WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Français
Lietuvių
日本語
Polski
Русский
Українська
Edit links

# Kahan summation algorithm

From Wikipedia, the free encyclopedia

In numerical analysis, the **Kahan summation algorithm** (also known as **compensated summation** [1]) significantly reduces the numerical error in the total obtained by adding a sequence of finite precision floating point numbers, compared to the obvious approach. This is done by keeping a separate *running compensation* (a variable to accumulate small errors).

In particular, simply summing *n* numbers in sequence has a worst-case error that grows proportional to *n*, and a root mean square error that grows as $\sqrt{n}$ for random inputs (the roundoff errors form a random walk).[2] With compensated summation, the worst-case error bound is independent of *n*, so a large number of values can be summed with an error that only depends on the floating-point precision.[2]

The algorithm is attributed to William Kahan.[3] Similar, earlier techniques are, for example, Bresenham's line algorithm, keeping track of the accumulated error in integer operations (although first documented around the same time[4]) and the Delta-sigma modulation[5] (integrating, not just summing the error).

**Contents** [hide]

## The algorithm [edit]

In pseudocode, the algorithm is:

```
function KahanSum(input)
    var sum = 0.0
    var c = 0.0                  // A running compensation for lost low-order bits.
    for i = 1 to input.length do
        var y = input[i] - c     // So far, so good: c is zero.
        var t = sum + y          // Alas, sum is big, y small, so low-order digits
of y are lost.
        c = (t - sum) - y        // (t - sum) recovers the high-order part of y;
subtracting y recovers -(low part of y)
        sum = t                  // Algebraically, c should always be zero. Beware
overly-aggressive optimizing compilers!
                                 // Next time around, the lost low part will be
added to y in a fresh attempt.
    return sum
```

### Worked example [edit]

This example will be given in decimal. Computers typically use binary arithmetic, but the principle being illustrated is the same. Suppose we are using six-digit decimal floating point arithmetic, *sum* has attained the value 10000.0, and the next two values of *input(i)* are 3.14159 and 2.71828. The exact result is 10005.85987, which rounds to 10005.9. With a plain summation, each incoming value would be aligned with *sum* and many low order digits lost (by truncation or rounding.) The first result, after rounding, would be 10003.1. The second result would be 10005.81828 before rounding, and 10005.8 after rounding. This is not correct.

However, with compensated summation, we get the correct rounded result of 10005.9.

Assume that *c* has the initial value zero.

```
  y = 3.14159 - 0                   y = input[i] - c
  t = 10000.0 + 3.14159
    = 10003.1                       Many digits have been lost!
  c = (10003.1 - 10000.0) - 3.14159 This must be evaluated as written!
    = 3.10000 - 3.14159             The assimilated part of y recovered, vs. the
original full y.
    = -.0415900                     Trailing zeros shown because this is six-digit
arithmetic.
sum = 10003.1                       Thus, few digits from input(i) met those of sum.
```

The sum is so large that only the high-order digits of the input numbers are being accumulated. But on the next step, *c* gives the error.

```
  y = 2.71828 - -.0415900           The shortfall from the previous stage gets
included.
    = 2.75987                       It is of a size similar to y: most digits meet.
  t = 10003.1 + 2.75987             But few meet the digits of sum.
    = 10005.85987, rounds to 10005.9
  c = (10005.9 - 10003.1) - 2.75987 This extracts whatever went in.
    = 2.80000 - 2.75987             In this case, too much.
    = .040130                       But no matter, the excess would be subtracted
off next time.
sum = 10005.9                       Exact result is 10005.85987, this is correctly
rounded to 6 digits.
```

So the summation is performed with two accumulators: *sum* holds the sum, and *c* accumulates the parts not assimilated into *sum*, to nudge the low-order part of *sum* the next time around. Thus the summation proceeds with "guard digits" in *c* which is better than not having any but is not as good as performing the calculations with double the precision of the input. However, simply increasing the precision of the calculations is not practical in general; if *input* is already double precision, few systems supply quadruple precision and if they did, *input* could then be quadruple precision.

## Accuracy [edit]

A careful analysis of the errors in compensated summation is needed to appreciate its accuracy characteristics. While it is more accurate than naive summation, it can still give large relative errors for ill-conditioned sums.

Suppose that one is summing *n* values $x_i$, for *i*=1,...,*n*. The exact sum is:

$$S_n = \sum_{i=1}^{n} x_i \text{ (computed with infinite precision)}$$

With compensated summation, one instead obtains $S_n + E_n$, where the error $E_n$ is bounded above by:[2]

$$|E_n| \leq \left[2\varepsilon + O(n\varepsilon^2)\right] \sum_{i=1}^{n} |x_i|$$

where ε is the machine precision of the arithmetic being employed (e.g. $\varepsilon \approx 10^{-16}$ for IEEE standard double precision floating point). Usually, the quantity of interest is the relative error $|E_n|/|S_n|$, which is therefore bounded above by:

$$\frac{|E_n|}{|S_n|} \leq \left[2\varepsilon + O(n\varepsilon^2)\right] \frac{\sum_{i=1}^{n} |x_i|}{\left|\sum_{i=1}^{n} x_i\right|}.$$

In the expression for the relative error bound, the fraction $\Sigma|x_i|/|\Sigma x_i|$ is the condition number of the summation problem. Essentially, the condition number represents the *intrinsic* sensitivity of the summation problem to errors, regardless of how it is computed.[6] The relative error bound of *every* (backwards stable) summation method by a fixed algorithm in fixed precision (i.e. not those that use arbitrary precision arithmetic, nor algorithms whose memory and time requirements change based on the data), is proportional to this condition number.[2] An *ill-conditioned* summation problem is one in which this ratio is large, and in this case even compensated summation can have a large relative error. For example, if the summands $x_i$ are uncorrelated random numbers with zero mean, the sum is a random walk and the condition number will grow proportional to $\sqrt{n}$. On the other hand, for random inputs with nonzero mean the condition number asymptotes to a finite constant as $n \rightarrow \infty$. If the inputs are all non-negative, then the condition number is 1.

Given a condition number, the relative error of compensated summation is effectively independent of *n*. In principle, there is the O($n\varepsilon^2$) that grows linearly with $n$, but in practice this term is effectively zero: since the final result is rounded to a precision ε, the $n\varepsilon^2$ term rounds to zero unless $n$ is roughly 1/ε or larger.[2] In double precision, this corresponds to an $n$ of roughly $10^{16}$, much larger than most sums. So, for a fixed condition number, the errors of compensated summation are effectively $O(\varepsilon)$, independent of $n$.

In comparison, the relative error bound for naive summation (simply adding the numbers in sequence, rounding at each step) grows as $O(\varepsilon n)$ multiplied by the condition number.[2] This worst-case error is rarely observed in practice, however, because it only occurs if the rounding errors are all in the same direction. In practice, it is much more likely that the rounding errors have a random sign, with zero mean, so that they form a random walk; in this case, naive summation has a root mean square relative error that grows as $O(\varepsilon\sqrt{n})$ multiplied by the condition number.[7] This is still much worse than compensated summation, however. Note, however, that if the sum can be performed in twice the precision, then ε is replaced by $\varepsilon^2$ and naive summation has a worst-case error comparable to the O($n\varepsilon^2$) term in compensated summation at the original precision.

By the same token, the $\Sigma|x_i|$ that appears in $E_n$ above is a worst-case bound that occurs only if all the rounding errors have the same sign (and are of maximum possible magnitude).[2] In practice, it is more likely that the errors have random sign, in which case terms in $\Sigma|x_i|$ are replaced by a random walk—in this case, even for random inputs with zero mean, the error $E_n$ grows only as $O(\varepsilon\sqrt{n})$ (ignoring the $n\varepsilon^2$ term), the same rate the sum $S_n$ grows, canceling the $\sqrt{n}$ factors when the relative error is computed. So, even for asymptotically ill-conditioned sums, the relative error for compensated summation can often be much smaller than a worst-case analysis might suggest.

## Alternatives [edit]

Although Kahan's algorithm achieves $O(1)$ error growth for summing $n$ numbers, only slightly worse $O(\log n)$ growth can be achieved by pairwise summation: one recursively divides the set of numbers into two halves, sums each half, and then adds the two sums.[2] This has the advantage of requiring the same number of arithmetic operations as the naive summation (unlike Kahan's algorithm, which requires four times the arithmetic and has a latency of four times a simple summation) and can be calculated in parallel. The base case of the recursion could in principle be the sum of only one (or zero) numbers, but to amortize the overhead of recursion one would normally use a larger base case. The equivalent of pairwise summation is used in many fast Fourier transform (FFT) algorithms, and is responsible for the logarithmic growth of roundoff errors in those FFTs.[8] In practice, with roundoff errors of random signs, the root mean square errors of pairwise summation actually grow as $O(\sqrt{\log n})$.[7]

Another alternative is to use arbitrary precision arithmetic, which in principle need no rounding at all with a cost of much greater computational effort. A way of performing exactly rounded sums using arbitrary precision is to extended adaptively using multiple floating-point components. This will minimize computational cost in common cases where high precision is not needed.[9][10] Another method that uses only integer arithmetic, but a large accumulator was described by Kirchner and Kulisch;[11] a hardware implementation was described by Müller, Rüb and Rülling.[12]

## Computer languages [edit]

In principle, a sufficiently aggressive optimizing compiler could destroy the effectiveness of Kahan summation: for example, if the compiler simplified expressions according to the associativity rules of real arithmetic, it might "simplify" the second step in the sequence `t = sum + y; c = (t - sum) - y;` to `((sum + y) - sum) - y;` then to `c = 0;`, eliminating the error compensation.[13] In practice, many compilers do not use associativity rules (which are only approximate in floating-point arithmetic) in simplifications unless explicitly directed to do so by compiler options enabling "unsafe" optimizations,[14][15][16][17] although the Intel C++ Compiler is one example that allows associativity-based transformations by default.[18] The original K&R C version of the C programming language allowed the compiler to re-order floating-point expressions according to real-arithmetic associativity rules, but the subsequent ANSI C standard prohibited re-ordering in order to make C better suited for numerical applications (and more similar to Fortran, which also prohibits re-ordering),[19] although in practice compiler options can re-enable re-ordering as mentioned above.

In general, built-in "sum" functions in computer languages typically provide no guarantees that a particular summation algorithm will be employed, much less Kahan summation.[*citation needed*] The BLAS standard for linear algebra subroutines explicitly avoids mandating any particular computational order of operations for performance reasons,[20] and BLAS implementations typically do not use Kahan summation.

The standard library of the Python computer language specifies an fsum function for exactly rounded summation, using the Shewchuk algorithm [10] to track multiple partial sums.

## See also [edit]

- Algorithms for calculating variance, which includes stable summation

## References [edit]

1. ^ Strictly, there exist other variants of compensated summation as well: see Higham, Nicholas (2002). *Accuracy and Stability of Numerical Algorithms (2 ed)*. SIAM. pp. 110–123.
2. ^ *a b c d e f g h* Higham, Nicholas J. (1993), "The accuracy of floating point summation", *SIAM Journal on Scientific Computing* **14** (4): 783–799, doi:10.1137/0914050
3. ^ Kahan, William (January 1965), "Further remarks on reducing truncation errors", *Communications of the ACM* **8** (1): 40, doi:10.1145/363707.363723
4. ^ Jack E. Bresenham, "Algorithm for computer control of a digital plotter", *IBM Systems Journal*, Vol. 4, No.1, January 1965, pp. 25–30
5. ^ H. Inose, Y. Yasuda, J. Murakami, "A Telemetering System by Code Manipulation – ΔΣ Modulation," IRE Trans on Space Electronics and Telemetry, Sep. 1962, pp. 204–209.
6. ^ L. N. Trefethen and D. Bau, *Numerical Linear Algebra* (SIAM: Philadelphia, 1997).
7. ^ *a b* Manfred Tasche and Hansmartin Zeuner *Handbook of Analytic-Computational Methods in Applied Mathematics* Boca Raton, FL: CRC Press, 2000).
8. ^ S. G. Johnson and M. Frigo, "Implementing FFTs in practice, in *Fast Fourier Transforms*, edited by C. Sidney Burrus(2008).
9. ^ Jonathan R. Shewchuk, Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, *Discrete and Computational Geometry*, vol. 18, pp. 305–363 (October 1997).
10. ^ *a b* Raymond Hettinger, Recipe 393090: Binary floating point summation accurate to full precision, Python implementation of algorithm from Shewchuk (1997) paper (28 March 2005).
11. ^ R. Kirchner, U. W. Kulisch, *Accurate arithmetic for vector processors*, Journal of Parallel and Distributed Computing 5 (1988) 250-270
12. ^ M. Muller, C. Rub, W. Rulling [1], *Exact accumulation of floating-point numbers*, Proceedings 10th IEEE Symposium on Computer Arithmetic (Jun 1991), doi 10.1109/ARITH.1991.145535
13. ^ Goldberg, David (March 1991), "What every computer scientist should know about floating-point arithmetic" (PDF), *ACM Computing Surveys* **23** (1): 5–48, doi:10.1145/103162.103163
14. ^ GNU Compiler Collection manual, version 4.4.3: 3.10 Options That Control Optimization, *-fassociative-math* (Jan. 21, 2010).
15. ^ *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*, section 5.9.7 Arithmetic Reordering Optimizations (retrieved March 2010).
16. ^ Börje Lindh, Application Performance Optimization, *Sun BluePrints OnLine* (March 2002).
17. ^ Eric Fleegal, "Microsoft Visual C++ Floating-Point Optimization", *Microsoft Visual Studio Technical Articles* (June 2004).
18. ^ Martyn J. Corden, "Consistency of floating-point results using the Intel compiler," *Intel technical report* (Sep. 18, 2009).
19. ^ Tom Macdonald, "C for Numerical Computing", *Journal of Supercomputing* vol. 5, pp. 31–48 (1991).
20. ^ BLAS Technical Forum, section 2.7 (August 21, 2001), Archived on Wayback Machine.

## External links [edit]

- Floating-point Summation, Dr. Dobb's Journal September, 1996

Categories: Computer arithmetic | Numerical analysis