On Floating-point Summation *

T. O. Espelid †

Abstract

In this paper we focus on some general error analysis results in floating-point summation. We emphasize analysis useful from both a scientific and a teaching point of view.

Keywords: Floating-point summation, rounding errors, orderings.
AMS subject classification. primary 65G05, secondary 65B10.

1 Introduction

The floating-point summation has, over the years, got considerable attention since Wilkinson's, [11, 12], famous backward analysis from the early sixties. Over the last five years several papers have been published focusing on this topic, e. g. Robertazzi and Schwartz, [10], Dixon and Mills, [3], Goldberg, [5], and Higham, [6]. Comprehensive lists of references can be found in [5, 6].

This author too studied this subject in the late seventies, Espelid [4]. My interest stemmed from teaching numerical analysis courses: all textbooks that I knew would present the backward result. I observed that in doing so they failed to give good advice to students on how to compute e. g. the denominator in Aitken extrapolation for a linearly converging series $\ldots, t_{i-1}, t_i, t_{i+1}, \ldots \rightarrow t$: $d = t_{i+1} - 2t_i + t_{i-1}$. The backward analysis is of course important, but in my opinion the true nature of floating-point summation becomes hidden if one gives results for the backwards analysis only.

This paper has been inspired by Higham's discussions in [6]. However, instead of experimenting with different orderings I want to highlight some main results from [4] and [6] in a less technical manner. I do hope that both scientists and teachers of numerical analysis courses may find this presentation as useful as I have over the years.

Before we start on the floating-point discussion let us first consider the following natural summation algorithm

*Published by: *SIAM Review, 37(4), Dec. 1995, 603-607.*

†Author's address: Department of Informatics, University of Bergen, Høyteknologisenteret, 5020 Bergen, Norway

Algorithm Sum

Initialize: Given a set of $n+1$ numbers $\{x_i\}_0^n$
while $n > 0$ **do**
Pick two numbers from the set, say x and y.
Compute $z \leftarrow x + y$.
Let z replace x and y in the set,
and put $n \leftarrow n - 1$.
end

Higham, [6], uses the term insertion for this type of algorithm which is discussed in [4] too. We may associate a binary summation tree with this algorithm: the leaves contain the values $x_i,\ i = 0, 1, 2, \ldots, n$, and the internal nodes contain the intermediate sums $s_i, i = 1, 2, \ldots, n$. Obviously we have for the root of the tree

$$s_n = \sum_{i=0}^{n} x_i.$$

Each s_i is the exact sum of a subset of the $n+1$ x-values. Thus to each s_i, or to each internal node in the binary summation tree, we may associate the indices corresponding to this particular subset of x-values.

By making different choices in the algorithm we may get different binary summation trees. Define two binary summation trees as different if there is one internal node in one of the trees, with a particular associated index set, which does not appear in the other tree. A natural question is then: how many different binary summation trees are there?

Let $g(m)$ be the number of different binary summation trees with m leaves. We find by simple counting, with $g(1) = 1$, that

$$g(m) = \frac{1}{2} \sum_{j=1}^{m-1} \binom{m}{j} g(j)g(m-j), \text{ for } m \geq 2.$$

The factor $1/2$ comes from symmetry considerations. Define $g(0) = -1$, then this can be written

$$\sum_{j=0}^{m} \binom{m}{j} g(j)g(m-j) = 0, \text{ for } m \geq 2.$$

Defining the generating function $F(x) = \sum_{m\geq 0} x^m g(m)/m!$ it is easy to see that $(F(x))^2 = 1 - 2x$ and using this fact we find $g(m) = 1 \cdot 3 \cdot 5 \cdots (2m-3) = (2m-3)!!$, known as double factorial. Thus the number of different ways to add the $n+1$ numbers increases enormously with n: 1, 3, 15, 105, 945, 10 395, 135 135, ... This expression for $g(m)$ is probably well known: e. g. it is given in Knuth [8, p. 639].

2 Floating-point summation

Assume that we have a machine with standard floating-point arithmetic with rounding. Let $\hat{x}$ denote a floating-point machine number approximating the exact number x.

Lemma 1 *Let u be the unit roundoff on a machine with standard floating-point arithmetic. Then the floating-point error may be represented both as*

$$fl(\hat{x}\ op\ \hat{y}) - (\hat{x}\ op\ \hat{y}) = (\hat{x}\ op\ \hat{y})\ \phi,\ |\phi| \leq u, \tag{1}$$

and

$$fl(\hat{x}\ op\ \hat{y}) - (\hat{x}\ op\ \hat{y}) = fl(\hat{x}\ op\ \hat{y})\ \psi,\ |\psi| \leq u, \tag{2}$$

*with $\psi = \phi/(1+\phi)$. op can be either +, -, * or /.*

(1) is the standard representation used for backward analysis, while (2) was introduced by Babuška [1]. Observe that the relation between ϕ and ψ implies that $-u/(1+u) \leq \phi \leq u$ and $-u \leq \psi \leq u/(1+u)$, a fact of no practical value.

The error in $\hat{x}$, as an approximation to x, will in the following be denoted $e_{\hat{x}} = \hat{x} - x$. Define $\hat{z} = fl(\hat{x} + \hat{y})$ then (2) implies

$$e_{\hat{z}} = e_{\hat{x}} + e_{\hat{y}} + \psi\hat{z},\ |\psi| \leq u. \tag{3}$$

Now assume that we implement Algorithm Sum on this machine using floating-point arithmetic. Assume furthermore that we make the same choices as with exact arithmetic producing, from the machine values $\{\hat{x}_i\}_{i=0}^n$, the intermediate machine sums $\hat{s}_i,\ i = 1, 2, \ldots, n$. Using (3) on each node in the binary summation tree we formulate the result in a theorem.

Theorem 1 *The total error in the computed sum $\hat{s}_n$, using Algorithm Sum (assuming no over-/under-flow), and producing intermediate machine sums $\hat{s}_i,\ i = 1, 2, \ldots, n$, may be written*

$$\hat{s}_n - s_n = \sum_{i=1}^{n} \psi_i \hat{s}_i + \sum_{j=0}^{n} e_{\hat{x}_j}, \text{ with } |\psi_i| \leq u, i = 0, 1, \ldots, n, \tag{4}$$

giving a bound for the error

$$|\hat{s}_n - s_n| \leq u \sum_{i=1}^{n} |\hat{s}_i| + \sum_{j=0}^{n} |e_{\hat{x}_j}|. \tag{5}$$

In my opinion this theorem contains the essence of floating-point summation. The initial errors have an effect on the final sum which is independent of the choices made in Algorithm Sum. The choices made will influence the final sum through the intermediate sums $\hat{s}_i$. Thus making choices that create small intermediate sums seems like the best advice in general. This result should appear in textbooks and will make floating-point summation less obscure to beginners in this field.

The typical student question about how to compute the denominator in Aitken acceleration is now trivial. Indeed the situation is even more pleasant than Theorem 1 indicates in this case due to the following well known result concerning cancellation.

Lemma 2 *Let $\hat{x}$ and $\hat{y}$ be machine numbers in a binary floating-point machine such that $\hat{x}\hat{y} < 0$ and $2|\hat{y}| \geq |\hat{x}| \geq |\hat{y}|$. Then $fl(\hat{x} + \hat{y}) = \hat{x} + \hat{y}$.*

The optimal way of computing the Aitken denominator is

$$d = (t_{i+1} - t_i) - (t_i - t_{i-1}).$$

Using a binary floating-point machine we get $\psi_1 = \psi_2 = 0$, giving only a small relative error due to the last subtraction since the terms in the sequence $\{t_i\}$ are approximately equal when i is large.

In order to do a complete backward analysis of the implementation of Algorithm Sum we may use (1) to obtain the following result

$$|e_{\hat{z}}| \leq (|e_{\hat{x}}| + |e_{\hat{y}}| + u|z|)(1 + u). \tag{6}$$

This gives a bound for the error in any internal node in the binary summation tree expressed by the error in each of it's children and the error introduced when computing $\hat{z}$ from these children expressed by the exact sum z. In order for the bound to be true we have to perturb the bound by the factor $(1+u)$ which is unnecessary if we replace z by $\hat{z}$. The nice property of (6) is that it is easy to use recursively in the binary summation tree (by induction in the height of the tree) giving an alternative to (5)

Theorem 2 *The total error in the computed sum $\hat{s}_n$ may be bounded using the exact intermediate sums s_i, $i = 1, 2, \ldots, n$,*

$$|\hat{s}_n - s_n| \leq (u \sum_{i=1}^{n} |s_i| + \sum_{j=0}^{n} |e_{\hat{x}_j}|)(1 + u)^h, \tag{7}$$

where h denotes the height of the binary summation tree and $\lceil \log_2(n + 1) \rceil \leq h \leq n$. Alternatively we may write

$$|\hat{s}_n - s_n| \leq (u \sum_{i=1}^{n} m_i|x_i| + \sum_{j=0}^{n} |e_{\hat{x}_j}|)(1 + u)^h, \tag{8}$$

where m_i is the distance from the leaf containing x_i to the root of the binary summation tree, with $1 \leq m_i \leq h$.

To prove (8) recall that s_i is the exact sum of a subset of the exact x-values. Let m_i be the number of additions x_i has been part of, either explicitly or embedded in some s_j, then this is equivalent to the distance from the leaf containing x_i to the root. With this observation (8) follows from (7). We see that while (5) and (7) are realistic upper bounds (may be achieved) on the error (8) may be a substantial overestimate if the terms in the sum are of different sign.

Minimizing the height in the binary summation tree, known as Linz/Babuška summation, [1, 9], gives $h = \lceil \log_2(n + 1) \rceil$ and thus the minimum maximum perturbation of each x_i in (8).

Recursive summation, that is $\hat{s}_1 = fl(\hat{x}_0 + \hat{x}_1)$ and $\hat{s}_i = fl(\hat{s}_{i-1} + \hat{x}_i)$, on the other hand gives $h = m_0 = n$ and $m_i = n - i + 1, i = 1, 2, \ldots, n$, establishing the most cited backward result by Wilkinson. The factor $(1+u)^h$ may be replaced by

$$(1+u)^h < \exp(uh),$$

giving a perturbation of the bound less than 1.1 if $uh \leq .1$. Let us consider a case where it is of little value to spend extra effort on the floating-point summation:

Theorem 3 *Assume that the relative initial error is uniformly bounded, e. g.* $|e_{\hat{x}_j}| \leq \varepsilon |x_j|$, *then*

$$|\hat{s}_n - s_n| \leq (uh + \varepsilon)(\sum_{i=0}^{n} |x_i|)(1+u)^h. \tag{9}$$

This follows directly from (8). We see from (9) that when $un < \varepsilon$ (recall $h \leq n$) then this backward analysis suggests that we should not worry about the choices in Algorithm Sum (at least for $un << 1$). On the other hand if $\varepsilon = O(1)u$ then the same analysis indicates that the summation error may be dominating and such an effort may be worthwhile. From (9) we see that Linz/Babuška summation will minimize this bound, but (7) suggests that we can do even better in many cases.

Since the number of different binary summation trees becomes enormous with large values of n it is out of the question, in general, to search all these for the optimal summation tree (that is: the tree that minimizes either (5) or (7) or both) for a given set of x-values. However, in some cases the optimal tree is easy to find.

1) If $x_i > 0,\ i = 0, 1, \ldots, n$, then we may choose the two smallest values x and y (each time) in Algorithm Sum to get the optimal tree. It is well known that this is optimal, see D. Huffman's procedure , [7], or Caprani [2]. If the set is sorted in advance then this procedure is in addition easy to apply.

Recursive summation is optimal in the equal sign case if the set is sorted in increasing order and each intermediate sum is less than all the remaining (except the smallest) x-values. An example is: $0 < x_{j-1} \leq \alpha x_j,\ j = 1, 2, \ldots, n$ with $0 < \alpha \leq (\sqrt{5} - 1)/2$.

Linz/Babuška summation is optimal if all x-values are approximately equal. In this case (9) gives a very nice relative error bound that is realistic too with h minimal.

2) In [6] a number of different strategies were considered when we have the different sign situation. None of the strategies was a general winner in the experiments performed in [6]. One of these strategies, the separation strategy, is to compute the sum of all positive and negative x-values separately, using a good strategy for each subproblem. As remarked in [6] this is almost never optimal.

I feel that such strategies appear in the literature due to the fact that results like (4), (5) and (7) never have, to this authors knowledge, been given in textbooks. Everyone knows that cancellation may be harmful, and indeed this separation strategy minimizes the number of possible cancellations! However, as illustrated by the theorems 1 and 2 and Lemma 2, cancellation in floating-point summation is the best thing that can happen and the more frequent the better! Here we should note that our Aitken example makes use of this particular fact. The trouble with cancellation is the conflict between large initial errors and the fact that the final sum may be quite small. If this is a major concern then one

should reformulate the problem to avoid the summation! If not, then take advantage of the cancellations!

Of course it is possible to experience cases where the separation strategy is optimal: assume that $x_i > 0,\ i = 0, 1, 2, \ldots, n-1$, and $x_n \leq -2\sum_{i=0}^{n-1} x_i$ then we see indeed that the separate summation of positive and negative values is optimal. This fact illustrates how hard it is to design an algorithm which is optimal in general and at the same time implies a reasonable amount of extra work relative to the n floating-point additions which is the original job to be done.

References

[1] I. Babuška. Numerical stability in mathematical analysis. In *Proc. IFIP Congress, Information Processing 68*, pages 11–23. North-Holland, Amsterdam, 1969.

[2] O. Caprani. Roundoff errors in floating-point summation. *BIT*, 15:5–9, 1975.

[3] L. C. W. Dixon and D. J. Mills. The effect of rounding errors on the available metric method. Technical Report 229, Numerical Optimization Centre, Hatfield Polytechnic, Hatfield, UK, 1990.

[4] T. O. Espelid. On floating-point summation. Report 67, Dept. of Appl. Math., Univ. of Bergen, 1978.

[5] David Goldberg. What every scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[6] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14:783–799, 1993.

[7] Donald Knuth. *Sorting and Searching*, volume 3 of *The art of computer programming.* Addison-Wesley, 1 edition, 1973.

[8] Donald Knuth. *Seminumerical algorithms*, volume 2 of *The art of computer programming.* Addison-Wesley, 2 edition, 1981.

[9] Peter Linz. Accurate floating-point summation. *Comm. ACM*, 13:361–362, 1970.

[10] T. G. Robertazzi and S. C. Schwartz. Best "orderings" for floating-point summation. *ACM Trans. Math. Softw.*, 14:101–110, 1988.

[11] J. H. Wilkinson. Error analysis in floating-point computation. *Numer. Math.*, 2:319–340, 1960.

[12] J. H. Wilkinson. Rounding errors in algebraic processes. Technical Report 32, Her Majesty's Stationary Office, London, 1963.