Kahan summation algorithm: Wikis

Kahan summation algorithm Quiz

Note: Many of our articles have **direct quotes from sources you can cite, within the Wikipedia article**! This article doesn't yet, but we're working on it! See more info or our list of citable articles.

Categories: Articles with example pseudocode > Numerical analysis > Computer arithmetic

Top rankings for Kahan summation algorithm

184th Top numerical analysis topics

Encyclopedia

From Wikipedia, the free encyclopedia

In numerical analysis, the **Kahan summation algorithm** (also known as **compensated summation**) 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).[1] 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.[1]

The algorithm is attributed to William Kahan[2]. Similar, earlier techniques are, for example, Bresenham's line algorithm, keeping tab of the accumulated error in integer operations (although first documented around the same time[3]) and the Delta-sigma modulation[4] (integrating, not just summing the error).

Contents

1 The algorithm
    1.1 Example Working
2 Accuracy
3 Alternatives
4 Computer languages
5 External links
6 References

The algorithm

In pseudocode, the algorithm is:

```
function kahanSum(input)
 var sum = input[1]
 var c = 0.0          //A running compensation for lost low-order bits.
 for i = 2 to input.length
  y = input[i] - c    //So far, so good: c is zero.
  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
  sum = t             //Algebraically, c should always be zero. Beware eagerly optimising compilers!
 next i               //Next time around, the lost low part will be added to y in a fresh attempt.
return sum
```

Example Working

Suppose *sum* has attained the value 100000 and the next two values of *input(i)* are 3.14159 and 2.71828; all are six-digit decimal floating point numbers. With a plain summation, each incoming value would be aligned with

More info on Kahan summation algorithm

- **Wikis**
  - Encyclopedia
    - The algorithm
      - **Example Working**
    - Accuracy
    - Alternatives
    - Computer languages
    - External links
    - References
  - Related links
- Quiz
  - Quiz

sum and many low order digits lost (by truncation or rounding) so that both values would be 3. However, with compensated summation, not so. Suppose that *c* has the value zero.

```
  y = 3.14159 - 0                 y = input[i] - c
  t = 100000 + 3.14159
    = 100003                      Many digits have been lost!
  c = (100003 - 100000) - 3.14159 This must be evaluated as written!
    = 3.00000 - 3.14159           The assimilated part of y recovered, vs. the original full y.
    = -0.141590                   The trailing zero because this is six-digit arithmetic.
sum = 100003                      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* is not zero...

```
  y = 2.71828 - -0.141590         The shortfall from the previous stage has another chance.
    = 2.85987                     It is of a size similar to y: most digits meet.
  t = 100003 + 2.85987            But few meet the digits of sum.
    = 100006                      Rounding is good, but even with truncation,
  c = (100006 - 100003) - 2.85987 This extracts whatever went in.
    = 3.00000 - 2.85987           In this case, too much.
    = .140130                     But no matter, the excess will be subtracted off next time.
sum = 100006                      (instead of 100005.85987)
```

Suppose that instead of 2.71828 the next value was 2.3, and *c* is -0.141590 as before.

```
  y = 2.30000 - -0.141590
    = 2.44159
  t = 100003 + 2.44159            This time not rounding up.
    = 100005
  c = (100005 - 100003) - 2.44159
    = 2.00000 - 2.44159
    = -.441590
sum = 100005                      (instead of 100005.44159)
```

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. This is not practical in general, however, becaues if *input* is already in double precision, few systems supply quadruple precision (and if this is available, *input* could be quadruple precision).

Accuracy

A more careful analysis of the errors in compensated summation is needed to fully appreciate its accuracy characteristics. On the one hand, it is far more accurate than naive summation; on the other hand, it can still give large relative errors for ill-conditioned sums.

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

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

With compensated summation, one instead obtains $S_n + E_n$, where the error E_n is bounded above by:[1]

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

where ε is the machine precision of the arithmetic being employed (e.g. $\varepsilon \approx 10^{-16}$ for 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.[5] 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.[1] 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 $n \to \infty$. 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 $1/\varepsilon$ or larger.[1] 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.[1] 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.[6] 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 ε^2 and naive summation is 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).[1] 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

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.[1] This has the advantage of requiring the same number of arithmetic operations as the naive summation (unlike Kahan's algorithm, which requires more arithmetic). 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 cascade summation is used in many fast Fourier transform (FFT) algorithms, and is responsible for the logarithmic growth of roundoff errors in those FFTs.[7] In practice, with roundoff errors of random signs, the root mean square errors of cascade summation actually grow as $O(\sqrt{\log n})$.[6]

Another alternative is to use arbitrary precision arithmetic, which in principle need do no rounding at all at the cost of much greater computational cost. A way of performing exactly rounded sums using arbitrary precision that is extended adaptively using multiple floating-point components, to minimize computational cost in common cases where high precision is not needed, was described by Shewchuk.[8][9]

Computer languages

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 expression *c = (t - sum) - y = ((sum + y) - sum) - y* to *c = 0*, eliminating the error compensation.[10] 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,[11][12][13][14] although the Intel C++ Compiler is one example that allows associativity-based transformations by default.[15] 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),[16] 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,[17] 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 to track multiple partial sums.

External links

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

References

1. ^ **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
2. ^ Kahan, William (January 1965), "Further remarks on reducing truncation errors", *Communications of the ACM* **8** (1): 40, doi:10.1145/363707.363723
3. ^ Jack E. Bresenham, "Algorithm for computer control of a digital plotter", *IBM Systems Journal*, Vol. 4, No.1, January 1965, pp. 25–30
4. ^ 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.
5. ^ L. N. Trefethen and D. Bau, *Numerical Linear Algebra* (SIAM: Philadelphia, 1997).
6. ^ **a b** Manfred Tasche and Hansmartin Zeuner *Handbook of Analytic-Computational Methods in Applied Mathematics* Boca Raton, FL: CRC Press, 2000).
7. ^ S. G. Johnson and M. Frigo, "Implementing FFTs in practice, in *Fast Fourier Transforms*, edited by C. Sidney Burrus (2008).
8. ^ Jonathan R. Shewchuk, Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, *Discrete and Computational Geometry*, vol. 18, pp. 305-363 (October 1997).
9. ^ Raymond Hettinger, Recipe 393090: Binary floating point summation accurate to full precision, Python implementation of algorithm from Shewchuk (1997) paper (28 March 2005).
10. ^ 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, http://www.validlab.com/goldberg/paper.pdf
11. ^ GNU Compiler Collection manual, version 4.4.3: 3.10 Options That Control Optimization, *-fassociative-math* (Jan. 21, 2010).
12. ^ *Compaq Fortran User Manual for Tru64 UNIX and Linux Alpha Systems*, section 5.9.7 Arithmetic Reordering Optimizations (retrieved March 2010).
13. ^ Börje Lindh, Application Performance Optimization, *Sun BluePrints OnLine* (March 2002).
14. ^ Eric Fleegal, "Microsoft Visual C++ Floating-Point Optimization", *Microsoft Visual Studio Technical Articles* (June 2004).
15. ^ Martyn J. Corden, "Consistency of floating-point results using the Intel compiler," *Intel technical report* (Sep. 18, 2009).
16. ^ Tom Macdonald, "C for Numerical Computing", *Journal of Supercomputing* vol. 5, pp. 31–48 (1991).
17. ^ BLAS Technical Forum, section 2.7 (August 21, 2001).

Categories: Computer arithmetic | Numerical analysis

Related links

Up to date as of November 16, 2009

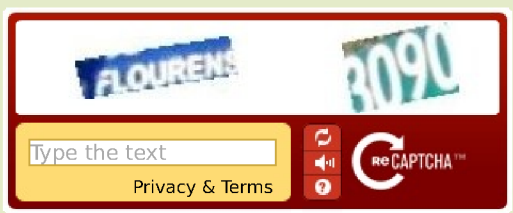
- What Every Computer Scientist Should Know About Floating-Point Arithmetic

Got something to say? Make a comment.

Your name

Your email address

Message



Submit Comment »