SIAM J. SCI. COMPUT.
Vol. 20, No. 5, pp. 1797–1806

A DISTILLATION ALGORITHM FOR FLOATING-POINT SUMMATION*

I. J. ANDERSON†

Abstract. The addition of two or more floating-point numbers is fundamental to numerical computations. This paper describes an efficient "distillation" style algorithm which produces a precise sum by exploiting the natural accuracy of compensated cancellation. The algorithm is applicable to all sets of data but is particularly appropriate for ill-conditioned data, where standard methods fail due to the accumulation of rounding error and its subsequent exposure by cancellation. The method uses only standard floating-point arithmetic and does not rely on the radix used by the arithmetic model, the architecture of specific machines, or the use of accumulators.

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

AMS subject classifications. 65G05, 65B10

PII. S1064827596314200

1. Introduction.

We consider the summation of n floating-point numbers $\{x_i\}_{i=1}^n$ using a t-digit, radix-β model for which addition and subtraction are *faithful*.

DEFINITION (from [19, p. 12]). *For floating-point numbers a and b and $\circ \in \{+, -, \times, /\}$, let $c = a \circ b$ exactly (assuming $b \neq 0$ if $\circ = /$). Let x and y be consecutive floating-point numbers with the same sign as c such that $|x| \leq |c| < |y|$. Then the floating-point arithmetic is called faithful if $fl(a \circ b) = x$ whenever $c = x$ and $fl(a \circ b)$ is either x or y whenever $c \neq x$.*

All arithmetics that conform to IEEE 754 [8] or IEEE 854 [9] satisfy this model.

Mathematically we write the summation as

$$s = \sum_{i=1}^{n} x_i.$$

However, numerical considerations, such as rounding errors, mean that we must be more specific about how this summation is performed. Various techniques and orderings of the data have been considered and an excellent survey and reference list are given by Higham [6]. In this section we summarize some of the useful ideas and techniques.

1.1. Various orderings of the data.

We begin with standard recursive summation,

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

An error analysis of the floating-point value of this sum produces

$$fl(s) = \sum_{i=1}^{n} x_i(1 + \delta_i), \tag{1}$$

*Received by the editors December 27, 1996; accepted for publication (in revised form) October 17, 1997; published electronically May 13, 1999.
http://www.siam.org/journals/sisc/20-5/31420.html
†School of Computing and Mathematics, University of Huddersfield, Huddersfield, HD1 3DH, UK (i.j.anderson@hud.ac.uk).

where $|\delta_i| \leq (n+1-i)\eta$, $\eta = \frac{1}{2}\beta^{1-t}$ is the unit round-off of the machine, $fl(s)$ is the floating-point value obtained by calculating s, and we ignore second-order terms in η. This gives the following error bound:

$$|e| = |s - fl(s)| \leq \sum_{i=1}^{n} |x_i|(n+1-i)\eta. \tag{2}$$

The main conclusion from this bound is that data values used at the beginning of the summation (x_1, x_2, x_3, ...) have a larger contribution to the error bound than the latter terms since they are involved in more of the additions. Thus we can minimize this error bound by simply ordering the data into increasing order of absolute magnitude [21]. Other methods for reducing the error bound include pairwise summation, which reduces the number of times each datum is involved in the summation, and hence the factor $(n+1-i)$ in (2) can be replaced by a factor proportional to $\log n$ [14].

Two more simple techniques are *Psum* and *insertion addition*, both of which express the error bound in terms of the partial sums generated in forming $fl(s)$ [6]. The former method orders the data so that the current partial sum is minimized, while the latter method sorts the data into increasing magnitude, sums the smallest two data, and inserts the result into the appropriate place in the ordered list.

The major problem with these methods that attempt to minimize the error bound is that two algorithms with wildly varying error bounds will frequently produce similar results. This is because error bounds are a measure of the worst rounding error that can occur and not the actual rounding error that is experienced [21]. Indeed, when the summation involves heavy cancellation (i.e., $\sum_{i=1}^{n} |x_i| \gg |\sum_{i=1}^{n} x_i|$), examples indicate that it is best to order the data into decreasing magnitude even though this maximizes the error bound. The heuristic argument of the success of the decreasing order for such data is based on the idea that since cancellation must take place at some stage, and that by definition this cancellation will involve the larger numbers, it is better that it happens at the beginning of the sum before the smaller numbers have been used and swamped by rounding errors [3, 20].

1.2. Compensation. We consider the compensated summation of floating-point numbers as described by Kahan [10]. This procedure involves estimating the rounding error from a floating-point addition and recycling it at the succeeding step of recursive addition.

ALGORITHM 1. COMPENSATED SUMMATION.

```
s=0; e=0;
for i=1:n;
   a=s; b=x(i)+e;
   s=a+b; e=(a-s)+b;
end
```

The error bound for this method is particularly attractive [7]:

$$|e| = |s - fl(s)| \leq \left(2\eta + \mathcal{O}(n\eta^2)\right) \sum_{i=1}^{n} |x_i|. \tag{3}$$

In practice, compensated summation performs well with most data sets and will frequently give results that are better than the method of recursive summation and its variants mentioned in the previous section. This method is recommended as an efficient and reliable summation algorithm for general data.

1.3. Relative error. Each of the error bounds above can be expressed in terms of the relative error

$$\frac{|e|}{|s|} = \frac{|s - fl(s)|}{|s|} \leq \eta\lambda R,$$

where $R = \sum_{i=1}^{n} |x_i| \,/\, |\sum_{i=1}^{n} x_i|$, the condition number of the data [7], and $\lambda = n$ for recursive, increasing, Psum, and insertion addition; λ is proportional to $\log n$ for pairwise addition, $\lambda = 2$ for compensated summation. If all the data have the same sign, then the condition number is $R = 1$ and the data are said to be well conditioned. In fact, in this case the summation is both forward and backward stable for practical values of $n \ll 1/\eta$ [21]. However, if the data are ill conditioned ($R \gg 1$), then the summation is only backward stable as shown in (1). Despite this backward stability, there is no guarantee that the relative error is small.

In the next section we present a new algorithm which is based on insertion and compensated summation and which exploits the accuracy of cancellation as observed by the success of decreasing ordering for ill-conditioned data. The need for error bounds is eliminated by showing that exact computations can be used throughout to produce a new set of data for which the sum is the same as the sum of the original data, but which has a condition number that is considerably less than the original value.

2. The deflation algorithm. Compensated summation is designed for the addition of two arbitrary numbers a and b. For faithful binary arithmetic the error estimate, $\hat{e} = fl((a - \hat{s}) + b)$, is exact (i.e., $\hat{e} = e$) for all floating-point values of a and b provided that $|a| \geq |b|$. However, for a general radix, β, this is no longer the case.

A special case of compensation is when we are adding two numbers of opposite sign, i.e., subtraction. Without loss of generality, we assume that $a \geq b \geq 0$. The expressions for the sum and the compensated error become

$$\hat{s} = fl(a - b) \qquad \text{and} \qquad \hat{e} = fl((a - \hat{s}) - b).$$

It has been shown that this error estimate is always exact irrespective of the arithmetic radix, β, provided that subtraction is faithful [19, 1]. It is worth emphasizing that the error estimate assumes $|a| \geq |b|$. For $|b| > |a|$ we must use $\hat{e} = fl((b - \hat{s}) - a)$. The choice of formula can be determined easily from the sign of $\hat{s}$.

We refer to the use of compensation in this manner as deflation since we replace two floating-point numbers with two new numbers which are no larger in magnitude but which have the difference between the respective pairs of numbers remaining exactly the same:

$$a \geq \hat{s} \qquad \text{and} \qquad b \geq |e| \qquad \text{and} \qquad a - b = \hat{s} - e.$$

Note that we have dropped the $\hat{}$ from the e, indicating that the value is exact.

2.1. The iterative deflation step. It is now clear how deflation can be used in an iterative manner to calculate the sum of a set of data. We summarize the new algorithm in the following steps.

1. Sort the numbers into decreasing order of absolute magnitude. Thus,

$$|x_1| \geq |x_2| \geq |x_3| \geq \cdots \geq |x_m|.$$

2. Find the first pair of consecutive numbers that have opposite sign. Let a and b be this pair of numbers.

3. Remove the original numbers a and b from the list of numbers.
4. Deflate these two numbers to produce two smaller numbers.
5. Replace the two deflated numbers into the list.
6. Resort the set of numbers into decreasing order of absolute magnitude. (This can be achieved easily by positioning the two new numbers in the appropriate position within the list of sorted numbers.)
7. Repeat the process from step 2 until all the numbers in the list are of the same sign.
8. Use the standard method of compensated summation to sum the remaining numbers.

By the time we get to step 8, we have a set of data which has condition number $R = 1$ and whose sum is *exactly* the same as the sum of the original set of data. No unrecovered errors have been made up to that point. Step 8 sums the new set of data using compensated addition which produces a value for the sum that has a relative error bound of 2η (plus second-order terms). The final sum not only is forward and backward stable, it has a relative error bounded by a quantity that is independent of the condition number of the original data.

2.2. Reduction. If we try to deflate two numbers whose magnitudes differ considerably, the deflation step produces the values $\hat{s} = a$ and $e = b$. The iterative step has not changed the data and so the same two values will be used on each subsequent iteration, ad nauseam. In order to avoid an infinite loop we reduce a to two smaller numbers, a_- and a_e, such that $a_e = \beta^k$ and k is the smallest integer such that

$$fl(a_- + a_e) = a_- + a_e = a. \tag{4}$$

The algorithm for reducing a is similar to the algorithm for computing the relative machine precision η.

ALGORITHM 2. REDUCTION.

```
t=a;
while ((a-t)!=a),
   u=t; t=t/2;
end
t=a-u; u=a-t;
```

Here `t` represents a_- and `u` represents a_e. The last instruction (`u=a-t;`) of the algorithm is only needed if the radix, β, is not a power of 2. It ensures that the value of a_e satisfies (4) and is not simply an arbitrary value such that $fl(a_- + a_e) = a$ with $a_- + a_e \neq a$.

We now have the two numbers a_e and b which we may deflate. If a_e and b have no overlap, then we may reduce a_e inductively until deflation is possible. Clearly, deflation must occur eventually since there is a finite number of floating-point numbers.

2.3. Convergence. The iterative process stops when all of the remaining deflated numbers are of the same sign, namely,

$$\sum_i |x_i| = \left| \sum_i x_i \right|.$$

For each iteration of this distillation algorithm we take the two numbers a and b and deflate them (via reduction if necessary) to produce the two numbers s and e, such that $a - b = s - e$. Note that

$$|a| + |b| > |s| + |b| > |s| + |e|,$$

and so the sum of the absolute values of the deflated numbers is strictly less than the sum of the absolute values of the numbers before deflation. Repeating this process iteratively produces a sequence of sums for the absolute values which is strictly decreasing and bounded below. Therefore the sequence must converge and a reductio ad absurdum argument shows that the convergence corresponds to all the deflated numbers having the same sign.

3. Practical considerations of the deflation algorithm.

3.1. Timings for the various methods. Given n data values, standard recursive, pairwise, and compensated summation all require $\mathcal{O}(n)$ floating-point operations (flops) although compensated summation is approximately four times slower than recursive and pairwise summation. Arranging the data into increasing or decreasing order of absolute magnitude requires an additional $\mathcal{O}(n \log n)$ operations, thus making these two methods considerably slower than recursive summation.

For the deflation algorithm presented here there are three main requirements that contribute to the overall operation count. First, the initial sort requires $\mathcal{O}(n \log n)$ operations. Second, the number of operations required to resort the outputs from each deflation is approximately $\mathcal{O}(n)$.[1] Finally, we must estimate the number of deflations that are required to reduce all the data to the same sign. We can bound the number of deflations by considering how many deflations are required to eliminate a single data value ($e = 0$). We note from the deflation process that the error term, e, has some trailing zero digits, the number of zero digits being at least as many as the number of digits of overlap between numbers a and b. Clearly, the error term must gain at least one trailing zero digit compared to the number of trailing zeros in b. Thus the largest number of deflations required to eliminate one number is equal to the number of digits in the floating-point representation of each number. For double precision binary, this is 53—a constant that is independent of n. So, the total number of deflations required to reduce all the data to the same sign is $\mathcal{O}(n)$. This agrees with practical experience. Data sets of varying sizes were summed using the algorithm above and the number of deflations used was recorded. The data were taken from a normal distribution of mean zero and variance 1:

n	4	16	64	256	1024	4096	16384	65536
number of deflations	4	11	44	122	509	2108	9522	32715

Here we note that the number of deflations is approximately linear in n and considerably smaller than the worst-case estimate of $53n$. Overall the operation count for this deflation algorithm is $\mathcal{O}(n^2) + \mathcal{O}(n \log n)$. Although the number of deflations required is very reasonable, the time taken to sort and resort the data makes the method prohibitively slow for large data sets.

3.2. Avoiding the direct sort. Sorting the data has several advantages, namely, deflating the two largest numbers of opposite sign reduces the condition number, R, as much as possible. Also, if the two numbers have the same exponent, or if $a/2 \leq b \leq 2a$, then the subtraction $s = a - b$ will be exact, thereby reducing the size of the problem [4, 19].

[1] Although a binary search method allows the new *positions* within the ordered list of the two deflated numbers to be calculated in $\mathcal{O}(\log n)$ time, the list still requires reordering to accommodate these new numbers. This latter step requires $\mathcal{O}(n)$ time [11, p. 83].

However, sorting is not essential since all numbers of one particular sign need to be deflated and eliminated for the algorithm to converge. The order in which this elimination takes place is not important. Further, deflation automatically sorts the data:

$$|e| = |s - fl(s)| = |s - s(1+\delta)| = |\delta s| \leq \eta |s|. \tag{5}$$

Since direct sorting and deflation both involve calculating $s = a - b$, it is worth using just deflation as an indirect sorting method as the program executes.

3.3. The new modified deflation algorithm.

1. The data are split into two sets: one for positive numbers and one for negative numbers. We shall refer to these sets as $\mathcal{P}$ and $\mathcal{N}$, respectively.
2. For each of these sets we remove the last number and deflate to produce a sum (difference), s, and an error term, e. The value s is returned to the end of the appropriate set $\mathcal{P}$ or $\mathcal{N}$ depending on its sign and the value e is discarded into an initially empty set $\mathcal{E}$ of "small" terms.
3. Repeat step 2 until one of the sets $\mathcal{P}$ or $\mathcal{N}$ is empty.
4. The data in the set $\mathcal{E}$ are redistributed to the sets $\mathcal{P}$ and $\mathcal{N}$ as appropriate and the data in these two sets are summed, using compensation, to obtain s_+ and s_-, respectively. The condition number of summation of the remaining terms is computed as $R = |(s_+ + s_-)/(s_+ - s_-)|$. If this estimate for R is not equal to 1, then the process is repeated from step 2.
5. The final sum is computed using compensated summation.

From (5) we observe that the numbers which are discarded into the set $\mathcal{E}$ are considerably smaller than the largest number from the sets $\mathcal{P}$ and $\mathcal{N}$. Thus, the smaller insignificant numbers are quickly rejected and the relatively large numbers remain in the sets $\mathcal{P}$ and $\mathcal{N}$ to be deflated as though the whole set of data had been sorted in the first place.

3.4. Reduction. As with the original deflation algorithm, it is possible that this method may be caught in an infinite loop. Consider, for example, the case where the set $\mathcal{P}$ contains one large number and the set $\mathcal{N}$ contains many small numbers. Steps 2 and 3 of the algorithm involve no useful deflations and all the data from $\mathcal{N}$ are discarded into $\mathcal{E}$ and then step 4 returns them to $\mathcal{N}$. At this stage it is appropriate to reduce the last number in $\mathcal{P}$ into two smaller numbers as described earlier. In practice it is more useful to extract more than just the last bit of the last number from the nonempty set $\mathcal{P}$ or $\mathcal{N}$. A recommended extraction is to split the data value into the approximate ratio $\sqrt{\eta}$ to $1 - \sqrt{\eta}$. This works for the case when one set has numbers that are much larger than the other set and also for the case where the set of smaller numbers has many more elements than the set of larger values.

4. Results. Six of the summation methods discussed in this paper have been compared for speed and accuracy. Each method was executed three times and the combined time is presented in Table 1. The original deflation algorithm (last column) has some entries missing for the larger data sets because the method was far too slow and impractical. Each method was written in FORTRAN 90 and run on a Sun Classic workstation. The UNIX command `timex` was used to obtain the timings correct to the nearest one-hundredth of a second. The data were sampled from the uniform distribution $[0, 1]$ using the FORTRAN 90 command `RANDOM_NUMBER` and the mean of the data (calculated using single precision recursive summation) was subtracted from each datum in order to induce ill conditioning.

TABLE 1

Total timings (in seconds) for three summation runs achieved using various methods and data sets.

Number of data	Summation method					
	Recursive	Compensated	Increasing	Decreasing	Modified deflation	Original deflation
256	3.50	3.50	3.53	3.60	3.56	3.93
1024	3.56	3.59	3.74	3.57	3.56	10.15
4096	3.60	3.62	4.10	4.16	3.68	108.12
16384	3.79	3.84	6.67	6.71	4.21	1619.19
65536	4.78	4.86	20.33	20.28	6.25	—
262144	8.64	9.48	91.10	90.92	14.76	—
1048576	23.86	27.24	515.39	515.38	49.15	—

We begin by observing that these results include computational overheads which have affected the timings. These overheads are most noticeable for the smaller data sets where 1024 data apparently can be summed as fast as 256 data. From this we can deduce that the majority of the time used for recursive summation is due to computational and system overheads and only a fraction of the time is due to the actual algorithm.

Another observation is that the original deflation algorithm is far too slow and impractical, whereas the modified deflation algorithm is considerably faster than the increasing and decreasing order summation algorithms, both of which use an $\mathcal{O}(n \log n)$ sorting routine. Indeed, the new deflation algorithm appears to be an $\mathcal{O}(n)$ algorithm. This seems to contradict the observation of Kahan, who states that distillation style algorithms (of which the new deflation method is an example) appear to be at least $\mathcal{O}(n \log n)$ [6]. The main reason for this discrepancy is that the modified algorithm does not take distillation to its natural conclusion and terminates considerably earlier when $fl(R) = 1$. If we allowed the modified deflation algorithm to continue until all the data had the same sign, then the algorithm would indeed be at least $\mathcal{O}(n \log n)$. It is the early, but accurate, termination condition that allows practical applications to run in $\mathcal{O}(n)$ time. It is of course possible to produce pathological data sets which require at least $\mathcal{O}(n \log n)$ time. This can be achieved only by using a data set that requires the main iterative step of the deflation algorithm to be executed at least $\mathcal{O}(\log n)$ times. For example, consider the data set of $n = 2m + 1$ floating-point numbers $\{1, X, X^2, \ldots, X^m, -X^m, \ldots, -X^2, -X\}$, where X is so large that $fl(X + 1) = X$. On each main iterative step, only the last positive number and the first negative number will deflate and so $n/2 - 1$ main iterations are needed to produce the required sum of 1. This set of data, as with all data sets that use so many iterations, is exceptionally ill conditioned and thus all of the standard, faster summation routines would produce results that were unreliable.

The accuracy of three of the methods is given in Table 2. Recursive and decreasing summation have been omitted since, in general, they are slightly less accurate than the method of increasing order of absolute magnitude. A more detailed comparison of these three methods can be found elsewhere [5, 6]. Since the two new deflation algorithms produce identical results, the values have been presented only once. Finally, an estimate of the exact summation, correct to single precision, has been obtained by summing the data in double precision. We note that for each value of n, the deflation algorithms produce exactly the same answer as that obtained using double precision. Compensated summation performs consistently better than the increasing order, and indeed occasionally produces the same accuracy as deflation.

TABLE 2
Accuracy achieved using various methods and data sets.

Number of data	Summation method			
	Increasing	Compensated	Deflation	Double
256	8.25524330e-6	9.11951065e-6	9.11951065e-6	9.11951065e-6
1024	2.60829926e-4	2.63065100e-4	2.63065100e-4	2.63065100e-4
4096	7.36564398e-4	7.34597445e-4	7.34686852e-4	7.34686852e-4
16384	3.62098217e-4	3.92526388e-4	3.92556190e-4	3.92556190e-4
65536	-1.49576068e-2	-1.49446130e-2	-1.49446428e-2	-1.49446428e-2
262144	-1.18375301e-0	-1.18361986e-0	-1.18361986e-0	-1.18361986e-0
1048576	-5.68951273e-0	-5.69462442e-0	-5.69462395e-0	-5.69462395e-0

In addition to the results given above, the new algorithm was tested on many different sets of data, and for each case the method never required more than two main iterative steps, with the majority of cases needing only one main iteration. One of the referees observed that since the error bound always satisfies (3), an alternative termination condition could be chosen, such as $fl(R) \leq 2$, which should not affect the accuracy significantly. For all the practical examples considered, the algorithm now converged after only one main iteration with the estimates for the final sums being identical to the previous estimates. Finally, it is easy to alter the algorithm so that the user can specify the termination condition, $fl(R) \leq \mu$, based on the required relative error bound.

5. Comparison with other methods. In its current form, this algorithm has some similarities to Pichat's algorithm for summing data using compensated addition [17]. In its basic form using a radix, $\beta = 2$ or 3, Pichat's algorithm sums the data using a variation of standard recursive addition as follows.

ALGORITHM 3. PICHAT'S METHOD.

```
s=x(1);
for i=2:n;
   a=s; s=a+x(i);
   e(i-1)=(a-s)+x(i);
end
e(n)=s;
```

The algorithm produces an estimate for the required sum together with the data set of errors produced at each step of the addition. Pichat suggests that an improved estimate for the required sum can be obtained by replacing the data set `x(1:n)` with the set `e(1:n)` and repeating the algorithm until the errors are suitably small. Pichat proves that this process converges in polynomial time, although several iterations may be needed for the errors to become sufficiently small.

The main problem with this method is that we are guaranteed only that the errors have been computed exactly when $|a| \geq |x(i)|$ and that the radix is either $\beta = 2$ or 3. This problem can be overcome since it is relatively easy to test for the condition $|a| < |x(i)|$, replacing the formula for the error estimate by `e(i-1)=(x(i)-s)+a;` where appropriate. Also, for radices, $\beta \geq 4$, the algorithm can be adapted by using a slightly more complicated form of compensation [18, 19].

Leuprecht and Oberaigner [13] apply Pichat's algorithm to a parallel environment, while Bohlender [2] extends the algorithm to compute the sum to arbitrary precision by storing the result as several nonoverlapping floating-point numbers. He also adds termination conditions if only limited precision is required. Unfortunately, this latter approach requires double length accumulators for each of the computations. In a

similar style, Kulisch and Miranker [12] propose a technique which involves dynamic length accumulators and the ability to extract the exponent of a floating-point number efficiently. A variation is given by Malcolm [15], who simulates high-precision accumulators by decomposing each datum into several lower-precision components. This effectively increases the number of data involved in the summation by a significant factor. Direct access to the exponents of the data is also required.

The main attractions of this new modified deflation algorithm over these existing methods are as follows:

- It employs only standard floating-point arithmetic and does not rely on the architecture of specific machines, nor does it use accumulators.
- It addresses the central problem with summation, namely the ill-conditioning of the data, by concentrating directly on reducing the value of R using deflation.
- By insisting on using compensated subtraction throughout, we are guaranteed that the summation will not overflow unnecessarily and that the errors are computed exactly, irrespective of the radix, β.

Another useful addition routine is Priest's "doubly compensated summation" which, like the deflation algorithms presented here, produces a value for the required sum that is independent of the condition number. Unfortunately, it has two minor drawbacks. First, as with the increasing, decreasing, and original deflation methods, the data must be ordered beforehand, and second, the algorithm assumes that all floating-point numbers satisfy

$$|b| \leq |a| \text{ implies } fl(a+b) \leq 2|a|. \tag{6}$$

This is reasonable for binary arithmetic, but using four-digit decimal arithmetic we have $fl(0.9999 + 0.9998) = 2.000$.[2]

6. Conclusions. We have presented a summation algorithm which is accurate and robust as well as being sufficiently fast to be used in practice. The algorithm uses deflation in an iterative manner to eliminate the ill conditioning of the data. Once the ill conditioning has been removed, a standard summation method is used to obtain the final sum. The method is backward stable and, unlike most addition routines, produces a sum which has a small relative error, irrespective of the original data.

Acknowledgments. I wish to thank C. K. Pink for his comments on an earlier draft of the paper and both referees for suggesting improvements to the manuscript.

REFERENCES

[1] I. J. Anderson, *A Note on Exact Compensation of Floating-Point Numbers*, Technical report RR9703, School of Computing and Mathematics, University of Huddersfield, Huddersfield, UK, 1996.

[2] G. Bohlender, *Floating-point computation of functions with maximum accuracy*, IEEE Trans. Comput., C-26 (1977), pp. 621–632.

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

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

[5] J. Gregory, *A comparison of floating point summation methods*, Comm. ACM, 15 (1972), p. 838.

[2] It has been shown that although cases of failure exist, they are exceedingly rare [1]. For example, on the Hewlett Packard 28 and 48G calculators, there are approximately 10^{30} pairs of values for a and b, of which fewer than 6000 pairs violate (6) (excluding overflow and underflow).

[6] N. J. Higham, *The accuracy of floating point summation*, SIAM J. Sci. Comput., 14 (1993), pp. 783–799.
[7] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, PA, 1996.
[8] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE standard 754-1985, IEEE Computer Society, New York, 1985.
[9] *A Radix-Independent Standard for Floating-Point Arithmetic*, ANSI/IEEE standard 854-1987, IEEE Computer Society, New York, 1987.
[10] W. Kahan, *Further remarks on reducing truncation errors*, Comm. ACM, 8 (1965), p. 40.
[11] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
[12] U. W. Kulisch and W. L. Miranker, *The arithmetic of the digital computer: A new approach*, SIAM Rev., 28 (1986), pp. 1–40.
[13] H. Leuprecht and W. Oberaigner, *Parallel algorithms for the rounding exact summation of floating point numbers*, Computing, 28 (1982), pp. 89–104.
[14] P. Linz, *Accurate floating-point summation*, Comm. ACM, 13 (1970), pp. 361–362.
[15] M. A. Malcolm, *On accurate floating-point summation*, Comm. ACM, 14 (1971), pp. 731–736.
[16] R. Marshall and K. Paul, *Pop-up Addition*, Tango Books, London, UK, 1984.
[17] M. Pichat, *Correction d'une somme en arithmetique a virgule flottante*, Numer. Math., 19 (1972), pp. 400–406.
[18] 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 Press, Los Alamitos, CA, 1991, pp. 132–143.
[19] 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; also available online from ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z.
[20] G. W. Stewart, *Afternotes on Numerical Analysis*, SIAM, Philadelphia, PA, 1996.
[21] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, Notes in Applied Science 32, Her Majesty's Stationery Office, London, UK, 1963.