Analysis and Application of Simply Compensated Summation

Prof. W. Kahan
Mathematics Dept., and Computer Science Dept.
University of California at Berkeley

Abstract
Summing a slowly convergent series $s = \Sigma_{k=0}^{\infty} x_k$, a two-line program

```
s := x_0 ;
for k = 1 to n do s := s + x_k ;
```

incurs n rounding errors due solely to additions. These errors accumulate to the point where each x_k is obscured by as many as $n+1-k$ rounding errors. The loss of accuracy can be severe if n is huge and almost every x_k is smaller in magnitude than the sum s to which it is added. *Counsel of Perfection* would reverse the order of summation to attenuate roundoff's effect, but this may be impractical if n is not known in advance and if x_k must be computed by recurrence for $k = 1, 2, \ldots, n$ in turn. An old trick we call *Compensated Summation* circumvents that loss of accuracy in sums of series, in trajectory calculations, and in numerical quadratures, and at so little extra cost as merits its availability to every general computer program serving those ends.

What Roundoff Does

Rounding off $s + x_k$ replaces it by $(s + x_k)(1 + b_k\varepsilon)$ where no more is known about b_k than that $|b_k| \leq 1$, and ε is a very tiny quantity intrinsic to a computer's floating-point arithmetic. Most computers, conforming to IEEE Standard 754, round to 24 or 53 sig. bits so their $\varepsilon = 2^{-24}$ or 2^{-53} resp. Hewlett-Packard 71B calculators round to 12 sig. dec. in conformity with IEEE 854 so their $\varepsilon = 5_{10}{-}11$; IBM /370s mostly chop to $\varepsilon = 16^{-5}$ for single-precision, $\varepsilon = 16^{-13}$ for double, $\varepsilon = 16^{-27}$ for extended. (CRAYs subtract in so aberrant a way that neither the foregoing nor what follows applies to them properly.)

Roundoff transforms the two-line program above into this:
$s_0 = x_0$, and $s_k = (s_{k-1} + x_k)(1 + b_k\varepsilon)$ for $k = 1, 2, \ldots$.
Therefore what it actually computes is

$$s_n = (1 + f_0\varepsilon)^n x_0 + \Sigma_{k=1}^{n} (1 + f_k\varepsilon)^{n+1-k} x_k$$

wherein little more can be said about f_k than that $|f_k| \leq 1$.
This sum can seem arbitrarily wrong. For example consider the sum

$$t := (10^{37} \tan(100\pi/180) + 1) - 10^{37} \tan(10^{20}\,\pi/180) ,$$

whose three terms ought to add up to just 1 despite that almost every computer computes $t = 0$ or else something far bigger than the ideal value 1 . The trouble is that the first sum rounds off as if the number " 1 " were " 0 ", after which the difference should cancel to 0 but for roundoff during calculations of the quantities $10^{37} \tan(\ldots)$, which should both be $-5.67128\ldots_{10}37$.

Backward Error Analysis

If t is wrong, yet it is no worse than if each term in its sum had been calculated only slightly less accurately than the best that can be expected. And in general, provided n is not huge, s_n is about as accurate as if the terms x_k had been computed as

$(1 + f_k\varepsilon)^{n+1-k} x_k$ instead, which is only moderately worse than the best that can be expected. This conclusion is typical of *Backward Error Analyses*; the algorithm so analysed works as well as can reasonably be expected unless the data is either exact or too near some singularity. The singularity here is at $n = \infty$.

When n is too huge, s_n can be far worse than is explainable by moderately perturbed end-figures among the summands x_k. For an extreme example try $n = 1/(4\varepsilon)$, $x_0 = 4$, $x_k = \varepsilon$ if $k > 0$; the two-line program gets $s_n = 4$, not the correct sum 4.25.

The Obvious Remedy: Double Precision

If hardware provides it, higher precision is the most economical way to cope with enormous numbers n of terms x_k. It adds at most two declarations to the two-line program:

```
Single Precision  x... ;
Double Precision  s ;
s := x0 ;
for  k = 1 to n  do  s := s + xk ;
```

It replaces ε in the error analyses above by ε^2 or something smaller, rendering the error of summation negligible compared with the uncertainty inherited with x_k. When supported directly by a computer's hardware, Double Precision operations take about as much time, surely less than twice as much, as Single.

But if Doubled Precision operations must be synthesized out of Single in software, Double may well take 5 to 10 times longer than Single, more on a CRAY. Thus, when x_k is already being computed at the highest precision supported directly by hardware or by the programming language in use, relief from errors in sums of vast numbers of terms must be sought elsewhere than from higher precision that runs too slowly or else is practically unavailable. *That unavailability is the only justification for what follows.*

The Unobvious Remedy: Compensated Summation

To compensate for the error committed when $s + x_k$ is rounded off let us estimate that error and record it in an auxiliary variable c which will be used the next time around the loop to cancel most of the error off. Here is a program that does that:

```
s := x0 ;  c := 0 ;
for k = 1 to ...  do {
    y := c + xk ;   ...  Computing  xk  costs most time.
    t := y + s ;    ...  Here is the worst rounding error,
    c := (s - t) + y ;  ...  and here is its estimate.
    s := t } .
```

This is *Compensated Summation*. When and why does it work? First let us take account of roundoff's contribution to the program:

$$s_0 = x_0, \quad c_0 = 0, \quad \text{and for } k = 1, 2, \ldots, n \text{ in turn}$$
$$y_k = (c_{k-1} + x_k)(1 + a_k\varepsilon),$$
$$s_k = (y_k + s_{k-1})(1 + b_k\varepsilon),$$
$$c_k = ((s_{k-1} - s_k)(1 + d_k\varepsilon) + y_k)(1 + e_k\varepsilon).$$

About the roundoff terms $a_k\varepsilon$, $b_k\varepsilon$, $d_k\varepsilon$, $e_k\varepsilon$ we know nothing but that their magnitudes cannot exceed ε. Our task is to determine bounds for the first few coefficients of an expansion in powers of ε for the computed sums of perturbed terms:

$$s_n = \sum_{k=1}^{n} (1 + (f_{nk}+g_{nk})\varepsilon)x_k \ , \quad c_n = -\sum_{k=1}^{n} g_{nk}\varepsilon x_k \ .$$

The intent of this expansion is to show how little each x_k would have to be perturbed to render the computed sum s_n exactly equal to the sum of the perturbed terms; this is a kind of *Backward Error-Analysis*. We shall see that the computed s_n is no worse than if only the last computed sig. dec. of every x_k had first been altered, and this remains so for all $n << 1/\varepsilon$, implying that the accuracy of s_n is limited only by the accuracies of the terms x_k for almost all practical purposes.

Outline of a Proof that It Works (Skip this on first reading.)
To avoid the blizzard of subscripts that afflicts D. E. Knuth's treatment in problem 19, p. 229 and 572-3, of his *Seminumerical Algorithms* (2d ed., 1981) Addison-Wesley, MA, we adopt a simpler notation that conveys only the essential form of our results:

$$\begin{aligned} \text{Given} \quad & S = (1 + (F_1+G_1)\varepsilon + (F_2(\varepsilon)+G_2(\varepsilon))\varepsilon^2)X \quad \text{and} \\ & C = -(G_1\varepsilon + G_2(\varepsilon)\varepsilon^2)X \ , \\ \text{let} \quad & y := (C + x)(1 + a\varepsilon) \ ; \\ & s := (y + S)(1 + b\varepsilon) \ ; \\ & c := ((S - s)(1 + d\varepsilon) + y)(1 + e\varepsilon) \ . \end{aligned}$$

Here S and C stand for s_{k-1} and c_{k-1}, X stands for any x_j with $j < k$, and x, s and c stand for x_k, s_k and c_k. $F_2(\varepsilon)$ and $G_2(\varepsilon)$ are polynomials in ε. About a, b, d, e we know nothing except that their magnitudes cannot exceed 1.

Substituting S and C into the last three equations invites too many errors in algebra to be done by hand reliably, so it has been submitted to computerized symbolic algebra systems ranging from SCRATCHPAD in 1972 to MuMath in 1984. The results are

$$\begin{aligned} s &= (1 + (f_1+g_1)\varepsilon + (f_2+g_2)\varepsilon^2)X \ + \ (1 + (h_1+q_1)\varepsilon + (h_2+q_2)\varepsilon^2)x \ , \\ c &= -(g_1 + g_2\varepsilon)\varepsilon X \ - \ (q_1 + q_2\varepsilon)\varepsilon x \ , \end{aligned}$$

where

$$\begin{aligned} f_1 &= F_1 \ , \quad g_1 = b \ , \quad h_1 = a-d \ , \quad q_1 = b+d \ , \\ f_2 &= f_2(\varepsilon) = F_2(\varepsilon) - (a-d)G_1 - (d+e)b + O(\varepsilon) \ , \\ g_2 &= g_2(\varepsilon) = F_1 b - G_1 d + (d+e)b + O(\varepsilon) \ , \\ h_2 &= h_2(\varepsilon) = -(ad+bd+be+de) + O(\varepsilon) \ , \\ q_2 &= q_2(\varepsilon) = ab+ad+bd+be+de + O(\varepsilon) \quad \text{as} \quad \varepsilon \to 0 \ . \end{aligned}$$

We interpret these equations to mean that each pass around the for-loop initializes the coefficients in s+c and c of x_k to $(1 + h_1\varepsilon + h_2\varepsilon^2)$ and $-(q_1 + q_2\varepsilon)\varepsilon$ respectively, and updates the coefficient of each previous x_j from $(1 + F_1\varepsilon + F_2\varepsilon^2)$ and $-(G_1 + G_2\varepsilon)\varepsilon$ respectively to $(1 + f_1\varepsilon + f_2\varepsilon^2)$ and $-(g_1 + g_2\varepsilon)\varepsilon$. Evidently $f_1 = F_1$ stays unchanged from its initial assignment h_1, so $|f_1| \le 2$; but G_1 starts at q_1, so $|G_1| \le 2$, and then is replaced by g_1 with $|g_1| \le 1$. If terms $O(\varepsilon)$ are ignored, G_2 starts at q_2 with $|G_2| \le 5$, and then goes to g_2 with $|g_2| \le |F_1| + |G_1| + 2 \le 6$ at first, ≤ 5 afterwards.

The interesting case is f_2. Still ignoring terms $O(\varepsilon)$, we see that F_2 is initialized to h_2, so $|F_2| \le 4$; then it goes to f_2 with $|f_2 - F_2| \le 2|G_1| + 2 \le 6$ at first, ≤ 4 afterwards. Therefore $|f_2| \le 4(k-j) + 6 + O(\varepsilon)$. Now our first conclusion may be drawn, valid in the usual case when the correction term c is so small compared with s that $s+c$ rounds to s. In that case our attempt to compute $\Sigma_{k=1}^{n} x_k$ produces instead a result s_n within one rounding error of a sum $\Sigma_{k=1}^{n} X_k$ whose perturbed terms X_k all satisfy $|X_k - x_k| \le (2 + (4(n-k)+6)\varepsilon + O(n\varepsilon)^2)\,\varepsilon\,|x_k|$.

A similar conclusion for s_n alone, regardless of c_n, entails estimation of f_1+g_1 and f_2+g_2. The result is that, rather than compute $\Sigma_{k=1}^{n} x_k$, compensated summation delivers a result $s_n = \Sigma_{k=1}^{n} X_k$ in which

$$|X_k - x_k| \le (3 + (4(n-k)+5)\varepsilon + O(n\varepsilon)^2)\,\varepsilon\,|x_k| .$$

Interpretation and Improvement of these Error Bounds

The last inequality says that, provided $n \ll 1/\varepsilon$, compensated summation's result is scarcely worse than if each summand x_k had first been obscured to the extent of roughly 3 *ulps* (*U*nits in its *L*ast *P*lace computed.) This is far less than the n ulps that afflict the early summands in the two-line summation program. But these are error *bounds*; the actual errors are certainly smaller and probably far smaller. Therefore we cannot expect the ratio

(two-line program's error)/(compensated summation's error)

to get so big as $n/3$ very often. What can we expect?

Conventional wisdom expects rounding errors to cancel each other quite often rather than always conspire to do their worst. On a machine that rounds correctly, as do those that conform to IEEE Standards 754 and 854, we might therefore expect that errors in the two-line program would more often accumulate to something like $\sqrt{(n/12)}$ ulps rather than n. The truth lies somewhere between.

When Σx_k is a very slowly convergent series, the terms x_k can change so slowly that rounding errors will mostly change slowly too, and hence reinforce instead of cancel. The extreme example $4 + \varepsilon + \varepsilon + \ldots + \varepsilon$ illustrates this possibility. That is why the two-line program's errors frequently amount to far more than $\sqrt{(n/12)}$ and yet far less than n ulps. But computers that chop instead of round, as do CRAYs and IBM /370s, incur errors not far from $n/2$ ulps almost all the time.

The error bounds for compensated summation are pessimistic too for a non-probabilistic reason. The following *Theorem* explains it for every commercially significant North American computer today except a CRAY :

> If S and T are two floating-point numbers stored by the computer in the same format (both single-, or both double-precision), and if $1/2 \le S/T \le 2$, then $S - T$ is representable exactly in the same format, and will be computed exactly by any computer that conforms to IEEE Standards 754/854 for Floating-Point Arithmetic.

(Nowadays almost all computers conform. Exceptions are mainly older designs like IBM /370s and DEC VAXs, which violate the theorem only when $S - T$ is so tiny that it underflows to 0.0 although $S \neq T$. CRAY arithmetics are bizarre; they violate the theorem also if $|S|$ and $|T|$ straddle a power of 2 and whichever is the smaller has 1 in its least significant bit.)

An early proof of the theorem is on p. 138 of P. H. Sterbenz's book " Floating- Point Computation " (1974) Prentice-Hall, N. J.

When s_k/s_{k-1} lies between 1/2 and 2, and if arithmetic obeys the theorem, then both rounding errors $d_k = e_k = 0$; and usually $a_k = 0$ too because only the leading digits of c_k are nonzero. Then compensated summation's result s_n is usually about as accurate as if at most a few terms x_k were perturbed by one ulp, and then all terms were added exactly, and then the sum rounded once. Accuracy like that is hard to surpass without using higher-precision arithmetic.

When computer arithmetic does not honor the theorem, compensated summation works as advertised provided every sum and difference is rounded to within an ulp or so. A few unconventional arithmetics exist that comply with this proviso; some are simulations of doubled-precision in software, and some encode numbers internally by sign-bits and logarithms of their magnitudes. But almost all unconventional arithmetics, like CRAY's, subtract magnitudes as if their last digits had first been perturbed, so a result after massive cancellation may differ significantly from the unperturbed difference of nearly equal operands; this invalidates compensated summation. Despite its invalidity, it almost always improves a sum's accuracy anyway, so one might believe it always worked but for counter-examples like the one I exhibited in " A Survey of Error-Analysis " in Proc. 1971 IFIP Congress, p. 1239 *et seq.* A cure for the invalidity on CRAYs is presented there too.

Example: A Slowly Convergent Series

When $x_k = \exp(-0.625 \ln(k)^{3/2})$ the series $\sum_1^\infty x_k$ converges very slowly, but the quickest way to compute its first seven sig. dec. is still the obvious way. The two-line program was run in single-precision binary floating-point arithmetic conforming to IEEE 754 on an IBM PC, and it produced the result $s = 5.145586$ after 4502 terms x_k were added. Computation stopped there because more terms could make no difference; each subsequent x_k would have been too small to alter the sum s after it was rounded off.

Compensated summation of 4502 terms produced 5.145461, which is correct so far, but the compensated sum s went further.

When should computation stop? That depends upon our estimate for the remainder $\sum_{N+1}^\infty x_k$ that is left after the first N terms have been added. Such estimates come from approximations of the given series by simpler ones that behave similarly enough as $N \to \infty$. For example, if the given series were enough like a geometric series we could use $x_N^2/(x_{N-1}-x_N)$ to estimate the remainder.

Our series has $x_k = 1/k^p$ for $p = (5/8)\sqrt{(\ln(k))}$, which grows so slowly that replacing it by an apt constant for all $k > N >> 1$ increases the sum of the series only moderately. From an integral $\int dx/x^p$ that slightly overestimates $\sum 1/k^p$ we find that

$$x_N^2/((x_{N-1} - x_N) (1 - 1/p))$$

moderately overestimates the remainder if $p = (5/8) (\ln(N))$ and N is big enough.

N had reached 219901 when that over-estimate first fell below a rounding error in compensated $s_N = 5.146056$, which matches s_∞ in all figures displayed. A smarter stopping criterion might have saved a significant fraction of the time spent by stopping sooner, but only compensated summation could save the last four figures.

Another computation of the series' sum on an IBM PC used IEEE 754 double-precision (53 sig. bits) but not so straightforward a program as before; that would have taken more than 250,000,000 terms. Instead, the first two terms of something like an Euler-Maclaurin summation formula supplied rigorous bounds for the remainder, after which 4,000,000 terms sufficed to calculate 5.146056069915... as the sum of the infinite series, provably correct in all digits displayed. Uncompensated summation would spoil the last six digits computed (the last three displayed).

Numerical Quadrature

Numerical quadrature approximates an integral $\int f(x)\,dx$ by a sum $\sum w_k\, f(x_k)$ of artfully selected samples $f(x_k)$ of the integrand, with artfully chosen weights w_k . Compensated summation permits that integral to be approximated as accurately as the integrand's accuracy allows, provided enough samples are drawn. But without compensated summation, attempts to improve accuracy by sampling more densely too often fail, though samples are numerous enough, because too much roundoff accumulates during summation.

Trajectory Calculations

A trajectory is the graph of the solution $y(\tau)$ of an ordinary differential equation with initial conditions:

$$y(0) = y_o \quad \text{and} \quad dy/d\tau = f(y,\tau) \quad \text{for} \quad \tau \geq 0 .$$

Here f is a given vector-valued function of the vector y and the scalar τ . The differential equation is equivalent to an integral equation $y(\tau) = y_o + \int_0^\tau f(y(\theta),\theta)\,d\theta$. For small time-increments $\Delta\tau$ this implies

$$y(\tau+\Delta\tau) = y(\tau) + \int_\tau^{\tau+\Delta\tau} f(y(\theta),\theta)\,d\theta ,$$

which resembles the formulas intended to compute a numerical approximation $Y(\tau)$ to $y(\tau)$:

$$Y(\tau+\Delta\tau) := Y(\tau) + F(Y(\tau),\tau;\ \Delta\tau,\ \ldots)\,\Delta\tau .$$

Here F is devised in a way intended to approximate the average of f along an unknown trajectory through $Y(\tau)$. The quality of that approximation, and consequently the quality of Y , depends upon details of the formula F ; all we need know now about those details is that the quality is expected to improve as $\Delta\tau \rightarrow 0$.

However closely F may approximate the desired average of f , roundoff can degrade it. Usually most of the damage occurs when the " + " operation in the last formula above is carried out;

the rounding error ηY in that operation amounts typically to at worst about an ulp (a unit in the last place) carried in Y, and results in the computation actually of

$$\begin{aligned} Y(\tau+\Delta\tau) &= Y(\tau) + F(Y(\tau),\tau;\ \Delta\tau,\ \ldots)\,\Delta\tau + \eta Y \\ &= Y(\tau) + (\ F(Y(\tau),\tau;\ \Delta\tau,\ \ldots) + \eta Y/\Delta\tau\ )\,\Delta\tau\ . \end{aligned}$$

The last expression shows how $\eta Y/\Delta\tau$ contaminates the average F as if f had been perturbed by a noise term $\eta Y/\Delta\tau$ that grows worse as $\Delta\tau \rightarrow 0$, thus jeopardizing any expected improvement in accuracy.

Fortunately, compensated summation suppresses ηY almost as if addition were carried out to about twice the precision actually carried. An auxiliary function $C(\tau)$ propagates a correction to compensate for the roundoff ηY. Here is how it works:

$Y(0) := y_0$; $C(0) := 0$;
...
$H := F(Y(\tau),\tau;\ \Delta\tau,\ \ldots)\,\Delta\tau + C(\tau)$; ... near enough.
$Y(\tau+\Delta\tau) := Y(\tau) + H$; ... rounded off to $Y(\tau) + H + \eta Y$.
$C(\tau+\Delta\tau) := (\ Y(\tau) - Y(\tau+\Delta\tau)\ ) + H$; ... $\doteq -\eta Y$.
... DON'T REMOVE PARENTHESES!

Provided the last subtraction and addition are performed exactly, as almost always happens automatically, $C(\tau+\Delta\tau)$ will carry the previous addition's rounding error forward to the next occasion when Y is incremented, thereby compensating for the error. Consequently the accuracy of Y will depend almost entirely upon how well F approximates the desired average of f even if $\Delta\tau$ is so tiny that many millions of additions occur. Let's see:

Example: Going Around in a Circle

The differential equations

$$dx/d\tau = -y\ ,\quad dy/d\tau = x,\quad x(0) = 1\ ,\quad y(0) = 0\ ,$$

have solutions $x(\tau) = \cos\tau$ and $y(\tau) = \sin\tau$. Given $T > 0$ and $\Delta\tau = T/N$ for some huge integer N, we wish to compute $X \doteq x(T)$ and $Y \doteq x(T)$ from this algorithm:

```
X := 1 ;  Y := 0 ;
For  n = 1  to  N  do
  {  X := X - Y*Δτ ;
     Y := Y + X*Δτ  } ;
X := X - Y*Δτ/2 .
```

In the absence of roundoff this algorithm would yield

$$\begin{aligned} X &= \cos(2N \arcsin(\Delta\tau/2)) = x(T) - T\,y(T)\,(\Delta\tau)^2/24 + \ldots\ , \\ Y &= \sin(2N \arcsin(\Delta\tau/2))/\cos(\arcsin(\Delta\tau/2)) \\ &= y(T) + (T\,x(T) + 3y(T))\,(\Delta\tau)^2/24 + \ldots\ . \end{aligned}$$

These formulas exhibit the error due to using a stepsize $\Delta\tau \neq 0$, but not the error due to roundoff. Compensated addition gets rid of the rounding errors in addition and subtraction, which do much more damage than the rounding errors in multiplication, by saving them in two auxiliary variables Cx and Cy thus:

```
X := 1 ;  Cx := 0 ;   Y := 0 ;  Cy := 0 ;
For  n = 1  to  N  do
  { H := Cx-Y*ΔT :  S := X+H ;  Cx := (X-S)+H ;  X := S ;
    H := Cy+X*ΔT ;  S := H+Y ;  CY := (Y-S)+H :  Y := S } ;
X := X + (Cx - Y*ΔT/2) .
```

The foregoing is the simplest form of compensated summation. A more elaborate form, tried out only to confirm that it was not worth trying because it got almost the same results, is this:

```
X := 1 ;  Cx := 0 ;   Y := 0 ;  Cy := 0 ;
For  n = 1  to  N  do
  {  H := Y*ΔT ;  S := (Cx-H)+X ;  Cx := ((X-S)-H)+Cx ;  X := S ;
     H := X*ΔT ;  S := (Cy+H)+Y ;  Cy := ((Y-S)+H)+Cy ;  Y := S ;
   } ;
X := X + (Cx - Y*ΔT/2) .
```

These computations were done in 1985 on an IBM PC in BASIC using first the BASICA interpreter, then the BASCOM compiler. Results were obtained for X and Y in Single-Precision first **Un**compensated, then **Com**pensated, and also in Double-Precision.
Single-Precision carries 24 sig. bits, almost 7 sig. dec.;
Double-Precision carries 56 sig. bits, almost 17 sig. dec.
BASICA rounds carelessly; BASCOM rounds carefully.

RESULTS obtained from the BASICA interpreter or the BASCOM compiler:

ΔT	1/4096	1/4096	1/16	1/16	1/1024	1/1024	1/4096
N	40960	40960	16000	16000	1024000	1024000	4096000
T = N ΔT	10	10	1000	1000	1000	1000	1000
S-P Unc. X	-.8388*410*	-.8390552	.4208555	.4208926	.5569*214*	.5623439	.5623*240*
S-P Com. X	-.8390715	-.8390715	.4208943	.4208918	.5623*481*	.5623462	.5623770
D-P Unc X	-.8390715	-.8390715	.4208918	.4208918	.5623463	.5623462	.5623770
x = cos(T)	-.8390715	-.8390715	.5623791	.5623791	.5623791	.5623791	.5623791
S-P Unc. Y	-.5440*174*	-.5440*112*	.9075*201*	.9075522	.8247543	.8269045	.8268*734*
S-P Com. Y	-.5440211	-.5440211	.9075*516*	.9075542	.8268969	.8269020	.8268809
D-P Unc. Y	-.5440211	-.5440211	.9075541	.9075541	.8269020	.8269020	.8268809
y = sin(T)	-.5440211	-.5440211	.8268795	.8268795	.8268795	.8268795	.8268795
Which BASIC ? :	BASICA	BASCOM	BASICA	BASCOM	BASICA	BASCOM	BASCOM

The results from BASICA show compensation subduing massive accumulations of error despite that it malfunctions occasionally because BASICA rounds carelessly. BASCOM rounds addition and subtraction according to IEEE Standard 754 for Floating-Point Arithmetic. Its results show that when rounding is careful the rounding errors behave almost like random variates, tending to cancel each other on average, so they accumulate spasmodically and sluggishly; their effect grows more like $\sqrt{N}$ than like N, the number of time-increments ΔT. Compensation wipes them out. But the foregoing example is a little atypical because it suffers no rounding errors during the computation of F.

When is Compensated Summation Worth the Bother?
That depends upon how the rounding error ηY compares with other errors. A *Backward Error Analysis* of the difference between the desired solution $y(\tau)$ of

$$y(0) = y_0 \quad \text{and} \quad dy/d\tau = f(y, \tau) \quad \text{for} \quad 0 \le \tau \le T$$

and the computed solution $Y(\tau)$ of

$$Y(0) := y_0 \quad \text{and} \quad Y(\tau+\Delta\tau) := Y(\tau) + F(Y(\tau),\tau;\ \Delta\tau, \ldots)\,\Delta\tau$$

reveals that $Y(\tau)$ may be regarded as a set of discrete samples of the exact solution $Y(\tau)$ of an initial value problem differing from the given one by three small perturbations:

$$Y(0) = y_0\ , \quad dY/d\tau = f(Y(\tau),\tau) + \Lambda(\tau)(\Delta\tau)^P + \delta F + \eta Y/\Delta\tau \qquad (*)$$

Here the three perturbing terms are piecewise continuous functions of τ with the following relations to sources of error:

$\Lambda(\tau)(\Delta\tau)^P$ is the local truncation error caused by using formula F instead of the desired but unknown average of f. The "order" of the formula is the positive integer P. In the absence of roundoff, this local truncation error would be the only perturbing term; it can be made arbitrarily tiny if $\Delta\tau$ is tiny enough. (Strictly speaking, Λ depends upon $\Delta\tau$ and Y as well as τ and F and f and y_0, but when $\Delta\tau$ is small enough it and Y can be overlooked.)

$\delta F \doteq$ (error during the computation of $F\,\Delta\tau$)$/\Delta\tau$; we regard this error as unavoidable. In any event, it need not be reduced below the uncertainty in f, which is itself often a mere approximation that models some physical situation.

$\eta Y/\Delta\tau$ is the term that compensated summation would eliminate.

(Because the perturbing terms are only piecewise continuous, the perturbed initial value problem $(*)$ might better be recast as an integral equation, but that is a technical detail that will not alter our conclusions about compensated summation's usefulness.)

Compensated summation appears to pay off only when the term $\eta Y/\Delta\tau$ in $(*)$ rises above the two preceding terms enough to justify the exra effort that its removal will cost. Sometimes no extra effort can be justified:

Compensated summation is not worth the bother for a differential equation $y' = f$ so strongly stable that its trajectory runs to a terminus nearly independent of initial conditions; then all error inherited from the perturbing terms in $(*)$ decays rapidly. Nor is the bother worth while for a differential equation so violently unstable that only the earliest few perturbations matter. Little is gained when so few timesteps $\Delta\tau$ are needed to reach T that a few errors ηY cannot add up to much, nor when arithmetic's precision so far exceeds the accuracy required of $Y(T)$ that huge numbers of errors ηY cannot add up to much. These are the cases when compensated summation does little good; but it does no harm either, and it does not cost much.

How much does compensated summation cost? A handfull of extra arithmetic operations will not be noticed unless F is almost as cheap to compute as in the Example of circular motion above. A more significant cost comes from allocating memory to hold vector $C(\tau)$ as well as $Y(\tau)$. The memory itself is cheap; what costs extra is the time it takes to move $C(\tau)$ as well as $Y(\tau)$ about in memory. If the vectors' dimension is huge, and if F takes just a few arithmetic operations per component to compute, that extra time will be noticeable; but in cases like this the number of timesteps is usually huge and the accuracy saved by compensated summation justifies its cost. Formulas F that tolerate larger (and therefore fewer) timesteps $\Delta\tau$ are generally complicated, often requiring an estimate for $\partial f(y,\tau)/\partial y$ as well as $f(y,\tau)$; then compensated summation adds negligibly to cost.

Compensated summation pays off when very many timestesp $\Delta\tau$ are needed to reach a goal $Y(T)$ that depends upon initial conditions $Y(0)$, and when F can be computed accurately enough only by using the computer's widest hardware-supported precision. Among such situations are simulations of the evolution of planetary systems, long-term computations of orbits of debris launched with satellites, and chemical reactions whose products depend upon initial concentrations of many reagents. Their trajectories are at worst weakly unstable, with errors that, at least initially, grow no faster than a polynomial of low degree as they propagate. The payoff is highest when the timestep $\Delta\tau$ is so tiny that $F\,\Delta\tau$ amounts to a small fraction of Y . Consequently, no payoff can be arbitrarily great because $\Delta\tau$ need never be smaller than small enough that the roundoff term δF dominate the truncation error $\Lambda(\tau)(\Delta\tau)^P$ in (*) above. The more elaborate the formula F , and then the higher its order P , the more modest can be the payoff from compensated summation; but then its cost becomes relatively more modest too.

Compensated summation can be adapted to higher-order differential equations and to multi-step numerical algorithms provided two principles are respected:

1. Whenever possible, exploit exact cancellation of differences like $Y(\tau+\Delta\tau) - Y(\tau)$ to generate no new rounding errors if $Y(\tau+\Delta\tau)$ and $Y(\tau)$ are close. (Recall the *Theorem* above.)

2. Think of $Y(\tau) + C(\tau)$ as an approximation to $y(\tau)$ computed at higher precision than $Y(\tau)$, but not computable directly because $Y(\tau) + C(\tau)$ rounds (usually) to $Y(\tau)$, so it has to be used indirectly.

For example, one algorithm to solve a second order differential equation $y'' = f(y)$ may be expressed conventionally in the form

$$Y(\tau + \Delta\tau) := 2Y(\tau) - Y(\tau - \Delta\tau) + F(Y(\tau), \Delta\tau)\Delta\tau^2 .$$

Roundoff obscures F badly, to an extent proportional to $\varepsilon/\Delta\tau^2$, unless compensated summation is used, which is accomplished by rewriting this algorithm thus:

$$H := ((Y(\tau)-Y(\tau-\Delta\tau)) + (C(\tau)-C(\tau-\Delta\tau))) + F(Y(\tau), \Delta\tau)\Delta\tau^2 ;$$
$$Y(\tau+\Delta\tau) := Y(\tau) + H ;$$
$$C(\tau+\Delta\tau) := (Y(\tau) - Y(\tau+\Delta\tau)) + H .$$

Compensated summation's principal justification is that it allows the error in Y to be analysed and controlled more easily because it is inherited almost entirely from the error inherent in F due to its own roundoff and to the size of $\Delta\tau$.

History - A Comedy of Errors

S. Gill invented compensated summation to perform long trajectory computations on the EDSAC at Cambridge, England, in the late 1940's. His paper in Proc. Camb. Phil. Soc. **47** (1951) p. 96-108 was widely appreciated but more widely misunderstood. Proof that it was misunderstood comes from texts published in the 1960's that recommended the " Runge-Kutta-Gill " method but presented its formulas in a way that did not compensate for roundoff; for an example see formula 25.5.12 on p. 896 of the *Handbook of Mathematical Functions*, ed. M. Abramowitz and Irene A. Stegun, National Bureau of Standards Applied Math. Series **55** (1964). And if the formulas had been presented correctly, they would not have worked in floating-point arithmetic, which had become commonplace by the late 1950's. Gill had devised his scheme for the Fixed-Point arithmetics ubiquitous among the earliest computers; his scheme became obsolete by 1960 and is now disadvantageous.

The first compensated summation algorithms intended for floating-point were published simultaneously in 1965 by R. Møller in *BIT* and by me in *Commun. ACM*, and the material has figured in my lecture notes for numerical analysis classes ever since. But hardly anyone else in America took much notice of the idea.

Throughout the 1960's, expert solvers of differential equations " knew " that the best achievable accuracy was achieved at some " optimal " stepsize $\Delta\tau$. At larger stepsizes, the error got worse as $F(...)$ diverged from the desired average of $f(...)$ like some positive power of $\Delta\tau$. At smaller stepsizes the error got worse like a multiple of $1/\Delta\tau$ because the number of steps, and the accrual of rounding errors, were proportional to $1/\Delta\tau$. (Look again at the perturbed initial value problem (*) above.) Texts plotted a U-shaped graph of Total Error vs. Stepsize; *see it now* on p. 31 of *Scientific Computing and Differential Equations* by G. H. Golub and J. M. Ortega (1992) Academic Press, N. Y. But the contribution of $1/\Delta\tau$ to that graph is spurious and is removed by compensated summation, after which the graph of Total Error levels out, as $\Delta\tau$ diminishes through practicable values, at a level determined by the irreducible uncertainties δF in the computation of each $F(...)$.

Will Rogers said that we suffer less from what we don't know than from what we " know " that isn't so. That may well explain what IBM did in 1967. At stake was a contract to supply computers to NASA. Competing with IBM were CDC and UNIVAC, both of whose computers could perform synthetic higher-precision floating-point arithmetic with some measure of hardware support that IBM /360s

lacked. And NASA insisted upon higher precison than IBM's 16 (roughly) sig. dec. IBM responded with the new 360/85 which introduced an " Extended " (Quadruple-Precision) format good for about 31 sig. dec. with nearly full hardware support; only division needed software. That format persists in to-day's IBM /370s, and is mimicked by a counterpart in DEC VAXs. But over the past two decades hardly anybody has used it, and hardware support for it has mostly atrophied without detracting noticeably from sales. Those " Extended " formats appear to have appeared, at great expense, before their time. Why did NASA insist?

During a SHARE (IBM Users' Group) meeting in 1972 or 1973, I eavesdropped upon a conversation at the bar. (I'm teetotal.) At issue was the arithmetic precision needed to achieve about 8 sig. dec. accuracy during very eccentric orbit calculations with hundreds of millions of timesteps. It seemed that 16 sig. dec. precison was inadequate, but 31 was more than enough, albeit too slow. Compensated summation was not being used; otherwise, by a rough calculation, 16 sig. dec. precision would have been ample, and IBM's and DEC's " Extended " precisions need not have been born prematurely.

Nowadays, because of larger memories, trigonometric functions can be computed to high accuracy faster than formerly, so orbit calculations that used to be performed in Cartesian coordinates can now be performed economically in Polar coordinates, which fluctuate less wildly than Cartesian. Consequently bigger steps Δr and fewer of them can be used, so less accuracy is lost.

Nowadays, the predominant computer arithmetics behave much better than they used to 25 years ago. To begin with, the predominant format used for orbits now is REAL*8 with 53 sig. bits (more than 15 sig. dec.) whereas precision then could be 27 to 56 but was most often CDC's and CRAY's 48 sig. bits (14 sig. dec.). Moreover, roundoff used to be accomplished predominantly by chopping, which is statistically biased, so error usually accumulated in proportion to the number of steps; nowadays almost every algebraic operation is correctly rounded in an unbiased way, so error often accumulates in proportion to the square root of the number of steps. In short, roundoff is less troublesome now than formerly, so old programs get better results than they used to.

Nowadays, computers have far more memory, run far faster, and cost far less than they used to, so orbit calculations proceed far farther than they used to, and far more errors accumulate. But, so far as I know, all major software packages for solving trajectory problems still do so without compensated summation.

Further Reading

Only the simplest form of compensated summation has been presented here. Other methods, and some more applications, were surveyed recently by N. J. Higham in " The Accuracy of Floating Point Summation " p. 783 - 799 of *SIAM J. Sci. Comput.* **14** (July 1993). In " Roundoff error in long-term orbital integrations using multistep methods ", soon to appear in *Celestial Mechanics and*

Dynamical Astronomy, G. D. Quinlan finds *Backward Difference* (rather than algebraically equivalent *Multistep*) formulas and compensated summation yield better accuracy than the other schemes he tried on a simple orbit calculation; here backward differences produce less error partly because they are smaller than the values differenced and partly because the act of differencing introduces usually no new rounding errors, thanks to the *Theorem* above.

That idea can be extended to provide practically arbitrarily high precision in portable software written in any common higher-level programming language; see " Algorithms for Arbitrary Precision Floating Point Arithmetic " by D. M. Priest, *Proc. 10th Symp. on Computer Arith.* (1991) ed. by P. Kornerup and D. Matula for the IEEE Computer Soc. Press, Calif. For much more detail see Priest's Ph. D. thesis " On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations " (1992) Math. Dept., Univ. of Calif. at Berkeley.

Altogether different schemes, based upon arbitrarily wide integer arithmetic even if implemented in floating-point, work better for extremely high precision computation. The best portable software for this purpose, written entirely in Fortran, has been placed in the public domain by Dr. David Bailey of NASA/Ames over the past few years; see *ACM Trans. on Math. Soft* **19** (1993) 288-319, or obtain software and documentation by electronic mail from
mp-request@nas.nasa.gov .

Conclusions
Compensated summation is an inexpensive way, surely cheaper than higher precision, to protect protracted sums from inaccuracy due solely to the process of summation. It is applicable to infinite series, to numerical quadrature, and to trajectory calculations, all of which sum large numbers of small increments, and has been proved valid whenever floating-point arithmetic behaves the way intuition would lead one to expect. I always use it. Why doesn't everybody? Either compensated summation is still too little known or there is something wrong with it that I still do not know.

Acknowledgment
The foregoing material has been assembled from lecture notes for my courses in Numerical Analysis going back almost thirty years. For much of that time I have been supported by the U. S. Office of Naval Research, the National Science Foundation, and/or the Dept. of Energy. Current contract numbers are N 00014-90-J-1372 (ONR) and ASC-9005933 (NSF).

Work remaining to be done for this paper:
Recompute the infinite series more accurately, and explain the Euler-Maclaurin-like approximation.

Replace BASIC programs by Fortran, and illustrate how biased rounding's bias is eliminated, and what happens after many more millions of steps around the circle. What about a real orbit?