Computer System Support for Scientific and Engineering Computation

Lecture 9 - May 31, 1988 (notes revised June 20, 1988)

1 Multiword Arithmetic

An earlier lecture mentioned that hardware containing an "add with carry" instruction will be able to implement multiprecision add very efficiently. It also mentioned that if the result of $a + b$ is the bit pattern s, then $a + b$ has a carryout exactly when $s < a$, where the compare is an unsigned one. Thus you could write a multiprecision package in C, since C has an unsigned compare operation.

Exercise : Code multiprecision add and subtract both in C and in assembly language on your local computer and compare how fast they run.

Here's the specification for a simple multiprecision package. A multiprecision number is an array $[\mathrm{D}_{n-1}\ \mathrm{D}_{n-2} \ldots \mathrm{D}_1\ \mathrm{D}_0]$. When the bits in these words are concatenated you get a $n \cdot w$ bit integer in two's complement, where w is the number of bits in a single word. `ADDLI(&D,&S,n)` should add the two multiprecision numbers D and S and put the result in D. `SUBLI(&D,&S,n)` should subtract S from D and put the result in D. The value of these functions is 0 if there is no overflow, +1 for positive overflow and −1 for negative overflow. The order in which the successive words D_0, D_1, ...are stored in memory (increasing or decreasing addresses) is left up to each implementation.

2 Simulating Single Precision in Double Precison

Recall the situation of a hardware designer who wants to simplify a design by building only double precision hardware. Can single precision be simulated by first rounding each arithmetic result correctly to double precision, then rounding that result correctly to single precision? Using our notation $[x]_p$ to mean x rounded correctly to p significant digits we need to know for which q, $[[x \otimes y]_q]_p = [x \otimes y]_p$, where $\otimes$ ranges over common operations. In the last lecture, we studied this situation for addition, subtraction, multiplication and division, and concluded that $q \geq 2p + 1$ is enough. In this section, we study the case of square roots.

Theorem 1 *If $[x]_p = x > 0$, then $[[\sqrt{x}]_q]_p = [\sqrt{x}]_p$ provided $q \geq 2p + 1$ for radix $\beta \geq 4$, and $q \geq 2p + 2$ for radix $\beta = 2$.*

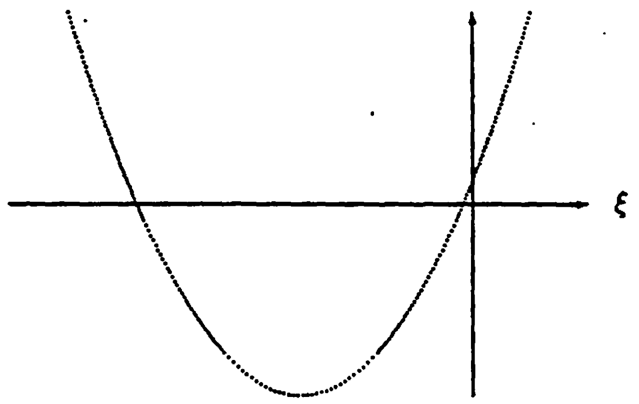


Figure 1: Graph of $4\xi^2 + 4(2N+1)\xi + 1$

Proof: Scale x so that $\beta^{p-1} \le \sqrt{x} < \beta^p$. Let $N = \lfloor\sqrt{x}\rfloor$. Then

$$\beta^{p-1} \le N \le \sqrt{x} \le N+1 \le \beta^p.$$

Define $\xi = \sqrt{x} - (N + \frac{1}{2})$ so that $0 \le |\xi| \le \frac{1}{2}$.

Now $4x = (2\xi + 2N + 1)^2 = 4\xi^2 + 4(2N+1)\xi + (2N+1)^2$ so that

$$\begin{aligned} 4\xi^2 + 4(2N+1)\xi &= 4\beta^{p-1}(x/\beta^{p-1}) - (2N+1)^2 \\ &= 8\left(\frac{1}{2}\beta^{p-1}(x/\beta^{p-1}) - N(N+1)/2\right) - 1 \\ &= 8(M) - 1, \end{aligned}$$

where M is an integer. That's because $\frac{1}{2}\beta^{p-1}$ is an integer since β is even, $N(N+1)/2$ is an integer because one of N or $N+1$ is even, and x/β^{p-1} is an integer because x has p digits and $x \ge \beta^{2p-2}$. Since either $8M - 1 \le -1$ or $8M - 1 \ge 7$ then also $4\xi^2 + 4(2N+1)\xi \le -1$ or $4\xi^2 + 4(2N+1)\xi \ge 7$.

Consider the first case. From the graph of $4\xi^2 + 4(2N+1)\xi + 1$, shown in Figure 1, we see that since this quantity is nonpositive, ξ must be left of the rightmost zero. From the quadratic formula, the zeros are

$$\frac{-(2N+1) \pm \sqrt{(2N+1)^2 - 1}}{2}.$$

Picking the larger root and multiplying through by $-(2N+1) - \sqrt{(2N+1)^2 - 1}$ gives

$$\xi \le \frac{-1}{2(2N+1) + \sqrt{4(2N+1)^2 - 4}} < \frac{-1}{8(N + \frac{1}{2})} < -\frac{1}{8}\beta^{-p},$$

thus $\sqrt{x} < N + \frac{1}{2} - \frac{1}{8}\beta^{-p}$. In a diagram

$\boxed{\quad N \quad} . \boxed{0111\ldots1}$ 111000

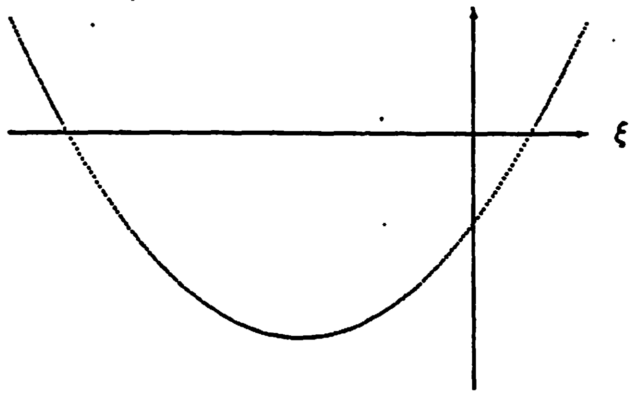


Figure 2: Graph of $4\xi^2 + 4(2N+1)\xi - 7$

$$\boxed{\quad N \quad} \; . \; \boxed{4999\ldots9}\; 875000$$

where each box holds p digits, and the top line represents the case when $\beta = 2$, the bottom line $\beta = 10$. From this we see that $[\sqrt{x}]_p$ is N. If the rightmost box becomes $\frac{1}{2}$ when rounding to q digits, then if N is odd the final rounding to p digits will result in $N+1$. So in order for $[[\sqrt{x}]_q]_p = [\sqrt{x}]_p$, the rightmost box must contain 0111...1. This will happen if the q rounding is to $2p+2$ digits or more (remember that the actual value of $\sqrt{x}$ is *less* than the number in the picture). For a radix greater than 2, only $2p+1$ digits are needed.

The second case is when $4\xi^2 + 4(2N+1)\xi - 7 \geq 0$. The graph of this quadratic, shown in Figure 2, has two positive regions. Since $|\xi| \leq \frac{1}{2}$, ξ must lie to the right of the rightmost zero. The analysis proceeds as in the first case and with the same conclusion about the value of q. This completes the proof.

Here's an example to show that the theorem is best possible. Let $x = (\beta^p - 1)\beta^p$. Then using the binomial theorem

$$\begin{aligned}
\sqrt{x} &= \sqrt{(\beta^p - 1)\beta^p} = \sqrt{\beta^{2p}(1-\beta^{-p})} = \beta^p(1 + (-\beta^{-p}))^{\frac{1}{2}} \\
&= \beta^p\left(1 + \frac{1}{2}(-\beta^{-p}) + \frac{\frac{1}{2}(\frac{1}{2}-1)}{2!}(-\beta^{-p})^2 + \frac{\frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)}{3!}(-\beta^{-p})^3 + \cdots\right) \\
&= \beta^p - \frac{1}{2} - \frac{1}{8}\beta^{-p} - \frac{1}{16}\beta^{-2p} - \ldots
\end{aligned}$$

So just as in the proof, $\sqrt{x} < N + \frac{1}{2} - \frac{1}{8}\beta^{-p}$, where $N = \beta^p - 1$. And N is odd. So in this example $[\sqrt{x}]_p = N$ but if $\beta = 2$, $[[\sqrt{x}]_{2p+1}]_p = N + 1$.

3 Error Analysis of the Quadratic

In this section we will perform an error analysis for finding the roots of a quadratic equation. The analysis is quite complex, even for such a simple problem. Every time hardware designers cook up new floating point hardware that invalidates the assumptions in an error analysis, the analysis has to be done all over again. This points up one advantage of using a standard like IEEE 754/854. Once the hard work of performing an error analysis for this standard is done, it can be shared by all users of the standard.

We're going to study the equation $Ax^2 - 2Bx + C = A(x - Z_+)(x - Z_-)$. In order to make the following formulas simpler, we have made a harmless substitution and used $-2B$ instead of the more usual B as the coefficient of x. The familiar formula for the roots is

$$Z_\pm = \frac{B \pm \sqrt{B^2 - AC}}{A}. \tag{1}$$

The roots satisfy the relations

$$\begin{aligned} Z_+ + Z_- &= \frac{2B}{A} & (2)\\ Z_+ - Z_- &= \frac{2\sqrt{B^2 - AC}}{A} & (3)\\ Z_+ Z_- &= \frac{C}{A} & (4)\end{aligned}$$

We will make heavy use of these formulas in our error analysis. Its easy to see that the formula for Z_- can be numerically unstable. If B^2 is much larger than $|AC|$, then there will be considerable roundoff error when computing $\sqrt{B^2 - AC} \approx |B|$ and when this is subtracted from B (or added to B if $B < 0$), cancellation will be disastrous. However, this is can be avoided by changing the formula: multiply the numerator and denominator by $B \mp \sqrt{B^2 - AC}$ to get the better formula

$$Z_\pm = \frac{C}{B \mp \sqrt{B^2 - AC}}. \tag{5}$$

What is less obvious is that when $B^2 \approx AC$, then no matter which formula we use, up to half the digits can be lost. This is a good example of the general principle mentioned earlier, that you should carry about twice as many digits in a calculation as you want in your final result.

3.1 Forward Error Analysis

We can make some simplifying assumptions about the coefficients of the quadratic. We can assume $A > 0$ (otherwise multiply the quadratic by -1) and we can assume $B > 0$ (otherwise multiply the roots Z_- and Z_+ by -1 and then swap them, so that Z_+ remains the largest root). We will now do a forward error analysis for the case $B^2 \gg |AC|$. Since $B > 0$,

$$\left|\frac{Z_+}{Z_-}\right| = \left|\frac{B + \sqrt{B^2 - AC}}{B - \sqrt{B^2 - AC}}\right| = \left|\frac{(B + \sqrt{B^2 - AC})^2}{AC}\right| \approx \frac{4B^2}{|AC|} \gg 1. \tag{6}$$

Using the classical model of roundoff, the computed value $z_\pm$ corresponding to $Z_\pm$ is

$$z_{\boldsymbol{\pm}} = (1 \pm 2\epsilon)\frac{B \boldsymbol{\pm} (1 \pm 2\epsilon)\sqrt{B^2 - AC}}{A}.$$

There are two different meanings of $\pm$. The bold $\boldsymbol{\pm}$ correlate with the two different roots of the quadratic. The nonbold $\pm$ represent uncorrelated roundoff errors.

The error of $(1 \pm 2\epsilon)$ in front of the square root requires some explanation. Because $B^2 \gg |AC|$, the roundoff error committed in computing AC doesn't contribute to the roundoff error in $B^2 - AC$. Thus the error in $B^2 - AC$ is $(1+\epsilon)^2$, one from the square and the other from the subtract. Taking the square root reduces that to $1+\epsilon$, but then you need to factor in another $1+\epsilon$ for the error committed in computing the square root. Using (2) and (3) we can rewrite the equation as

$$z_{\pm} = (1 \pm 2\epsilon)\left(\frac{Z_+ + Z_-}{2} \pm (1 \pm 2\epsilon)\frac{Z_+ - Z_-}{2}\right) \tag{7}$$

$$z_+ = (1 \pm 2\epsilon)Z_+\left(1 \pm \epsilon(1 - \frac{Z_-}{Z_+})\right) \tag{8}$$

Since $|Z_-/Z_+| < 1$ from (6), $1 - 2\epsilon < 1 \pm \epsilon(1 - Z_-/Z_+) < 1 + 2\epsilon$, so we end up with

$$z_+ = (1 \pm 2\epsilon)(1 \pm 2\epsilon)Z_+ \approx (1 \pm 4\epsilon)Z_+. \tag{9}$$

To get the other root, go back to (7) to get

$$\begin{aligned} z_- &= (1 \pm 2\epsilon)(Z_- \pm \epsilon(Z_+ - Z_-)) \\ &= (1 \pm 2\epsilon)\left(1 \pm \epsilon(\frac{Z_+}{Z_-} - 1)\right)Z_- \\ &\approx (1 \pm 3\epsilon)(1 \pm \epsilon\frac{Z_+}{Z_-})Z_- \\ &\approx \left(1 \pm (3 + \frac{4B^2}{|AC|})\epsilon\right)Z_- \end{aligned} \tag{10}$$

where we used (6) in the last step. So we see that when $B^2 \gg |AC|$ and $B > 0$, the smaller root Z_- may have lost as many as $3 + (4B^2/|AC|)$ ulps. So it is quite possible that none of the figures in Z_- are meaningful.

As we mentioned above, we can avoid this disastrous cancellation by picking whichever formula of (1) and (5) doesn't involve cancellation. Or more precisely

Algorithm 1 (Quadratic Formula) *Let $D = B^2 - AC$. Then if $D < 0$ compute the complex roots using $Z_{\pm} = (B \pm i\sqrt{-D})/A$, otherwise compute the real roots using $S = B + \sqrt{D} \cdot sign(B)$, $Z_- = C/S$ and $Z_+ = S/A$.*

3.2 Backward Error Analysis

We will now perform a backward error analysis of the quadratic formula.

Theorem 2 *The computed roots $z_{\pm}$ differ from the true roots $Z_{\pm}$ by at most a few ulps more than if the coefficients A, B, and C had first been perturbed by at most a few ulps.*

Proof: The actual value of D is

$$d = B^2(1+\kappa) - AC(1+\pi). \tag{11}$$

Each greek letter in this proof will represent 1 or 2 rounding errors. In the equation above, we used the version of error estimate suitable even for machines without a guard digit, and each of κ, π actually represent two rounding errors. If $d < 0$ then

$$z_{\pm} = \frac{B}{A}(1+\beta) \pm i\frac{\sqrt{-d}}{A}(1+\delta). \tag{12}$$

To do a backward error analysis, we need to find A', B', C', and $D' = B'^2 - A'C'$ so that

$$z_\pm = \frac{B' \pm i\sqrt{-D'}}{A'} = \frac{B}{A}(1+\beta) \pm i\frac{\sqrt{-d}}{A}(1+\delta). \qquad (13)$$

This will exhibit $z_\pm$ as *exactly* the roots of a slightly perturbed quadratic. Suppose we let

$$\begin{aligned} A' &= A && (14)\\ B' &= B(1+\beta) && (15)\\ C' &= C(1+\gamma). && (16) \end{aligned}$$

Then clearly the real parts of (13) are equal. To compute γ, we equate the imaginary parts and get

$$\begin{aligned} \sqrt{-D'} &= \sqrt{-d}(1+\delta)\\ B^2(1+\beta)^2 - AC(1+\gamma) &= (1+\delta)^2\left(B^2(1+\kappa) - AC(1+\pi)\right)\\ 1+\gamma &= \frac{B^2}{AC}\left((1+\beta)^2 - (1+\delta)^2(1+\kappa)\right) + (1+\delta)^2(1+\pi)\\ 1+\gamma &\approx \frac{B^2}{AC}(2\beta - 2\delta - \kappa) + (1+\delta)^2(1+\pi). \end{aligned}$$

Since $d < 0$, then $D \le 0$ and $B^2/|AC| \le 1$ so γ is just a few ulps. You might wonder why $d < 0$ implies $D < 0$. On almost all machines (the Cray is an exception), the products $B \cdot B$ and $A \cdot C$ are computed internally to full precision and then rounded in some fashion. Since if $x < y$ then $[x]_p \le [y]_p$ for all the standard types of rounding, then if $B^2 < AC$ it follows that the computed values satisfy $b^2 \le ac$.

To deal with the case $d \ge 0$ note that

$$\begin{aligned} s &= (B + \sqrt{d}\cdot \mathrm{sign}(B))(1+\sigma)\\ z_- &= \frac{C}{s}(1+\zeta_-)\\ z_+ &= \frac{s}{A}(a+\zeta_+). \end{aligned}$$

In a simple backward error analysis, the computed quantity is exactly the solution of a slightly perturbed problem. This was the case when $d < 0$. If only life were so simple. In many cases, even after perturbing the original problem, we still don't get the computed quantity exactly, but only within a few ulps. This is the situation we are in now. If we let

$$\begin{aligned} A' &= A\\ B' &= B\sqrt{\frac{1+\kappa}{1+\pi}}\\ C' &= C, \end{aligned}$$

then calculation shows that $\frac{1+\delta}{1+\zeta_-}Z_-$ and $\frac{1}{(1+\zeta_+)(1+\delta)}Z_+$ are exact roots of $A'x^2 - 2B'x + C' = 0$, where

$$1+\delta = \frac{|s|}{|b| + \sqrt{b^2 - ac}} = (1+\sigma)\sqrt{1+\pi}\,\frac{\frac{|b|}{1+\kappa} + \sqrt{b^2-ac}}{|b| + \sqrt{b^2-ac}}.$$

So when $d \ge 0$, the actual roots differ by just a few ulps from the exact roots of a perturbed equation one of whose coefficients differs from the original by a few ulps.

3.3 Multiple Roots

What we really want to know about the roots computed from algorithm 1 is how many significant digits they contain. The forward error analysis answered that question when $B^2 \gg |AC|$ (equation(9)). What about the general case? Backward error analysis told us that the computed roots differed only a few ulps from the roots of a perturbed equation. But perhaps the roots of the perturbed equation are nowhere near the roots of the original equation. We can analyze this by looking at the quadratic $AZ_+^2 - 2BZ_+ + C = 0$ and considering the root $Z_+ = Z_+(A,B,C)$ to be a function of the coefficients. Then if we differentiate implicitly with respect to the variable A, we get $Z_+^2 + 2AZ_+\frac{\partial Z_+}{\partial A} - 2B\frac{\partial Z_+}{\partial A} = 0$ and solving for $\frac{\partial Z_+}{\partial A}$ gives

$$\frac{\partial Z_+}{\partial A} = \frac{-Z_+^2}{A(2Z_+ - \frac{2B}{A})} = \frac{-Z_+^2}{A(2Z_+ - Z_+ - Z_-)} = \frac{-Z_+^2}{A(Z^+ - Z_-)}$$

using (2) and (3). We can repeat this calculation differentiating with respect to B and C to get

$$\begin{aligned} \frac{\partial Z_+}{\partial A} &= \frac{-Z_+^2}{A(Z^+ - Z_-)} \\ \frac{\partial Z_+}{\partial B} &= \frac{2Z_+}{A(Z^+ - Z_-)} \\ \frac{\partial Z_+}{\partial C} &= \frac{-1}{A(Z^+ - Z_-)}. \end{aligned} \tag{17}$$

We conclude that when the two roots are close together, a small change in any one of the coefficients will make a large difference in the value of the root x. So in this case, backwards error analysis doesn't give us any information about how many significant digits algorithm 1 generates. We will need to do a forward error analysis.

When the roots are nearly equal, the crucial roundoff error comes from D.

$$\begin{aligned} d &= B^2(1 \pm \kappa) - AC(1 \pm \pi) \\ &= A^2\left(\frac{B^2 - AC}{A^2}(1 \pm \kappa) - \frac{C}{A}(\pm\kappa \pm \pi)\right) \\ &= (1 \pm \kappa)A^2\left(\left(\frac{Z_+ - Z_-}{2}\right)^2 + \frac{\pm\kappa \pm \pi}{1 \pm \kappa}Z_+Z_-\right) \end{aligned}$$

where we used (3) and (4). Taking square roots gives

$$\frac{\sqrt{d}}{A} = (1 \pm \kappa/2)\sqrt{\left(\frac{Z_+ - Z_-}{2}\right)^2 + \frac{\pm\kappa \pm \pi}{1 \pm \kappa}Z_+Z_-}\,.$$

To simplify this, we use the fact that if $p \geq q \geq 0$ then

$$p - q \leq \sqrt{p^2 - q^2} \leq \sqrt{p^2 + q^2} \leq p + q$$

so $p \pm q$ is at least as large as $\sqrt{p^2 \pm q^2}$. Using this together with $Z_+ \approx Z_-$, we conclude that

$$\frac{\sqrt{d}}{A} = (1 \pm \kappa/2)\left(\frac{Z_+ - Z_-}{2} \pm \sqrt{\left|\frac{\pm\kappa \pm \pi}{1 \pm \kappa}\right| |Z_+Z_-|}\right)$$

$$\approx \quad (1 \pm \kappa/2)\left(\frac{\sqrt{D}}{A} \pm |Z_{\pm}|\sqrt{\left|\frac{\pm\kappa \pm \pi}{1 \pm \kappa}\right|}\right).$$

If Z_+ has p significant figures in base β, then the errors κ, π are about β^{-p} so that $\sqrt{|\kappa \pm \pi|}$ is about $\beta^{-p/2}$ and the error term $|Z_+|\sqrt{|\kappa \pm \pi|}$ has about half as many digits as $|Z_+|$. From the formulas

$$\begin{aligned} Z_{\pm} &= \frac{B}{A} \pm i\frac{\sqrt{-d}}{A} \\ Z_+ &= \frac{B}{A} + \mathrm{sign}(B)\frac{\sqrt{d}}{A} \\ Z_- &= \frac{C}{B + \mathrm{sign}(B)\sqrt{d}} \end{aligned}$$

we can see that the error in computing d swamps out the other errors, and so when $Z_+ \approx Z_-$, we lose at most half the digits.

Is there a way to compute the roots to full accuracy when they are nearly equal? One solution is to compute $D = B^2 - AC$ in double precision, so that when we lose half the figures, we are back to full single precision. If double precision isn't available we can simulate it by splitting each number into 2 parts. Then multiplying two such quantities together will require 4 single precision multiplies. Another trick is to scale A, B, and C to integers and use the remainder function. See the paper *Rational Arithmetic in Floating Point* for details.

As the zeros of a quadratic get closer, the formulas for the partial derivatives (17) suggest that the rounding error grows like $1/|Z_+ - Z_-|$. On the other hand, we have just showed that the rounding error never consumes more than half the digits. Why does the rounding error plateau? One possible explanation involves scaling the quadratic to make the coefficents integers. The constraint that the coefficients are integers puts a limit on how close the roots can be. This is studied in a paper by Mignotte.

3.4 General Comments on Quadratics

The argument above used the explicit formula given in algorithm 1. It turns out that we don't need any explicit formulas. If all we know is that there is some algorithm satisfying Theorem 2, then we can prove that you can compute roots which have at least half their figures correct, and when the zeros are far apart, the roots are in error by at most $\max(Z_+, Z_-)/|Z_+ - Z_-|$ ulps.

When we use the naive formula for solving the quadratic and when $\lambda = AC/B^2 \approx 0$, formula (10) shows that the error is about $1/|\lambda|$ ulps. This is no accident. A recent paper by James Demmel in *Mathematics of Computation* (April 1988) shows that in general, you lose about $1/|\lambda|$ ulps where λ is the distance to a singularity. With the naive formula, the singularity occurs when $\lambda = 0$ so the distance is just $|\lambda|$. Another example occurs when the roots become equal, $\lambda = 1$. In this case, equation (17) suggests that the roundoff error is $1/(|1 - \lambda|)$, which again is the reciprocal of the distance to the singular set.

The quadratic equation also illustrates another principle, namely that you are more likely to find problems with an algorithm if you perform it in single precision. Imagine that you carry out your calculations with p bits of precision, that you print out and compare the first n bits with the correctly computed result, where $n < p$. The roundoff error will have to exceed $p - n$ bits before you notice it. If you are using the naive formula for solving quadratics, the roundoff error is $1/|\lambda|$ ulps, as we observed above, so we will observe a discrepancy between what is actually computed and what should be computed only if $1/|\lambda| < 2^{n-p}$. If we use double precision ($p = 53$) and compare with an answer correctly computed to single precision ($n = 24$) we will notice a discrepancy only when $|\lambda| < 2^{-29} = .000000002$.

4 Single Precision vs Double Precision

We have previously stressed the importance of doing calculations in single precision. However, there are some calculations where no matter how careful you are, single precision just isn't enough. Consider the recurrence $x_{n+1} = 4Vx_n(1 - x_n)$ which arises in discrete dynamical systems. Suppose that we choose $V = 0.997068882545$ and compute the x's using a program like this.

```
double v = 0.997068882545
float fnf(x) {
    float tmp1, tmp2;

    tmp1 = 4*v*x;
    tmp2 = tmp1*(1 - x);
    return(tmp2);
}
x = v;
for i = 1 to n {
    for j=1 to 20
        x = fnf(x);
    print i, x;
}
```

Figure 3 shows the results of running this program first in single precision with 24 significant bits, and then in double precision with 53 significant bits. In the program, the constant v# ends with a # which causes it to be stored in double precision, even in the single precision version of the program. When run in double precision, we discover that $x_{n+20} \approx x_n$. However, roundoff error destroys this relationship when computed in single precision.

```
----------- Program ------------------------- Results ---------------

D:ITERN.BAS                         | i =  83    x = .9970688825449633
defdbl a-z            : REM Double  | i =  84    x = .9970688825449634
v# = 0.997068882545                 | i =  85    x = .9970688825449633
def fnf(x)                          | i =  86    x = .9970688825449634
   local o1, o2                     | i =  87    x = .9970688825449633
   o1 = 4*v#*x :  o2 = o1*(1-x)     | i =  88    x = .9970688825449634
   fnf = o2 :  end def              | i =  89    x = .9970688825449633
x = v#                              | i =  90    x = .9970688825449634
input "How many loops", n           | i =  91    x = .9970688825449633
print "Initial   x = ";x            | i =  92    x = .9970688825449634
for i=1 to n                        | i =  93    x = .9970688825449633
   for j=1 to 20                    | i =  94    x = .9970688825449634
      x = fnf(x) :  next j          | i =  95    x = .9970688825449633
   print "i = ";i;"    x = ";x      | i =  96    x = .9970688825449634
   next i                           | i =  97    x = .9970688825449633
end                                 | i =  98    x = .9970688825449634
                                    | i =  99    x = .9970688825449633
                                    | i =  100   x = .9970688825449634


D:ITERN.BAS                         | i =  3     x = .7717265486717224
defsng a-z            : REM Single  | i =  4     x = .7567538619041443
v# = 0.997068882545                 | i =  5     x = .9641148447990417
def fnf(x)                          | i =  6     x = .3844239115715027
   local o1, o2                     | i =  7     x = .5180658102035522
   o1 = 4*v#*x :  o2 = o1*(1-x)     | i =  8     x = .9968630075454712
   fnf = o2 :  end def              | i =  9     x = .1617649644613266
x = v#                              | i =  10    x = .712332010269165
input "How many loops", n           | i =  11    x = .6466721296310425
print "Initial   x = ";x            | i =  12    x = .9350763559341431
for i=1 to n                        | i =  13    x = .1641836315393448
   for j=1 to 20                    | i =  14    x = .7653287053108215
      x = fnf(x) :  next j          | i =  15    x = .5457952618598938
   print "i = ";i;"    x = ";x      | i =  16    x = .9818066954612732
   next i                           | i =  17    x = .3048334121704102
end                                 | i =  18    x = .1700769513845444
                                    | i =  19    x = 1.167876459658146E-002
                                    | i =  20    x = .232066810131073
```

Figure 3: Single and double-precision summation

5 Compensated Summation

Consider the problem of computing a very long summation $\sum_0^N x_i$. There are three common situations where this calculation arises. The obvious one is estimating an infinite series. Two other applications are numerical quadrature and ordinary differential equations. We briefly indicate how the numerical solution of an ordinary differential equation of the form $\frac{dy}{dt} = f(y)$ results in computing a long sum. Using the fundamental theorem of calculus

$$\begin{aligned} y(t+\tau) - y(t) &= \int_t^{t+\tau} y'(s)ds = \int_t^{t+\tau} f(y(s))ds \\ y(t+\tau) &= y(t) + \tau\left(\frac{1}{\tau}\int_t^{t+\tau} f(y(s)ds)\right) \\ &\approx y(t) + \tau \cdot \mathrm{av}(f). \end{aligned}$$

The quantity that we have called av(f) is an approximation to the integral. The integral can't be computed exactly, but represents an average of f. Different approximations form the basis for different solution methods. But no matter which method is used, to compute the value of y at $\bar{t}$, you start with the value at t and add many summands ($(\bar{t} - t)/\tau$ of them) to get the value of $f(\bar{t})$.

The obvious program for computing a long sum is

```
S = X[0]
for j = 1 to N {
    S = S + X[j]  /* S[j] = S[j-1] + X[j] */
 }
```

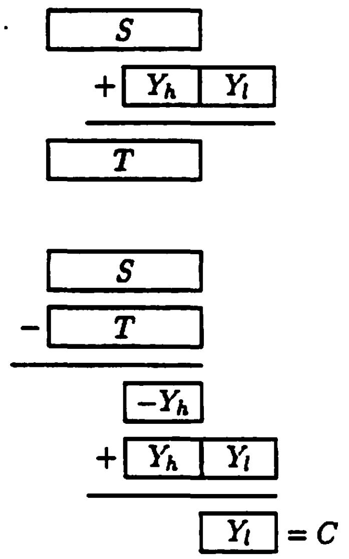
When N is very large, then classical error analysis tells us that

$$s_N = \sum_0^N x_j(1 \pm \epsilon)^{N+1-j} \approx \sum_0^N x_j(1 \pm (N+1-j)\epsilon).$$

In other words, the error bounds are huge. In particular, the contribution of the early summands can be completely lost due to roundoff error. The simplest way to improve the accuracy of long summations is to compute them in a higher precision. If they are already being computed in the highest precision, there is a method called *Compensated Summation* that can improve the accuracy at the cost of only 3 extra operations. A technique similar to this for fixed point numbers was first used around 1950 by S. P. Gill. The method explained below was introduced by Møller in a paper in BIT around 1960, and independently by Kahan. The program works like this.

```
S = X[0];
C = 0;
for j = 1 to N {
    Y = X[j] + C;       /* Y[j] = X[j] + C[j-1] */
    T = S + Y;          /* T[j] = S[j-1] + Y[j] */
    C = (S - T) + Y;    /* C[j] = (S[j-1] - T[j]) + Y[j] */
    S = T;              /* S[j] = T[j] */
}
```

To see why this improves accuracy, consider the following diagram of the procedure.



Each time we add in a summand, there is a correction factor C which will be added in on the next loop. So first we add the correction C_{j-1} from the previous loop to X_j, giving us a corrected summand Y_j. Then we add this summand to the running sum S_{j-1}. The low order bits of Y (namely Y_l) are lost in the sum. Next we compute the high order bits of Y by computing $S_{j-1} - T_j$. When we add that back into Y we will have recovered the low order bits of Y. These are the bits that were lost in the first sum in the diagram. They become the correction factor for the next loop. Figure 4 is a numerical example illustrating how well the method works.

The explanation given above is only a heuristic: it doesn't always hold exactly, as the following example in decimal with 5 digits of precision shows.

$$\begin{array}{rcllll}
S_{j-1} & = & 0.99998 & & & \\
Y_j & = & 0.99998 & & & \\
S_{j-1} + Y_j = T_j & = & 1.99996 & \rightarrow & 2.00000 & \\
S_{j-1} & = & & & 0.99998 & \\
(S_{j-1} - T_j) & = & & & -1.00002 & \rightarrow \quad -1.00000 \\
Y_j & = & & & & \qquad\quad .99998 \\
(S_{j-1} - T_j) + Y_j = C_j & = & & & & \qquad -0.00002
\end{array}$$

The expected correction factor is −0.00004, but the algorithm yields −0.00002 instead. In a paper in BIT, S. Linnainmaa introduced refinements to the algorithm to try and avoid this anomaly. But a better approach is to simply recognize that the explanation given above is not intended to be exact, and instead rely on an error analysis. The error analyis of compensated summation is surprisingly hard. The result is that

$$s_N = \sum_{0}^{N} x_j(1 \pm \epsilon)^k + O(N\epsilon^2)\sum_{0}^{N} |x_j|,$$

where k is a small integer.

16

```
-------------------- Edit --------------------
|     D:COMPSUM.BAS  Line 3     Col 44  Insert|
|   defsng a-z                                |
|   s = 0 :  sc = 0 :  c = 0 :  ds# = 0       |
|   for n=1 to 100001 step 2                  |
|      x = 6930/(n*n - 0.25)                  |
|      s = s+x :  ds# = ds# + x               |
|      y = x+c :  tc = sc + y                 |
|      c = (sc-tc) + y :  sc = tc             |
|      next n                                 |
|   print "Single sum      = ";s              |
|   print "Compensated sum = ";sc             |
|   print "Double sum      = ";ds#            |
|   print "  3465*pi       = "; 3465*4*atn(1#)|
|   end                                       |
-----------------------------------------------
```

```
---------------- Run ----------------
|Single sum      =  10884.833984375   |
|Compensated sum =  10885.583984375   |
|Double sum      =  10885.5838892153  |
|  3465*pi       =  10885.61854468863 |
--------------------------------------
```

Figure 4: Compensated Summation at work.

Comparison of MULTI-WORD INTEGER ADD/SUBTRACT Programs in ...

— Assembly language, using ADD-with-CARRY, SUB'T-with-BORROW on i8086, μ68020, NS 32032, DEC VAX, IBM 370 (but not on MIPS or some other RISC machines.)

— C .

Program specifications:

$$\left\{ \begin{array}{ll} \text{ADDLI}(\&D, \&S, n) \ \ldots & D = D+S \quad \text{and} \\ \text{SUBLI}(\&D, \&S, n) \ \ldots & D = D-S , \end{array} \right.$$

where destination $D = [D_{n-1}\ D_{n-2} \cdots D_1\ D_0]$
and source $S = [S_{n-1}\ S_{n-2} \cdots S_1\ S_0]$

are INT arrays with at least n words $(n \geq 0)$, each regarded as a 2's complement n-word integer, overwrite D with $D \pm S$ and return zero for ADDLI and SUBLI if no overflow occurs. However, if
positive overflow occurs, return +1 instead.
negative - - - - .. −1 ...

Note: The word-order is at the programmer's discretion, so &D can point to D_0 or D_{n-1} or something else.

Kahan 27 May

CORRECTED VERSION May 31/88

$\circledast$ is $\sqrt{\ }$:

$$\boxed{\begin{array}{l}\text{If } [x]_p = x > 0, \text{ then} \\ [[\sqrt{x}]_q]_p = [\sqrt{x}]_p \\ \text{PROVIDED } q \ge 2p+1 \text{ for radix } \beta \ge 4, \\ \qquad\qquad\quad q \ge 2p+2 \text{ for radix } \underline{\beta = 2} \text{ (BINARY)}\end{array}}$$

Proof: Scale x so that $\beta^{p-1} \le \sqrt{x} < \beta^p$, and

ASSUME $[\sqrt{x}]_{p+1} \ne \sqrt{x}$, else we would be done.

$\therefore$ The INTEGER $N = \lfloor \sqrt{x} \rfloor$ satisfies

$$\beta^{p-1} \le N < \sqrt{x} < N+1 \le \beta^p .$$

Define $\xi = \sqrt{x} - (N + 1/2)$, so $0 < |\xi| < 1/2$.

since $[\sqrt{x}]_{p+1} \ne \sqrt{x}$.

Now $4\xi^2 + 4\cdot(2N+1)\cdot\xi + (2N+1)^2 = (2\xi + 2N+1)^2 = 4x \ge 4\beta^{2p-2}$

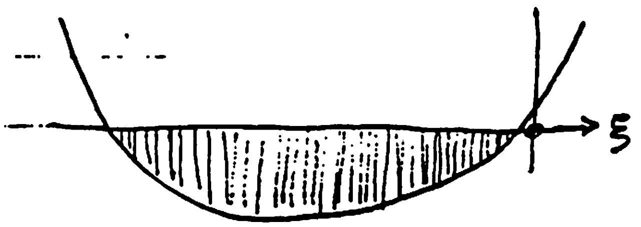
i.e. $4\xi^2 + 4\cdot(2N+1)\xi = 4\beta^{p-1}\cdot(x/\beta^{p-1}) - (2N+1)^2$

$$= 8\cdot\{\tfrac{1}{2}\beta^{p-1}\cdot(x/\beta^{p-1}) - \tfrac{N(N+1)}{2}\} - 1$$

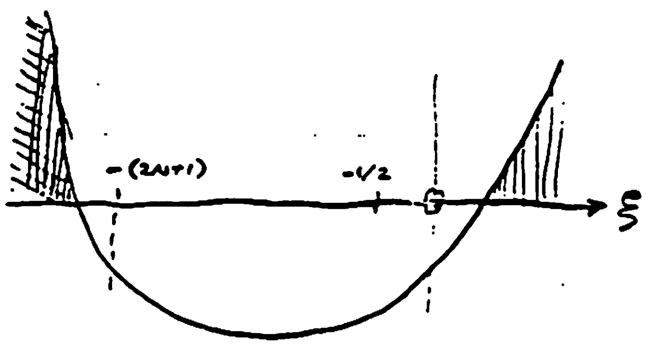
INTEGERS!

$$= 8\cdot\{\text{INTEGER}\} - 1$$

$\therefore$ $4\xi^2 + 4(2N+1)\xi \le -1$ or $4\xi^2 + 4(2N+1)\xi \ge +7$.



ξ lies BETWEEN zeros of $4\xi^2 + 4(2N+1)\xi + 1$



ξ lies BEYOND positive zero of $4\xi^2 + 4(2N+1)\xi - 7$

$$\xi = \sqrt{x} - (N+\tfrac{1}{2}) \neq 0.$$

$$4\xi^2 + 4(2N+1)\xi + 1 \leq 0 \quad \text{or} \quad 0 \leq 4\xi^2 + 4(2N+1)\xi - 7$$

$$\xi \leq \frac{-1}{2(2N+1)+\sqrt{4(2N+1)^2-4}} \quad \text{or} \quad \xi \geq \frac{7}{2(2N+1)+\sqrt{4(2N+1)^2-28}}$$

$$< \frac{-1}{8(N+\frac{1}{2})} < -\tfrac{1}{8}\beta^{-p} \qquad\qquad > \frac{7}{8(N+1)} > \tfrac{7}{8}\beta^{-p}$$

$$\therefore \quad \sqrt{x} < N + \tfrac{1}{2} - \tfrac{1}{8}\beta^{-p} \quad \text{or} \quad \sqrt{x} > N + \tfrac{1}{2} + \tfrac{7}{8}\beta^{-p}$$

$$\boxed{N}.\boxed{4999\cdots9}875 \qquad\qquad \boxed{N}.\boxed{5000\cdots0}875$$

$$\boxed{N}.\boxed{0111\cdots1}111 \qquad\qquad \boxed{N}.\boxed{1000\cdots0}111$$

$$\therefore \quad \text{If } \beta \geq 4, \quad [\,[\sqrt{x}\,]_{2p+1}\,]_p = [\sqrt{x}\,]_p \,.$$

$$\text{If } \beta = 2, \quad [\,[\sqrt{x}\,]_{2p+2}\,]_p = [\sqrt{x}\,]_p \,.$$

<u>Example:</u> $x = (\beta^p - 1)\beta^p$

$$\sqrt{x} = \beta^p - \tfrac{1}{2} - \underline{\tfrac{1}{8}\beta^{-p}} - \tfrac{1}{16}\beta^{-2p} - \cdots$$

$\therefore$ When $\beta \geq 4$, we really do need $q \geq 2p+1$

$\beta = 2$ ——— $q \geq 2p+2$

to ensure $[\,[\sqrt{x}\,]_q\,]_p = [\sqrt{x}\,]_p$.

ERROR-ANALYSIS OF THE QUADRATIC

$$Ax^2 - 2Bx + C \equiv A\cdot(x - Z_+)\cdot(x - Z_-)$$

has ZEROS $$Z_\pm = \left(B \pm \sqrt{B^2 - AC}\right)/A \qquad \circledast$$

$$\left(\text{i.e.} \quad \begin{aligned} Z_+ + Z_- &= 2B/A \\ Z_+ - Z_- &= 2\sqrt{B^2-AC}/A \\ Z_+ \cdot Z_- &= C/A \end{aligned} \right)$$

The formula $\circledast$ above is NUMERICALLY UNSTABLE

- when $B^2 >> |AC|$ ... then ALL figures carried can be lost
- when $B^2 \doteqdot AC$ ... then HALF the " " " " .

To analyze the behavior of $\circledast$, we introduce first some SIMPLIFYING ASSUMPTIONS:

Say $A > 0$; else reverse signs of $A, B, C,$
Say $B > 0$; else swap & reverse signs of Z_+ , Z_- .

$$\therefore \quad Z_+ = \left(B + \sqrt{B^2 - AC}\right)/A$$

has $\mathrm{Re}(Z_+) > 0$.

and $$|Z_+/Z_-| = \left|\frac{B + \sqrt{B^2-AC}}{B - \sqrt{B^2-AC}}\right| = \left|\frac{(B+\sqrt{B^2-AC})^2}{AC}\right| \geq$$

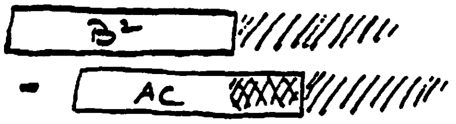
$$>> 1 \quad \text{when } B^2 >> AC$$

$$Z_\pm = \big(B \pm \sqrt{B^2 - AC}\,\big)/A \qquad \textcircled{\pm}$$

$B > 0$ $\qquad$ $A > 0$

WHAT is actually computed when $B^2 \gg |AC|$?

$$z_\pm = (1 \pm 2\varepsilon)\cdot\Big(B \pm (1 \pm 2\varepsilon)\sqrt{B^2 - AC}\Big)/A$$



$$z_\pm = (1 \pm 2\varepsilon)\left(\frac{(Z_+ + Z_-)}{2} \pm (1 \pm 2\varepsilon)\frac{(Z_+ - Z_-)}{2}\right)$$

Recall $\;|Z_+/Z_-| \approx \frac{4B^2}{|AC|} \gg 1.$

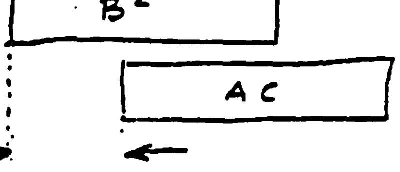
$$z_+ \doteq (1 \pm 3\varepsilon)\cdot Z_+\cdot\Big(1 \pm \underbrace{\varepsilon\cdot\frac{Z_-}{Z_+}}_{\text{O.K.}}\Big) \doteq (1 \pm 4\varepsilon)\cdot Z_+$$

$$z_- \doteq (1 \pm 3\varepsilon)\cdot\big(Z_- \pm \varepsilon Z_+\big)$$

$$= (1 \pm 3\varepsilon)\cdot Z_-\cdot\Big(1 \pm \varepsilon\cdot\frac{Z_+}{Z_-}\Big)$$

$$\doteq \Big(1 \pm \underbrace{\big(3 + \tfrac{4B^2}{|AC|}\big)}\varepsilon\Big)\cdot Z_-$$

DIGITS LOST



$$Z_\pm = (B \pm \sqrt{B^2 - AC})/A \qquad \circledast$$

There is a BETTER FORMULA :

$D := B^2 - AC$ ... DISCRIMINANT

```
If D<0 then COMPLEX $Z_\pm := \frac{B \pm \imath\sqrt{-D}}{A}$
           else REAL ROOTS $Z_\pm$ thus:
             { $S := B + \mathrm{SIGN}(\sqrt{D}, B)$ ;
               $Z_- := C/S$ ;
               $Z_+ := S/A$ .
```

Note : Special cases $A=0$, $A=\infty$, $C=0$, $C=\infty$ are skipped.

Note : NO CANCELLATION except for D.

NUMERICAL STABILITY :

Computed roots $z_\pm$ differ from respective true roots $Z_\pm$ by a few ulps more than if coefficients A, B, C had first been perturbed by a few ulps.

$\therefore$ Roots are INACCURATE only if they are too nearly COINCIDENT .

Proof: BETTER FORMULA actually computes ...

$$\boxed{d = B^2\cdot(1+\kappa) - AC\cdot(1+\pi)}$$

If $d<0$, $\quad z_\pm = \frac{B}{A}\cdot(1+\beta) \pm \imath\frac{\sqrt{d}}{A}\cdot(1+\delta)$

Note $AC > B^2$ here.

Replace A by $a=A$

B by $b = B\cdot(1+\beta)$

C by $c = C\cdot(1+\gamma)$ where

$$1+\gamma = (1+\pi)(1+\delta)^2 + \underset{\text{fraction}}{\left(\frac{B^2}{AC}\right)}\cdot\underbrace{\left\{(1+\beta)^2-(1+\delta)^2(1+\kappa)\right\}}_{\{2\beta+2\delta-\kappa+\cdots\}}$$

$z_\pm$ are roots EXACTLY of $az^2-2bz+c=0$.

If $d\geq 0$, $\quad s = (B+\sqrt{d}\cdot\text{sign}(B))\cdot(1+\sigma)$

$z_- = (C/s)\cdot(1+\varsigma_-)$

$z_+ = (s/A)\cdot(1+\varsigma_+)$.

Now replace A by $a=A$, C by $c=C$, B by $b=B\sqrt{\frac{1+\kappa}{1+\pi}}$;

define $1+\delta = |s|/(|b|+\sqrt{b^2-ac}) = (1+\sigma)\sqrt{1+\pi}\,\frac{(|b|/(1+\kappa)+\sqrt{b^2-ac})}{(|b|+\sqrt{b^2-ac})}$;

Then $\left(\frac{1+\delta}{1+\varsigma_-}\right)z_-$ and $\frac{z_+}{(1+\varsigma_+)(1+\delta)}$ are roots EXACTLY of

$$az^2-2bz+c=0.$$

In both cases, the computed roots $z_\pm$ differ from the true roots $Z_\pm$ by at most a few ulps more than if the coefficients A, B, C had each been perturbed by at most a few ulps.

$$Ax^2 - 2Bx + C = A\cdot(x - Z_+)\cdot(x - Z_-) .$$

How do Z_+ and Z_- change when coefficients A, B, C are perturbed by a few ulps ?

$$Az^2 - 2Bz + C = 0 \Rightarrow \begin{cases} \dfrac{\partial z}{\partial A} = -\frac{1}{2} z^2/(Az - B) \\ \dfrac{\partial z}{\partial B} = -\frac{1}{2} z/(Az - B) \\ \dfrac{\partial z}{\partial C} = -\frac{1}{2}/(Az - B) \end{cases}$$

$$2(Az - B) = A\cdot(2Z - Z_+ - Z_-)$$

$$\therefore \quad \frac{\partial Z_+}{\partial(A,B,C)} = (Z_+^2,\ Z_+,\ 1)/(Z_- - Z_+)$$

$$= (Z_+^2,\ Z_+,\ 1)\cdot\left(\frac{\mp A/2}{\sqrt{B^2 - AC}}\right)$$

$\therefore$ Zeros are hypersensitive to perturbations in coefficients just when they are too nearly coincident, i.e. when $B^2 \doteqdot AC$.

In this case, as we shall see,

at most half the figures carried can be lost.

Error when $B^2 \doteq AC$, so $Z_+ \doteq Z_-$.

$$d = B^2\cdot(1+\kappa) - A\cdot C\cdot(1+\pi)$$
$$= (1+\kappa)\cdot A^2\cdot\left(\left(\frac{Z_+ - Z_-}{2}\right)^2 + \left(\frac{\kappa-\pi}{1+\kappa}\right)\cdot Z_+ Z_-\right)$$

$$\frac{1}{1+\kappa}\cdot\frac{\sqrt{d}}{A} = \sqrt{\left(\frac{Z_+ - Z_-}{2}\right)^2 + \left(\frac{\kappa-\pi}{1+\kappa}\right) Z_+ Z_-}$$
$$= \left(\frac{Z_+ - Z_-}{2}\right) \pm \underset{\sim\sim\sim}{\sqrt{\left|\frac{\kappa-\pi}{1+\kappa}\right| \, |Z_+ Z_-|}}$$

since if $p \geq q \geq 0$ then

$0 \leq p - q \leq \sqrt{p^2 - q^2}$ and $\sqrt{p^2+q^2} \leq p+q$.

So, error in Complex $\frac{B}{A} \pm i\frac{\sqrt{-d}}{A}$

or in Real $\left\{ \begin{array}{l} \dfrac{B + \sqrt{d}\ \mathrm{sign}(B)}{A}, \text{ and} \\ \dfrac{C}{B + \sqrt{d}\ \mathrm{sign}\,B} \end{array} \right.$

is <u>Relatively</u> of the order of $\sqrt{|\kappa - \pi|}$ at worst,

i.e. at most HALF the figures carried can be lost.

Summary

The BETTER FORMULA to solve

$$Ax^2 - 2Bx + C = 0$$

produces roots a few ulps worse than if the coefficients A, B, C had been perturbed by a few ulps.

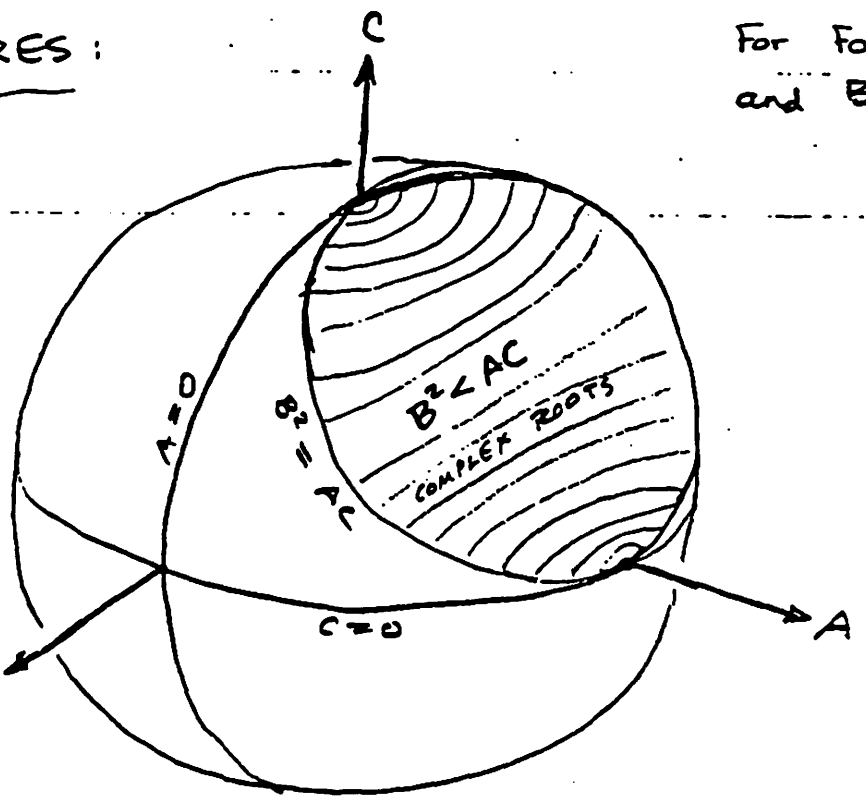
In consequence, computed roots $Z_\pm$ are correct in at least half the figures carried, and otherwise in error by at most a modest multiple of

$$\frac{\max|Z_\pm|}{|Z_+ - Z_-|} \text{ ulps}\ldots$$

$$\boxed{Az^2 - 2Bz + C = 0}$$

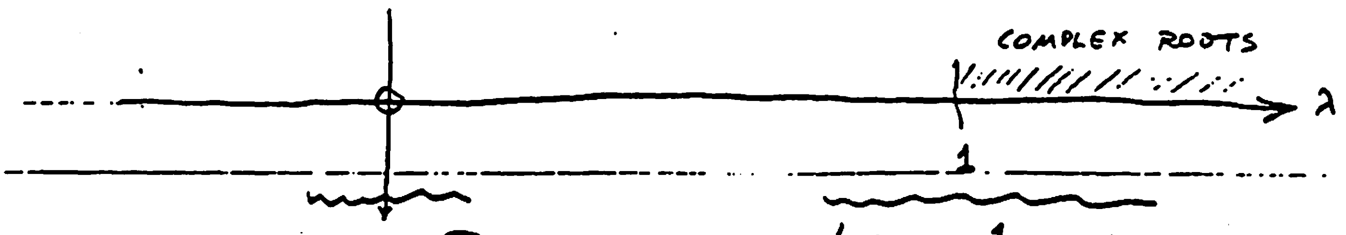
Pictures:

For Formula ⊛ and Better Formula



Circle "$B^2=AC$" loses ½ the figures carried.
Circles "$A=0$" and "$C=0$" lose all figures in ⊛.

How do lost figures in ⊛ depend on closeness to latter two circles ?
Use $\lambda = AC/B^2$ as dimensionless parameter:



Error in ⊛ is $(1/\lambda)$ ulps.

Lose $\frac{1}{|1-\lambda|}$ ulps up to ½ figures carried.

If you did not know that $\textcircled{B}$ is an unsatisfactory formula, what are the chances that random testing might reveal the danger ?

Carry p bits of precision
Print out n bits ; $n < p$.

$\therefore$ Observe loss only if it exceeds $p-n$ bits,

i.e. if $\lambda = AC/B^2$ is smaller than 2^{n-p}.

e.g. take $p = 53$ (double-precision)
$n = 24$ (single-precision)
need $|\lambda| < 2^{-29} = 0.000\,000\,002$.
UNLIKELY.

Similarly, to observe loss in accuracy due to nearly double roots, $|1-\lambda|$ would have to be smaller than 2^{n-p} ; and if $n < p/2$ the loss could NEVER be observed!

$\therefore$ DEBUG IN SINGLE PRECISION!

$$Az^2 - 2Bz + C = 0$$

```
D := B^2 - AC ;
if D < 0 then  Z± = (B ± i√(−D)) / A
     else { S := B + SIGN(√D, B);
            Z₋ := C/S;
            Z₊ := S/A .
```

To maintain full accuracy in all cases, compute $D := B^2 - AC$ in at least DOUBLE PRECISION,

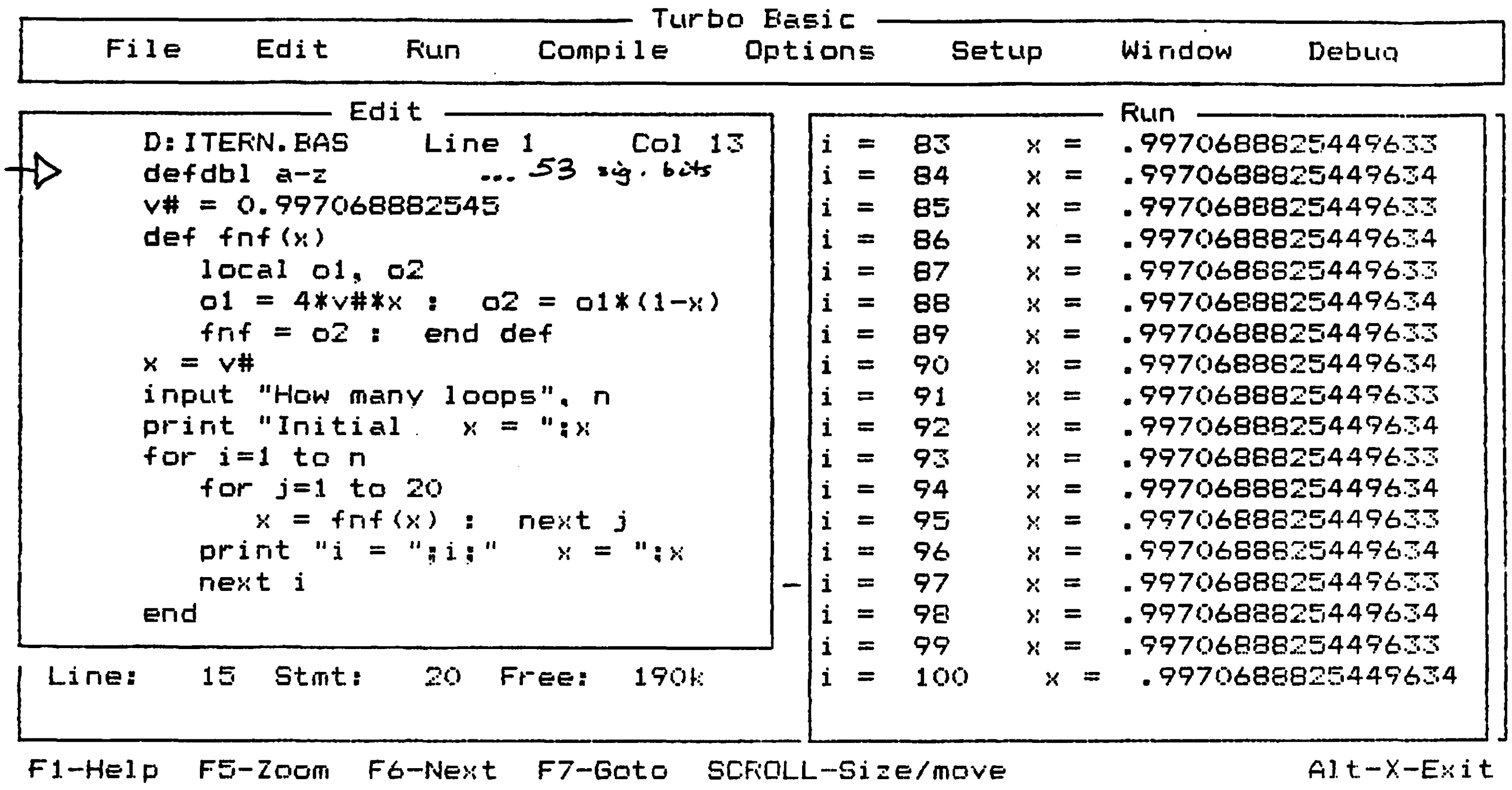
hence EXACTLY when cancellation is severe.

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.)

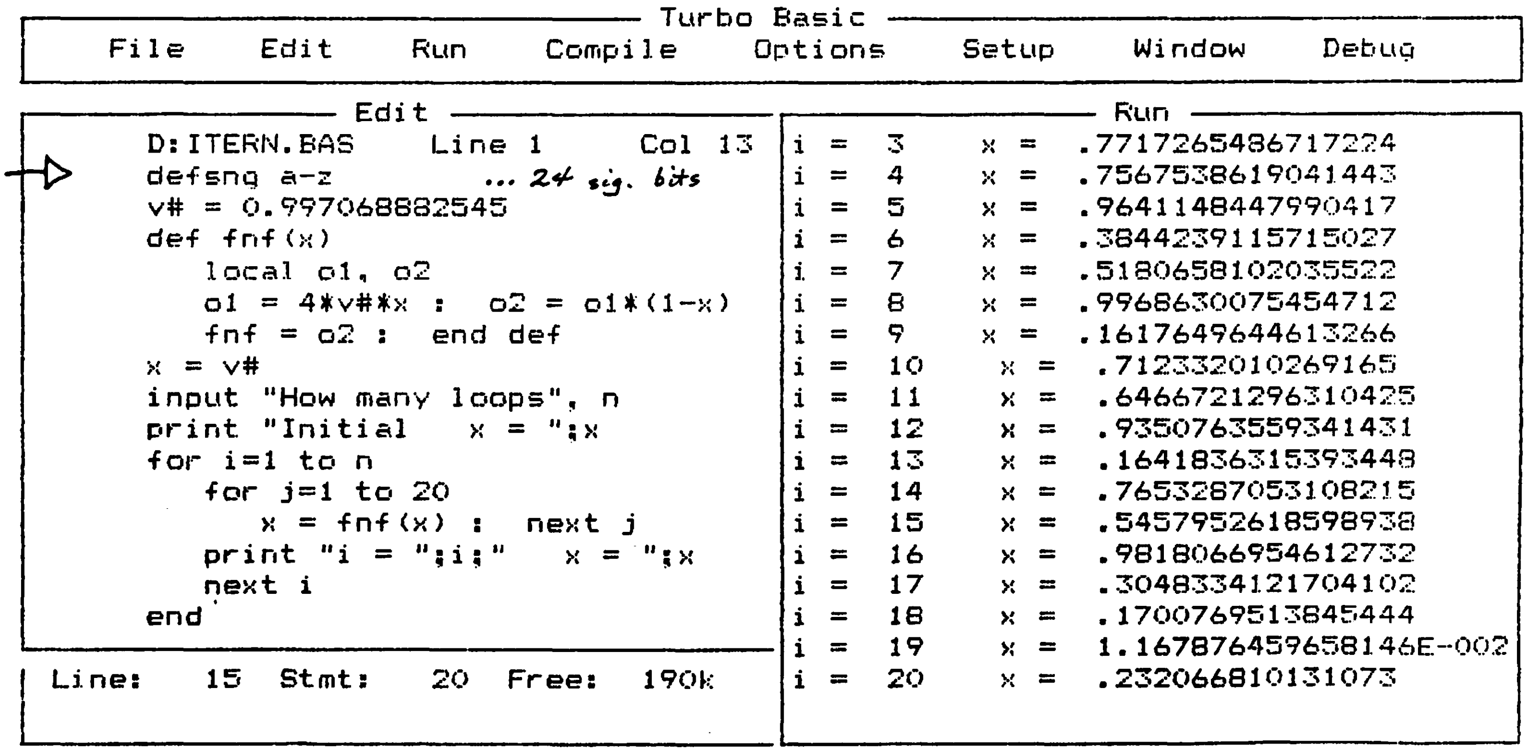
53 sig. bits   vs.   24 sig. bits



```
                              Turbo Basic
     File      Edit      Run      Compile      Options      Setup      Window      Debug

                    Edit                                             Run
     D:ITERN.BAS     Line 1     Col 13          i  =  83     x  =  .997068882544963 3
     defdbl a-z            ... 53 sig. bits     i  =  84     x  =  .997068882544963 4
     v# = 0.997068882545                        i  =  85     x  =  .997068882544963 3
     def fnf(x)                                 i  =  86     x  =  .997068882544963 4
        local o1, o2                            i  =  87     x  =  .997068882544963 3
        o1 = 4*v#*x  :  o2 = o1*(1-x)           i  =  88     x  =  .997068882544963 4
        fnf = o2 :  end def                     i  =  89     x  =  .997068882544963 3
     x = v#                                     i  =  90     x  =  .997068882544963 4
     input "How many loops", n                  i  =  91     x  =  .997068882544963 3
     print "Initial   x = ";x                   i  =  92     x  =  .997068882544963 4
     for i=1 to n                               i  =  93     x  =  .997068882544963 3
        for j=1 to 20                           i  =  94     x  =  .997068882544963 4
           x = fnf(x) :  next j                 i  =  95     x  =  .997068882544963 3
        print "i = ";i;"   x = ";x              i  =  96     x  =  .997068882544963 4
        next i                                  i  =  97     x  =  .997068882544963 3
     end                                        i  =  98     x  =  .997068882544963 4
                                                i  =  99     x  =  .997068882544963 3
 Line:   15  Stmt:   20  Free:  190k            i  =  100     x  =  .997068882544963 4

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



```
                              Turbo Basic
     File      Edit      Run      Compile      Options      Setup      Window      Debug

                    Edit                                             Run
     D:ITERN.BAS     Line 1     Col 13          i  =  3     x  =  .771726548671722 4
     defsng a-z           ... 24 sig. bits      i  =  4     x  =  .756753861904144 3
     v# = 0.997068882545                        i  =  5     x  =  .964114844799041 7
     def fnf(x)                                 i  =  6     x  =  .384423911571502 7
        local o1, o2                            i  =  7     x  =  .518065810203552 2
        o1 = 4*v#*x  :  o2 = o1*(1-x)           i  =  8     x  =  .996863007545471 2
        fnf = o2 :  end def                     i  =  9     x  =  .161764964461326 6
     x = v#                                     i  =  10     x  =  .712332010269165
     input "How many loops", n                  i  =  11     x  =  .646672129631042 5
     print "Initial   x = ";x                   i  =  12     x  =  .935076355934143 1
     for i=1 to n                               i  =  13     x  =  .164183631539344 8
        for j=1 to 20                           i  =  14     x  =  .765328705310821 5
           x = fnf(x) :  next j                 i  =  15     x  =  .545795261859893 8
        print "i = ";i;"   x = ";x              i  =  16     x  =  .981806695461273 2
        next i                                  i  =  17     x  =  .304833412170410 2
     end                                        i  =  18     x  =  .170076951384544 4
                                                i  =  19     x  =  1.16787645965814 6E-002
 Line:   15  Stmt:   20  Free:  190k            i  =  20     x  =  .232066810131073
```

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

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

Compensated Summation

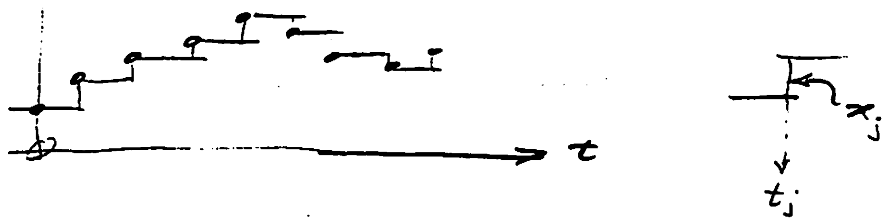
DESIRED: $S_N = \sum_0^N x_j$ for HUGE N.

e.g. – Infinite series $N \doteq \infty$
– Numerical Quadrature
– Trajectory Problems
(Ordinary Differential Equations)

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

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

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



USUAL PROGRAM:

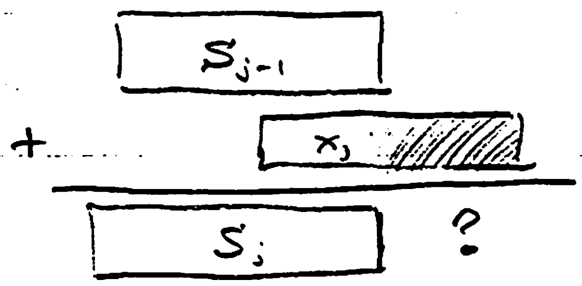
```
$S_0 := X_0$ ;
for $j = 1$ to $N$ do $S_j := S_{j-1} + X_j$ ;
```

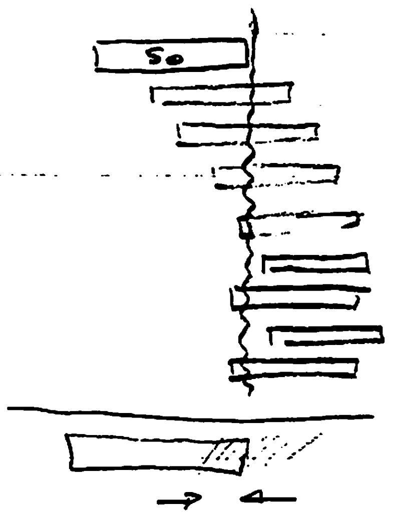
Troublesome when N is HUGE

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

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

COMPENSATED SUMMATION



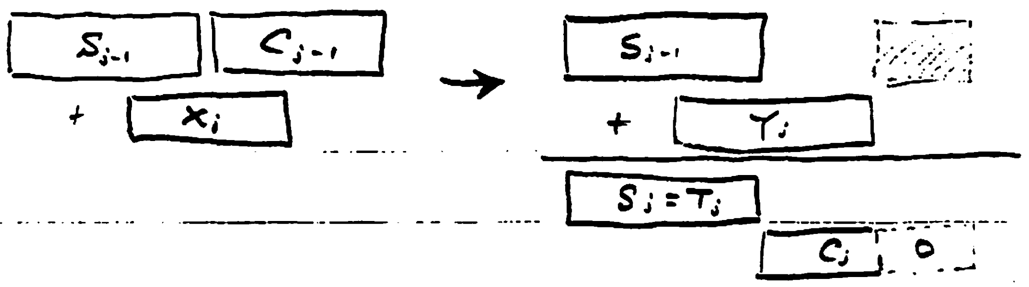


N ulps of S_N.

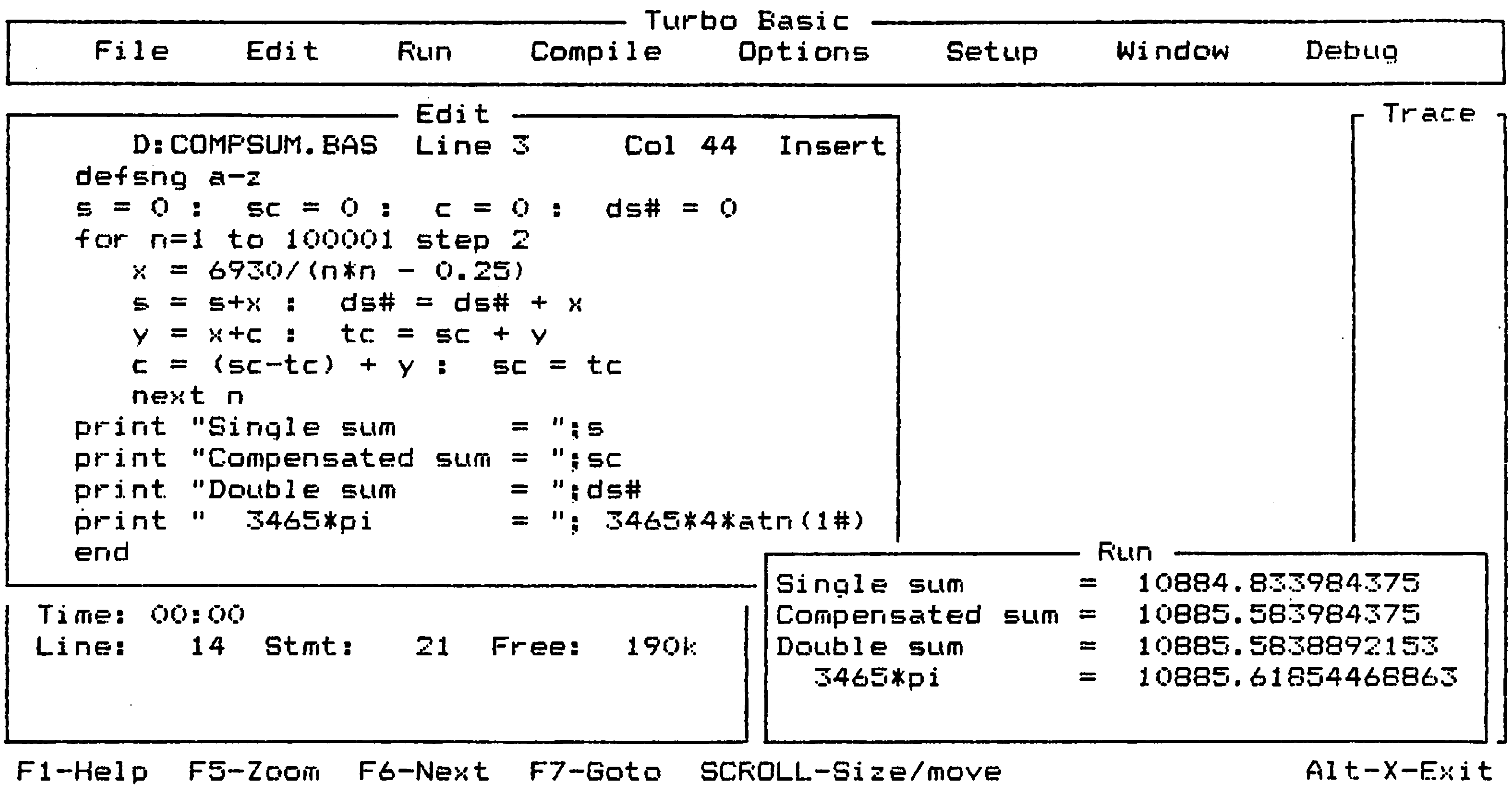
COMPENSATED PROGRAM:

(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.



```
Edit
   D:COMPSUM.BAS  Line 3     Col 44   Insert
defsng a-z
s = 0 :   sc = 0 :   c = 0 :   ds# = 0
for n=1 to 100001 step 2
   x = 6930/(n*n - 0.25)
   s = s+x :  ds# = ds# + x
   y = x+c :  tc = sc + y
   c = (sc-tc) + y :  sc = tc
   next n
print "Single sum      = ";s
print "Compensated sum = ";sc
print "Double sum      = ";ds#
print "  3465*pi       = "; 3465*4*atn(1#)
end
```

```
Time: 00:00
Line:   14  Stmt:   21  Free:  190k
```

```
Run
Single sum      =  10884.833984375
Compensated sum =  10885.583984375
Double sum      =  10885.5838892153
  3465*pi       =  10885.61854468863
```

S: Ordinary SUMMATION

SC: COMPENSATED SUMMATION

DS#: SUM using DOUBLE-PRECISION ADDs.

BUT

Trouble : - - -

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

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

<u>EXAMPLE</u> to 5 sig. dec.

$$S_{j-1} = 0.99998$$
$$Y_j = 0.99998$$
$$S_{j-1} + Y_j = T_j = 1.99996 \rightarrow 2.0000$$
$$S_{j-1} = 0.99998$$
$$(S_{j-1} - T_j) = -1.0000\underline{\underline{2}} \rightarrow -1.0000$$
$$C_j = -0.0000\underline{\underline{2}}$$

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

FAILURE ?

(This CAN'T HAPPEN in BINARY, or <u>CHOPPED</u> Hexadecimal)

<u>CURE</u> : See papers by S. Linnainmaa in BIT , 1980 ... . | ?

NOT REALLY NECESSARY !