nu$ produces a lower and upper bound for s. After developing the theoretical background in Section 5, we present a fast algorithm for computing a K-fold faithful rounding $\texttt{Res}_\nu$ of $\sum p_i$ in Section 6. Moreover we show that the sequence $\texttt{Res}_\nu$ is non-overlapping.

In the following Section 7 we develop a rounding-to-nearest algorithm for $s = \sum p_i$. This is the ultimate accuracy of an approximation of s by a single floating-point number; however, it necessarily comes with a burden: To compute a faithful rounding of s, it suffices to know s up to some error margin, whereas the rounded-to-nearest result may ultimately require to know s exactly, namely if s is the midpoint of two adjacent floating-point numbers.

Our Algorithm 4.5 (`AccSum`) presented in Part I for computing a faithfully rounded result of $\sum p_i$ has the charming property that the computing time is proportional to the logarithm of the condition number: The more difficult the problem is, the more computing time is needed. This is also true for our rounding-to-nearest algorithm, however, the "difficulty" depends on the nearness of the exact result $\sum p_i$ to the midpoint of two adjacent floating-point numbers. In Section 7 we also give algorithms for computing $\sum p_i$ with directed rounding.

For our Algorithm 4.5 (`AccSum`) presented in Part I the vector length was limited to about $\sqrt{\texttt{eps}^{-1}}$. In Section 8 we extend the range of applicability to vector lengths near $\texttt{eps}^{-1}$. We conclude the paper with computational results on a Pentium 4, Itanium 2 and Athlon 64 processor. For all algorithms presented in Part I and II of this paper and in [18] we put a Matlab reference code on `http://www.ti3.tu-harburg.de/rump` .

As in [18] and [22], all theorems, error analysis and proofs are due to the first author of the present paper.

2. Notation and basic facts.

We use the notation and a number of results of Part I [22] of this paper. For convenience, some of the main results are summarized in the following, for more details, cf. [22].

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

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

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 summation may be performed in any order. We assume floating-point operations in rounding to nearest corresponding to the IEEE 754 arithmetic standard [10].

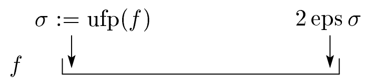

FIG. 2.1. *The unit in the first place and unit in the last place of a normalized floating-point number*

In Part I 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 $2\mathtt{eps} \cdot \mathrm{ufp}(f)$. 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 $\mathtt{eps}\sigma\mathbb{Z}$, which can be interpreted as a set of fixed point numbers with smallest positive number $\mathtt{eps}\sigma$. Of course, $\mathbb{F} \subseteq \mathtt{eta}\mathbb{Z}$. These two concepts, the unit in the first place $\mathrm{ufp}(\cdot)$ together with $f \in \mathbb{F} \Rightarrow f \in 2\mathtt{eps} \cdot \mathrm{ufp}(f)\mathbb{Z}$ 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

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

$$\sigma' = 2^m,\ m \in \mathbb{Z} \text{ and } \sigma' \ge \sigma \quad \Rightarrow \quad \mathtt{eps}\sigma'\mathbb{Z} \subseteq \mathtt{eps}\sigma\mathbb{Z} \tag{2.4}$$

$$r \in \mathtt{eps}\sigma\mathbb{Z},\ |r| \le \sigma \quad \text{and} \quad \mathtt{eps}\sigma \ge \mathtt{eta} \quad \Rightarrow \quad r \in \mathbb{F} \tag{2.5}$$

$$a, b \in \mathbb{F} \cap \mathtt{eps}\sigma\mathbb{Z} \quad \text{and} \quad \delta := \mathrm{fl}(a+b) - (a+b) \quad \Rightarrow \quad \mathrm{fl}(a+b),\ a+b,\ \delta \in \mathtt{eps}\sigma\mathbb{Z} \tag{2.6}$$

$$n\mathtt{eps} \le 1,\ a_i \in \mathbb{F} \quad \text{and} \quad |a_i| \le \sigma \quad \Rightarrow \quad \begin{array}{l} |\mathrm{fl}(\sum_{i=1}^n a_i)| \le n\sigma \quad \text{and} \\ |\mathrm{fl}(\sum_{i=1}^n a_i) - (\sum_{i=1}^n a_i)| \le \frac{n(n-1)}{2}\mathtt{eps}\sigma \end{array} \tag{2.7}$$

$$a, b \in \mathbb{F},\ a \neq 0 \quad \Rightarrow \quad \mathrm{fl}(a+b) \in \mathtt{eps} \cdot \mathrm{ufp}(a)\mathbb{Z} \,, \tag{2.8}$$

see (2.9) through (2.20) in Part I of this paper. The fundamental error bound for floating-point addition is

$$f = \mathrm{fl}(a+b) \quad \Rightarrow \quad f = a + b + \delta \ \text{ with } |\delta| \le \mathtt{eps} \cdot \mathrm{ufp}(a+b) \le \mathtt{eps} \cdot \mathrm{ufp}(f) \le \mathtt{eps}|f| \,, \tag{2.9}$$

cf. (2.19) in part I. 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. In many estimations we desperately need this factor. Also note that (2.9) is also true in the underflow range, in fact addition (and subtraction) is exact if $\mathrm{fl}(a \pm b) \in \mathbb{U}$.

Next we note a useful sufficient criterion [cf. (2.21) in Part I] to decide that no error occurred in the sum of two floating-point numbers. For $a, b \in \mathbb{F}$ and $\sigma = 2^k$, $k \in \mathbb{Z}$,

$$\begin{array}{llll} a, b \in \mathtt{eps}\sigma\mathbb{Z} \quad \text{and} \quad \mathrm{fl}(|a+b|) < \sigma & \Rightarrow & \mathrm{fl}(a+b) = a+b & \text{and} \\ a, b \in \mathtt{eps}\sigma\mathbb{Z} \quad \text{and} \quad |a+b| \le \sigma & \Rightarrow & \mathrm{fl}(a+b) = a+b \,. & \end{array} \tag{2.10}$$

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

$$f \notin \mathbb{U} \quad \text{and} \quad |f| \neq \mathrm{ufp}(f) \quad \Rightarrow \quad \mathrm{pred}(f) = f - 2\mathtt{eps} \cdot \mathrm{ufp}(f) \quad \text{and} \quad f + 2\mathtt{eps} \cdot \mathrm{ufp}(f) = \mathrm{succ}(f) \,,$$

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

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

$$f - 2\mathtt{eps} \cdot \mathrm{ufp}(f) \le \mathrm{pred}(f) < \mathrm{succ}(f) \le f + 2\mathtt{eps} \cdot \mathrm{ufp}(f)\ . \tag{2.11}$$

A main concept of both parts of this paper is *faithful rounding* [5, 20, 4]. A floating-point number f is called a faithful rounding of a real number r if there is no other floating-point number between f and r. It follows that $f = r$ in case $r \in \mathbb{F}$.

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*

$$\mathrm{pred}(f) < r < \mathrm{succ}(f)\ . \tag{2.12}$$

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

The following sufficient criterion was given in Lemma 2.4 in Part I. Note that for the computation of a faithful rounding of a real number r we only need to know r up to a certain error margin. The rounded-to-nearest $\mathrm{fl}(r)$ computed by Algorithm 7.4, however, ultimately requires to know r precisely, namely if r is the midpoint between two adjacent floating-point numbers.

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 of both parts of this paper is the error-free transformation of the sum of floating-point numbers. For the sum of two floating-point numbers Knuth [11] gave an algorithm transforming $a+b = x+y$ for general $a, b \in \mathbb{F}$ with $x = \mathrm{fl}(a+b)$. It requires 6 floating-point operations. The following faster version by Dekker [5] applies if a, b are somehow sorted.

ALGORITHM 2.4. *Compensated summation of two floating-point numbers.*

```
function $[x, y] = \mathtt{FastTwoSum}(a, b)$
    $x = \mathrm{fl}(a + b)$
    $q = \mathrm{fl}(x - a)$
    $y = \mathrm{fl}(b - q)$
```

In Part I, Lemma 2.6 we analyzed the algorithm as follows.

LEMMA 2.5. *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 2.4 (`FastTwoSum`) applied to a, b. Then*

$$x + y = a + b\ , \quad x = \mathrm{fl}(a+b) \quad \text{and} \quad |y| \le \mathtt{eps} \cdot \mathrm{ufp}(a+b) \le \mathtt{eps} \cdot \mathrm{ufp}(x)\ . \tag{2.13}$$

Furthermore,

$$q = \mathrm{fl}(x - a) = x - a \quad \text{and} \quad y = \mathrm{fl}(b - q) = b - q\ , \tag{2.14}$$

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

3. Error-free transformations.

The main principle of the error-free vector transformation in Part I is the extraction of a vector into a sum of higher order parts and a vector of lower order parts. The splitting is chosen so that the higher order parts add without error.

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

```
function $[\tau, p'] = \texttt{ExtractVector}(\sigma, p)$
    $\tau = 0$
    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
```

The following was proved in Part I, Theorem 3.4.

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 $M \in \mathbb{N}$ and $|p_i| \le 2^{-M}\sigma$ for all i. Then*

$$\sum_{i=1}^{n} p_i = \tau + \sum_{i=1}^{n} p'_i \ , \quad \max |p'_i| \le \texttt{eps}\sigma \ , \quad |\tau| \le (1 - 2^{-M})\sigma < \sigma \quad \text{and} \quad \tau \in \texttt{eps}\sigma\mathbb{Z} \ . \tag{3.1}$$

Based on that we derived in Part I a transformation of a vector p_i into an approximation of its sum and a remainder part, namely $\sum p_i = \tau_1 + \tau_2 + \sum p'_i$, an error-free transformation. Now we refine this algorithm and its analysis by introducing an offset, by a parameterized stopping criterion and by allowing for huge vector lengths.

ALGORITHM 3.3. *Transformation of a vector $p^{(0)}$ plus offset ϱ (depending on parameter Φ).*

```
function $[\tau_1, \tau_2, p^{(m)}, \sigma, M] = \texttt{Transform}(p^{(0)}, \varrho)$
    $\mu = \max(|p_i^{(0)}|)$
    if $\mu = 0$, $\tau_1 = \varrho$, $\tau_2 = p^{(m)} = \sigma = 0$, return, end if
    $M = \lceil \log_2 \left(\mathrm{length}(p^{(0)}) + 2\right) \rceil$
    $\sigma_0 = 2^{M + \lceil \log_2(\mu) \rceil}$
    $t^{(0)} = \varrho$, $m = 0$
    repeat
        $m = m + 1$
        $[\tau^{(m)}, p^{(m)}] = \texttt{ExtractVector}(\sigma_{m-1}, p^{(m-1)})$
        $t^{(m)} = \mathrm{fl}(t^{(m-1)} + \tau^{(m)})$
        $\sigma_m = \mathrm{fl}(2^M \texttt{eps} \sigma_{m-1})$
    until $|t^{(m)}| \ge \mathrm{fl}(\Phi \sigma_{m-1})$ or $\sigma_{m-1} \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$    % Parameter $\Phi$ to be specified
    $\sigma = \sigma_{m-1}$
    $[\tau_1, \tau_2] = \texttt{FastTwoSum}(t^{(m-1)}, \tau^{(m)})$
```

Note that this is a preliminary version of Transform with all intermediate results uniquely identified using indices to ease readability and analysis. The original Algorithm 4.1 (Transform) in Part I was rewritten into the final version Algorithm 4.4 by omitting indices, overwriting results and an additional check for zero. Completely similarly Algorithm 3.3 is to be rewritten in an actual implementation.

LEMMA 3.4. *Let $\tau_1, \tau_2, p^{(m)}, \sigma$ be the results of Algorithm 3.3* (Transform) *applied to a nonzero vector of floating-point numbers $p_i^{(0)}, 1 \le i \le n$, and let $\varrho \in \mathbb{F}$. Define $M := \lceil \log_2(n + 2) \rceil$ and assume $2^M \texttt{eps} < 1$. Furthermore, assume that $\varrho \in \texttt{eps}\sigma_0\mathbb{Z}$ is satisfied for $\mu := \max_i |p_i^{(0)}|$ and $\sigma_0 = 2^{M + \lceil \log_2 \mu \rceil}$. Assume the parameter Φ in the "until"-condition to be a power of 2 satisfying $\texttt{eps} \le \Phi \le 1$. Denote $s := \sum_{i=1}^{n} p_i^{(0)}$.*

Then Algorithm 3.3 will stop, and

$$s + \varrho = t^{(m-1)} + \tau^{(m)} + \sum_{i=1}^{n} p_i^{(m)} \tag{3.2}$$

$$\max |p_i^{(m)}| \le \mathtt{eps}\sigma_{m-1} \;, \quad |\tau^{(m)}| \le (1-2^{-M})\sigma_{m-1} < \sigma_{m-1} \quad \text{and} \quad t^{(m-1)}, \tau^{(m)} \in \mathtt{eps}\sigma_{m-1}\mathbb{Z} \tag{3.3}$$

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

$$\tau_1 + \tau_2 = t^{(m-1)} + \tau^{(m)} \;, \quad |\tau_2| \le \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) \;, \quad \tau_1 = \mathrm{fl}(\tau_1 + \tau_2) = \mathrm{fl}(t^{(m-1)} + \tau^{(m)}) = t^{(m)} \tag{3.4}$$

is satisfied for the final value of m. If $\sigma_{m-1} \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ is satisfied for the final value of m, then the vector $p^{(m)}$ is entirely zero. If $\sigma_{m-1} > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ is satisfied for the final value of m, then

$$\mathrm{ufp}(\tau_1) \ge \Phi\sigma_{m-1} \;. \tag{3.5}$$

Remark. The assumption $\varrho \in \mathtt{eps}\sigma_0\mathbb{Z}$ is necessary to ensure that $\mathrm{ufp}(\varrho)$ is not too small: If Algorithm `Transform` does not stop for $m = 1$, then we will show that no rounding error occurs in the computation of $t^{(1)} = \mathrm{fl}(t^{(0)} + \tau^{(1)})$. Without the assumption on $\varrho = t^{(0)}$ this need not be true if $0 < \varrho \le \mathtt{eps}|\tau^{(1)}|$.

Proof of Lemma 3.4. For $\varrho = 0$, Algorithm `Transform` is identical to Algorithm 4.1 in Part I [22] of this paper with the parameter $\Phi = 2^{2M}\mathtt{eps}$. Algorithm 4.1 in Part I is analyzed in Lemma 4.2 in Part I with the stronger assumption $2^{2M}\mathtt{eps} \le 1$.

Carefully going through the proof of Lemma 4.2 in Part I we identify the necessary changes to prove Lemma 3.4. We see that $2^M\mathtt{eps} < 1$ suffices to guarantee that Algorithm 3.3 (`Transform`) stops. The assumption $\varrho \in \mathtt{eps}\sigma_0\mathbb{Z}$ assures that (3.3) is satisfied for $m = 1$, and $t^{(0)} = \varrho$ gives (3.2) for $m = 1$. In the induction step we used $\sigma_{m-2} \ge \mathtt{eps}^{-1}\mathtt{eta}$ to verify that no rounding error has occurred in the previous computation of $2^{2M}\mathtt{eps}\sigma_{m-2}$ in the "until"-condition. This is also true for $\Phi\sigma_{m-2}$ under our assumption $\mathtt{eps} \le \Phi \le 1$. Next we proved

$$|t^{(m-1)}| < \sigma_{m-2} \tag{3.6}$$

in (4.6) in Part I to assure that no rounding error occurs in the computation of $t^{(m-1)} = \mathrm{fl}(t^{(m-2)} + \tau^{(m-1)})$. But (3.6) follows by the previous "until"-condition on $|t^{(m-1)}|$ and $\mathtt{eps} \le \Phi \le 1$. This proves (3.2) and (3.3) for all values of m, and (3.4) follows as well. If $\sigma_{m-1} \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ is satisfied for the final value of m, then (3.3) implies $|p_i^{(m)}| \le \frac{1}{2}\mathtt{eta}$, hence $p_i^{(m)} = 0$ for all i. Finally, $\mathtt{eps} \le \Phi \le 1$ assures again that no rounding error occurred in the computation of $\Phi\sigma_{m-1}$ if $\sigma_{m-1} > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$ and (3.5) follows. The lemma is proved. □

A special application of Algorithm 3.3 (`Transform`) with $\varrho \ne 0$ might be the following. Suppose it is known that one component, p_1 say, of a vector p is much larger in magnitude than the others. Then the call `Transform`$(p(2:n), p_1)$, in Matlab notation, may reduce the number of loops since then σ_0 only depends on the smaller components $p_2, \cdots, p_n$ and not on p_1. We will use this in Section 6, where we will present Algorithm 6.4 (`AccSumK`) to compute a faithfully rounded result of K-fold accuracy.

The following lemma adapts some parts of the analysis of the original Algorithm 4.1 (`Transform`) in Part I and includes the offset ϱ. Note that the code in (3.7) without offset is the same as for Algorithm 4.5 (`AccSum`) in Part I.

Lemma 3.5. *Let p be a nonzero vector of n floating-point numbers and $\varrho \in \mathbb{F}$. Let `res` be computed as follows:*

$$\begin{array}{ll} [\tau_1, \tau_2, p', \sigma] = \mathtt{Transform}(p, \varrho) & \%\ \textit{Parameter } \Phi \textit{ replaced by } 2^{2M}\mathtt{eps} \\ \mathtt{res} = \mathrm{fl}(\tau_1 + (\tau_2 + (\sum_{i=1}^{n} p'_i))) & \end{array} \tag{3.7}$$

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

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

$$s + \varrho = \tau_1 + \tau_2 + \sum_{i=1}^n p_i' \quad \text{and} \quad \max |p_i'| \le \mathtt{eps}\sigma \ , \tag{3.8}$$

$$\mathrm{fl}(\tau_1 + \tau_2) = \tau_1 \ , \quad \tau_1, \tau_2 \in \mathtt{eps}\sigma\mathbb{Z} \quad \text{and} \quad |\tau_2| \le \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) \ , \tag{3.9}$$

$$|s + \varrho - \mathtt{res}| < 2\mathtt{eps}(1 - 2^{-M-1})\mathrm{ufp}(\mathtt{res}) \ . \tag{3.10}$$

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

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

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

$$\mathrm{ufp}(\tau_1) \ge 2^{2M}\mathtt{eps}\sigma \ . \tag{3.11}$$

If the exponent $2M$ *in the parameter* Φ *is changed into another integer, then* res *need not be a faithful rounding of* $s + \varrho$. *Abbreviate*

$$\begin{array}{rcccl} \tau_3 & = & \mathrm{fl}(\sum_{i=1}^n p_i') & = & \sum_{i=1}^n p_i' - \delta_3 \ , \\ \tau_2' & = & \mathrm{fl}(\tau_2 + \tau_3) & = & \tau_2 + \tau_3 - \delta_2 \ , \\ \mathtt{res} & = & \mathrm{fl}(\tau_1 + \tau_2') & = & \tau_1 + \tau_2' - \delta_1 \ . \end{array} \tag{3.12}$$

Then

$$s + \varrho = r + \delta \ \textit{ and } \ \mathtt{res} = \mathrm{fl}(r) \quad \textit{for} \quad r := \tau_1 + \tau_2' \ \textit{ and } \ \delta := \delta_2 + \delta_3 \ . \tag{3.13}$$

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

$$|\tau_3| = |\mathrm{fl}(\sum_{i=1}^n p_i')| \le n\mathtt{eps}\sigma \ , \tag{3.14}$$

$$|\tau_2'| \le (1 + \mathtt{eps})|\tau_2 + \tau_3| \le (1 + \mathtt{eps})(\mathtt{eps} \cdot \mathrm{ufp}(\tau_1) + n\mathtt{eps}\sigma) < |\tau_1| \ , \tag{3.15}$$

$$|\mathtt{res}| \ge \frac{5}{8}|\tau_1| \ , \tag{3.16}$$

$$2\mathtt{eps}^{-1}|\delta| < (1 - 2^{-M})|\mathtt{res}| < |\mathtt{res}| \ . \tag{3.17}$$

Proof. The only difference to the assumptions of Lemma 4.3 in Part I of this paper is the additional parameter ϱ in `Transform`. The proof and the assertions of Lemma 4.3 in Part I, however, are based on Lemma 4.2 in Part I and the error-free transformation $s = \tau_1 + \tau_2 + \sum p_i'$. With the offset ϱ this changes into $s + \varrho = \tau_1 + \tau_2 + \sum p_i'$ as by Lemma 3.4. Carefully going through the proof of Lemma 4.3 in Part I we see that basically s has to be replaced by $s + \varrho$, and the assertions (3.8) through (3.11) follow. The abbreviations (3.12) are the same as in (4.13) in the proof of Lemma 4.3 in Part I, implying (3.13). The remaining assertions (3.14) through (3.17) repeat corresponding assertions in the proof of Lemma 4.3 in Part I for the case $\sigma > \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$. □

4. The sign of a sum. In Remark 2 following Lemma 4.3 in Part I of this paper we saw that the exponent in the lower bound $2^{2M}\mathtt{eps}\sigma_{m-1}$ in the "until"-condition cannot be decreased without jeopardizing faithful rounding. In the refined version of Algorithm 3.3 (`Transform`) we saw in Lemma 3.4 that most properties remain valid when replacing the parameter Φ by a factor as small as eps, i.e. the lower bound in the "until"-condition reads $\mathtt{eps}\sigma_{m-1}$.

The smaller Φ, the less iterations are necessary. If only the sign of the sum $\sum p_i$ is to be computed and not necessarily a faithful rounding, then Φ may be decreased below $2^{2M}\mathtt{eps}$. Next we will show that if Φ is replaced by $2^M\mathtt{eps}$, then the sign of res as computed by Algorithm 4.5 (`AccSum`) in Part I is still equal to the sign of the sum $\sum p_i$, and this not only for n bounded by $2^{2M}\texttt{eps} \le 1$ but only requiring $2^M\texttt{eps} < 1$. Moreover, $2^M\texttt{eps}$ is the smallest possible choice for Φ.

Algorithm 4.1. *Rigorous computation of* $\operatorname{sign}(\sum p_i)$*, also for huge lengths.*

```
function $S = \texttt{AccSign}(p)$
    $[\tau_1, \tau_2, p', \sigma] = \texttt{Transform}(p, 0)$        % Parameter $\Phi$ replaced by $2^M\texttt{eps}$
    $S = \operatorname{sign}(\tau_1)$
```

Theorem 4.2. *Let S be the result of Algorithm 4.1 (`AccSign`) applied to a vector of floating-point numbers $p_i, 1 \le i \le n$. Assume that $2^M\texttt{eps} < 1$ is satisfied for $M := \lceil \log_2(n+2) \rceil$.*

Then Algorithm `AccSign` will stop, and

$$S = \operatorname{sign}\Big(\sum_{i=1}^{n} p_i\Big) \ . \tag{4.1}$$

The parameter Φ cannot be replaced by a power of 2 smaller than $2^M\texttt{eps}$ without jeopardizing (4.1). The algorithm needs $(4m+2)n + \mathcal{O}(m)$ flops for m "repeat-until"-loops in `Transform`.

Proof. Define $s := \sum_{i=1}^n p_i$. Without loss of generality assume the vector p to be nonzero. The assumptions of Lemma 3.4 are satisfied, and (3.2), (3.4) and $\varrho = 0$ imply

$$s = \tau_1 + \tau_2 + \sum_{i=1}^{n} p'_i \ . \tag{4.2}$$

If $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then p' is entirely zero and $s = \tau_1 + \tau_2$, so that $\tau_1 = \mathrm{fl}(\tau_1 + \tau_2) = \mathrm{fl}(s)$.

If $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then $\tau_1 = t^{(m)}$ and the "until"-condition in Algorithm 3.3 (`Transform`) imply $|\tau_1| \ge 2^M\texttt{eps}\sigma$, so that $|\tau_2| \le \texttt{eps}|\tau_1|$ and (3.3) give

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

and $\operatorname{sign}(s) = \operatorname{sign}(\tau_1)$ by (4.2).

To see that Φ is optimal, consider

$$p^{(0)} = \begin{bmatrix} c & c & c & c & -\frac{1}{8}\texttt{eps}^{-1} & -4 + 4\texttt{eps} \end{bmatrix} \in \mathbb{F}^6 \quad \text{with} \quad c := \frac{1}{32}\texttt{eps}^{-1} + 1 \tag{4.3}$$

in a floating-point format with relative rounding error unit $\texttt{eps} \le \frac{1}{64}$. Then $M = 3, \mu = \frac{1}{8}\texttt{eps}^{-1}$ and $\sigma = \texttt{eps}^{-1}$. Hence rounding tie to even implies $p^{(1)} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 4\texttt{eps} \end{bmatrix}$ and $\tau^{(1)} = 4 \cdot \frac{1}{32}\texttt{eps}^{-1} - \frac{1}{8}\texttt{eps}^{-1} - 4 = -4$ in the notation of Algorithm 3.3 (`Transform`). With the parameter $\Phi = 2^{M-1}\texttt{eps}$ the first inequality in the "until"-condition would read $|t^{(m)}| \ge 4$, so the "repeat-until"-loop would be finished. However, $t^{(0)} = 0$, so $\tau_1 = -4$ but $s = +4\texttt{eps}$. $\square$

Note that for the example in (4.3) with the parameter $\Phi = 2^{M-1}\texttt{eps}$ in `AccSign` it would also not help to compute `res` as in Algorithm 4.5 (`AccSum`) in Part I because $\tau_2 = 0$ and $\mathrm{fl}(\sum p_i^{(1)}) = 4$ by rounding tie to even, so that $\texttt{res} = 0$ but $s = 4\texttt{eps}$.

The improved version for sign determination is applicable in single precision for dimensions up to $8.3 \cdot 10^6$, and in double precision for dimensions up to $4.5 \cdot 10^{15}$.

5. K-fold faithful rounding.

Sometimes the precision of the given floating-point format may not be sufficient. One possibility for improvement is the use of multiple precision arithmetic such as in [2, 3, 7, 6]. Frequently such packages support a special long precision data type.

An alternative may be to put together a long number by pieces of floating-point numbers; for example XBLAS uses this to simulate quadruple precision [14]. For two and more pieces this technique was used in [21] to compute highly accurate inclusions of the solution of systems of linear equations. Later this technique was also called *staggered correction*.

To gain as much accuracy as possible, the pieces may be required to be non-overlapping, i.e. their bit representations should have no bit in common [19, 20]. Another approach is to assume that two numbers may only overlap in bits where those of the first one are zero [24].

In this section we define *K-fold faithful rounding*. Recall that by 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

$$\text{pred}(f) < r < \text{succ}(f) \ . \tag{5.1}$$

Before extending this definition to K-fold faithful rounding, we collect some properties of the ordinary faithful rounding.

LEMMA 5.1. *Let $r \in \mathbb{R}$ be given, and let $f \in \mathbb{F}$ be a faithful rounding of r. Then*

$$f \notin \mathbb{U} \quad \Rightarrow \quad |r - f| < 2\texttt{eps} \cdot \text{ufp}(r) \quad \textit{and} \quad |r - f| < 2\texttt{eps} \cdot \text{ufp}(f) \tag{5.2}$$

and

$$f \in \mathbb{U} \quad \Rightarrow \quad |r - f| < \texttt{eta} \ . \tag{5.3}$$

PROOF. If $f \in \mathbb{U}$, then $f \pm \texttt{eta}$ are the neighbors of f, and the definition of faithful rounding (5.1) yields $|r - f| < \texttt{eta}$. This proves (5.3).

To prove (5.2) we assume $f \notin \mathbb{U}$ and without loss of generality $f > 0$. For f not being a power of 2 we have $\text{ufp}(f) \le \text{ufp}(r)$, so (5.1) and (2.11) in Lemma 2.1 imply

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

If f is a power of 2, then this is also true if $f \le r < \text{succ}(f)$ because $\text{succ}(f) = (1 + 2\texttt{eps})\text{ufp}(f)$. If $(1 - \texttt{eps})f = \text{pred}(f) < r < f$ then

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

The lemma is proved. □

Lemma 5.1 is formulated for general $r \in \mathbb{R}$. For our main application, the approximation of the sum $s = \sum p_i$ of floating-point numbers p_i, we have $p_i \in \texttt{eta}\mathbb{Z}$ and therefore $s \in \texttt{eta}\mathbb{Z}$. Thus (5.3) reads in this case $f \in \mathbb{U} \Rightarrow r = f$. Before we define K-fold faithful rounding, we introduce the concept of a non-overlapping sequence of floating-point numbers.

DEFINITION 5.2. *A sequence of floating-point numbers $f_1, \cdots, f_k \in \mathbb{F}$ is called* non-overlapping, *if*

$$\begin{array}{lll} |f_{i+1}| < 2\texttt{eps} \cdot \text{ufp}(f_i) & \quad \textit{if} & f_i \ne 0 \\ f_{i+1} = 0 & \quad \textit{if} & f_i = 0 \end{array} \tag{5.4}$$

holds true for $1 \le i < k$.

REMARK. Note that trailing zeros are possible, and that a sequence of zeros is by definition non-overlapping.

The definition implies that for $f_i f_{i+1} \neq 0$ the binary expansions of f_i and f_{i+1} do not have a bit in common. The following properties hold true.

LEMMA 5.3. *Let $f_1, \cdots, f_k$ be a non-overlapping sequence of floating-point numbers. Then*

$$\text{ufp}(f_k) \le \texttt{eps}^{k-1}\text{ufp}(f_1) \ . \tag{5.5}$$

Moreover,

$$|\sum_{\nu=1}^{k} f_\nu| < 2\text{ufp}(f_1) \ . \tag{5.6}$$

REMARK. Note that Lemma 5.3 is applicable to any subsequence $f_{i_1}, \cdots, f_{i_m}$ with $i_1 > \cdots > i_m$ because it is non-overlapping as well.

PROOF. Let (f, g) be a non-overlapping sequence. Then the definition (5.4) of non-overlapping implies

$$\text{ufp}(g) \le \texttt{eps} \cdot \text{ufp}(f) \ , \tag{5.7}$$

and (5.5) follows. To prove (5.6) we first show

$$|f| + 2\text{ufp}(g) \le 2\text{ufp}(f) \ . \tag{5.8}$$

If $f \in \mathbb{U}$, then the definition of the set $\mathbb{U}$ and (5.5) imply $\text{ufp}(g) \le \frac{1}{2}\texttt{eta}$ and therefore $g = 0$. If $f \notin \mathbb{U}$, then $|f| \le \text{pred}(2\text{ufp}(f)) = 2(1 - \texttt{eps})\text{ufp}(f)$, and (5.7) implies (5.8). Without loss of generality we can omit trailing zeros in the sequence $f_1, \cdots, f_k$ and assume $f_k \neq 0$. Then

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

proving (5.6). □

Our aim is a sequence $(f_1, \cdots, f_k)$ such that $\sum f_\nu$ has a small relative error with respect to s, i.e.

$$|s - \sum_{\nu=1}^{k} f_\nu| < 2\texttt{eps}^k|s| \ . \tag{5.9}$$

For that we extend the Definition 2.2 of ordinary faithful rounding to a sequence $f_1, \cdots, f_k$.

DEFINITION 5.4. *A sequence $f_1, \cdots, f_k \in \mathbb{F}$ is called a (k-fold)* faithful rounding *of $s \in \mathbb{R}$ if*

$$f_i \in \square(s - \sum_{\nu=1}^{i-1} f_\nu) \qquad \text{for } 1 \le i \le k \ . \tag{5.10}$$

The sequence is called a strongly faithful rounding *if it is in addition non-overlapping.*

Of course, a strongly faithful rounding is faithful, but the converse is not true. However, the following lemma shows that the members of a sequence representing a faithful rounding may overlap in at most one bit, and subnormal numbers can only occur at the end.

LEMMA 5.5. *Suppose the sequence $f_1, \cdots, f_k \in \mathbb{F}$ is a faithful rounding of some $s \in \mathbb{R}$. If $|f_{i-1}| \ge \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ for $1 < i \le k$, then*

$$|f_i| \le 2\texttt{eps} \cdot \text{ufp}(f_{i-1}) \qquad \text{with equality if and only if} \quad \text{ufp}(f_i) = |f_i| \ . \tag{5.11}$$

Moreover, $|f_m| > 2\texttt{eta}$ implies $f_i \notin \mathbb{U}$ for $1 \le i < m$, and $|f_m| < \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ implies $f_i = 0$ for $m < i \le k$.

Proof. Let $f_1, \cdots, f_k \in \mathbb{F}$ be a faithful rounding of $s \in \mathbb{R}$. Then by definition

$$\sum_{\nu=1}^{m-1} f_\nu + \mathrm{pred}(f_m) < s < \sum_{\nu=1}^{m-1} f_\nu + \mathrm{succ}(f_m) \qquad \text{for } 1 \le m \le k \ . \tag{5.12}$$

Assume $f_i \ge 0$. Then using (5.12) for $m = i$ and $m = i - 1$ gives

$$\mathrm{pred}(f_i) < s - \sum_{\nu=1}^{i-1} f_\nu < \mathrm{succ}(f_{i-1}) - f_{i-1} \in \mathbb{F} \ ,$$

so

$$\mathrm{ufp}(f_i) \le f_i \le \mathrm{succ}(f_{i-1}) - f_{i-1} \le 2\mathtt{eps} \cdot \mathrm{ufp}(f_{i-1}) \ ,$$

where the last inequality follows by the assumption $|f_{i-1}| \ge \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$. The case $f_i < 0$ is treated similarly, and this proves (5.11), and also the very last statement of the lemma. For $f_m \ge 0$ and $f_m > 2\mathtt{eta}$, (5.12) yields

$$\mathrm{succ}(f_{m-1}) - f_{m-1} > s - \sum_{\nu=1}^{m-2} f_\nu - f_{m-1} > \mathrm{pred}(f_m) \ge 2\mathtt{eta} \ .$$

Hence $f_{m-1} \notin \mathbb{U}$, and an induction argument finishes the proof. $\square$

Next we show that faithful rounding is sufficient to satisfy our anticipated bound (5.9) on the relative error with respect to s.

Proposition 5.6. *If a sequence $f_1, \cdots, f_k \in \mathbb{F}$ is a faithful rounding of some $s \in \mathbb{R}$, then*

$$|s - \sum_{\nu=1}^{k} f_\nu| \quad < \quad \begin{cases} 2\mathtt{eps}^k \mathrm{ufp}(s) & \text{if } f_k \notin \mathbb{U} \\ 2\mathtt{eps}^k \mathrm{ufp}(f_1) & \text{if } f_k \notin \mathbb{U} \\ \mathtt{eta} & \text{if } f_k \in \mathbb{U} \ . \end{cases} \tag{5.13}$$

Moreover, a sequence representing a faithful rounding need not to be strongly faithful, i.e. may be overlapping.

Proof. For $0 \le i \le k$ define $\Delta_i := s - \sum_{\nu=1}^{i} f_\nu$. By assumption, $f_i \in \square(\Delta_{i-1})$ for $1 \le i \le k$. If $f_k \in \mathbb{U}$, then (5.13) follows by Lemma 5.1. Suppose $f_k \notin \mathbb{U}$, so that Lemma 5.5 implies $f_i \notin \mathbb{U}$ for $1 \le i \le k$. Now Lemma 5.1 yields $|\Delta_i| < 2\mathtt{eps} \cdot \min\big(\mathrm{ufp}(\Delta_{i-1}), \mathrm{ufp}(f_i)\big)$, hence

$$\mathrm{ufp}(\Delta_i) \le \mathtt{eps} \cdot \min\big(\mathrm{ufp}(\Delta_{i-1}), \mathrm{ufp}(f_i)\big) \tag{5.14}$$

and

$$\mathrm{ufp}(\Delta_k) \le \mathtt{eps}^{k-1} \mathrm{ufp}(\Delta_1) < 2\mathtt{eps}^k \cdot \min\big(\mathrm{ufp}(s), \mathrm{ufp}(f_1)\big) \ .$$

This proves (5.13) because the rightmost expression is a power of 2. Finally consider

$$f_1 = 1 - \mathtt{eps}, \quad f_2 = \mathtt{eps} \quad \text{and} \quad s = 1 - \mathtt{eps}^3 \ ,$$

then

$$\begin{aligned} f_1 + \mathrm{pred}(f_2) = 1 - \mathtt{eps}^2 \ &< \ s \ < \ 1 + 2\mathtt{eps}^2 = f_1 + \mathrm{succ}(f_2) \\ \mathrm{pred}(f_1) = 1 - 2\mathtt{eps} \ &< \ s \ < \ 1 = \mathrm{succ}(f_1) \end{aligned}$$

implies that (f_1, f_2) is a faithful rounding of s. However, it is overlapping by $|f_2| \ge 2\mathtt{eps} \cdot \mathrm{ufp}(f_1) = \mathtt{eps}$. $\square$

Note the difference to Algorithm 4.8 (`SumK`) in [18]. There, the result can also be stored in an array of length K and it is of a quality *as if* computed in K-fold precision, i.e. with relative rounding error unit $\mathtt{eps}^K$. So the *precision* is K-fold, the *accuracy*, however, depends on the condition number of the sum. The accuracy of the K-fold faithfully rounded result is independent of the condition number.

It is straightforward to derive from Algorithm 4.5 (`AccSum`) in Part I a new algorithm computing a result *as if* computed in K-fold precision, similar to `SumK` in [18]. This fact has been mentioned independently to the first author by Nicolas Louvet [15].

6. Algorithms for K-fold accuracy. Let floating-point numbers $p_1, \cdots, p_n$ be given. Next we will derive an algorithm to compute a sequence $f_1, \cdots, f_K \in \mathbb{F}$ representing a strongly faithful rounding of the sum $s := \sum p_i$.

LEMMA 6.1. *Let for a nonzero vector p of n floating-point numbers and $\varrho \in \mathbb{F}$ the assumptions of Lemma 3.5 be satisfied. Assume* $\texttt{res} \in \mathbb{F}$ *has been computed with the code given in (3.7) in Lemma 3.5. Set $f_1 :=$* $\texttt{res}$*, so that $f_1 \in \square(s+\varrho)$. Furthermore, suppose $f_2 \in \square(s+\varrho-f_1)$ for some $f_2 \in \mathbb{F}$. Then f_1, f_2 are non-overlapping.*

PROOF. If $f_2 \notin \mathbb{U}$, then (5.2) and (3.10) imply

$$\begin{aligned}|f_2| &\le |s+\varrho-f_1| + |s+\varrho-f_1-f_2| \le |s+\varrho-f_1| + 2\texttt{eps}\cdot\mathrm{ufp}(s+\varrho-f_1)\\ &\le (1+2\texttt{eps})|s+\varrho-f_1| < 2\texttt{eps}(1+2\texttt{eps})(1-2^{-M-1})\mathrm{ufp}(f_1)\\ &< 2\texttt{eps}\cdot\mathrm{ufp}(f_1)\ ,\end{aligned}$$

proving that f_1 and f_2 are non-overlapping. If $f_2 \in \mathbb{U}$, then $f_2 \in \square(s+\varrho-f_1)$ implies $f_2 = s+\varrho-f_1$, and the assertion follows similarly. □

When applying the code given in Lemma 3.5 to a vector p and $\varrho \in \mathbb{F}$, it calculates a faithful rounding $\texttt{res}$ of $s+\varrho$, and also extracts p into some vector p'. However, we have no *equation* relating s, ϱ, $\texttt{res}$ and p', but only an *estimation* which follows from $\texttt{res} \in \square(s+\varrho)$. The only *equation* between the mentioned quantities we know up to now is $s+\varrho = \tau_1+\tau_2+\sum p'_i$ as by Lemma 3.5.

However, to be able to repeatedly apply Lemma 6.1 to produce a non-overlapping sequence $(f_1, \cdots, f_K)$, we need a faithful rounding of $s+\varrho-\texttt{res}$. This can be done by applying the code given in (3.7) in Lemma 3.5 to ϱ and the vector p appended by $-\texttt{res}$. However, this would extract the entire vector $[p, -\texttt{res}]$ again and is ineffective. In the following Algorithm 6.2 (TransformK) we give an effective way to compute a single floating-point number R relating s, ϱ, $\texttt{res}$ and p' by an equation. It paves the way to compute a strongly faithful sequence of K numbers $f_1, \cdots, f_K$ approximating s, thus establishing an approximation of K-fold accuracy.

ALGORITHM 6.2. *Error-free vector transformation including faithful rounding.*

```
function [res, $R$, $p'$] = TransformK($p$, $\varrho$)
    [$\tau_1$, $\tau_2$, $p'$, $\sigma$] = Transform($p$, $\varrho$)        % Parameter $\Phi$ replaced by $2^{2M}$eps
    res = $\mathrm{fl}(\tau_1 + (\tau_2 + (\sum_{i=1}^n p'_i)))$
    $R = \mathrm{fl}(\tau_2 - (\texttt{res} - \tau_1))$
```

LEMMA 6.3. *Let p be a nonzero vector of n floating-point numbers and let $\varrho \in \mathbb{F}$. Abbreviate $s := \sum p_i$. Define $M := \lceil \log_2(n+2) \rceil$, and assume that $2^{2M}\texttt{eps} \le 1$. Furthermore, assume that $\varrho \in \texttt{eps}\sigma_0\mathbb{Z}$ is satisfied for $\mu := \max_i |p_i|$ and $\sigma_0 = 2^{M+\lceil \log_2 \mu \rceil}$. Let* $\texttt{res}$*, R and p' be the results of Algorithm 6.2* (TransformK) *applied to p and ϱ. Then* $\texttt{res}$ *is a faithful rounding of $s+\varrho$ and*

$$s + \varrho - \texttt{res} = R + \sum_{i=1}^{n} p'_i\ . \tag{6.1}$$

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

$$R \in \texttt{eps}\sigma'\mathbb{Z} \tag{6.2}$$

is satisfied for $\mu' := \max_i |p'_i|$ and $\sigma' := 2^{M+\lceil \log_2 \mu' \rceil}$. Moreover, no rounding error occurs in the computation of R, i.e.

$$\Delta := \texttt{res} - \tau_1 = \mathrm{fl}(\texttt{res} - \tau_1) \quad \textit{and} \quad R = \tau_2 - \Delta = \mathrm{fl}(\tau_2 - \Delta)\ . \tag{6.3}$$

Algorithm 6.2 (`TransformK`) *requires* $(4m+3)n+\mathcal{O}(m)$ *flops for* m *executions of the "repeat-until"-loop in* `Transform`.

Remark 1. The main purpose of Algorithm `TransformK` is to replace the pair τ_1, τ_2 by the pair $\texttt{res}, R$ without error. This makes it suitable for cascading, see Algorithm 6.4 (`AccSumK`).

Remark 2. The main part of the proof of Lemma 6.3 will be to show that no rounding error can occur in the computation of R, that is $R = \tau_2 - (\texttt{res} - \tau_1)$. The proof is involved and moved to the Appendix.

Remark 3. As is Matlab convention, an output parameter, in this case M for `Transform`, has been omitted. The output parameter σ is not needed but added for clarity.

Proof of Lemma 6.3. For the computation of $\tau_1, \tau_2, p', \sigma$ and `res`, Algorithm 6.2 (`TransformK`) uses the same piece of code as in (3.7) in Lemma 3.5, so the assertions of Lemma 3.5 are valid. Hence the assertions follow if $\texttt{res} = 0$.

First assume $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$. Then $p'_i = 0$ for all i by Lemma 3.5, and (3.8) and (3.9) imply $s + \varrho = \tau_1 + \tau_2$ and $\texttt{res} = \text{fl}(\tau_1 + \tau_2) = \tau_1$, so $R = \tau_2$ and (6.1) and (6.3) follow. The case $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$ is finished.

Henceforth we assume $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$. We use the notation in Lemma 3.5, especially (3.12). Then $|\tau'_2| < |\tau_1|$ by (3.15), and $\texttt{res} = \text{fl}(\tau_1 + \tau'_2)$ and (2.14) in Lemma 2.5 imply

$$\Delta := \texttt{res} - \tau_1 = \text{fl}(\texttt{res} - \tau_1) \in \mathbb{F} \; . \tag{6.4}$$

Next we have to show $R \in \mathbb{F}$, i.e.

$$R = \text{fl}(\tau_2 - \Delta) = \tau_2 - \Delta \tag{6.5}$$

provided $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$. This proof is quite involved and left to the Appendix. It verifies (6.3). Now (6.4) and (6.5) yield $R = \tau_1 + \tau_2 - \texttt{res}$, and (3.8) gives

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

This proves (6.1). To see (6.2) note that (2.8), (3.11) and (2.4) imply $\Delta = \texttt{res} - \tau_1 \in \texttt{eps} \cdot \text{ufp}(\tau_1)\mathbb{Z} \subseteq 2^{2M}\texttt{eps}^2\sigma\mathbb{Z}$. Furthermore, (3.9) and $2^{2M}\texttt{eps} \le 1$ yield $\tau_2 \in \texttt{eps}\sigma\mathbb{Z} \subseteq 2^{2M}\texttt{eps}^2\sigma\mathbb{Z}$, and (2.6) gives

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

But (3.8) and the definition of σ' imply $\texttt{eps}\sigma' \le 2^M\texttt{eps}^2\sigma$, and (6.2) follows by (2.4). □

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

Algorithm 6.4. *Accurate summation computing* K*-fold faithful rounding.*

```
function $\texttt{Res} = \texttt{AccSumK}(p, K)$
    $p^{(0)} = p,\ R_0 = 0$
    for $k = 1 : K$
        $[\texttt{Res}_k, R_k, p^{(k)}] = \texttt{TransformK}(p^{(k-1)}, R_{k-1})$
        if $\texttt{Res}_k \in \mathbb{U}$, $\texttt{Res}_{k+1..K} = 0$, return; end if
    end for
```

Proposition 6.5. *Let* p *be a vector of* n *floating-point numbers and abbreviate* $s := \sum p_i$. *Define* $M := \lceil \log_2(n+2) \rceil$, *and assume that* $2^{2M}\texttt{eps} \le 1$. *Let* $1 \le K \in \mathbb{N}$ *be given, and let* `Res` *be the result vector of Algorithm 6.4* (`AccSumK`) *applied to* p *and* K.

Then $\texttt{Res}_1, \cdots, \texttt{Res}_K$ *is a strongly faithful rounding of* s. *Furthermore,*

$$s = \sum_{\nu=1}^{k} \texttt{Res}_\nu + R_k + \sum_{i=1}^{n} p_i^{(k)} \quad \textit{for } 1 \le k \le K \ . \tag{6.6}$$

Abbreviate $\overline{\texttt{Res}} := \sum_{k=1}^{K} \texttt{Res}_k$. *Then* $\texttt{Res}_k \in \mathbb{U}$ *for some* $1 \le k \le K$ *implies* $\overline{\texttt{Res}} = s$, *and* $\texttt{Res}_k \notin \mathbb{U}$ *implies*

$$|s - \overline{\texttt{Res}}| < 2\texttt{eps}^K \mathrm{ufp}(s) \le 2\texttt{eps}^K |s| \tag{6.7}$$

and

$$|s - \overline{\texttt{Res}}| < 2\texttt{eps}^K \mathrm{ufp}(\texttt{Res}_1) \le 2\texttt{eps}^K |\texttt{Res}_1| \ . \tag{6.8}$$

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

Remark 2. For $k \ge 3$, say, it might be advantageous to eliminate zero summands in the vectors $p^{(k)}$. Assuming that it seems unlikely that extracted vectors $p^{(k)}$ are ill-conditioned for $k \ge 1$ and using the fact that the new σ needs not to be computed in subsequent calls to TransformK, AccSumK needs $(4m + 5K + 3)n + \mathcal{O}(m + K)$ flops if the "repeat-until"-loop in the first extraction TransformK is executed m times.

Remark 3. Also note that the limited exponent range poses no problem to achieve K-fold accuracy. Since the exact result s is a sum of floating-point numbers, it follows $s \in \texttt{eta}\mathbb{Z}$, and also $s - \sum f_k \in \texttt{eta}\mathbb{Z}$ for arbitrary floating-point numbers f_k. So for a certain K, the sum s is *always* stored exactly in $\sum_{k=1}^{K} \texttt{Res}_k$. The maximum K can be estimated solely by the exponent range because the sequence $(\texttt{Res}_1, \cdots, \texttt{Res}_K)$ is non-overlapping. In IEEE 754 single precision at most $K \le 12$ summands are needed, in IEEE 754 double precision at most $K \le 40$.

Remark 4. In an actual implementation, of course, R and the vector p can be reused. So, for example, the main statement in Algorithm 6.4 (AccSumK) reads $[\texttt{Res}_k, R, p] = \texttt{TransformK}(p, R)$.

Proof of Proposition 6.5. For zero input vector p the assertions are evident; henceforth we assume p to be nonzero.

For $k = 0$ the assumptions of Lemma 6.3 are satisfied, so $\texttt{Res}_1$ is a faithful rounding of $s = \sum p_i^{(0)}$. Moreover, by (6.2) the assumptions of Lemma 6.3 are satisfied for $k \ge 1$ as well. With $s_k := \sum p_i^{(k)}$ this means

$$\begin{array}{rcccl} s & = & s_0 & = & \texttt{Res}_1 + R_1 + s_1 \\ & & & = & \texttt{Res}_1 + \texttt{Res}_2 + R_2 + s_2 \\ & & & = & \cdots \\ & & & = & \sum_{k=1}^{K} \texttt{Res}_k + R_K + s_K \ , \end{array}$$

which is true because $\texttt{Res}_k \in \mathbb{U}$ implies R_k and all $p_i^{(k)}$ to be zero by Lemma 6.3. This proves (6.6), and that $\texttt{Res}_k$ is a faithful rounding of $s - \sum_{\nu=1}^{k-1} \texttt{Res}_\nu$ for $1 \le k \le K$. Thus the sequence $\texttt{Res}_1, \cdots, \texttt{Res}_K$ is a faithful rounding of s, and Proposition 5.6 proves (6.7) and (6.8). Moreover, Lemma 6.1 implies $\texttt{Res}_1, \cdots, \texttt{Res}_K$ to be non-overlapping, thus $\texttt{Res}_1, \cdots, \texttt{Res}_K$ is even a strongly faithful rounding of s. $\square$

Now it becomes clear why so much effort was spent (see the Appendix) to prove (6.1) and (6.2): It is the key to cascade TransformK in Algorithm 6.4 (AccSumK) without increasing the length of the input vector. Especially for large gaps in the input vector, R_{k-1} in AccSumK may be large compared to $p^{(k-1)}$. In that case a number of case distinctions were necessary in the proof in the Appendix to assure that no rounding error can occur in the computation of R_k in TransformK.

7. Rounding to nearest and directed rounding. Using the results of the previous section we can derive algorithms for computing $s := \sum p_i$ with directed rounding, i.e. $\texttt{resD} := \max\{f \in \mathbb{F} : f \le s\}$ for rounding downwards, and $\texttt{resU} := \min\{f \in \mathbb{F} : s \le f\}$ for rounding upwards. Note that $\texttt{resD} = \texttt{resU}$ is equivalent to $s \in \mathbb{F}$. As an example we display Algorithm 7.1 (DownSum) for rounding downwards, its counterpart looks similarly. The proof of correctness follows straightforwardly by Lemma 6.3 and the definition of faithful rounding.

ALGORITHM 7.1. *Accurate summation with rounding downwards.*

```
function $\texttt{resD} = \texttt{DownSum}(p)$
    $[\texttt{res}, R, p'] = \texttt{TransformK}(p, 0)$   % $s - \texttt{res} = R + \sum p'_i$
    $\delta = \texttt{TransformK}(p', R)$                % $\delta \in \Box(s - \texttt{res})$
    if $\delta < 0$, $\texttt{resD} = \text{pred}(\texttt{res})$     % $s < \texttt{res}$
    else $\texttt{resD} = \texttt{res}$                    % $\texttt{res} \le s$
    end if
```

Next we show how to compute the rounded to nearest result of the sum of a vector of floating-point numbers. If res is a faithful rounding of $s := \sum p_i$ and $\texttt{res} \ne s$, then res must be one of the immediate floating-point neighbors of s. Define for $c \in \mathbb{F} \setminus \{\pm\infty\}$ with finite predecessor and successor

$$\mathcal{M}^-(c) := \frac{1}{2}\big(\text{pred}(c) + c\big) \quad \text{and} \quad \mathcal{M}^+(c) := \frac{1}{2}\big(c + \text{succ}(c)\big). \tag{7.1}$$

Then $\mathcal{M}^-(\texttt{res}) < s < \mathcal{M}^+(\texttt{res})$ implies $\text{fl}(s) = \texttt{res}$,

$$\begin{array}{lll} s < \mathcal{M}^-(\texttt{res}) \Rightarrow \text{fl}(s) = \text{pred}(\texttt{res}) & \text{and} & \mathcal{M}^+(\texttt{res}) < s \Rightarrow \text{fl}(s) = \text{succ}(\texttt{res}) \ , \\ s < \mathcal{M}^+(\texttt{res}) \Rightarrow \text{fl}(s) \ne \text{succ}(\texttt{res}) & \text{and} & \mathcal{M}^-(\texttt{res}) < s \Rightarrow \text{fl}(s) \ne \text{pred}(\texttt{res}) \ , \end{array} \tag{7.2}$$

and for the cases $\mathcal{M}^-(\texttt{res}) = s$ and $s = \mathcal{M}^+(\texttt{res})$ the rounding depends on the tie. So if $\text{sign}(\texttt{res} - s)$ is known, then the rounding to nearest $\text{fl}(s)$ of s can be calculated by computing a faithful rounding of $s - \mu$ with $\mu \in \{\mathcal{M}^-(\texttt{res}), \mathcal{M}^+(\texttt{res})\}$, and it can be decided which of $\mathcal{M}^-$ or $\mathcal{M}^+$ to choose for μ. This is true because a faithful rounding of $s - \mu$ detects the sign with certainty. The sign of $\texttt{res} - s$ could be calculated by calling $\texttt{AccSum}(s, -\texttt{res})$ using Algorithm 4.5 (AccSum) in Part I. However, we can do much better. The following lemma shows how enough information can be extracted from AccSum to decide which of pred(res) or succ(res) the true rounding to nearest $\text{fl}(s)$ can only be in case $\text{fl}(s) \ne \texttt{res}$.

LEMMA 7.2. *Let p be a vector of n floating-point numbers, define $M := \lceil \log_2(n+2) \rceil$ and assume $2^{2M}\texttt{eps} \le 1$. Let* res *and δ_1 be computed as follows:*

$$\begin{array}{ll} [\tau_1, \tau_2, p'] = \texttt{Transform}(p, 0) & \\ \tau'_2 = \text{fl}(\tau_2 + (\sum_{i=1}^n p'_i)) & \textit{\% Parameter } \Phi \textit{ replaced by } 2^{2M}\texttt{eps} \\ [\texttt{res}, \delta_1] = \texttt{FastTwoSum}(\tau_1, \tau'_2) & \end{array} \tag{7.3}$$

Abbreviate $s := \sum_{i=1}^n p_i$. Then $\delta_1 = 0$ implies $\text{fl}(s) = \texttt{res}$, and

$$\text{fl}(s) \in \begin{cases} \{\text{pred}(\texttt{res}), \texttt{res}\} & \textit{if } \delta_1 < 0 \\ \{\texttt{res}, \text{succ}(\texttt{res})\} & \textit{if } \delta_1 > 0 \ . \end{cases} \tag{7.4}$$

PROOF. The first statement in Algorithm 2.4 (FastTwoSum) is $\texttt{res} = \text{fl}(\tau_1 + \tau'_2)$, so the result res computed in (7.3) is the same as in (3.7) for $\varrho = 0$. Moreover, the assumptions of Lemma 3.5 are satisfied, so res is a faithful rounding of s. This means $\text{fl}(s) \in \{\text{pred}(\texttt{res}), \texttt{res}, \text{succ}(\texttt{res})\}$. If the final value of σ in Transform satisfies $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then the vector p' is entirely zero and $\text{fl}(s) = \text{fl}(\tau_1 + \tau_2) = \texttt{res}$. Henceforth we may assume $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$.

We use the same notation as in Lemma 3.5, especially (3.12). Then $|\tau_2'| < |\tau_1|$ by (3.15), and Lemma 2.5 yields $\texttt{res} + \delta_1 = \tau_1 + \tau_2'$ [as in (3.12)], where the error δ_1 is the second result of $\texttt{FastTwoSum}(\tau_1, \tau_2')$. By (3.13) we know $s = \tau_1 + \tau_2' + \delta$, so that $\delta_1 \le 0$ and (3.17) imply

$$s = \texttt{res} + \delta_1 + \delta < \texttt{res} + \frac{1}{2}\texttt{eps}|\texttt{res}| \le \mathcal{M}^+(\texttt{res}) \ .$$

Thus (7.2) shows $\mathrm{fl}(s) \ne \mathrm{succ}(\texttt{res})$. The case $\delta_1 \ge 0$ follows similarly, and the case $\delta_1 = 0$ follows. $\square$

By Lemma 7.2 it can be decided which $\mu \in \{\mathcal{M}^-(\texttt{res}), \mathcal{M}^+(\texttt{res})\}$ has to be subtracted from s to decide whether $\mathrm{fl}(s)$ is equal to $\texttt{res}$ or to one of its neighbors. However, μ is not a floating-point number but could only be subtracted in two parts by splitting μ into $\texttt{res}$ and $\delta_{\texttt{res}} := \mu - \texttt{res} \in \mathbb{F}$. Rather than extracting the vector p appended by $-\texttt{res}$ and $-\delta_{\texttt{res}}$ again, it is much better to use the already extracted vector p'. Fortunately, our analysis in Lemma 6.3 gives an equation for $s - \texttt{res}$ in terms of p_i', namely $s - \mu = s - \texttt{res} - \delta_{\texttt{res}} = R + \sum p_i' - \delta_{\texttt{res}}$ for R as computed in Algorithm 6.2 ($\texttt{TransformK}$). Now Algorithm 3.3 ($\texttt{Transform}$) allows for one extra parameter to be added to $\sum p_i'$, but not two. The following lemma shows that one parameter $R' \in \mathbb{F}$ is sufficient and $\texttt{Transform}$ can be applied directly to p' and R'.

LEMMA 7.3. *Let p be a vector of n floating-point numbers, define $M := \lceil \log_2(n+2) \rceil$, and assume $2^{2M}\texttt{eps} \le 1$. Let $\texttt{res}$, R and p' be the results of Algorithm 6.2 ($\texttt{TransformK}$) applied to p and $\varrho = 0$. Let $\delta_{\texttt{res}} \in \{\frac{1}{2}(\mathrm{pred}(\texttt{res}) - \texttt{res}), \frac{1}{2}(\mathrm{succ}(\texttt{res}) - \texttt{res})\}$ and assume $\delta_{\texttt{res}} \in \mathbb{F}$.*

Then $R - \delta_{\texttt{res}} \in \mathbb{F}$. Moreover, if the vector p' is nonzero, then $\delta_{\texttt{res}} \in \texttt{eps}\sigma'\mathbb{Z}$ is satisfied for $\mu' := \max_i |p_i'|$ and $\sigma' := 2^{M + \lceil \log_2 \mu' \rceil}$.

PROOF. For the computation of $\texttt{res}$ we use the same code as in (3.7), the assumptions of Lemma 3.5 and Lemma 6.3 are satisfied and we may use results of the lemmas and the proofs, especially $\Delta = \texttt{res} - \tau_1$ and $R = \tau_2 - \Delta$.

Assume the final value of σ in $\texttt{Transform}$ satisfies $\sigma \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$. Then by Lemma 3.5 the vector p' is entirely zero, $s = \tau_1 + \tau_2$ and $\texttt{res} = \mathrm{fl}(\tau_1 + \tau_2) = \tau_1$. Thus $\Delta = \texttt{res} - \tau_1 = 0$ and $R = \tau_2$. If the final value of m in Algorithm 3.3 ($\texttt{Transform}$) is $m = 1$, then $\max |p_i| \le \mu \le 2^{-M}\sigma$ so that $|s| \le n\mu < \sigma$ implies $s \in \mathbb{U}$ and $\texttt{res} = s \in \mathbb{U}$, which means $\delta_{\texttt{res}} = \texttt{eta}/2 \notin \mathbb{F}$, a contradiction. Therefore, the final value m satisfies $m > 1$, and the "until"-condition in yields $|\texttt{res}| = |\tau_1| = |t^{(m-1)}| < 2^{2M}\texttt{eps}\sigma_{m-2} = 2^M\sigma \le 2^{M-1}\texttt{eps}^{-1}\texttt{eta}$. Hence $|\delta_{\texttt{res}}| \le 2^{M-2}\texttt{eta}$ and $|R| = |\tau_2| \le \texttt{eps}|\tau_1| \le 2^{M-1}\texttt{eta}$, and $R - \delta_{\texttt{res}} \in \mathbb{F}$ follows.

Henceforth we assume $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$. Then the assumption $\delta_{\texttt{res}} \in \mathbb{F}$ implies $|\texttt{res}| \ge \texttt{eps}^{-1}\texttt{eta}$ and

$$\delta_{\texttt{res}} \in \begin{cases} \frac{1}{2}\texttt{eps} \cdot \mathrm{ufp}(\texttt{res})\mathbb{Z} & \text{if } |\texttt{res}| = \mathrm{ufp}(\texttt{res}) \\ \texttt{eps} \cdot \mathrm{ufp}(\texttt{res})\mathbb{Z} & \text{if } |\texttt{res}| \ne \mathrm{ufp}(\texttt{res}) \end{cases} \tag{7.5}$$

and

$$|\delta_{\texttt{res}}| \le \texttt{eps} \cdot \mathrm{ufp}(\texttt{res}) \ . \tag{7.6}$$

We know by (3.16) and (3.11) that

$$|\texttt{res}| > \frac{5}{8}|\tau_1| > 2^{2M-1}\texttt{eps}\sigma \ge 2^{M+1}\texttt{eps}\sigma \ .$$

Moreover (3.8) gives $\sigma' \le 2^M\texttt{eps}\sigma$, so that (7.5) and (2.4) prove $\delta_{\texttt{res}} \in 2^M\texttt{eps}^2\sigma\mathbb{Z} \subseteq \texttt{eps}\sigma'\mathbb{Z}$.

To show $R - \delta_{\texttt{res}} \in \mathbb{F}$, we distinguish two cases. First, assume $|\tau_1| < \sigma$. Then $|\tau_2| \le \texttt{eps}|\tau_1| < \texttt{eps}\sigma$ and $\tau_2 \in \texttt{eps}\sigma\mathbb{Z}$ as in (3.9) yields $\tau_2 = 0$. Hence $R \in \mathbb{F}$ gives $R = \tau_1 - \texttt{res} = \mathrm{fl}(\tau_1 - \texttt{res})$, and (2.8) shows $R \in \texttt{eps}\cdot\mathrm{ufp}(\texttt{res})\mathbb{Z} \subseteq \frac{1}{2}\texttt{eps}\cdot\mathrm{ufp}(\texttt{res})\mathbb{Z}$. Regarding (7.5) and (2.10) it suffices to show $|R - \delta_{\texttt{res}}| \le \frac{1}{2}\mathrm{ufp}(\texttt{res})$. By (3.12), (2.9), $\tau_2 = 0$, (3.14) and (3.11),

$$\begin{aligned} |R| &= |\texttt{res} - \tau_1| = |\tau_3 - \delta_2 - \delta_1| \le |\tau_3| + \texttt{eps}|\tau_3| + |\delta_1| \le (1 + \texttt{eps})n\texttt{eps}\sigma + \texttt{eps} \cdot \mathrm{ufp}(\texttt{res}) \\ &\le (2^M - 1)\texttt{eps}\sigma + \texttt{eps} \cdot \mathrm{ufp}(\texttt{res}) \le (2^M - 1)2^{-2M}\mathrm{ufp}(\tau_1) + \texttt{eps} \cdot \mathrm{ufp}(\texttt{res}) \ . \end{aligned} \tag{7.7}$$

Now (3.16) implies $\mathrm{ufp}(\tau_1) \le 2\mathrm{ufp}(\mathtt{res})$, so that (7.6) and the assumption $2^{2M}\mathtt{eps} \le 1$ yield

$$|R - \delta_{\mathtt{res}}| \le \big((2^M - 1)2^{-2M+1} + 2\mathtt{eps}\big)\mathrm{ufp}(\mathtt{res}) \le 2^{-M+1}\mathrm{ufp}(\mathtt{res}) \le \frac{1}{2}\mathrm{ufp}(\mathtt{res}) \ .$$

This finishes the first case. Second, assume $|\tau_1| \ge \sigma$. Then as in (10.7) we see

$$|R| = |\tau_2 - \Delta| \le \mathtt{eps}(1 + 3\mathtt{eps})|\tau_1| + 2^M\mathtt{eps}\sigma \ , \tag{7.8}$$

and by (2.8) we know $\Delta = \mathtt{res} - \tau_1 \in \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) \subseteq \mathtt{eps}\sigma\mathbb{Z}$. We distinguish two subcases. First, suppose $|\mathtt{res}| \le \sigma$. Then (3.9) yields $R = \tau_2 - \Delta \in \mathtt{eps}\sigma\mathbb{Z} \subseteq \frac{1}{2}\mathtt{eps}\sigma\mathbb{Z}$. Regarding (7.5) implies $\delta_{\mathtt{res}} \in \frac{1}{2}\mathtt{eps} \cdot \mathrm{ufp}(\mathtt{res})\mathbb{Z} \subseteq \frac{1}{2}\mathtt{eps}\sigma\mathbb{Z}$, and with (2.10) it suffices to show $|R - \delta_{\mathtt{res}}| \le \frac{1}{2}\sigma$. This is seen with (7.8), (3.16), (7.6) and

$$|R - \delta_{\mathtt{res}}| \le \mathtt{eps}(1 + 3\mathtt{eps})\frac{8}{5}\sigma + 2^M\mathtt{eps}\sigma + \mathtt{eps}\sigma \le 3\mathtt{eps}\sigma + 2^{-M}\sigma < \frac{1}{2}\sigma.$$

This leaves us, secondly, with $|\tau_1| \ge \sigma$ and $|\mathtt{res}| > \sigma$. In this case (7.5) and (3.9) imply $\Delta, \delta_{\mathtt{res}} \in \mathtt{eps}\sigma\mathbb{Z}$ and $R = \tau_2 - \Delta \in \mathtt{eps}\sigma\mathbb{Z}$ as well, so with (2.10) it suffices to show $|R - \delta_{\mathtt{res}}| \le \sigma$. We have $t^{(0)} = \varrho = 0$ in Algorithm 3.3, so for the final value of m in the "until"-condition (Transform) and with (3.3) we have

$$|\tau_1| = |\mathrm{fl}(t^{(m-1)} + \tau^{(m)})| < (1 + \mathtt{eps})(2^{2M}\mathtt{eps}\sigma_{m-2} + \sigma) = (1 + \mathtt{eps})(2^M + 1)\sigma < (2^M + 2)\sigma \ .$$

Now (3.12) and (3.15) give

$$|\mathtt{res}| \le (1 + \mathtt{eps})|\tau_1 + \tau_2'| \le (1 + 2\mathtt{eps})^2|\tau_1| + 2^M\mathtt{eps}\sigma < (2^M + 3)\sigma \ ,$$

so that (7.6) yields $|\delta_{\mathtt{res}}| \le 2^M\mathtt{eps}\sigma$. Using (7.8) and $2^{2M}\mathtt{eps} \le 1$ we conclude

$$|R - \delta_{\mathtt{res}}| < (2^M + 4)\mathtt{eps}\sigma + 2^{M+1}\mathtt{eps}\sigma \le 2^{M+2}\mathtt{eps}\sigma \le \sigma \ .$$

The lemma is proved. □

We note that Lemma 7.3 does not remain true for $\varrho \ne 0$ as is seen by $\varrho = 1$, the 1-element vector $p = \mathtt{eps}^2$ and the choice $\delta_{\mathtt{res}} = -\mathtt{eps}$. However, we do not need this case in the sequel.

Now we can state the algorithm to compute the rounded-to-nearest result of a sum of floating-point numbers.

ALGORITHM 7.4. *Accurate summation with rounding to nearest.*

```
function resN = NearSum($p$)
    $[\tau_1, \tau_2, p'] = \mathtt{Transform}(p, 0)$                 % Parameter $\Phi$ replaced by $2^{2M}\mathtt{eps}$
    $\tau_2' = \mathrm{fl}(\tau_2 + (\sum_{i=1}^n p_i'))$
    $[\mathtt{res}, \delta] = \mathtt{FastTwoSum}(\tau_1, \tau_2')$              % $\mathtt{res} + \delta = \tau_1 + \tau_2'$
    if $\delta = 0$, resN = res, return, end if                  % $\mathrm{fl}(s) = \mathtt{res}$
    $R = \tau_2 - (\mathtt{res} - \tau_1)$                          % $s - \mathtt{res} = R + \sum p_i'$
    if $\delta < 0$                                              % $\mathrm{fl}(s) \in \{\mathrm{pred}(\mathtt{res}), \mathtt{res}\}$
        $\gamma = \mathrm{pred}(\mathtt{res}) - \mathtt{res}$                       % $\mathtt{res} + \gamma = \mathrm{pred}(\mathtt{res})$
        if $\gamma = -\mathtt{eta}$, resN = res, return, end if   % $\mathcal{M}^-(\mathtt{res}) \notin \mathbb{F} \Rightarrow s = \mathtt{res}$
        $\delta' = \gamma/2$                                       % $\mu := \mathtt{res} + \delta' = \mathcal{M}^-(\mathtt{res})$
        $\delta'' = \mathtt{TransformK}(p', R - \delta')$                % $s - \mu = R - \delta' + \sum p_i',\ \delta'' \in \Box(s - \mu)$
        if $\delta'' > 0$, resN = res                              % $s > \mathcal{M}^-(\mathtt{res})$
            elseif $\delta'' < 0$, resN = pred(res)                % $s < \mathcal{M}^-(\mathtt{res})$
            else resN = $\mathrm{fl}(\mathtt{res} + \delta')$                       % $s = \mathcal{M}^-(\mathtt{res})$
        end if
    else                                                         % $\mathrm{fl}(s) \in \{\mathtt{res}, \mathrm{succ}(\mathtt{res})\}$
        the case $\delta > 0$ is treated similarly
    end if
```

THEOREM 7.5. *Let p be a vector of n floating-point numbers, define $M := \lceil \log_2(n+2) \rceil$, and assume $2^{2M}\texttt{eps} \le 1$. Let* `resN` *be the result of Algorithm 7.4* (`NearSum`) *applied to p.*

Then $\texttt{resN} = \text{fl}(s)$ is the rounded-to-nearest exact sum $s := \sum_{i=1}^{n} p_i$ in the sense of IEEE 754.

REMARK 1. When the floating-point rounding "fl" is omitted in Algorithm 7.4, then we will show that no rounding error can occur.

REMARK 2. Note that numerical evidence suggests that it seems worth to check for $\delta = 0$ to save the second extraction by `TransformK`. If the "repeat-until"-loop in the first and second extraction by `TransformK` is executed m and m' times, respectively, and using the fact that the second extraction unit σ needs not to be computed, then `NearSum` needs $(4m + 4m' + 4)n + \mathcal{O}(m + m')$ flops.

REMARK 3. Note that we generally assume that no overflow occurs, however, this is easily treated by some scaling.

REMARK 4. We used the predecessor and successor of a floating-point number. These are especially easy to calculate if directed rounding as in IEEE 754 is available. Otherwise, for example, they may computed in rounding to nearest using Algorithm 3.5 (`NextPowerTwo`) from Part I of this paper, or by the algorithms presented in [23].

PROOF OF THEOREM 7.5. The internal result `res` is computed in the same way as in (7.3), and the assumptions of Lemma 7.2 are satisfied. The quantity δ_1 in Lemma 7.2 is the same as δ in `NearSum`. Therefore, $\text{fl}(s) \in \{\text{pred}(\texttt{res}), \texttt{res}, \text{succ}(\texttt{res})\}$, and $\delta = 0$ implies $\text{fl}(s) = \texttt{res}$. The quantity R is computed exactly the same way as in Algorithm 6.2 (`TransformK`), so (6.1) implies $s - \texttt{res} = R + \sum p'_i$.

If $\delta < 0$, then (7.4) implies $\text{fl}(s) \in \{\text{pred}(\texttt{res}), \texttt{res}\}$. The quantity $\gamma = \text{pred}(\texttt{res}) - \texttt{res}$ is always in $\mathbb{F}$, and $\gamma = -\texttt{eta}$ together with $\texttt{res} \in \square(s)$ and $s = \sum p_i \in \texttt{eta}\mathbb{Z}$ implies $s = \texttt{res}$. If $\gamma \ne -\texttt{eta}$, then $\delta' = \frac{1}{2}(\text{pred}(\texttt{res}) - \texttt{res}) \in \mathbb{F}$ and $\mathcal{M}^-(\texttt{res}) = \texttt{res} + \delta'$. The quantity $\delta_{\texttt{res}}$ in Lemma 7.3 is the same as δ', so $R - \delta' \in \mathbb{F}$. Moreover, using $\sigma' = 2^{M + \lceil \log_2 \mu' \rceil}$ with $\mu' := \max_i |p'_i|$ as in Lemma 6.3, (6.2) and Lemma 7.3 prove $R, \delta', R - \delta' \in \texttt{eps}\sigma'\mathbb{Z}$ for the first extraction of $(p, 0)$, so that the assumptions of Lemma 6.3 for the application of Algorithm 6.2 (`TransformK`) to $(p', R - \delta')$ are satisfied. Therefore $\delta'' \in \square(R - \delta' + \sum p'_i)$, where $R - \delta' + \sum p'_i = s - \texttt{res} - \delta' = s - \mathcal{M}^-(\texttt{res})$. Hence $\text{sign}(s - \mathcal{M}^-(\texttt{res})) = \text{sign}(\delta'')$. So the assertion follows by (7.2) if $\delta'' \ne 0$. Finally, if $\delta'' = 0$, then $s = \mathcal{M}^-(\texttt{res})$ and $\text{fl}(s) = \text{fl}(\mathcal{M}^-(\texttt{res}))$.

The case $\delta > 0$ is programmed and treated similarly, and the theorem is proved. □

In contrast to Algorithm 4.5 (`AccSum`) in Part I, the computing time of Algorithm 7.4 (`NearSum`) depends on the exponent variation of the vector entries rather than on the condition number of the sum. However, it seems unlikely that the maximum number of 40 extractions for IEEE 754 double precision is achieved in other than constructed examples.

There is an apparent contradiction, namely that the computing time of `AccSum` is proportional to the logarithm of the condition number, but that of `NearSum` with only one extra call of Algorithm 6.2 (`TransformK`) is not. However, the computing time of `AccSum` to compute a faithfully rounded result is proportional to $t_f := \log(\text{cond}(\sum p_i))$, but that of `NearSum` is proportional to $t_N := \log(\text{cond}(R - \delta' + \sum p'_i))$, where $R - \delta' + \sum p'_i$ is the difference between the exact sum s and one of the "switching points" $\mathcal{M}^-(\texttt{res})$ or $\mathcal{M}^+(\texttt{res})$. Note that, subject to the size of the exponent range, t_N/t_f can be arbitrarily large.

Finally we mention that combining the results of Sections 6 and 7 we can easily define an algorithm producing a sequence of K floating-point numbers representing a K-fold rounded-to-nearest result of $s = \sum p_i$. Consider Algorithm 6.4 (`AccSumK`) changed in such a way that only the last member $\texttt{Res}_K$ of the sequence $(\texttt{Res}_1, \cdots, \texttt{Res}_k)$ is computed in rounding-to-nearest using a piece of code similar to that in Algorithm 7.4

(`NearSum`). Then (5.14) implies in the notation of the proof of Proposition 5.6

$$|s - \sum_{k=1}^{K} \mathtt{Res}_k| = |\Delta_K| \le \mathtt{eps} \cdot \mathrm{ufp}(\Delta_{K-1}) \le \mathtt{eps}^K \mathrm{ufp}(s) \le \mathtt{eps}^K |s| \ .$$

Hence $\sum \mathtt{Res}_k$ is of the same accuracy as a nearest floating-point approximation in a floating-point grid with relative rounding error unit $\mathtt{eps}^K$, and the last member $\mathtt{Res}_K$ is $s - \sum_{k=1}^{K-1} \mathtt{Res}_k$ rounded-to-nearest.

8. Vectors of huge length.

In IEEE 754 double precision our summation algorithm computes a faithful rounding of the exact result for up to about $6.7 \cdot 10^7$ summands, this restriction imposed by the assumption $2^{2M}\mathtt{eps} \le 1$. This should suffice for most practical purposes. In IEEE 754 single precision with $\mathtt{eps} = 2^{-24}$, the number of summands is restricted to $n = 4094$, which may be an obstacle for the application of our algorithms.

In Section 4 we showed that using $2^M\mathtt{eps}$ for the parameter Φ in Algorithm 3.3 (`Transform`) we have enough information to calculate the sign of a sum. This was also true for huge vector lengths. Next we show how to compute a faithfully rounded result of the sum for huge vector lengths. For this we continue to extract p'_i by `Transform` until the error term of the summation of the extracted vector is small enough compared to τ_1. This is done in the following Algorithm 8.1 (`AccSumHugeN`).

Algorithm 8.1. *Accurate summation with faithful rounding for huge n.*

```
function res = AccSumHugeN($p$)
    $[\tau_1, \tau_2, q^{(0)}, \sigma, M] = \mathtt{Transform}(p, 0)$          % Parameter $\Phi$ replaced by $2^{M+3}\mathtt{eps}$
    if $\sigma \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$, $\mathtt{res} = \tau_1$, return, end if
    $k = 0$; $\sigma_0 = \mathrm{fl}(2^M \mathtt{eps}\sigma)$
    repeat
        $k = k + 1$
        $[\tau^{(k)}, q^{(k)}] = \mathtt{ExtractVector}(\sigma_{k-1}, q^{(k-1)})$
        $\sigma_k = \mathrm{fl}(2^M \mathtt{eps}\sigma_{k-1})$
    until  $|\tau_1| \ge \mathrm{fl}\big((2^{2M+1}\mathtt{eps})\sigma_{k-1}\big)$   or   $\sigma_{k-1} \le \frac{1}{2}\mathtt{eps}^{-1}\mathtt{eta}$
    $\tilde\tau^{(1)} = \mathrm{fl}\big(\tau^{(1)} + (\tau^{(2)} + \cdots + (\tau^{(k)} + (\tau_2 + (\sum_{i=1}^n q_i^{(k)}))) \cdots)\big)$
    $\mathtt{res} = \mathrm{fl}(\tau_1 + \tilde\tau^{(1)})$
```

Proposition 8.2. *Let p be a vector of n floating-point numbers, assume $2^{M+3}\mathtt{eps} \le 1 < 2^{2M}\mathtt{eps}$ and $\mathtt{eps} \le 1/512$. Let `res` be the result of Algorithm 8.1 (`AccSumHugeN`) applied to p.*

Then `res` is a faithful rounding of $s := \sum_{i=1}^n p_i$. The algorithm needs $(4m + 4K + 3)n + \mathcal{O}(m + K)$ flops for m "repeat-until"-loops within `Transform` and K denoting the final value of k.

Remark 1. For IEEE 754 single precision with $\mathtt{eps} = 2^{-24}$ this limits the vector length to a little over 2 million rather than $n = 4094$ for Algorithm `AccSum`. For IEEE 754 double precision the vector length is now limited to about $1.1 \cdot 10^{15}$.

Remark 2. For huge vector lengths near the admissible maximum, however, the algorithm becomes rather inefficient because few bits, possibly only three bits, are extracted in each execution of `Transform` .

Remark 3. We mention that the parameters can be adjusted so that the weaker assumption $2^{M+2}\mathtt{eps} \le 1$ suffices. However, this extracts ultimately only two bits at a time.

Remark 4. The assumption $1 < 2^{2M}\mathtt{eps}$ imposes no restriction because otherwise Algorithm 4.5 (`AccSum`) in Part I can (and for better performance should) be used.

Proof of Proposition 8.2. Algorithm `Transform` is called with $\varrho = 0$. So $2^M\texttt{eps} \le \frac{1}{8}$ and Lemma 3.4 imply

$$s = \tau_1 + \tau_2 + \sum_{i=1}^{n} q_i^{(0)} \ , \tag{8.1}$$

$$\max |q_i^{(0)}| \le \texttt{eps}\sigma, \tag{8.2}$$

$$\tau_1 = \mathrm{fl}(\tau_1 + \tau_2) \quad \text{and} \quad |\tau_2| \le \texttt{eps}|\tau_1| \ . \tag{8.3}$$

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

$$\varphi = 2^M\texttt{eps} \le \frac{1}{8} \ , \ \ \texttt{eps} \le \frac{1}{512} \ , \ \ M \ge 5 \quad \text{and} \quad \sigma_k = \varphi^{k+1}\sigma \ \text{ for } \ 0 \le k \le K \ , \tag{8.4}$$

and the choice of Φ and (3.5) gives

$$|\tau_1| \ge 8\varphi\sigma \ . \tag{8.5}$$

We first show that we may assume

$$\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta} \qquad \text{and} \qquad |\tau_1| \ge \texttt{eps}^{-1}\texttt{eta} \tag{8.6}$$

without loss of generality. To see that, first assume that σ, computed by Algorithm `Transform`, is in the underflow range. Then $q^{(0)}$ is the zero vector by Lemma 3.4, and $\mathrm{fl}(s) = \texttt{res}$ by (8.1). Second, assume $|\tau_1| < \texttt{eps}^{-1}\texttt{eta}$. Then (8.5) implies $8\varphi\sigma \le |\tau_1| < \texttt{eps}^{-1}\texttt{eta}$, so that by $\sigma_0 = \varphi\sigma < \frac{1}{8}\texttt{eps}^{-1}\texttt{eta}$ the "repeat-until"-loop is finished for $k = 1$. Moreover $|\tau_2| \le \texttt{eps} \cdot \mathrm{ufp}(\tau_1) \le \frac{1}{2}\texttt{eta}$ implies $\tau_2 = 0$, and by Theorem 3.2, $|q_i^{(1)}| \le \texttt{eps}\sigma_0$, so that the vector $q_i^{(1)}$ is identically zero. Therefore $\texttt{res} = \mathrm{fl}(\tau_1 + \tau^{(1)}) = \mathrm{fl}(s)$ by (8.7), and `res` is again a faithful rounding of s.

By definition of σ_0 and (8.2), $\max_i |q_i^{(0)}| \le 2^{-M}\sigma_0$, so the assumptions of Theorem 3.2 are satisfied for the first call of `ExtractVector` with $k = 1$. Therefore,

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

If $\sigma_0 \le \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then the loop finishes and a possible rounding error in the computation of σ_1 does no harm since σ_1 is not used. If $\sigma_0 > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$, then σ_1 is computed without rounding error and can safely be used. Note that the assumption $1 < 2^{2M}\texttt{eps}$ implies that no rounding error occurs in the computation of $\mathrm{fl}\big((2^{2M+1}\texttt{eps})\sigma_{k-1}\big)$ in the "until"-condition.

Again, the assumptions of Theorem 3.2 are satisfied for the next call of `ExtractVector`, and repeating this argument shows

$$s = \tau_1 + \tau_2 + \sum_{\nu=1}^{k} \tau^{(\nu)} + \sum_{i=1}^{n} q_i^{(k)} \ , \ \ |\tau^{(k)}| < \sigma_{k-1} \quad \text{and} \quad \max_i |q_i^{(k)}| \le 2^{-M}\sigma_k \tag{8.7}$$

for k between 1 and its final value K. The same argument as before applies to possible rounding errors in the computation of σ_K. Denote

$$\tau^{(K+1)} := \tau_2 + \big(\sum_{i=1}^{n} q_i^{(K)}\big) \quad \text{and} \quad \tilde{\tau}^{(K+1)} := \mathrm{fl}\big(\tau_2 + (\sum_{i=1}^{n} q_i^{(K)})\big) = \tau^{(K+1)} - \delta_{K+1}, \tag{8.8}$$

$$\tilde{\tau}^{(k)} := \mathrm{fl}\big(\tau^{(k)} + \tilde{\tau}^{(k+1)}\big) = \tau^{(k)} + \tilde{\tau}^{(k+1)} - \delta_k \quad \text{for } 1 \le k \le K \ . \tag{8.9}$$

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

$$\tilde{\tau}^{(1)} = \sum_{k=1}^{K+1} \tau^{(k)} - \sum_{k=1}^{K+1} \delta_k = \sum_{k=1}^{K} \tau^{(k)} + \tau_2 + \sum_{i=1}^{n} q_i^{(K)} - \sum_{k=1}^{K+1} \delta_k \ . \tag{8.10}$$

Denote $\tau' := \sum q_i^{(K)}$ and $\tilde\tau' := \mathrm{fl}\big(\sum q_i^{(K)}\big)$. Then by definition (8.8), (8.7), (2.7) and $n+2 \le 2^M$,

$$\begin{aligned} |\delta_{K+1}| &= |\tilde\tau^{(K+1)} - \tau^{(K+1)}| = |\mathrm{fl}(\tau_2+\tilde\tau') - (\tau_2+\tilde\tau') + (\tau_2+\tilde\tau') - (\tau_2+\tau')| \\ &\le \mathtt{eps}|\tau_2 + \mathrm{fl}\big(\textstyle\sum q_i^{(K)}\big)| + |\mathrm{fl}\big(\textstyle\sum q_i^{(K)}\big) - \textstyle\sum q_i^{(K)}| \\ &\le \mathtt{eps}\,\big(|\tau_2| + n2^{-M}\sigma_K + \tfrac12 n(n-1)2^{-M}\sigma_K\big) \\ &< \mathtt{eps}\,\big(|\tau_2| + 2^{M-1}\sigma_K\big)\ . \end{aligned} \tag{8.11}$$

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

$$|\tilde\tau^{(k)}| \le \Big(\sum_{\nu=0}^{K+1-k}(1+\mathtt{eps})^{\nu+1}\varphi^\nu\Big)\sigma_{k-1} + (1+\mathtt{eps})^{K+2-k}|\tau_2| \quad \text{for } 1 \le k \le K+1\ . \tag{8.12}$$

For $k = K+1$ this follows by (8.5)

$$|\tilde\tau^{(K+1)}| \le (1+\mathtt{eps})|\tau_2 + \mathrm{fl}\big(\textstyle\sum q_i^{(K)}\big)| \le (1+\mathtt{eps})(|\tau_2| + n2^{-M}\sigma_K) < (1+\mathtt{eps})(|\tau_2|+\sigma_K)\ ,$$

again using (2.7), and the induction step uses

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

by (8.9), and $|\tau^{(k)}| < \sigma_{k-1}$ by (8.7). Hence (8.4) gives

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

and by (8.9) and (8.4) it follows

$$\mathtt{eps}^{-1}|\delta_k| \le |\tilde\tau^{(k)}| \le \frac{147}{128}\varphi^k\sigma + (1+\mathtt{eps})^{K+2-k}|\tau_2| \quad \text{for } 1\le k\le K\ . \tag{8.13}$$

Next we need an upper bound on K, the final value of k. Denote $\mathtt{eps} = 2^{-m}$, then (8.4) gives $m \ge 9$ and $M - m \le -3$. If $K \ge 2$, then the "until"-condition, (8.5) and (8.4) imply $8\varphi\sigma \le |\tau_1| < 2^{2M+1}\mathtt{eps}\sigma_{K-2} = 2^{M+1}\sigma_{K-1} = 2^{M+1}\varphi^K\sigma$, so that $8\varphi \le 2^M\varphi^K$. Hence $K-1 \le \frac{M-3}{m-M}$, so that

$$K \le \frac{M}{3} \quad\text{and}\quad K \le \frac{m-3}{3}\ , \tag{8.14}$$

which is by (8.4) also true for $K<2$. Therefore, $m \ge 9$ yields

$$(1+\mathtt{eps})^K \le (1+2^{-m})^{\frac{m-3}{3}} < \frac{129}{128} \quad\text{and}\quad (1+\mathtt{eps})^2\frac{(1+\mathtt{eps})^K-1}{\mathtt{eps}} \le \frac{1}{128}\mathtt{eps}^{-1}\ . \tag{8.15}$$

Now the definition of res, (8.13), (8.3), (8.15), (8.4) and (8.5) yield

$$\begin{aligned} |\mathtt{res}| &\ge (1-\mathtt{eps})\big(|\tau_1| - |\tilde\tau^{(1)}|\big) \\ &\ge (1-\mathtt{eps})\big(|\tau_1| - \tfrac{147}{128}\varphi\sigma - (1+\mathtt{eps})^{K+1}\mathtt{eps}|\tau_1|\big) \\ &\ge \tfrac{127}{128}|\tau_1| - \tfrac{147}{128}\varphi\sigma\ . \end{aligned} \tag{8.16}$$

Hence (8.5) yields

$$|\mathtt{res}| > \frac{217}{256}|\tau_1|\ , \tag{8.17}$$

and (8.6) implies $\mathtt{res} \notin \mathbb{U}$. Now set $r := \tau_1 + \tilde\tau^{(1)}$, so that $\mathtt{res} = \mathrm{fl}(r)$. Then (8.7) and (8.10) imply

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

TABLE 8.1
Minimum treatable vector length n by Algorithm 8.1 depending on K

K	single precision	double precision
1	8190	$1.3 \cdot 10^{8}$
2	65534	$6.8 \cdot 10^{10}$
3	$2.6 \cdot 10^{5}$	$1.0 \cdot 10^{12}$
4	$5.2 \cdot 10^{5}$	$4.3 \cdot 10^{12}$
5	$1.0 \cdot 10^{6}$	$1.7 \cdot 10^{13}$

We will show $2|\delta| < \mathtt{eps}|\mathtt{res}|$ and apply Lemma 2.3 to demonstrate that `res` is a faithful rounding of the true result s. First we need to bound δ_{K+1}. If $\sigma_{K-1} \in \mathbb{U}$, then (8.7) implies $q_i^{(K)} = 0$ for all i, and (8.8) gives $\delta_{K+1} = 0$. Otherwise the "until"-condition and (8.11) yield

$$2^{M-1}\sigma_K = 2^{2M-1}\mathtt{eps}\sigma_{K-1} \le \frac{1}{4}|\tau_1| \quad \text{and} \quad \mathtt{eps}^{-1}|\delta_{K+1}| \le (\mathtt{eps} + \frac{1}{4})|\tau_1| \ .$$

Hence (8.13), (8.15), $\mathtt{eps} \le \frac{1}{512}$ and $\varphi\sigma \le \frac{1}{8}|\tau_1|$ show

$$\begin{aligned} \mathtt{eps}^{-1}|\delta| &< \tfrac{147}{128}\tfrac{\varphi}{1-\varphi}\sigma + (1+\mathtt{eps})^2\tfrac{(1+\mathtt{eps})^K-1}{\mathtt{eps}}|\tau_2| + \mathtt{eps}^{-1}|\delta_{K+1}| \\ &\le \tfrac{147}{112}\varphi\sigma + \left(\tfrac{1}{128} + \mathtt{eps} + \tfrac{1}{4}\right)|\tau_1| \\ &\le \tfrac{217}{512}|\tau_1| \ , \end{aligned} \tag{8.18}$$

and (8.17) gives $2|\delta| < \mathtt{eps}|\mathtt{res}|$. Therefore $\mathtt{res} \notin \mathbb{U}$ and Lemma 2.3 prove that `res` is a faithful rounding of s. The proof is finished. □

With (8.14) we see that for IEEE 754 double precision at most $K = 17$ extractions are possible, and we can also estimate the minimum treatable vector length n by Algorithm 8.1 (`AccSumHugeN`) depending on the number of extra extractions. Suppose the "repeat-until"-loop in Algorithm 8.1 (`AccSumHugeN`) has been executed K times, and assume

$$M \le \frac{mK+2}{K+1} \ . \tag{8.19}$$

Then the first call of `Transform` implies $|\tau_1| \ge 8\varphi\sigma$ and a little computation using $\mathtt{eps} = 2^m$ yields

$$|\tau_1| \ge 2^{M+3-m} \ge 2^{2M+1}\mathtt{eps}\varphi^K \ ,$$

so that the "until"-condition is satisfied. That means, if M satisfies (8.19), then the "repeat-until"-loop in `AccSumHugeN` is executed at most K times. It follows the *minimum* treatable vector lengths n with K loops, which are summarized in Table 8.1. As expected, `AccSumHugeN` becomes inefficient for a vector length approaching $\mathtt{eps}^{-1}$; in IEEE 754 single precision about 10^6 and in double precision about $2 \cdot 10^{13}$ may be a reasonable limit for n. As before an algorithm with K-fold faithful rounding and rounding-to-nearest for input vectors of huge length may be developed as well.

9. Computational results.

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

All algorithms are tested in three different environments, namely Intel Pentium 4, Intel Itanium 2 and AMD Athlon 64. We carefully choose compiler options to achieve best possible results, see Table 9.1.

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 $q_i = \mathrm{fl}\left((\sigma + p_i) - \sigma\right)$ in Algorithm 3.1 (`ExtractVector`), which is optimized into $q_i = p_i$. This can, of course, be avoided by setting appropriate compiler options; however, this may slow down the whole computation. In this specific case the

Table 9.1
Testing environments

	CPU, Cache sizes Peak performance (for summation)	Compiler, Compile options Memory bandwidth (approx.)
I)	Intel Pentium 4 (2.53GHz)	Intel Visual Fortran 9.1
	L2: 512KB	`/O3 /QaxN /QxN [/Op`, see Table 9.2`]`
	2.53GFlops	1GB/s
II)	Intel Itanium 2 (1.4GHz)	Intel Fortran 9.0
	L2: 256KB, L3: 3MB	`-O3`
	2.8GFlops	2.6GB/s
III)	AMD Athlon 64 (2.2GHz)	GNU gfortran 4.1.1
	L2: 512KB	`-O3 -fomit-frame-pointer`
		`-march=athlon64 -funroll-loops`
	2.2GFlops	3GB/s

Table 9.2
Compile options for Pentium 4, Intel Visual Fortran 9.1

Algorithm	Necessity of compile option `/Op`
`SSum`	No
`Sum2`	Yes, for `TwoSum`
`XBLAS`	Yes, for `TwoSum` and `FastTwoSum`
`Priest`, `PriestS`	Yes, for `FastTwoSum`
`Malcolm`, `MalcK`, `MalcN`, `MalcS`, `LAccu`, `LAccuK`, `LAccuN`, `LAccuS`	Yes, for `Split`
`AccSum`, `AccSumK`, `NearSum`, `AccSumHugeN`, `AccSign`	Basically, no

second author suggested a simple trick to overcome this by using $q_i = \mathrm{fl}\left(|\sigma + p_i| - \sigma\right)$ instead. This does not change the intended result since $|p_i| \le \sigma$ is assumed in the analysis (Theorem 3.2), it avoids unintended compiler optimization, and it does not slow down the computation. For the other algorithms to be tested we had to use, however, 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 9.2.

To test the algorithms presented in this paper, examples for huge condition numbers larger than $\mathtt{eps}^{-1}$ were generated by Algorithm 6.1 in [18], where a method to generate a vector whose summation is arbitrarily ill-conditioned is described. All tests are performed in IEEE 754 double precision except those for `AccSumHugeN`, which are performed in IEEE 754 single precision.

First, test results for Algorithm 6.4 (`AccSumK`) for K-fold faithfully rounded results are presented. For all examples we choose dimension $n = 1000$ and small condition number around 10^3 or so and watch the effect of increasing values of K. If the condition number of the initial sum and of sums of intermediately extracted vectors does not exceed $\mathtt{eps}^{-1}$ significantly, we may expect the computing time to grow linearly in K.

Competitors for K-fold rounding are Malcolm's summation [16] and the long accumulator [12]. In fact, both approaches produce the bit representation of the exact result in an intermediate step: Malcolm's in an overlapping sequence of floating-point numbers, and the long accumulator in a non-overlapping sequence. From this it is not too difficult to extract a result of K-fold accuracy; the corresponding algorithms are denoted by `MalcK` and `LAccuK`, respectively. In Table 9.3 we normed the computing time of `AccSum` to 1, practically the same computing time as for `AccSumK` for $K = 1$.

Indeed we observe the linear dependency of `AccSumK` in K. For `MalcK` and `LAccuK` there is not too much dependency on K since the exact result of the sum is computed anyway; only some additional effort to

TABLE 9.3
Measured computing times for `AccSumK`*,* $n = 1000$*, cond* $\sim 10^3$*, for all environments time of* `AccSum` *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		
K	`MalcK`	`LAccuK`	`AccSumK`	`MalcK`	`LAccuK`	`AccSumK`	`MalcK`	`LAccuK`	`AccSumK`
1	17.5	91.7	1.0	14.4	54.1	1.0	6.0	28.0	1.0
2	17.1	89.8	1.6	15.5	57.4	1.4	6.1	28.8	1.8
3	18.4	89.8	2.2	15.5	57.1	1.8	6.2	28.4	3.1
4	19.2	89.8	3.4	15.9	57.4	2.5	6.3	28.6	3.9
5	20.0	91.8	3.9	16.2	57.6	2.9	6.5	29.1	4.8
6	22.4	89.8	4.6	16.5	57.5	3.4	6.7	28.9	5.6
7	23.7	91.8	5.7	16.9	57.7	4.1	6.9	28.8	6.9
8	27.5	95.8	6.4	17.4	57.9	4.5	7.2	28.9	7.7
9	30.0	93.8	7.0	17.9	58.1	4.9	7.6	29.3	8.5
10	32.2	93.9	8.0	18.5	58.4	5.6	7.9	29.5	9.8

TABLE 9.4
Measured computing times for `NearSum`*,* $n = 1000$*, for all environments time of* `AccSum` *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		
$\text{cond}_{\text{near}}$	`MalcN`	`LAccuN`	`NearSum`	`MalcN`	`LAccuN`	`NearSum`	`MalcN`	`LAccuN`	`NearSum`
10^{8}	17.3	92.2	1.7	14.7	57.2	1.7	5.5	28.3	1.8
10^{16}	16.9	92.3	1.7	14.7	58.0	1.7	5.5	27.5	1.8
10^{24}	16.5	92.3	2.2	14.9	58.4	2.0	5.7	28.2	2.3
10^{32}	17.3	94.1	2.2	14.9	58.4	2.0	5.7	27.9	2.3
10^{40}	16.6	90.6	2.7	15.4	58.8	2.5	5.9	29.0	2.8
10^{48}	17.3	92.3	3.2	15.4	59.0	2.6	5.9	28.3	3.2
10^{56}	17.3	94.2	3.1	15.5	58.9	2.7	6.1	29.1	3.2
10^{64}	17.7	92.3	3.7	15.7	61.7	2.9	6.4	29.1	3.7

extract from this a sequence of K floating-point numbers is necessary. On AMD Athlon 64 and for large K, Malcolm's approach is superior to `AccSumK`. Note that the accuracy of the results $\sum \texttt{Res}_k$ may in fact be better than K-fold since there may be sequences of adjacent zeros in the bit representation of the exact result, producing gaps between adjacent $\texttt{Res}_k$.

Next we tested `NearSum`. Challenging examples for rounding to nearest have an exact sum near the midpoint of two adjacent floating-point number. The usual condition number for the sum $\sum p_i$ is $\sum |p_i| / |\sum p_i|$, cf. (4.25) in Part I of this paper, measuring the amount of cancelation. For rounding to nearest we may use

$$\text{cond}_{\text{near}} := \frac{\sum |p_i|}{\min\{|\frac{1}{2}(f + \text{succ}(f)) - \sum p_i| : f \in \mathbb{F}\}} , \tag{9.1}$$

measuring the nearness of the sum to a "switching-point" for rounding to nearest. In Table 9.4 we normed again the computing time for `AccSum` to 1, so the additional amount of computing time to go from faithful rounding to rounding to nearest is monitored. Again, Malcolm's algorithm and the long accumulator compute a bit representation of the exact sum, and from there it is not difficult to derive the rounded-to-nearest result. The corresponding algorithms are denoted by `MalcN` and `LAccuN`, respectively. In all examples we choose the dimension $n = 1000$.

For small condition number the ratio of computing times for `NearSum` shows the expected factor 2 compared

Table 9.5

Measured computing times for `AccSumHugeN` *in single precision, cond* $= 10^8$*, time of* `SSum` *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`	`Malcolm`	`LAccu`	`AccSum`	`Sum2`	`XBLAS`	`Malcolm`	`LAccu`	`AccSum`
10,000	40.0	150.0	888.9	777.8	25.6	7.9	55.9	858.3	930.5	32.5
20,000	32.7	122.7	772.7	681.8	20.0	7.9	55.6	900.0	940.5	32.8
40,000	32.7	122.7	681.8	681.8	23.6	7.9	55.6	867.6	940.5	36.9
80,000	30.8	108.3	911.5	607.6	45.8	6.9	48.3	1058.8	816.7	35.8
160,000	4.5	15.8	125.5	94.1	21.6	6.4	44.4	961.5	743.4	36.1
320,000	3.6	12.7	176.9	70.8	23.3	6.0	40.4	1398.7	682.0	48.3
640,000	3.6	13.3	176.1	76.6	29.3	4.3	28.4	965.1	472.0	53.4
1,280,000	3.5	13.0	242.3	71.9	36.9	4.2	26.5	1253.2	445.1	63.0

to `AccSum` caused by the additional call of `TransformK`. Also as expected, the computing time of `MalcN` and `LAccuN` is almost independent of the condition number. For $n = 1000$ some 43 bits are extracted in double precision at a time. Hence for condition number 10^{64} which is about 2^{213} we need some 5 extractions, and this is reflected in the computing time ratio of `NearSum` to `AccSum` in Table 9.4. Note that condition numbers exceeding 10^{16} occur only in very special applications.

In Figures 5.1 and 5.3 in Part I of this paper we displayed the MFlop-rates for different algorithms. They showed that `AccSum` achieves a much better MFlop-rate than Malcolm's or long accumulator algorithm. For `AccSumK` and `NearSum` this is similar; for small condition numbers they achieve about 85% to 95%, for larger ones between 105% up to 160% and a little more of the MFlop-rate of `AccSum`. This corresponds to about 50% to 80% of the peak performance.

Next we tested `AccSumHugeN`. For double precision we may use `AccSum` until dimension $n = 6.4 \cdot 10^7$. For such large dimensions we would basically measure cache misses rather than performance of the algorithms. Therefore we rewrote all algorithms in single precision. We use the same names `Malcolm`, `LAccu` and `AccSum` as before, so `AccSum` in the following Tables 9.5 and 9.6 refers to `AccSumHugeN`. For increasing dimension fewer and fewer bits can be extracted by `AccSumHugeN` at a time, thus requiring more and more extractions. We tested dimensions from $n = 10,000$ up to $n = 1,280,000$, which is the range of applicability of `AccSumHugeN` where the dimension is too large for `AccSum`. All examples are generated to have condition number 10^8.

The results for Pentium 4 and Itanium 2 architectures are displayed in Table 9.5, where now the computing time of `SSum`, the ordinary single precision recursive summation, is normed to 1. Moreover, the ratio of computing time for single precision `Sum2` [18] and `XBLAS` [14, 1] to `SSum` is displayed. Note that both deliver a result "as if" calculated in twice single precision (i.e. 48 bits precision). So for condition number 10^8 we can expect almost full accuracy of the result, but for condition numbers $2^{48} \sim 2 \cdot 10^{14}$ and above we cannot expect a single correct digit of the computed result; hence the comparison to the other algorithms is not quite fair.

As in Part I of this paper we observed a significant performance drop for larger dimensions due to cache misses. So, as also explained in Part I, algorithms `Sum2` and `XBLAS` do not become relatively faster, but the reference `SSum` gets abruptly slower at a certain dimension. As expected, the computing time for `AccSum` grows slowly with increasing dimension. Both `Malcolm` and `LAccu` suffer severely from the small sizes of the internal accumulators. Note that although `AccSum` computes a result of much better quality, namely a faithfully rounded result, it is faster than `XBLAS` up to dimensions where cache misses appear.

The results for AMD Athlon architecture are displayed in Table 9.6, left the ratio of computing times relative

TABLE 9.6
Test `AccSumHugeN` *in single precision on AMD Athlon 64 (2.2GHz), GNU gfortran 4.1.1, cond* $= 10^8$. *Left: Measured computing times, time of* `SSumU` *normed to 1, Right: Measured MFlops*

n	Sum2	XBLAS	Malcolm	LAccu	AccSum	SSum	SSumU	Malcolm	AccSum
10,000	12.5	31.1	203.5	325.4	43.5	333	2193	119	1159
20,000	11.0	26.1	170.0	272.3	33.8	548	1838	119	1253
40,000	9.6	23.4	151.9	243.9	37.6	551	1645	119	1180
80,000	9.5	23.0	205.3	240.4	39.8	551	1623	158	1264
160,000	5.3	12.6	111.4	130.1	25.9	539	874	157	1182
320,000	4.6	11.0	157.6	114.0	27.3	539	767	185	1207
640,000	4.5	10.9	156.2	112.4	34.3	536	757	184	1216
1,280,000	4.4	10.9	222.2	112.1	45.2	538	756	191	1189

to recursive summation `SSumU` with unrolled loops, right MFlops. As explained in Part I this is the only architecture out of the three where unrolled loops speed up recursive summation `SSum`. For $n = 10,000$ the speed up is a factor 6.6, i.e. `SSumU` is more than 6 times faster than `SSum`. We programmed also the other algorithms `XBLAS` etc. with unrolled loops, but observed almost no difference.

Again there is a drop in performance at a certain dimension. The MFlop-rates are displayed in the right half of Table 9.6. For `Sum2`, `XBLAS` and `LAccu` those are 1200, 690 and 200, respectively, for all dimensions, so they are not displayed. So the MFlop-rate of `AccSum` is significantly slower than that of `SSumU` for small dimension, and becomes superior for larger dimensions.

Finally we tested `AccSign`. To compute the sign of a sum of floating-point numbers is very simple from the intermediate result of Malcolm's or the long accumulator algorithm. Since a bit representation of the exact sum is available, the sign is immediately available. However, the exact sum is *always* computed, no matter how well- or ill-conditioned the problem is.

For `AccSign` we use the weak "until"-condition with the factor $2^M\texttt{eps}$ rather than $2^{2M}\texttt{eps}$ in `AccSum`. We proved that this is weakest possible, just sufficient to guarantee the correct sign. Another competitor is now Priest's doubly compensated summation. The computed approximation `res` of $s := \sum p_i$ satisfies [19, 20] the error estimate $|\texttt{res} - s| \le 2\texttt{eps}|s|$, therefore $\operatorname{sign}(\texttt{res}) = \operatorname{sign}(s)$.

The adapted algorithms by Malcolm and Priest to compute $\operatorname{sign}(s)$ are denoted by `MalcS` and `PriestS`, respectively. To save space we omit results for the long accumulator. The following Table 9.7 shows the results for fixed vector length $n = 1000$. The first column "cond" denotes the condition number of the sum; all computations are performed in double precision. Note again that condition numbers up to 10^{16} are the generic case.

As can be seen from Table 9.7, where the time for `AccSum` is normed to 1, the improved stopping criterion in `AccSign` pays for non-extreme condition numbers. The improvement becomes negligible for huge condition numbers. The corresponding ratios for the long accumulator `LAccuS` are at best 38, 20 and 10 for the three architectures and largest condition number 10^{64}, and for "small" condition number, the generic case, about 110, 51 and 30, respectively.

Finally we tested our algorithms for the special case of zero sums. For Malcolm's, the long accumulator or Priest's algorithm this does not make much difference, but for `AccSign` and `AccSum` it does. In this case the input vector has to be extracted completely, until the final extracted vector is entirely zero. Note that the condition number of a zero sum is infinity.

The results are displayed in Table 9.8, where the time for `AccSign` is normed to 1. All computations are performed in double precision. The first column "Exp. range" depicts $53(1 - \log_{\texttt{eps}}(e_{max} - e_{min}))$, where

TABLE 9.7
Measured computing times for `AccSign`, $n = 1000$, *for all environments time of* `AccSum` *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		
cond	`PriestS`	`MalcS`	`AccSign`	`PriestS`	`MalcS`	`AccSign`	`PriestS`	`MalcS`	`AccSign`
10^{8}	35.4	10.1	0.55	36.8	9.5	0.71	13.8	3.6	0.43
10^{16}	35.4	10.0	0.91	36.4	9.5	0.94	13.8	3.5	0.74
10^{24}	26.7	7.6	0.93	29.3	7.7	0.95	10.5	2.7	0.78
10^{32}	23.6	6.8	0.95	29.4	7.8	0.95	10.5	2.7	0.78
10^{40}	18.2	5.6	0.95	18.7	5.0	0.97	7.8	2.0	0.83
10^{48}	17.3	5.2	0.96	15.0	4.0	0.97	7.0	1.8	0.68
10^{56}	17.4	5.3	0.96	15.0	4.0	0.97	7.0	1.8	0.83
10^{64}	17.5	5.4	0.95	15.1	4.0	0.97	5.7	1.5	0.85

TABLE 9.8
Measured computing times for zero sums, $n = 1000$, *for all environments time of* `AccSign` *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		
Exp. range	`PriestS`	`MalcS`	`LAccu`	`PriestS`	`MalcS`	`LAccu`	`PriestS`	`MalcS`	`LAccu`
106	33.9	9.3	63.9	24.2	7.0	32.6	15.8	3.9	22.4
159	32.0	9.2	61.8	24.7	7.0	32.2	15.3	3.9	22.1
212	21.2	6.2	42.8	16.7	4.6	21.6	10.0	2.7	14.9
265	14.6	4.5	30.2	12.3	3.5	16.1	7.6	2.0	11.1
318	12.3	3.8	24.2	9.7	2.8	12.8	6.1	1.7	8.9
371	10.1	3.2	20.6	8.3	2.3	10.6	4.8	1.3	7.3
424	8.3	2.6	17.0	7.0	2.0	9.1	4.1	1.1	6.2
477	7.2	2.3	15.2	6.2	1.7	8.0	3.6	1.1	5.5
530	6.6	2.1	13.8	5.6	1.5	7.1	3.2	1.0	4.9
583	6.1	1.9	12.6	4.8	1.4	6.4	2.9	0.9	4.5

e_{max} and e_{min} denote the largest and smallest exponent of the summands, so basically 53 times the number of bits covered by the summands. Zero sums are fortunate for Malcolm's and the long accumulator approach because the *exact* sum *has* to be computed. This is seen from the computational results. Here for AMD Athlon architecture Malcolm's algorithm outperforms `AccSign` for large exponent range.

Finally we display the achieved MFlop-rates for Itanium and Athlon architecture in Figure 9.1; for Pentium 4 it looks similarly. As can be seen, `AccSign` achieves for not too large exponent range a little better rate than `AccSum`. The MFlop-rates for the other algorithms do not change since in any case the exact bit representation is calculated.

10. Appendix.

Following we prove (6.5), that is we show

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

under the assumptions of Lemma 6.3 and for $\sigma > \frac{1}{2}\texttt{eps}^{-1}\texttt{eta}$. Note that (10.1) implies $R = \text{fl}(\tau_2 - \Delta) = \tau_2 - \Delta$. Since σ is a power of 2, we have

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

We already know by (6.4) that

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

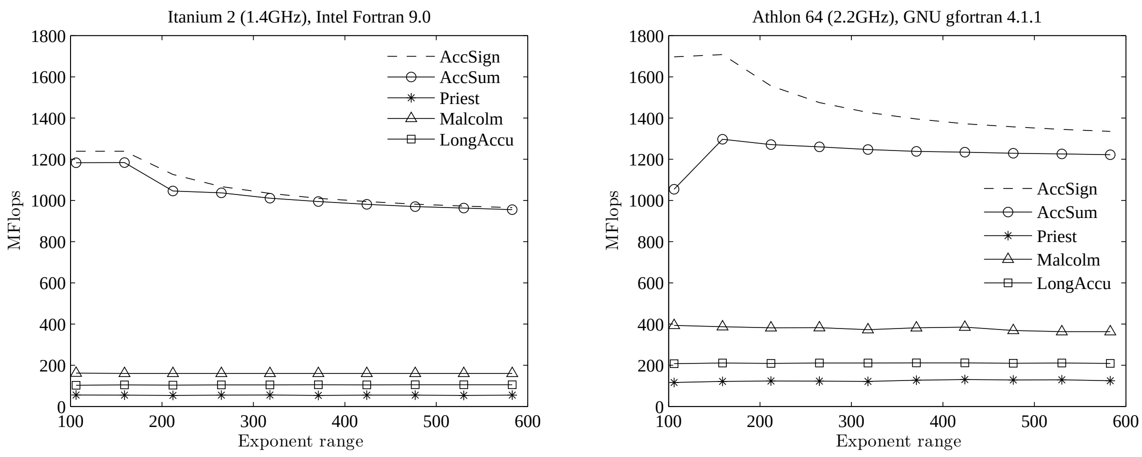

FIG. 9.1. *Measured MFlops for varying exponent ranges*

We first prove some facts, namely

$$\sigma \le |\tau_1| \quad \Rightarrow \quad \Delta, \tilde{R} \in \mathtt{eps}\sigma\mathbb{Z} \ , \tag{10.4}$$

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

$$\frac{1}{2}\mathtt{eps}^{-1}\sigma \le |\tau_1| \quad \text{and} \quad \mathtt{res} \ne \tau_1 \quad \Rightarrow \quad \tau_2 \cdot \Delta \ge 0 \quad . \tag{10.6}$$

If $\sigma \le |\tau_1|$, then $\mathtt{res} = \mathrm{fl}(\tau_1 + \tau_2')$, (2.8) and (2.4) yield $\mathtt{res} \in \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) \subseteq \mathtt{eps}\sigma\mathbb{Z}$, so $\mathtt{res}, \tau_1, \tau_2 \in \mathtt{eps}\sigma\mathbb{Z}$ from $\Delta, \tilde{R} \in \mathtt{eps}\sigma\mathbb{Z}$ and (3.9), proving (10.4).

Furthermore, (3.12), (3.14), (3.9) and the assumption $\frac{1}{2}\mathtt{eps}^{-1}\sigma \le |\tau_1|$ imply $\sigma \le 2\mathtt{eps} \cdot \mathrm{ufp}(\tau_1)$ and

$$\begin{aligned} |\tau_2'| &\le (1+\mathtt{eps})|\tau_2 + \tau_3| \le |\tau_2| + \mathtt{eps}|\tau_2| + (2^M - 2)\mathtt{eps}\sigma \\ &< |\tau_2| + 2^{M+1}\mathtt{eps}^2\mathrm{ufp}(\tau_1) \le |\tau_2| + 2^{-M+1}\mathtt{eps} \cdot \mathrm{ufp}(\tau_1) \\ &\le |\tau_2| + \frac{1}{2}\mathtt{eps} \cdot \mathrm{ufp}(\tau_1) \ , \end{aligned}$$

so that $\tau_1 = \mathrm{fl}(\tau_1 + \tau_2)$ and $\mathtt{res} = \mathrm{fl}(\tau_1 + \tau_2')$ imply (10.5). If $\mathtt{res} \ne \tau_1$, then $\mathrm{sign}(\tau_2) = \mathrm{sign}(\tau_2') = \mathrm{sign}(\mathtt{res} - \tau_1) = \mathrm{sign}(\Delta)$ by (10.3) and the monotonicity of rounding, so (10.6).

PROOF OF (10.1). We use the notation in Lemma 3.5, especially (3.12). We distinguish several cases.

First, assume $|\tau_1| < \sigma$. Then (3.9) yields $\tau_2 \in \mathtt{eps}\sigma\mathbb{Z}$ and $|\tau_2| \le \mathtt{eps} \cdot \mathrm{ufp}(\tau_1) < \mathtt{eps}\sigma$, so $\tau_2 = 0$ and $\tilde{R} = -\Delta \in \mathbb{F}$ by (10.3).

Second, assume $\sigma \le |\tau_1| < \frac{3}{5}\mathtt{eps}^{-1}\sigma$. Then (3.12), (2.9), (3.15) and (3.14) imply

$$\begin{aligned} |\tilde{R}| = |\tau_2 - \Delta| &= |\tau_2 - \tau_2' + \delta_1| \le |\tau_2 - \tau_2'| + \mathtt{eps}|\tau_1 + \tau_2'| \\ &\le |\tau_3| + \mathtt{eps}|\tau_2 + \tau_3| + \mathtt{eps}|\tau_1| + \mathtt{eps}(1+\mathtt{eps})|\tau_2 + \tau_3| \\ &\le \mathtt{eps}|\tau_1| + \mathtt{eps}(2+\mathtt{eps})|\tau_2| + (1+\mathtt{eps})^2|\tau_3| \\ &\le \mathtt{eps}(1+3\mathtt{eps})|\tau_1| + 2^M\mathtt{eps}\sigma \\ &< \sigma \ , \end{aligned} \tag{10.7}$$

and (2.5), (10.4) and (10.2) prove $\tilde{R} \in \mathbb{F}$.

Henceforth, we may assume without loss of generality $\mathtt{res} \ne \tau_1$ because otherwise $\Delta = \mathtt{res} - \tau_1 = 0$ and $\tilde{R} = \tau_2 \in \mathbb{F}$. For the remaining cases $\frac{3}{5}\mathtt{eps}^{-1}\sigma \le |\tau_1|$, so (10.5) yields

$$\mathtt{res} \in \{\mathrm{pred}(\tau_1), \mathrm{succ}(\tau_1)\} \ . \tag{10.8}$$

Third, assume $\frac{3}{5}\texttt{eps}^{-1}\sigma \le |\tau_1| < \texttt{eps}^{-1}\sigma$. Then $\mathrm{ufp}(\tau_1) = \frac{1}{2}\texttt{eps}^{-1}\sigma$ and

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

by Lemma 2.1. But $|\tau_2| \le \texttt{eps} \cdot \mathrm{ufp}(\tau_1) = \frac{1}{2}\sigma$, so that $\tau_2 \cdot \Delta \ge 0$ by (10.6) gives

$$|\tilde{R}| = |\tau_2 - \Delta| \le \sigma \ .$$

Hence (2.5), (10.4) and (10.2) prove $\tilde{R} \in \mathbb{F}$ also for that case.

Fourth and last, assume $|\tau_1| \ge \texttt{eps}^{-1}\sigma$. We first show that in Algorithm 3.3 (`Transform`) the "repeat-until"-loop is executed only once, i.e. the final value for m is $m = 1$. To establish a contradiction, suppose the final value of m satisfies $m \ge 2$. Then the "until-condition" with Φ replaced by $2^{2M}\texttt{eps}$ implies

$$|t^{(m-1)}| < \mathrm{fl}(2^{2M}\texttt{eps}\sigma_{m-2}) = 2^M\sigma_{m-1} = 2^M\sigma \le 2^{-M}\texttt{eps}^{-1}\sigma \le \frac{1}{4}\texttt{eps}^{-1}\sigma \ .$$

On the other hand, $\sigma = \sigma_{m-1}$, (3.3) and $|\tau_1| \ge \texttt{eps}^{-1}\sigma$ imply

$$|t^{(m-1)}| = |\tau_1 + \tau_2 - \tau^{(m)}| > (1 - \texttt{eps})|\tau_1| - \sigma \ge (1 - \texttt{eps})\texttt{eps}^{-1}\sigma - \sigma > \frac{1}{2}\texttt{eps}^{-1}\sigma \ ,$$

a contradiction. Hence $m = 1$ is the final value of m in Algorithm 3.3 (`Transform`), and (3.3) in Lemma 3.4 yields

$$\tau_1 + \tau_2 = \varrho + \tau^{(1)} \quad \text{and} \quad |\tau^{(1)}| \le (1 - 2^{-M})\sigma \ . \tag{10.9}$$

Next we prove that (10.8) implies

$$|\tau_1| = \texttt{eps}^{-1}\sigma \quad \text{and} \quad |\Delta| = \sigma \ . \tag{10.10}$$

Note that by $|\tau_1| \ge \texttt{eps}^{-1}\sigma$ the distance of τ_1 to its predecessor or successor is at least σ. We distinguish two cases. First, assume $\tau_1 = \varrho$. Then (10.9) implies $\tau_2 = \tau^{(1)}$ and therefore $|\tau_2| \le (1 - 2^{-M})\sigma$. Hence (3.12), (3.14) and $2^{2M}\texttt{eps} \le 1$ yield

$$|\tau_2'| \le (1 + \texttt{eps})|\tau_2 + \tau_3| \le (1 + \texttt{eps})\big(1 - 2^{-M} + n\texttt{eps}\big)\sigma \le (1 + \texttt{eps})\big(1 - 2^M\texttt{eps} + (2^M - 2)\texttt{eps}\big)\sigma < \sigma \ .$$

The assumption $|\tau_1| \ge \texttt{eps}^{-1}\sigma$, $\texttt{res} = \mathrm{fl}(\tau_1 + \tau_2') \ne \tau_1$ and Lemma 2.1 prove $|\tau_1| = \texttt{eps}^{-1}\sigma$, $|\texttt{res}| = \mathrm{pred}(|\tau_1|) = |\tau_1| - \sigma$ and $|\Delta| = |\texttt{res} - \tau_1| = \sigma$, and thus (10.10). Second, assume $\tau_1 \ne \varrho$. In this case $|\tau^{(1)}| < \sigma$ by Lemma 3.4, $|\tau_1| \ge \texttt{eps}^{-1}\sigma$ and $\tau_1 = \mathrm{fl}(\tau_1 + \tau_2) = \mathrm{fl}(\varrho + \tau^{(1)}) \ne \varrho$ together with Lemma 2.1 imply $|\tau_1| = \texttt{eps}^{-1}\sigma$, $|\varrho| = \mathrm{pred}(|\tau_1|) = |\tau_1| - \sigma$ and $\tau_1\tau_2 \le 0$. Now (10.6) yields $\mathrm{sign}(\tau_1) = -\mathrm{sign}(\texttt{res} - \tau_1)$, and therefore $|\tau_1| > |\texttt{res}| = |\tau_1| - \sigma$ by (10.8). This proves (10.10).

Finally we have $|\tau_2| \le \texttt{eps} \cdot \mathrm{ufp}(\tau_1) = \sigma$, we know $\tau_2\Delta \ge 0$ by (10.6), and together with $|\Delta| = \sigma$ this certifies $|\tilde{R}| = |\tau_2 - \Delta| \le \sigma$. Therefore (10.4) together with (2.10) yield (10.1). This finishes the proof of (6.5). $\square$

11. Summary.

We presented algorithms to calculate the sign of a sum, summation with K-fold faithfully rounded, with directed rounding and rounded-to-nearest result. The paper contains as well the ingredients to compute a rounded to nearest result in K-fold accuracy. All our algorithms use only floating-point addition, subtraction and multiplication in one working precision, no branches in the inner loops and no special operations. Similar algorithms for dot products are easily developed using the error-free transformation `TwoProduct` of a product of two floating-point numbers into a sum (cf. [5], see also [18]). For all algorithms presented in Part I and II of this paper and in [18] 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 thank the two anonymous referees for their thorough reading and most valuable and inspiring 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 and 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] *XBLAS: A Reference Implementation for Extended and Mixed Precision BLAS.* http://crd.lbl.gov/~xiaoye/XBLAS/.
[2] D. Bailey, *A Fortran-90 based multiprecision system*, ACM Trans. Math. Software, 21 (1995), pp. 379–387.
[3] R. Brent, *A Fortran Multiple-Precision Arithmetic Package*, tech. report, Dept. of Computer Science, Australian National University, Canberra, 1975.
[4] M. Daumas and D. Matula, *Validated Roundings of Dot Products by Sticky Accumulation*, IEEE Trans. Comput., 46 (1997), pp. 623–629.
[5] T. Dekker, *A Floating-Point Technique for Extending the Available Precision*, Numerische Mathematik, 18 (1971), pp. 224–242.
[6] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, *MPFR: A multiple-precision binary floating-point library with correct rounding*, ACM Trans. Math. Softw., 33 (2007), 15 pages.
[7] *GNU multiple precision arithmetic library (GMP), version 4.1.2.*, 2003. Code and documentation available at `http://swox.com/gmp`.
[8] S. Graillat, P. Langlois, and N. Louvet, *Improving the Compensated Horner Scheme with a Fused Multiply and Add*, Proceedings of the ACM symposium on Applied computing, CMS-109, pp. 1323–1327, 2006.
[9] N. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadelphia, 2nd ed., 2002.
[10] *ANSI/IEEE 754-1985, Standard for binary floating-point arithmetic*, 1985.
[11] D. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2, Addison Wesley, Reading MA, 1969.
[12] U. Kulisch and W. Miranker, *Arithmetic Operations in Interval Spaces*, Computing, Suppl., 2 (1980), pp. 51–67.
[13] P. Langlois and N. Louvet, *Solving Triangular Systems more Accurately and Efficiently*, Tech. Report RR2005-02, Laboratoire LP2A, University of Perpignan, 2005.
[14] 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.
[15] N. Louvet, *private communication*, 2006.
[16] M. Malcolm, *On accurate floating-point summation*, Comm. ACM, 14 (1971), pp. 731–736.
[17] A. Neumaier, *Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen*, Zeitschrift für Angew. Math. Mech. (ZAMM), 54 (1974), pp. 39–51.
[18] T. Ogita, S. Rump, and S. Oishi, *Accurate Sum and Dot Product*, SIAM Journal on Scientific Computing (SISC), 26 (2005), pp. 1955–1988.
[19] D. Priest, *Algorithms for Arbitrary Precision Floating Point Arithmetic*, in Proc. 10th Symposium on Computer Arithmetic, P. Kornerup and D. Matula, eds., Grenoble, France, IEEE Computer Society Press, pp. 132–145, 1991.
[20] ———, *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`.
[21] S. Rump, *Kleine Fehlerschranken bei Matrixproblemen*, PhD thesis, Universität Karlsruhe, 1980.
[22] S. Rump, T. Ogita, and S. Oishi, *Accurate Floating-point Summation I: Faithful Rounding.* submitted for publication in SISC, 200–2007.
[23] S. Rump and P. Zimmermann, *Interval operations in rounding to nearest.* submitted for publication, 2007.
[24] J. Shewchuk, *Adaptive precision floating-point arithmetic and fast robust geometric predicates*, Discrete Comput. Geom., 18 (1997), pp. 305–363.
[25] 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.
[26] 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.
[27] G. Zielke and V. Drygalla, *Genaue Lösung linearer Gleichungssysteme*, GAMM Mitt. Ges. Angew. Math. Mech., (2003), pp. 7–108.