HAL
open science

A New Distillation Algorithm for Floating-Point Summation

Yong-Kang Zhu, Jun-Hai Yong, Guo-Qin Zheng

To cite this version:

Yong-Kang Zhu, Jun-Hai Yong, Guo-Qin Zheng. A New Distillation Algorithm for Floating-Point Summation. SIAM Journal on Scientific Computing, 2005, 26, pp.2066-2078. 10.1137/030602009 . inria-00517618

HAL Id: inria-00517618

https://inria.hal.science/inria-00517618v1

Submitted on 17 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A NEW DISTILLATION ALGORITHM FOR FLOATING-POINT SUMMATION*

YONG-KANG ZHU†, JUN-HAI YONG†, AND GUO-QIN ZHENG†

Abstract. The summation of n floating-point numbers is ubiquitous in numerical computations. We present a new distillation algorithm for floating-point summation which is stable, efficient, and accurate. The algorithm iteratively "distills" the summands without discarding any significant digit until the partial sums cannot change the whole sum. It uses standard floating-point arithmetic and does not rely on the choice of radix or any other specific assumption. Furthermore, the error bound of our algorithm is independent of n and less than 1 ulp.

Key words. floating-point summation, rounding error, distillation

AMS subject classifications. 65G05, 65B10

DOI. 10.1137/030602009

1. Introduction.

Rounding errors frequently occur in computer floating-point computations [7]. The problem studied in this paper is how to maximize the accuracy of the summation of n floating-point numbers. Using infinite precision arithmetic may be a good choice, but both the time complexity and the space complexity are considerable. Because sums of floating-point numbers are ubiquitous in scientific computing [9], we should choose a method whose accuracy and cost are both acceptable.

Many people [1, 5, 6, 8, 9, 14, 15, 16, 20] have contributed to this area and produced a large number of algorithms. In this section, we will make a classification of the floating-point summation methods. The floating-point numbers considered here are t-digit and radix-β model. We use $fl(a \circ b)$ ($\circ \in \{+, -, \times, /\}$) to represent the floating-point arithmetic that appears in this paper.

1.1. Recursive summation.

The well-known characteristic of the summation of n floating-point numbers is that different ordering of summations can give different sums due to rounding errors [17]. Recursive summation with various orderings is one class of algorithms intending to reduce the rounding error. The standard recursive summation [1] is

$$s = (\cdots + ((x_1 + x_2) + x_3) + \cdots + x_{n-1}) + x_n, \tag{1.1}$$

where x_i ($i = 1, 2, \ldots, n$) are n floating-point numbers. Three orderings, increasing, decreasing, and Psum, of recursive summation have been compared by Higham [9]. Increasing ordering of recursive summation is the method which sorts x_i by order of increasing absolute value before using (1.1), and decreasing ordering of recursive summation is just the opposite. The ordering of Psum is determined by minimizing, in turn, $|x_1|, |\hat{S}_2|, \ldots, |\hat{S}_{n-1}|$, where $\hat{S}_i$ ($i = 2, 3, \ldots, n-1$) is the temporary sum of $\sum_{j=1}^{i} x_j$. Clearly, Psum degenerates into increasing ordering when all the x_i have the

*Received by the editors November 27, 2003; accepted for publication (in revised form) October 8, 2004; published electronically July 13, 2005. This research was supported by the Chinese 973 Program (2004CB719400) and the National Science Foundation of China (60403047).

†School of Software, Tsinghua University, Beijing, 100084, People's Republic of China (zhuyk@tsinghua.org.cn, yongjh@tsinghua.edu.cn, gqzheng@tsinghua.edu.cn). The second author's research was supported by a Foundation for the Author of National Excellent Doctoral Dissertation of People's Republic of China (200342).

same sign. Increasing ordering and Psum ordering are preferable to the decreasing ordering when all the summands are positive (or negative). However, the decreasing ordering is recommended when there is heavy cancellation in the sum, that is, when $|\sum_{i=1}^{n} x_i| \ll \sum_{i=1}^{n} |x_i|$ (see [9]). The common disadvantage of these methods is that sorting is time-consuming.

1.2. Using high-precision accumulators. Although these kinds of methods use almost the same algorithms as recursive summation, we classify it here since it has special assumptions and a different error analysis.

Demmel and Hida [5] analyze four algorithms for summing n floating-point numbers x_i using at least one accumulator with higher precision. They assume that there exist several extra precise floating-point registers with F ($F > f$) significant bits, where f is the number of significant bits in the x_i. The first two algorithms, mentioned in their paper, sort x_i by decreasing magnitude and use one F accumulator to sum the floating-point numbers using recursive summation, while the other two use more F accumulators to avoid sorting.

To guarantee that each of these summation algorithms always obtains a result whose rounding error is about 1.5 ulp (unit in last place [10]), the number of summands must have an upper limit. Demmel and Hida consider various values of f and F arising from computations in the IEEE floating point standard. The limits of n for the four algorithms are provided and some proofs are made in [5]. According to their conclusions, we can choose the cheapest algorithm to use for any particular n. However, in practice, there is no such accumulator available in computer when we use the highest precision type of floating-point numbers.

1.3. Compensated summation. Compensated summation is derived by Kahan [12] from Gill who notices that the rounding error in the sum of two numbers could be estimated by subtracting one of the numbers from the sum [9].

Algorithm 1. Compensated summation.

//the input data are n floating-point numbers, x_i, $i = 1, 2, \ldots, n$.

//the output is the sum, s.

//e, *temp*, and y are floating-point numbers used as temporary variables.

1. $s \leftarrow 0$, $e \leftarrow 0$.
2. For $i \leftarrow 1$ to n
   (a) $temp \leftarrow s$.
   (b) $y \leftarrow fl(x_i + e)$.
   (c) $s \leftarrow fl(temp + y)$.
   (d) $e \leftarrow fl(fl(temp - s) + y)$.
3. End.

This method is recommended as an efficient and reliable summation algorithm for general data [1], but it has two weaknesses. One is, $e \leftarrow fl(fl(temp - s) + y)$ is not always the exact correction, since it is based on the assumption that $|temp| \geq |y|$ and $\beta \leq 3$ (see [14]). The other one is, the addition $y \leftarrow fl(x_i + e)$ does not always satisfy $fl(x_i + e) = x_i + e$. When $\sum_{i=1}^{n} |x_i| \gg |\sum_{i=1}^{n} x_i|$, compensated summation does not yield a small error. However, when all the summands have the same sign, this method guarantees perfect relative accuracy (as long as $nu \leq 1$, where u is the unit roundoff) [9]. The improved Kahan–Babuska summation (iKBS) [18] (IV,1) is a variation of the compensated summation. This method can obtain higher accuracy to some extent than compensated summation for sums with heavy cancellation ($\sum_{i=1}^{n} |x_i| \gg |\sum_{i=1}^{n} x_i|$). More variations of the compensated summation are given

and compared in [10, 18, 19, 20]. They use slightly more complicated operations to compensate the error, and some of them sort the original data before implementing compensated summation.

1.4. Distillation algorithms. Distillation algorithms are first summarized by Kahan (see [9]). For summation of x_i $(i = 1, 2, \ldots, n)$, they iteratively construct floating-point numbers $x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}$ such that $\sum_{i=1}^n x_i^{(k)} = \sum_{i=1}^n x_i$, terminating when $x_n^{(k)}$ approximates $\sum_{i=1}^n x_i$ with relative error at most $u = \frac{1}{2}\beta^{1-t}$ (see [10]).

Anderson presents a new algorithm, named modified deflation, for floating-point summation in [1], which is a kind of distillation algorithm. This algorithm mainly utilizes the high performance of compensated summation method when all the summands have the same sign. It deflates (see (1.3)) two floating-point numbers with opposite signs iteratively until the sum of positive numbers, s_+, and the sum of negative numbers, s_-, satisfy

$$\left| fl\big(fl(s_+ + s_-)/fl(s_+ - s_-)\big) \right| = 1, \tag{1.2}$$

which is a compromise to the condition that all the residual numbers have the same sign. And one deflation is a special case of compensation,

$$\hat{s} = fl(a + b), \quad \hat{e} = fl(fl(a - \hat{s}) + b), \tag{1.3}$$

where $|a| \geq |b|$ and $ab \leq 0$. It has been proven that the deflation (1.3) is always exact, namely $a + b = \hat{s} + \hat{e}$, irrespective of the radix β (see [1, 20]). However, it is clear that we have $\hat{s} = a$ and $\hat{e} = b$ when $|a| \gg |b|$, which will result in an infinite loop. To avoid this flaw, Anderson affiliates the reduction algorithm that reduces a to two numbers w and v, where v is much smaller than w and $fl(w + v) = w + v = a$. After several times of reduction, the value of v is small enough to make one deflation such that $\hat{s} \neq v$ and $\hat{e} \neq b$. When (1.2) is satisfied, it uses Kahan's compensated summation to sum up the remainders. Since each step of deflation is exact, the error is produced at the last step. Thus, the error bound of this method is very small as long as $nu \leq 1$ (see section 1.3).

1.5. Other methods. Two additional methods are also compared by Higham in [9]. They are pairwise summation (also known as cascade summation) and insertion method. The pairwise method [15] can work efficiently in parallel. For example when $n = 8$, the pairwise summation can be written as $S = ((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8))$. The insertion method is based on sorting. First, it sorts all the x_i by order of increasing magnitude. And then, it removes the first two numbers and inserts the sum of them into the remainder summands, maintaining the increasing order. The process is repeated until the final sum is obtained. However, Higham [9] indicates that "*no one method is uniformly more accurate than the others.*" Here the methods that he refers to are recursive summation of three orderings, in section 1.1, compensated summation, in section 1.3, and the two methods mentioned in this subsection.

In the next section, we present a new distillation algorithm for floating-point summation and analyze the error bound of it. We get inspiration from the error analysis in section 2 and make an improvement to our algorithm in section 3. The new algorithms are compared with other methods in section 4. Finally, conclusions are drawn in section 5.

2. A new distillation algorithm. Each floating-point number, say y, can be represented in the form

$$y = \pm 0.y_1 y_2 \ldots y_t \times \beta^{e_y}, \tag{2.1}$$

where each digit y_i satisfies $0 \le y_i \le \beta - 1$, $y_1 \ne 0$ for normalized numbers, and $y_1 = 0$ for denormalized numbers [10]. We do not consider the floating-point overflow and underflow in our algorithms. In the first subsection, we present an exact addition algorithm for two arbitrary floating-point numbers. Then we use this algorithm iteratively to construct a new distillation algorithm for the summation of n floating-point numbers. The last subsection is the error analysis for the algorithm.

2.1. Exact addition. To make every step of summation absolutely accurate, we consider how to obtain the sum and error of addition between two arbitrary floating-point numbers exactly. The difference between our method and the deflation method (see section 1.4) is that it is not required that the two floating-point numbers have opposite signs.

We divide the rounding error into two classes, the truncation error and the carry error. Given two floating-point numbers, a and b, where $|a| \ge |b|$, we consider the floating-point addition between a and b. When the exponents of a and b are not equal, b, whose exponent is smaller, will right-shift the mantissa to make its exponent increase until its exponent equals that of a. If the bits discarded by right-shifting have at least one significant digit, the truncation error occurs. Most machines have several extra digits [2, 3], such as guard digits, used for rounding when truncation error occurs. For example, $t = 4$ and $\beta = 10$, we assume $a = 0.4122 \times 10^6$ and $b = 0.5952 \times 10^3$. Then, when the machine has a guard digit, we have $fl(a+b) = 0.4128 \times 10^6$ and the error is -0.4800×10^1, where the digit "9" in b is saved and rounded up. However, when no guard digit existed, we have $fl(a+b) = 0.4127 \times 10^6$ and the error is 0.9520×10^2 since the digit "9" in b is simply discarded. This kind of error can be avoided. Our method first defines a floating-point number b'. Second, let b' have the same value as b, and then clears the last Δe bit(s) of the mantissa of b', where Δe is the exponent difference between a and b'. Thus, when $fl(a+b')$ is implemented, all of the bits discarded by right-shifting b' are 0, which means no truncation error occurs whether the machine has extra digits or not. And we have $r = b - b' = fl(b - b')$, namely $a + b = (a + b') + r$.

However, $fl(a+b')$ does not always equal $a+b'$, since the carry error may occur under two conditions when $fl(a+b')$ is being implemented. The first condition is that the exponent of $fl(a+b')$ is bigger than that of a. The other one is that the bit discarded by floating-point normalizing for the sum is a significant digit. After obtaining the carry error c, we let $e \leftarrow fl(r+c)$, where e is just the error that satisfies $a + b = s + e$. We emphasize that the sum s does not always satisfy $s = fl(a+b)$, since the value of s is obtained from $s = fl(a+b')$, which may have 1 ulp difference with the value of $fl(a+b)$. But this does not influence the accuracy of our distillation algorithm in the next subsection. We will prove the validity of Algorithm 2 after describing it in detail.

Algorithm 2. Exact addition.

// The inputs are two arbitrary floating-point numbers, a and b.

// The outputs are the sum s and the error e, where $a + b = s + e$.

// $EXP(x)$ represents the exponent of the floating-point number x.

1. If $EXP(a) < EXP(b)$, then swap a and b.

2. If $EXP(a) \geq EXP(b) + t$, then
   (a) $s \leftarrow a$, $e \leftarrow b$.
   (b) Go to Step 11.
3. $r \leftarrow 0$, $c \leftarrow 0$.
4. $b' \leftarrow b$.
5. If $EXP(a) = EXP(b)$, then
   (a) $s \leftarrow fl(a + b)$.
   (b) Go to Step 9.
6. Clear the last $\big(EXP(a) - EXP(b)\big)$ digit(s) of the mantissa of b'. Namely, set them zero.
7. $s \leftarrow fl(a + b')$.
8. $r \leftarrow fl(b - b')$.
9. If $EXP(s) > EXP(a)$, then
   If $EXP(s) > EXP(fl(a - s))$, then $c \leftarrow fl(fl(a - s) + b')$.
   Otherwise, $c \leftarrow fl\big(fl(a - fl(s/2)) + fl(b' - fl(s/2))\big)$.
10. $e \leftarrow fl(r + c)$.
11. End.

When $EXP(a) \geq EXP(b) + t$, Step 2 sets s and e directly because all the significant digits of b will be cleared by Step 6. Thus, if the algorithm stops from Step 2, then it is clear that $a + b = s + e$. Otherwise, we have $EXP(b) = EXP(b')$ after Step 6. Thus, we have

$$r = fl(b - b') = b - b'. \tag{2.2}$$

If $EXP(s) > EXP(a)$, the carry error may occur. We will prove that the carry error c obtained from Step 9 is exact, namely $c = a + b' - s$.

Proof. We have $EXP(s) = EXP(a) + 1$ and $ab' > 0$ when the carry error occurs. Without loss of generality, we assume $a, b' > 0$, $a = 0.a_1a_2 \ldots a_t \times \beta^{e_a}$, $b' = 0.b'_1b'_2 \ldots b'_t \times \beta^{e'_b}$, and $s = 0.s_1s_2 \ldots s_t \times \beta^{e_{a+1}}$, $s_1 = 1$ or 2.

If $EXP(fl(a-s)) < EXP(s)$, we have $fl(a-s) = a-s$. Furthermore, Steps 2 and 5 guarantee b' is a multiple of ulp(a). Thus, we have $c = fl(fl(a-s)+b') = a-s+b'$.

Otherwise, we have $EXP(fl(a-s)) = EXP(s)$. If $s_1 = 1$, then $a_1+b'_1+1-\beta \geq a_1$, namely $a_1 = b'_1 = \beta - 1$ and $a_1 + b'_1$ gets a carry from the next bit. Thus, $a_i = b'_i = s_{i+1} = \beta-1$, $i = 1, 2, \ldots, t-1$, and $a_t+b'_t \geq \beta$. But, since $EXP(fl(a-s)) = EXP(s)$, we must have $a_t = b'_t = 0$. So $s_1 \neq 1$.

Then $s_1 = 2$. This case happens only when $a_i = b'_i = \beta - 1$, $i = 1, 2, \ldots, t - 1$, and $a_t + b'_t \geq \beta$. Moreover, the rounding operation must produce a carry when s is normalized. So we have $s_1 = 2$, $s_i = 0$, $i = 2, 3, \ldots, t$. For example, $t = 4$ and $\beta = 10$, we assume $a = 0.9999$ and $b' = 0.9997$. Then, $a + b' = 1.9996$. When 1.9996 is being normalized, the rounding operation discards the last digit "6," and produces a carry. We obtain $s = fl(a + b') = 0.2000 \times 10^1$. Hence, we have $s/2 = fl(s/2) = 0.1 \times \beta^{e_a+1} > a \geq b'$, $fl(a - s/2) = a - s/2$, and $fl(b' - s/2) = b' - s/2$. Therefore, $c = fl\big(fl(a - fl(s/2)) + fl(b' - fl(s/2))\big) = a + b' - s$. ∎

With (2.2), we have

$$a + b = s + r + c. \tag{2.3}$$

Then we will prove that $e = fl(r + c) = r + c$.

Proof. Consider the addition between two floating-point numbers, a and b, where $|a| \geq |b|$. If $c = 0$, then $e = fl(r) = r$. If $c \neq 0$, then we have $ab > 0$ and a is

a normalized number because $EXP(s) > EXP(a)$. Without loss of generality, we assume $a, b > 0$. According to (2.1), we define a and b as

$$a = 0.a_1a_2\ldots a_t \times \beta^{e_a}, \quad a_1 \neq 0,$$
$$b = 0.b_1b_2\ldots b_t \times \beta^{e_b}.$$

After Step 6 in Algorithm 2, we have

$$\begin{array}{rcccccccccccccc} & 0 & . & a_1 & a_2 & \ldots & a_{t-i+1} & \ldots & a_{t-1} & a_t & & & & \times & \beta^{e_a} \\ +) & 0 & . & 0 & 0 & \ldots & b_1 & \ldots & b_{i-1} & b_i & 0 & \ldots & 0 & \times & \beta^{e_a} \\ \hline & s_1 & . & s_2 & s_3 & \ldots & s_{t-i+2} & \ldots & s_t & s_{t+1} & & & & \times & \beta^{e_a} \end{array} \tag{2.4}$$

where $a_1 \neq 0$ and $s_1 \neq 0$. After normalizing s to $0.s_1s_2\ldots s_t \times \beta^{e_a+1}$, the last digit, say s_{t+1}, is rounded off. Thus, we have

$$c = \hat{s} - s = \pm x \times \beta^{e_a - t}, \quad 0 < x \leq \beta - 1, \tag{2.5}$$

where $\hat{s} = s_1.s_2\ldots s_ts_{t+1} \times \beta^{e_a}$, s is the normalized number for $\hat{s}$, c equals the value obtained from Step 9 in Algorithm 2, and the value of x is determined by the rounding mode of a certain floating-point arithmetic. With (2.2) and (2.4), we have

$$\begin{array}{rrcccccccccccc} & b = & & 0 & . & b_1 & b_2 & \ldots & b_i & b_{i+1} & \ldots & b_t & \times & \beta^{e_b} \\ -) & b' = & & 0 & . & b_1 & b_2 & \ldots & b_i & 0 & \ldots & 0 & \times & \beta^{e_b} \\ \hline & r = & & 0 & . & 0 & 0 & \ldots & 0 & b_{i+1} & \ldots & b_t & \times & \beta^{e_b} \\ +) & c = & \pm & 0 & . & 0 & 0 & \ldots & x & 0 & \ldots & 0 & \times & \beta^{e_b} \\ \hline & e = & & 0 & . & 0 & 0 & \ldots & e_1 & e_2 & \ldots & e_{t-i+1} & \times & \beta^{e_b} \end{array} \tag{2.6}$$

where $1 \leq i \leq t$. Thus the length of mantissa of $r + c$ is t at most, i.e., $e = fl(r + c) = r + c$. $\square$

Therefore $a + b = s + e$, which means Algorithm 2 is an exact addition.

2.2. Distillation. With the exact addition algorithm above, we present a new distillation algorithm to sum the floating-point numbers iteratively so as to obtain a result as accurate as possible. First, the algorithm divides the summands into two sets: one for positive numbers and one for negative numbers. Second, using Algorithm 2 recursively to obtain the partial sums of the two sets respectively. The error produced in each exact addition is redistributed to the two sets according to its sign. Then use Algorithm 2 again to get the whole sum from the partial sums. If neither of the partial sums can change the value of the whole sum, then stop. Otherwise, repeat the above operations. We will describe our distillation algorithm in detail.

ALGORITHM 3. DISTILLATION.

//The inputs are n floating-point numbers, $x_1, x_2, \ldots, x_n$.

//The output is the sum, s

1. Initialize two empty sets, $\mathcal{P}$ and $\mathcal{N}$. The positive numbers in $x_1, x_2, \ldots, x_n$ are distributed into $\mathcal{P}$, and the negative numbers are distributed into $\mathcal{N}$.
2. $s \leftarrow 0$.
3. $e_1 \leftarrow 0, \quad e_2 \leftarrow 0$.
   //Outer loop (Steps 4–16)
4. If $e_1 \neq 0$, then put e_1 into the end of $\mathcal{P}$ or $\mathcal{N}$ depending on its sign.
5. If $e_2 \neq 0$, then put e_2 into the end of $\mathcal{P}$ or $\mathcal{N}$ depending on its sign.
6. $s' \leftarrow s$.

7. $n_1 \leftarrow$ (The current size of $\mathcal{P}$).
8. $s_+ \leftarrow 0$.
9. For $i \leftarrow 1$ to n_1 //Inner loop 1
   (a) Remove the head number of $\mathcal{P}$, say a.
   (b) $b \leftarrow s_+$.
   (c) Add a and b using Algorithm 2 to obtain e and the new s_+.
   (d) If $e \neq 0$, then redistribute e to the end of $\mathcal{P}$ or $\mathcal{N}$ depending on its sign.
10. Add s and s_+ using Algorithm 2 to obtain e_1 and the new s.
11. $s'' \leftarrow s$.
12. $n_2 \leftarrow$ (The current size of $\mathcal{N}$).
13. $s_- \leftarrow 0$.
14. For $i \leftarrow 1$ to n_2 //Inner loop 2
   (a) Remove the head number of $\mathcal{N}$, say a.
   (b) $b \leftarrow s_-$.
   (c) Add a and b using Algorithm 2 to obtain e and the new s_-.
   (d) If $e \neq 0$, then redistribute e to the end of $\mathcal{P}$ or $\mathcal{N}$ depending on its sign.
15. Add s and s_- using Algorithm 2 to obtain e_2 and the new s.
16. If $s \neq s'$ or $s \neq s''$, then go to Step 4. //Outer loop (Steps 4–16)
17. $s \leftarrow fl\big(s + fl(e_1 + e_2)\big)$.
18. End.

The algorithm cancels the significant digits between positive summands and negative summands by order of decreasing magnitude. So the algorithm can end in finite number of steps. Furthermore, the process of distillation does not discard any significant digit since the addition operation is exact. Therefore, the error is produced only in the last step.

2.3. Error analysis. When Algorithm 3 reaches Step 17 and before effectuating Step 17, e_1, e_2, and the sum of the remainder numbers in $\mathcal{P}$ and $\mathcal{N}$ may have significant digits relative to s. Thus the error of s, before Step 17 is carried out, is

$$|e_0| = \left| \sum_{i=1}^{n_+} p_i + \sum_{i=1}^{n_-} n_i + e_1 + e_2 \right|. \tag{2.7}$$

To obtain the bound of the error, we analyze each term of (2.7). For Algorithm 2, we have

$$EXP(s_{ab}) \geq EXP(e) + t, \tag{2.8}$$

where s_{ab} represents the s in Algorithm 2 to avoid confusion. After Algorithm 3 finishes Steps 9 and 14, all the numbers in $\mathcal{P}$ and $\mathcal{N}$ are generated by Algorithm 2. Since $|s_+|$ and $|s_-|$ are increasing gradually, with (2.8), we have

$$\max\big((EXP(s_+), EXP(s_-))\big) \geq EXP(x) + t, \quad x \in \mathcal{P} \cup \mathcal{N}. \tag{2.9}$$

When $s = s'$ and $s = s''$ (see Step 16 in Algorithm 3), we have

$$\begin{aligned} EXP(s) &\geq EXP(s_+) + t, \\ EXP(s) &\geq EXP(s_-) + t. \end{aligned}$$

With (2.9), we have

$$EXP(s) \geq EXP(x) + 2t, \quad x \in \mathcal{P} \cup \mathcal{N}.$$

According to (2.1), we obtain

$$\begin{aligned} |x| &= 0.x_1x_2\ldots x_t \times \beta^{EXP(x)} \\ &\le \beta^{EXP(x)} \\ &\le \beta^{EXP(s)-2t}. \end{aligned}$$

Hence, we have

$$\left|\sum_{i=1}^{n_+} p_i\right| \le |n_+ \cdot \max(p_i)| \le n_+ \cdot \beta^{EXP(s)-2t}, \quad p_i \in \mathcal{P}, \tag{2.10}$$

and

$$\left|\sum_{j=1}^{n_-} n_j\right| \le |n_- \cdot \min(n_j)| \le n_- \cdot \beta^{EXP(s)-2t}, \quad n_j \in \mathcal{N}, \tag{2.11}$$

where n_+ and n_- represent the sizes of $\mathcal{P}$ and $\mathcal{N}$, respectively. Thus,

$$|e'| = \left|\sum_{i=1}^{n_+} p_i + \sum_{j=1}^{n_-} n_j\right| \le \max(n_+, n_-) \cdot \beta^{EXP(s)-2t}, \tag{2.12}$$

where $p_i \in \mathcal{P}$ and $n_j \in \mathcal{N}$. After $f \leftarrow fl(s + fl(e_1 + e_2))$ (see Step 17 in Algorithm 3), we have

$$|e''| = \big|(s + e_1 + e_2) - fl(s + fl(e_1 + e_2))\big| \le 0.5 \quad \text{(ulp)}, \tag{2.13}$$

where we assume that the floating-point arithmetic $fl(x \pm y)$ is *correctly rounded* (*when the error is less than* 0.5 ulp, *the floating-point number is said to be correctly rounded*) [11]. Therefore, according to (2.7), (2.12), and (2.13), the error bound for Algorithm 3 is

$$\begin{aligned} |e| &\le |e'| + |e''| \\ &\le \max(n_+, n_-) \cdot \beta^{EXP(s)-2t} + 0.5\,\text{ulp} \\ &= \max(n_+, n_-) \cdot \beta^{-t} + 0.5 \quad \text{(ulps)}. \end{aligned} \tag{2.14}$$

And we can approximate the error bound as

$$|e| \le n\beta^{-t} + 0.5 \quad \text{(ulps)}. \tag{2.15}$$

Therefore, when $n\beta^{-t} \ll 1$, the error bound is 0.5 ulp.

3. Improvement. From the above error analysis, we can estimate the upper (lower) limit of the next sum of $\mathcal{P}$ ($\mathcal{N}$) according to the current value of s_+ (s_-). This gives us an inspiration to reduce the error bound of Algorithm 3.

When Steps 9 and 14 in Algorithm 3 are finished, we have (2.9). Thus, we can get the upper bound for the next $|s_+ + s_-|$, say $|s'_+ + s'_-|$. So, we have

$$\begin{aligned} |s'_+ + s'_-| &= \left|\sum_{i=1}^{n_+} p_i + \sum_{j=1}^{n_-} n_j\right| \\ &\le \max\left(\left|\sum_{i=1}^{n_+} p_i\right|, \left|\sum_{j=1}^{n_-} n_j\right|\right) \\ &\le \max(|p_i|, |n_j|) \cdot \max(n_+, n_-) \\ &\le \beta^{\max((EXP(s_+), EXP(s_-))-t} \cdot \max(n_+, n_-), \end{aligned} \tag{3.1}$$

where $p_i \in \mathcal{P}$ and $n_j \in \mathcal{N}$. Then we modify Step 16 in Algorithm 3 to obtain Algorithm 4.

ALGORITHM 4. NEW DISTILLATION.
// Steps 1–15, 17, 18, which are the same ones as in Algorithm 3.
16′. (a) $\hat{s} \leftarrow \beta^{\max((EXP(s_+),EXP(s_-))-t} \cdot \max(n_+, n_-)$.
(b) If $fl(s + \hat{s}) = s$, then go to Step 17.
(c) Go to Step 4. //Outer loop (Steps 4–16′)

So the new terminating condition is $fl(s + \hat{s}) = s$. Thus, we have

$$\begin{aligned} |e'| &= \left| \sum_{i=1}^{n_+} p_i + \sum_{j=1}^{n_-} n_j \right| \\ &\le \hat{s} \\ &\le 0.5\,\text{ulp}. \end{aligned} \tag{3.2}$$

However, the final operation is unchanged, see Step 17 in Algorithm 3. Therefore, with (2.13), we have

$$|e| \le |e'| + |e''| \le 1\,\text{ulp}. \tag{3.3}$$

Comparing with (2.15) we reduce the error bound to a constant. Thus, the sum produced by Algorithm 4 is more reliable than Algorithm 3 when n is extremely huge. Actually, the sum of remaining numbers in $\mathcal{P}$ and $\mathcal{N}$ is much less than the estimated value $\hat{s}$ in most cases. We broaden the error bound to some extent in order to make it independent of n.

When adding two binary floating-point numbers, say a and b, if $|a| \ge |b|$, then the sum and error can be correctly obtained with

$$\hat{s} = fl(a + b), \quad \hat{e} = fl(fl(a - \hat{s}) + b), \tag{3.4}$$

proven by Dekker [4], Higham [10], and Knuth [14]. Here, only three flops (floating-point operations) are needed in one addition.

ALGORITHM 2′. BINARY EXACT ADDITION.
// The inputs are two arbitrary binary floating-point numbers, a and b.
// The outputs are the sum s and the error e, where $a + b = s + e$.
1. If $EXP(a) < EXP(b)$, then swap a and b.
2. $s \leftarrow fl(a + b)$.
3. $e \leftarrow fl(fl(a - s) + b)$.

So, when algorithms are implemented with binary floating-point arithmetic, the following algorithm can be used instead of Algorithm 4.

ALGORITHM 5. NEW DISTILLATION (BINARY).
// This algorithm is the same as Algorithm 4 except:
// changing "using Algorithm 2" in Steps 9(c), 10, 14(c), and 15
// into "using Algorithm 2′."

4. Results. The algorithms are implemented with Microsoft Visual C++ 6.0 on Pentium IV 1.7 GHz processor. The software environment is Windows 2000 Professional. The type of floating-point numbers used in our algorithms is *double* ($\beta = 2$, $t = 53$), which conforms to IEEE 754 standard [2, 13]. So, Algorithm 5 can be used here. We test the accuracy of the algorithms by comparing the results with the value

TABLE 4.1

Relative errors (in ulps) produced by summation algorithms. Size of random data set $N =$ 100,000, $|\Delta E| < 100$.

Summation methods	Well-conditioned data	
	Example 1	Example 2
Recursive [1]	250	291
Compensated [9]	0	0
iKBS [18]	0	0
Modified deflation [1]	0	0
Algorithm 3	0	0
Algorithm 4	0	0
Algorithm 5	0	0

Summation methods	Random data	
	Example 1	Example 2
Recursive [1]	990	4,799
Compensated [9]	2	2
iKBS [18]	1	1
Modified deflation [1]	1	1
Algorithm 3	0	0
Algorithm 4	0	0
Algorithm 5	0	0

Summation methods	The first ill-conditioned data	
	Example 1	Example 2
Recursive [1]	148,784,046,618	56,355,983,702
Compensated [9]	875,848,218	2,800,479,914
iKBS [18]	97,220,890	300,008,692
Modified deflation [1]	1	0
Algorithm 3	0	0
Algorithm 4	0	0
Algorithm 5	0	0

Summation methods	The second ill-conditioned data	
	Example 1	Example 2
Recursive [1]	1,061,818,183,723,985,500	1,877,614,346,812,393,000
Compensated [9]	112,163,757,435,528	136,635,642,414,552
iKBS [18]	34,374,137,666,768	6,369,588,672,926
Modified deflation [1]	0	0
Algorithm 3	0	0
Algorithm 4	0	0
Algorithm 5	0	0

figured out by Java BigDecimal class. The way of comparison is that we save all the summands, generated by the randomizer, and the results, produced by a certain floating-point summation algorithm, to a data file, and then use a Java program to read the file and verify the accuracy of the result. And the running time of each algorithm is the average of 20 uniform tests.

The accuracy of seven methods is given in Table 4.1. Relative errors in Table 4.1 are computed as $|s - s_i|/\mathrm{ulp}(s_i)$, where s is the result calculated by a certain method listed in the table, and s_i is the result produced by the infinite precision summation method. Four kinds of data sets used in our tests are well-conditioned data, random data, and two ill-conditioned data generated in different ways. The well-conditioned data are the data whose condition number R [10] is $R = 1$, and the ill-conditioned data are $R \gg 1$, where $R = \sum_{i=1}^{n} |x_i| / |\sum_{i=1}^{n} x_i|$. The first ill-conditioned data are generated as follows. First, we randomly generate one floating-point number, a. Second, let $b = -a$ and randomly change the last 20 digits of b's mantissa. Then use the same method repeatedly until all summands are obtained.

TABLE 4.2
Time costs (in milliseconds) of summation algorithms. Size of data set $N = 2{,}000{,}000$.

Data attributes	Well-conditioned data							
	Recu.	Comp.	iKBS	Incr.	M.D.	Algo. 3	Algo. 4	Algo. 5
$\lvert\Delta E\rvert<1000$	15	38	278	3864	135	423	322	296
$\lvert\Delta E\rvert<500$	13	36	280	3860	137	469	347	305
$\lvert\Delta E\rvert<200$	15	38	282	3866	139	596	431	314
$\lvert\Delta E\rvert<50$	13	34	284	3883	147	782	650	364
$\Delta E=0$	15	37	281	3771	140	667	664	297
Data attributes	Random data							
	Recu.	Comp.	iKBS	Incr.	M.D.	Algo. 3	Algo. 4	Algo. 5
$\lvert\Delta E\rvert<1000$	13	40	291	3863	4923	446	341	309
$\lvert\Delta E\rvert<500$	12	38	287	3863	3058	491	369	316
$\lvert\Delta E\rvert<200$	12	37	281	3866	1630	626	449	337
$\lvert\Delta E\rvert<50$	13	36	300	3864	366	810	676	396
$\Delta E=0$	11	35	273	3777	161	695	695	332
Data attributes	The first ill-conditioned data							
	Recu.	Comp.	iKBS	Incr.	M.D.	Algo. 3	Algo. 4	Algo. 5
$\lvert\Delta E\rvert<1000$	13	36	275	3851	5927	552	451	408
$\lvert\Delta E\rvert<500$	15	35	278	3850	3694	601	490	410
$\lvert\Delta E\rvert<200$	16	37	278	3844	1598	756	615	425
$\lvert\Delta E\rvert<50$	12	39	279	3851	344	754	750	393
$\Delta E=0$	13	39	283	3781	160	652	651	315
Data attributes	The second ill-conditioned data							
	Recu.	Comp.	iKBS	Incr.	M.D.	Algo. 3	Algo. 4	Algo. 5
$\lvert\Delta E\rvert<1000$	14	36	281	3694	444	753	747	374
$\lvert\Delta E\rvert<500$	14	37	273	3595	441	757	744	436
$\lvert\Delta E\rvert<200$	14	36	282	3728	408	765	748	403
$\lvert\Delta E\rvert<50$	11	39	286	3863	365	822	759	431
$\Delta E=0$	15	37	280	3785	162	693	689	328

Recu. represents standard recursive summation [1].
Comp. represents compensated summation [9].
iKBS represents the improved Kahan–Babuska summation in Neumaier's paper [18].
Incr. represents increasing ordering of recursive summation [9].
M.D. represents the modified deflation algorithm [1].
Algos. 3−5 are Algorithms 3−5 presented in this paper.

The second ill-conditioned data are generated by the method mentioned in [1]: after randomly generating n floating-point numbers, the mean of the data (calculated using recursive summation) is subtracted from each datum.

The observation is that three methods (Algorithms 3−5) always generate results without any error. The biggest relative error produced by Anderson's modified deflation algorithm [1] is 1 ulp in our tests.

The running times of eight summation methods are compared in Table 4.2. Besides well-conditioned data, random data, and ill-conditioned data, five data attributes are considered. They are $|\Delta E| < 1000$, $|\Delta E| < 500$, $|\Delta E| < 200$, $|\Delta E| < 50$, and E is a constant, where E is the exponent of a floating-point number and ΔE is the exponent difference between two arbitrary summands. We must emphasize that when it is the second ill-conditioned data, the data attributes, listed in Table 4.2, are attributes of the original data used to generate the second ill-conditioned data.

Since the first four summation methods in Table 4.2 are independent of the data attributes, the time complexities of them are stable. On the contrary, Anderson's modified deflation method degenerates to compensated summation when the initial data are well-conditioned. So it performs well in such a condition. But when the data are the first ill-conditioned or random, the time it consumes increases considerably.

It is noticeable that when the data are $\Delta E = 0$ the modified deflation method performs very well, although the data set is not well-conditioned. The reason is that when two floating-point numbers, a and b, satisfy $ab < 0$ and $EXP(a) = EXP(b)$, the error estimation for $fl(a+b)$ is zero since neither a truncation error nor a carry error occurs under this condition. And the sum, $\hat{s}_o = fl(a+b)$, generated in the last deflation, is a multiple of ulp(a) or ulp(b), and also a multiple of ulp(x), where x is the new number to be deflated whose sign is opposite to $\hat{s}_o$. So, no error will be generated by all deflations. Thus, the main loop time of the algorithm is always one in this case.

Another observation is that the modified deflation method also works fast when the data are the second ill-conditioned data. We have tested the actual $|\Delta E|$ of the second ill-conditioned data, not the $|\Delta E|$ of the original data used to generate them. The result is that the actual $|\Delta E|$ is one when E is a constant, and is about 15 in the other four cases. Thus, the reduction (see section 1.4) is hardly needed. Since after reduction the bigger number will be returned to the set of remaining summands, which will increase the size of data set and the main loop time, we can conclude that the modified deflation algorithm can work fast when the actual $|\Delta E|$ is not very big. With $|\Delta E|$ increasing gradually, it works slower (see Table 4.2) due to a big amount of reduction.

Algorithms 3−5 are more stable than the modified deflation algorithm (see Table 4.2). Algorithm 5 is faster than Algorithm 4 since Algorithm $2'$ is much more simple than Algorithm 2 when it is binary floating-point arithmetic. The longest running time of Algorithm 4 is less than one fourth that of the increasing ordering of recursive summation [9]. In most cases, Algorithms 4 and 5 can finish after two outer loops. We can make up the worst case of the original data as the form of

$$x_1, -x_1, x_2, -x_2, \ldots, x_l, -x_l, y_1, y_2, \ldots, y_m, \quad m \gg l, \tag{4.1}$$

where

$$fl(x_i + x_{i+1}) = x_i, \quad i = 1, 2, \ldots, l-1$$

and

$$fl(x_l + y_j) = x_l, \quad j = 1, 2, \ldots, m.$$

In this case, Algorithm 4 needs at least $l+1$ outer loops to obtain the result since many useless additions occur, that is, the partial sums are unchanged in most additions before all the x_i have been cancelled. What is more, the total number of summands decreases little after each outer loop. However, Algorithm 2 directly returns the sum and the error without implementing any floating-point arithmetic when the exponent difference between two summands is more than t, so Algorithm 4 can work fast as well. We let $l = 30$ and $l + m = 2,000,000$ in our numerical tests. The result is Algorithm 4 uses 32 outer loops to complete the summation. And the running time of it is about 4013 milliseconds, which is commensurate with that of increasing ordering of recursive summation [9] (see Table 4.2). Moreover, this case hardly occurs in practice.

5. Conclusions. We have presented a new distillation algorithm for floating-point summation, which is accurate and fast. Our algorithm iteratively manipulates the summands without discarding any significant digit until the partial sums cannot change the whole sum. The error bound of our new algorithm (Algorithm 4 or 5) is irrespective of n and less than 1 ulp. The running time is much shorter than increasing

ordering of recursive summation [9], and the algorithm is more stable than Anderson's distillation algorithm [1], tested with diversified data sets.

Acknowledgment. The authors appreciate the comments and suggestions of the anonymous reviewers.

REFERENCES

[1] I. J. Anderson, *A distillation algorithm for floating-point summation*, SIAM J. Sci. Comput., 20 (1999), pp. 1797–1806.

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

[3] *A Radix-Independent Standard for Floating-Point Arithmetic, Standard* 854-1987, ANSI/IEEE, New York, 1987.

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

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

[6] T. O. Espelid, *On floating-point summation*, SIAM Rev., 37 (1995), pp. 603–607.

[7] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Comput. Surveys, 23 (1991), pp. 5–48.

[8] J. Gregory, *A comparision of floating point summation methods*, Commun. ACM, 15 (1972), p. 838.

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

[10] N. J. Higham, *Accuracy and Stability of Numerical Algorithms, Second Edition*, SIAM, Philadelphia, 2002.

[11] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied Interval Analysis*, Springer, London, 2001.

[12] W. Kahan, *Further remarks on reducing truncation error*, Commun. ACM, 8 (1965), p. 40.

[13] W. Kahan, *Lecture Notes on the Status of IEEE Standard* 754 *for Binary Floating-Point Arithmetic*, University of California, Berkeley, CA, 1996.

[14] D. E. Knuth, *The art of computer programming, Vol.* 2: *Seminumerical Algorithms*, 3rd ed., Addison–Wesley, Reading, MA, 1998.

[15] P. Linz, *Accurate floating-point summation*, Commun. ACM, 13 (1970), pp. 361–362.

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

[17] E. Mizukami, *The Accuracy of Floating Point Summations for CG-Like Methods*, Technical report 486, Department of Computer Science, Indiana University-Bloomington, Bloomington, IN, 1997.

[18] A. Neumaier, *Rundungsfehleranalyse einiger verfahren zur summation endlicher summen*, Z. Angew. Math. Mech., 54 (1974), pp. 39–51.

[19] D. M. Priest, *Algorithms for arbitrary precision floating point arithmetic*, in Proceedings of the 10th IEEE Symposium on Computer Arithmetic, P. Kornerup and D. W. Matula, eds., IEEE Computer Society, Los Alamitos, CA, 1991, pp. 132–143.

[20] D. M. Priest, *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*, Ph.D. thesis, Mathematics Department, University of California, Berkeley, CA, 1992.