Computer System Support for Scientific and Engineering Computation

Lecture 10 - June 2, 1988 (notes revised June 21, 1988)

1 Multiword Arithmetic

Recall that we earlier discussed adding integer multiword arithmetic operations to the definition of C. The advantage of doing this would be to gain efficiency over writing multiprecision integer routines in C. Such a program for add might look like this:

```
unsigned c, a[], s[];

c = 0
for i = 1 to n {
    if (c == 0) {
        s[i] = a[i] + b[i];
        c = (s[i] < a[i]);
    } else {
        s[i] = a[i] + b[i] + 1;
        c = (s[i] <= a[i]);
    }
}
```

How does this compare with an assembly language coded routine? A representative from HP reports that on the HP precision architecture, an assembly coded routine takes $5 + 5N$ cycles to add two N word arrays. The C version takes $5 + (13 + C)N$ cycles, where C varies between 0 and 1 depending on how often there is a carry. On the other hand, the MIPS contingent reports that a MIPS processor takes from 11-14 cycles per iteration (depending on the carries). But if an add with carry intruction were added to the MIPS architecture, it would only speed up to run at 8 cycles per iteration.

2 Nonstandard Numerical Theories

Just as the real world is filled with bitterly controversial topics like abortion and evolution, so is the numerical analysis world. In this section we discuss some of these controversial theories.

2.1 Significance Arithmetic

Significance arithmetic was invented in the early sixties by Ashenhurst and Metropolis. The idea is that if rounding errors have contaminated the lowest k bits of a number, those bits should be thrown away. In significance arithmetic, the contaminated bits are thrown away as the computation proceeds. The hope is that the final answer will have at least the required number of significant digits. If so, a complicated error analysis would be unnecessary. For example, if a number were only accurate to 5 digits on a decimal machine that carries 13 digits, instead of representing it as $3.141592653589 \times 10^0$, you would represent it as $0.000000031416 \times 10^8$. You might reject this theory on philosophical grounds, noting that in real life context suggests that 3 is exact, but 3×10^{17} is most likely not exact. However, we readily discern a more precise reason why significance arithmetic is not a good idea.

When you are given a number like 3.14×10^0 in significance arithmetic, the assumption is that the error is about 1 place in the last digit, that is 1×10^{-2}. If the error in 3.1416 is 3×10^{-2}, we have to either underestimate it by writing 3.14 or overestimate it by writing 3.1. In either case, we have lost information. So no matter what rules you come up with for discarding contaminated bits, once those rules have been picked you can cook up an example whose true error bounds differ quite a bit from what is implied by the number of bits in the final answer. In binary, this could differ as much as $\frac{1}{2}$ bit per arithmetic operation. Thus after n computations, the best error bound could differ from the implied error bound by as much as a factor of $2^{n/2}$ either way.

2.2 Probabilistic Analysis

Another approach to roundoff error is to deal with probabilities. Since most interesting numerical calculations involve many arithmetic operations, we could appeal to the central limit theorem which says that when you add together many random variables with comparable variances, you get a nearly normal distribution. But of course when you perform a particular calculation, you're not interested in the statistics of roundoff error, but rather the roundoff error in your particular calculation. If you are doing a crucial calculation, having a incorrect error bound could be disastrous. It doesn't help you to know that the error bound is correct "on average".

An argument in favor of probabilistic analysis says that incorrect error bounds are quite rare, and if you consider the cost of an incorrect error bound times the probability of such an event, things work out well in the long run. There are two problems with this argument. The first is that you often can't measure the cost of a wrong answer. The second is that rare events occur in the tail of the normal distribution, and convergence of a sum of random variables is fastest at the center, and slowest at the tails. In fact, the farther out in the tail you are, the slower the convergence.

Another problem with the probabilistic approach is that in most computation, only two or three roundoff errors really affect the final answer. In computing the roots of a quadratic equation, the error is serious only when b^2 and ac mostly cancel, so only the two roundoff errors incured in computing each of b^2 and $4ac$ really matter. Another example is linear equations. If you use Gaussian elimination to convert a matrix to upper triangular form, almost all the roundoff error will be in the diagonal element corresponding to the smallest pivot, and in that element the roundoff error will be almost entirely due to a few arithmetic operations. So modelling roundoff error as a normal distribution is often not realistic. However, this doesn't mean that there is no place for probabilistic analysis. It is

useful when testing programs. You can run your program with many different inputs, and compare the computed output to the correct answer. If the statistics of the resulting errors doesn't match your probabilistic erro- analysis, you can suspect there is either an error in your program or in your analysis of it.

2.3 Interval Arithmetc

In interval arithmetic, the error bounds of quantities are carried throughout the calculation. For example, if we know that the variable X lies somewhere in the interval $[\check{x}, \hat{x}]$, and Y lies somewhere in the interval $[\check{y}, \hat{y}]$, then X+Y lies somewhere in $[\check{x}+\check{y}, \hat{x}+\hat{y}]$. When computing these bounds you must round *down* (towards $-\infty$) when computing $\check{x}+\check{y}$, and round *up* (towards $+\infty$) when computing $\hat{x}+\hat{y}$. Because most pre-IEEE hardware doesn't allow you to round both up and down, you have to estimate the effect of rounding. This is usually done by multiplying by $1 \pm \epsilon$, as in classical error analysis.

There are a number of problems with interval arithmetic. The first has to do with quantities that can be represented exactly without any rounding error. For example when computing $3 + 7$ there is no rounding error. However, in an implementation of interval analysis in which you multiply by $1 \pm \epsilon$ to estimate rounding error of the bounds, you would lose the information that $3 + 7$ is exact.

The second problem has to do with the fact that in interval arithmetic, to perform an operation on an interval, you perform that operation on the endpoints of the interval. And you do this step by step for each arithmetic operation. This can be too pessimistic. For example, suppose you want to compute $y = x^2 - x$, and suppose that $x \in [0.49, 0.51]$. Now the graph of $x^2 - x$ is a parabola whose minimum is at $x = 0.5$, so $[0.49, 0.51]$ is carried to $[-.25, -.2499]$. But in a program that computes $y = x^2 - x$ by first computing $z = x^2$ and then $y = z - x$, we first get that $[.49, 0.51]$ is carried to $[.2401, .2601]$ and then carried to $[-.2601, -.2299]$, which is much too pessimistic. The problem is that interval analysis assumes the errors are uncorrelated, which is not the case in this example. However there is a trick you could use, namely rewrite $x^2 - x$ as $(x - \frac{1}{2})^2 - \frac{1}{4}$. For more complicated expressions finding the trick may not be so easy, and for functions of several variables it isn't possible in general.

When we analyze a calculation with interval analysis, we start with an interval and then perform the calculation on that interval. As long as all the steps in the calculation are continuous, at each step the interval gets transformed into another interval. But in a calculation involving two or more variables, intervals get replaced by boxes (squares, cubes, etc). And when we subject a box to a continuous operation, it will in general get transformed into a twisted shape that is not a box. In order to proceed to the next step of interval analysis, we will have to replace that twisted shape by a box that contains it. If we're very unlucky, this new box might include a singularity that was quite far away from the original twisted shape (see Figure 1). However, this problem is not quite as serious as it seems. First of all, as long as the operations are differentiable, they are locally linear and so will take small boxes to shapes that are almost boxes. If the boxes start to get too large, they can be partitioned into a set of smaller boxes, and we can operate on each box independently, taking their union at the end (see Figure 2). Of course this gets expensive in large dimensions. Dividing each edge of the box in half increases the number of boxes by 2^d in dimension d.

Besides these drawbacks, using interval arithmetic makes calculations noticeably slower. Most programmers don't care enough about error analysis to slow down their calculations

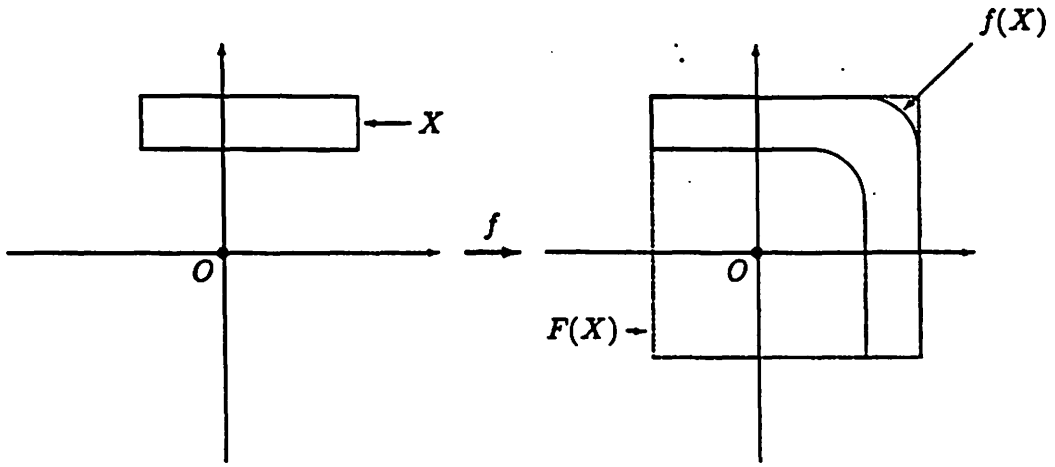


Figure 1: An unexceptional 2-dimensional interval X maps to a region $f(X)$ whose smallest enclosing interval is $F(X)$, which also contains the origin.

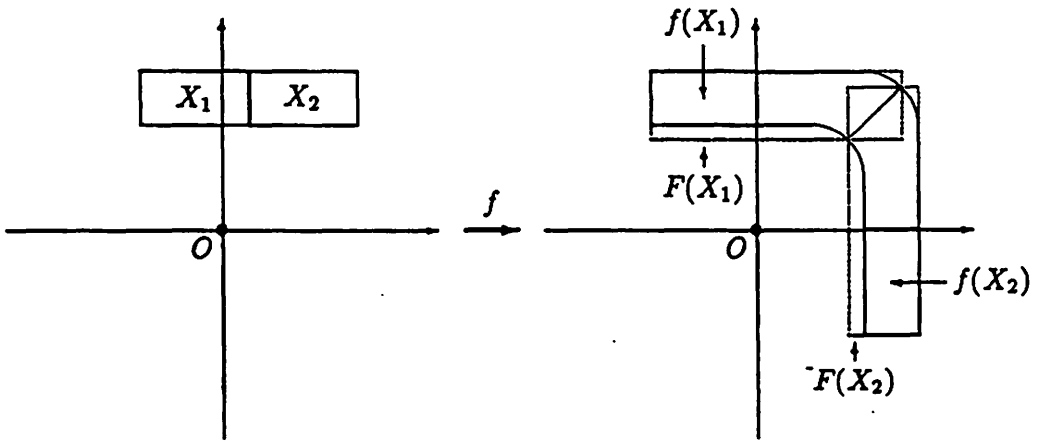


Figure 2: Previous problem improved by subdivision.

that much. It is probably for this reason that none of the standard languages offer any support for interval analysis. This makes interval programming awkward, and in turn decreases its use still further. For all these reasons, interval analysis is not widely used. However, it is useful for searching and optimization problems (see the next section), and it does provide a very useful language for describing errors.

Two references for interval arithmetic are *Methods and Applications of Interval Analaysis* by R. E. Moore (published by SIAM, 1979) and *Introduction to Interval Computations* by G. Alefeld and J. Herzberger (Academic Press, 1983). The first is easy to read, the second is much more mathematical.

2.4 Iteration

There is a class of numerical problems that can be cast in the form of finding a fixed point for a function ϕ, that is solving $\phi(x) = x$. In many cases, we can solve this equation by iterating $x_{i+1} = \phi(x_i)$. For example, to find the roots of $f(x) = 0$, we search for the fixed point of $\phi(x) = x - \frac{f(x)}{f'(x)}$. A useful fact to know about fixed points is

Theorem 1 (Brouwer) *If B is a closed and bounded convex region, and ϕ is a continuous function satisfying $\phi(B) \subseteq B$, then ϕ has a fixed point.*

A *closed* region is one that contains its boundary. Thus $\{(x,y) : x^2 + y^2 < 1\}$ is not closed, but $\{(x,y) : x^2 + y^2 \leq 1\}$ is closed. In one dimension (B is an interval) the proof is easy. For two dimensions it's harder. See Graves's calculus book or *Algebraic Topology: An introduction* by William Massey. For the general case see *Topology From the Differentiable Viewpoint* by John Milnor. The Brouwer theorem doesn't say anything about how many fixed points there might be. One such result is

Theorem 2 *If ϕ is a contraction mapping, that is $|\phi(x) - \phi(x')| < |x - x'|$, then ϕ has a unique fixed point.*

Interval arithmetic is quite useful for solving fixed point problems. Instead of using boxes starting with size 0 and growing due to roundoff error, start with a large box, so large that you know it contains a fixed point (you can use the Brouwer theorem to find such a box). If there is a fixed point in B, it must be in $B \cap \phi(B)$, and then in $B \cap \phi(B) \cap \phi(B \cap \phi(B)), \ldots$. As we iterate, the boxes should close in on the fixed point, not only locating its position, but also providing an error bound. As mentioned above, the image of a box is not necessarily a box, so we have to replace the image by a box containing it. If this expansion isn't too bad, the boxes will close in on a solution.

Besides Newton's method, another example of iteration arises in solving linear equations, that is $A\vec{x} = \vec{b}$. Suppose we have a program P that takes A and $\vec{b}$ and produces an approximate solution $\vec{z} \leftarrow P(A, \vec{b})$ so that $A\vec{z} \approx \vec{b}$. Then we can compute a residual $\vec{r_0} = \vec{b} - A\vec{z} = A(\vec{x} - \vec{z})$. Now we iterate and compute $\vec{c} \leftarrow P(a, \vec{r_0})$, so that $A\vec{c} \approx \vec{r_0} = A(\vec{x} - \vec{z})$ and thus $\vec{c} \approx \vec{x} - \vec{z}$. Usually $\vec{z} + \vec{c}$ will be a better approximation to $\vec{x}$ than $\vec{z}$, and in fact will usually have twice as many correct figures. We can continue the process by letting $\vec{r_1} = \vec{b} - A(\vec{z} + \vec{c})$. This process only works if $\vec{r_i}$ is computed accurately. If our previous estimate for $\vec{x}$ is any good at all, $\vec{b}$ and $A\vec{z}$ will be close, and when we subtract them to compute $\vec{r_i}$, there will be a lot of cancellation. However, if we compute $A\vec{z} - b$ in double precision (or if we were already using double precision, then compute it in quadruple precision), then the calculation should be good enough. And in fact there is a theorem that says that except for certain pathological cases, computing in double precision is as good as

you can do. That is, using higher precision to compute $A\vec{x} - b$ won't improve the accuracy of the final (single precison) result.

Iteration can also be used to solve nonlinear equations. The method uses the multidimensional version of Taylor's theorem.

$$\vec{f}(\vec{x} + \vec{y}) = \vec{f}(\vec{x}) + \vec{f}'(\vec{x})\vec{y} + \cdots.$$

In this equation, $\vec{f}'(\vec{x})$ is a matrix, called the *Jacobian*. If $\vec{x}$ is close to the zero of $\vec{f}$, then $\vec{f}(\vec{x} + \vec{y}) = 0$ for a small value of $\vec{y}$, and the high order terms of Taylor's formula are negligible. Thus we can refine our first guess $\vec{x}$ by adding $\vec{y}$ to it, where $\vec{y}$ is computed from $\vec{f}'(\vec{x})\vec{y} = -\vec{f}(\vec{x})$. And this is a linear equation of the type we just discussed. This method of solving non-linear equations is used in the ACRITH package.

2.5 The Super-Accumulator

Even if $\vec{r_0} = \vec{b} - A\vec{x}$ is computed in double precision, it might not be exact. That is, it might not have the same value as if it were computed in infinite precision and then rounded. Kulisch and Miranker have proposed using a *super-accumulator* to compute inner products exactly. Their theory is explained in *Computer Arithmetic in Theory and Practice* (Academic Press, 1981) and *A New Approach to Scientific Computation* (Academic Press, 1983). The crux of their method is to accurately compute results to single precision by doing all calculations in single precision except for inner products, which are computed exactly in the super-accumulator.

Kulisch and Miranker do not describe their methods as an encoding of multiple-precision arithmetic, but that's what's really going on. Consider again the algorithm for solving $a\vec{x} = \vec{b}$:

$$\begin{array}{ll} A\vec{x_1} \approx \vec{b} & x_1 \leftarrow P(A, \vec{b}) \\ \vec{r_1} = \vec{b} - A\vec{x_1} & x_2 \leftarrow P(A, \vec{r_1}) \\ \vec{r_2} = \vec{b} - A\vec{x_1} - A\vec{x_2} & \vec{x_3} \leftarrow P(A, \vec{r_2}) \end{array}$$

The solution to the equation is $\vec{x_1} + \vec{x_2} + \vec{x_3} + \cdots$, which is really a multiple precision encoding. At each step of the iteration, we must operate on each of the $\vec{x_i}$ separately. And, at each step we must recompute $A\vec{x_1}$ in the super-accumulator, unless we are willing to store in memory all the bits of $A\vec{x_1}$, which could be a lot of storage if the largest and smallest summands differ dramatically in size.

2.6 ACRITH

ACRITH is a package that runs on IBM mainframes and utilizes the Kulisch-Miranker super-acumulator theory. Here is one anomaly that occurs on release 3 of that system. Entering the following nonlinear system and initial guesses

$$\begin{array}{rcl@{\qquad}rcl} a - 2 \times 10^{-9} & = & 0 & a & = & 2 \times 10^{-9} \\ b - 5 \times 10^{8} & = & 0 & b & = & 5 \times 10^{8} \\ x - (1 - a) & = & 0 & x & = & 1 \\ x + ay - 1 & = & 0 & y & = & 1 \\ 3x - by + (b-1)z - (2 - 3a) & = & 0 & z & = & 2 \end{array}$$

causes ACRITH to fail with the message SINGULAR JACOBIAN. However, renaming the variables to

$$\begin{aligned} z - 2\times 10^{-9} &= 0 & z &= 2\times 10^{-9} \\ y - 5\times 10^{8} &= 0 & y &= 5\times 10^{8} \\ x - (1-z) &= 0 & x &= 1 \\ x + zb - 1 &= 0 & b &= 1 \\ 3x - yb + (y-1)a - (2-3z) &= 0 & a &= 2 \end{aligned}$$

successfully computes a solution with $a = 1$ and the other values unchanged. All the variables are computed to within one figure in the last place. The exact output of ACRITH is shown below :

Unknown	Result	Last correction
A	0.1000000000000000D+01	-0.3289D-25
B	0.1000000000000000D+01	-0.3289D-25
X	0.9999999980000000D+00	-0.1283D-16
Y	0.5000000000000000D+09	-0.0000D+00
Z	0.2000000000000000D-08	-0.3053D-25

Unknown	Result
A	(0.9999999999999999D+00 , 0.1000000000000001D+01)
B	(0.9999999999999999D+00 , 0.1000000000000001D+01)
X	(0.9999999979999999D+00 , 0.9999999980000001D+00)
Y	(0.5000000000000000D+09 , 0.5000000000000001D+09)
B	(0.1999999999999999D-08 , 0.2000000000000001D-08)

In this example, it is probably not the Kulisch-Miranker theory that is at fault, but rather a problem with the implementation. It appears that ACRITH orders the variables in alphabetical order. Since the matrix associated with this nonlinear system is very nearly singular, changing the order of the rows in the matrix is enough to make the difference between success and failure.

COMPENSATED SUMMATION

If "$A := B + C$"

produces, $a = (b+c)\cdot(1 \pm \varepsilon)$,

then COMPENSATED SUMMATION

produces $S_N = \sum_0^N x_j \cdot (1 \pm \varepsilon)^{\text{Small Integer}}$

$+ O(N\varepsilon^2) \sum_0^N |x_j|$.

(Small Integer < 6)

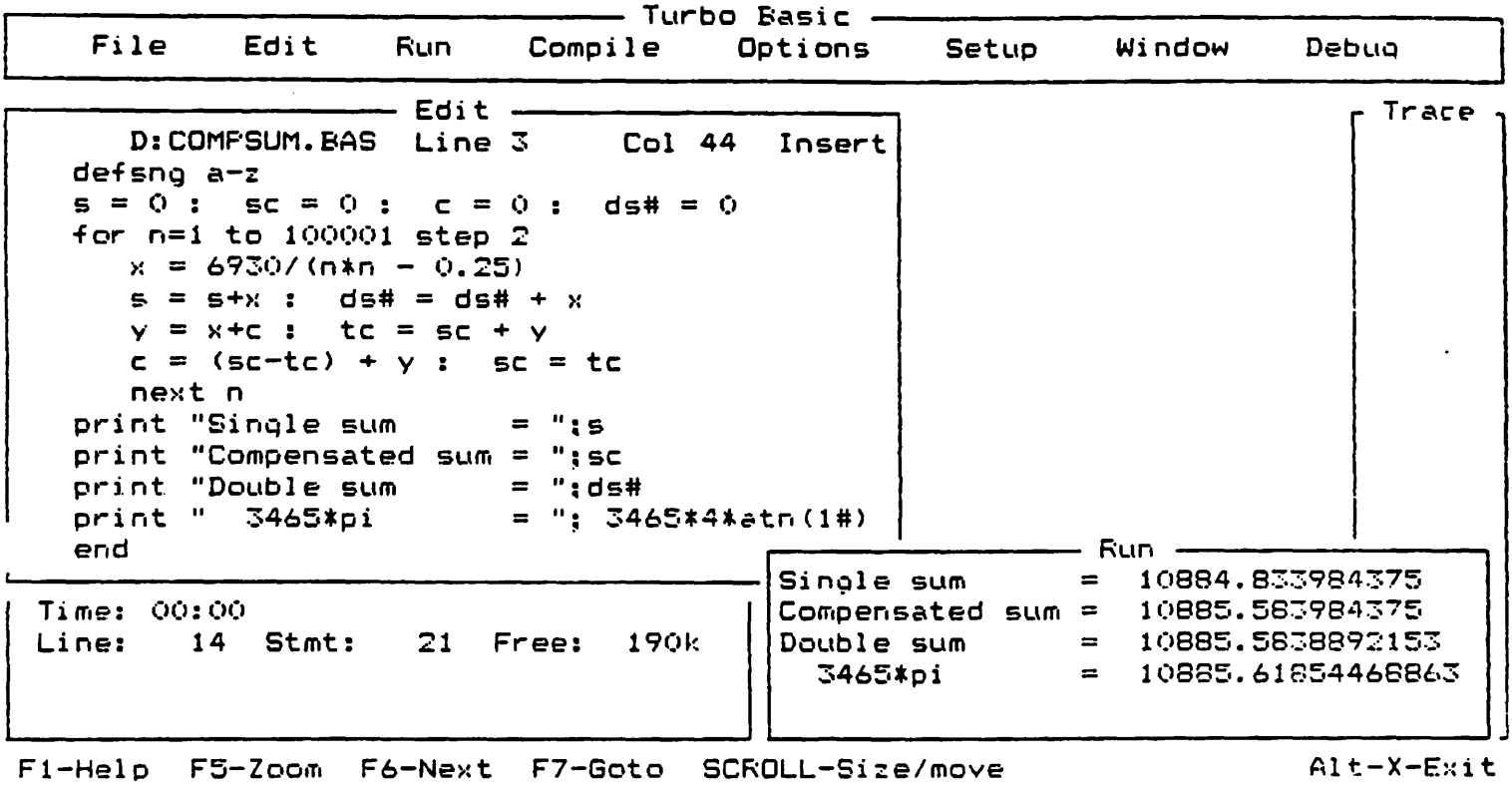
PROOF : SURPRISINGLY HARD !

But now it works for

NON-CONVENTIONAL FLOATING-POINT

e.g. Logarithmic Representation.

as well as ROUNDED DECIMAL, etc.



S: Ordinary Summation

SC: Compensated Summation

DS#: Sum using Double-Precision ADDs

BUT

$$Y_j = X_j + C_{j-1}$$
$$T_j = S_{j-1} + Y_j$$
$$C_j = (S_{j-1} - T_j) + Y_j$$
$$S_j = T_j$$

Trouble : ---

What if $S_{j-1} - T_j$ is NOT EXACT ?

EXAMPLE to 5 sig. dec.

$$S_{j-1} = 0.99998$$
$$Y_j = 0.99998$$
$$S_{j-1} + Y_j = T_j = 1.99996 \rightarrow 2.0000$$
$$S_{j-1} = 0.99998$$
$$(S_{j-1} - T_j) = -1.00002 \rightarrow -1.0000$$
$$C_j = -0.00004$$

But $(S_{j-1} - T_j) + Y_j = -0.00004$ ◁

FAILURE ?

(This CAN'T HAPPEN in BINARY, or CHOPPED Hexadecimal)

CURE : See papers by S. Linnainmaa in BIT, 1980 ... .

NOT REALLY NECESSARY !

?

Compensated Summation

DESIRED: $S_N = \sum_0^N x_j$ for HUGE N.

e.g. – Infinite series $N \doteq \infty$

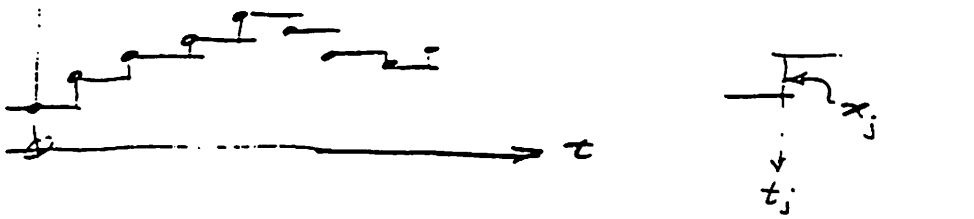
– Numerical Quadrature

– Trajectory Problems
(Ordinary Differential Equations)

$$\frac{dS}{dt} = f(S)$$

$$\rightarrow \quad S(t+\tau) = S(t) + \tau \cdot F(S(t))$$

$$F \doteq \underbrace{\int_t^{t+\tau} f(S(\theta))\, d\theta}_{\uparrow} = \text{Average}(f).$$



USUAL PROGRAM:

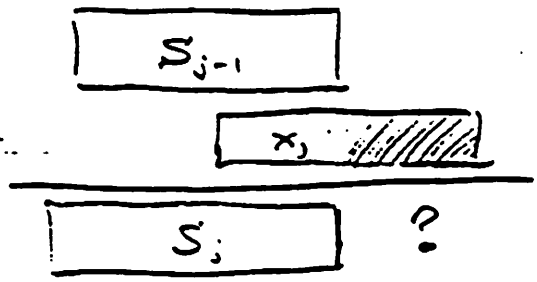
```
S_0 := X_0 ;
for j = 1 to N do S_j := S_{j-1} + X_j ;
```

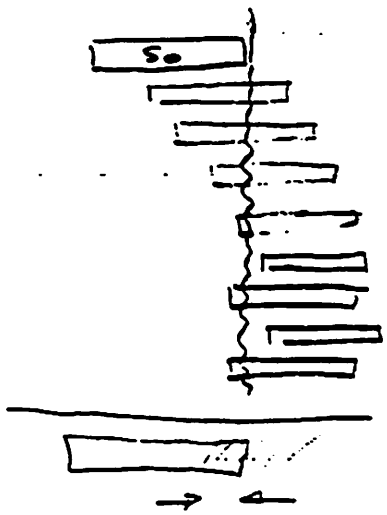
Troublesome when N is HUGE

$$\ldots \quad s_N = \sum_0^N x_j \cdot (1 \pm \epsilon)^{N+1-j}$$

$$\doteq \sum_0^N x_j \cdot (1 \pm (N+1-j)\epsilon)$$

COMPENSATED SUMMATION





COMPENSATED PROGRAM: (cf. S. R. Gill, 1950 !)

$S_0 := X_0$; $C_0 := 0$;

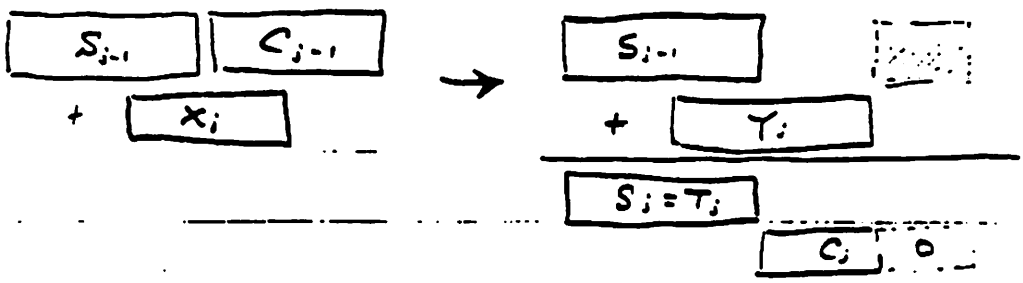
for $j = 1$ to N do

{ $Y_j := X_j + C_{j-1}$;

$T_j := S_{j-1} + Y_j$;

$C_j := (S_{j-1} - T_j) + Y_j$;

$S_j := T_j$; }



Adds about an ulp of error to X_j.

$$A z^2 - 2Bz + C = 0$$

$D := B^2 - AC$;

if $D < 0$ then $z_\pm = \dfrac{B \pm \sqrt{-D}}{A}$

else $\begin{cases} S := B + \text{SIGN}(\sqrt{D}, B) ; \\ z_- := C/S ; \\ z_+ := S/A \ . \end{cases}$

To maintain full accuracy in all cases, compute $D := B^2 - AC$ in at least DOUBLE PRECISION, hence EXACTLY when cancellation is severe.

Alternatives to DOUBLE PRECISION :

- simulate DOUBLE PRECISION using SINGLE !!
- scale A, B, C to integers and use REMAINDER function to compute $D = B^2 - AC$ to near full accuracy despite cancellation.

(See paper on RATIONAL ARITHMETIC in FLOATING-POINT.)

53 sig. bits vs. 24 sig. bits

Turbo Basic

File Edit Run Compile Options Setup Window Debug

Edit

```
D:ITERN.BAS     Line 1      Col 13
defdbl a-z              ... 53 sig. bits
v# = 0.997068882545
def fnf(x)
   local o1, o2
   o1 = 4*v#*x :  o2 = o1*(1-x)
   fnf = o2 :  end def
x = v#
input "How many loops", n
print "Initial    x = ";x
for i=1 to n
   for j=1 to 20
      x = fnf(x) :  next j
   print "i = ";i;"    x = ";x
   next i
end
```

Line: 15 Stmt: 20 Free: 190k

Run

```
i =  83     x =  .9970688825449633
i =  84     x =  .9970688825449634
i =  85     x =  .9970688825449633
i =  86     x =  .9970688825449634
i =  87     x =  .9970688825449633
i =  88     x =  .9970688825449634
i =  89     x =  .9970688825449633
i =  90     x =  .9970688825449634
i =  91     x =  .9970688825449633
i =  92     x =  .9970688825449634
i =  93     x =  .9970688825449633
i =  94     x =  .9970688825449634
i =  95     x =  .9970688825449633
i =  96     x =  .9970688825449634
i =  97     x =  .9970688825449633
i =  98     x =  .9970688825449634
i =  99     x =  .9970688825449633
i =  100    x =  .9970688825449634
```

F1-Help F5-Zoom F6-Next F7-Goto SCROLL-Size/move Alt-X-Exi

Turbo Basic

File Edit Run Compile Options Setup Window Debug

Edit

```
D:ITERN.BAS     Line 1      Col 13
defsng a-z              ... 24 sig. bits
v# = 0.997068882545
def fnf(x)
   local o1, o2
   o1 = 4*v#*x :  o2 = o1*(1-x)
   fnf = o2 :  end def
x = v#
input "How many loops", n
print "Initial    x = ";x
for i=1 to n
   for j=1 to 20
      x = fnf(x) :  next j
   print "i = ";i;"    x = ";x
   next i
end
```

Line: 15 Stmt: 20 Free: 190k

Run

```
i =  3      x =  .7717265486717234
i =  4      x =  .7567538619041443
i =  5      x =  .9641148447990417
i =  6      x =  .3844239115715027
i =  7      x =  .5180658102035522
i =  8      x =  .9968630075454712
i =  9      x =  .1617649644613266
i =  10     x =  .7123320102691650
i =  11     x =  .6466721296310425
i =  12     x =  .9350763559341431
i =  13     x =  .1641836315393448
i =  14     x =  .7653287053108215
i =  15     x =  .5457952618598938
i =  16     x =  .9818066954612732
i =  17     x =  .3048334121704102
i =  18     x =  .1700769513845444
i =  19     x =  1.167876459658146E-002
i =  20     x =  .2320668101310730
```

$x_{n+1} := 4 \cdot V \cdot x_n \cdot (1 - x_n)$, $V = 0.997068882545$

tends to $x_{n+20} = x_n$.