Computer System Support for Scientific and Engineering Computation

Lecture 6a - May 19, 1988 (notes revised June 20, 1988)

1 Floating Point Problems

Two problems follow that illustrate the kinds of puzzles that afflict floating point users. The first puzzle is the difference between the results obtained from a simple Fortran program run on two models of the DEC VAX family; we still don't know why the results are different. The second puzzle is a challenge to Fortran and Pascal (but perhaps not C) programmers.

1.1 Unexpected Results

Consider the program shown in Figure 6a-1 and the results obtained when executed on two machines running the same version of the operating system.

It is tempting to try various hypotheses which might explain the different results. For example, suppose the constant 0.01 was being computed in single precision. Similarly, suppose that the compiler neglected the READ*8 declaration, or failed to sign extend the constant to double precision. Those hypotheses must be discarded as possibilities, since they are inconsistent with the results.

Another possibility is that the compilers might not be generating the same code because of differences in the architectures. This explanation is quite plausible since, on the 8650, for example, some of the instructions are not implemented in hardware.

1.2 Understanding underflow and overflow

The following problem was assigned in the previous lecture. The formal solution is in Appendix A, and a neater way to compute both p and q will be discussed in the next lecture.

Exercise: Exhibit a program that starts from any three given positive numbers x, y, z and computes $p := x \cdot y \cdot z$ in some order that avoids undeserved over/underflow. Do likewise for $q := x \cdot y / z$.

To compute p: Sort x, y, z so that, say,

$$x \geq y \geq z > 0,$$

and then compute

$$x = (x \cdot y) \cdot z.$$

```
Welcome to Computer Center VAX 8650 VAX/VMS 4.7

$ type precis.for
REAL*8 X,Y
        X=0.
        Y=0.01
        DO 10 I=1,30000
        X=X+Y
10      CONTINUE
        WRITE(1,20)X
20      FORMAT(1X,'X= ',D30.20)
        END

$ run precis
$ type for0001.dat
X=      0.30000000000000888178D+03


Welcome to Computer Center VAX 11/780 VAX/VMS 4.7

$ run precis
$ type for001.dat
X=      0.29999963378907138178D+03
```

To compute q: We could sort x, y, $1/z$, except that $1/z$ might give overflow or underflow.
Discussion: The solution to this exercise illustrates a problem which is not widely recognized. The problem is that given a relatively short program, the proof can be quite long. In fact, we might ask "When the proof is longer than the program, why trust the proof?"

The **program for p** could be implemented in a single line, except on machines with gradual underflow. The proof is provided, and is lengthy compared to the size of the program. As for the **programs for q**, the first program is quite short, and works on machines which allow branching on overflow and underflow. The second program, however, must handle several cases. The proof of that program is quite long and tedious, even though what we are computing is trivial.

There are many important math libraries, such as IMSL, NAG, and EISPACK, which have been developed at great cost, much of which has been borne by the public. From the examples above, one can see that the expense of proving the validity of this code is enormous. The lesson here might simply be "Have nothing to do with floating point". The return on investment, at least from a management perspective, is quite small. For others, however, floating-point correctness is becoming more and more important. For, as other styles of arithmetic are introduced, we must try to avoid invalidating the current proofs of correctness. This concern is becoming increasingly important for operating systems designers, who must provide concurrency, multiple tasks, parallel computations, etc. Even compiler writers, who for the most part take the attitude that the compiler defines the language, must be more aware of floating point issues.

2 The Classical Model of Roundoff

The previous chapter introducted the notion of rounding and discussed several schemes used in rounding midway cases. For those cases, it was shown that the relative error introduced is at most one-half of a unit in the last place (*ulp*). Specifically, $ulp(x) = \beta^{e+1-p}$ when $\beta^e < x < \beta^{e+1}$. Thus, given the relative precision of x, $\frac{1ulp(x)}{|x|}$, we can say that a certain error will be no bigger than $1/2ulp(x)$, where *ulp* is bounded by the inequality

$$\beta^{-p} \le \frac{1ulp(x)}{|x|} \le \beta^{1-p}.$$

The classical model of roundoff was first clearly explained by J.H. Wilkinson. The idea is that if you round an arbitrary real number, X, to the nearest floating point number, x, with p significant digits of radix β, then

$$|x - X| \le \frac{1}{2}ulp(X),$$

ignoring underflow and overflow for now. Thus, the rounded value x is related to the true value, X; $x = X(1 \pm \xi)$, where the relative error ξ is $\xi \le \epsilon := \beta^{1-p}/2$. For example, consider rounding π to an approximation pi of 4 significant digits ($p = 4$, $\beta = 10$) :

$$\begin{aligned}
\pi &= 3.14159265\ldots \\
pi &= 3.142 \\
&= \pi + 0.00040735\ldots \\
&= \pi \cdot (1 + \frac{0.00040735\ldots}{\pi}) \\
pi &= \pi \cdot (1 + 0.0001297\ldots)
\end{aligned}$$

Using the model,

$$x = X \cdot (1+\xi),$$

we can conclude that

$$\xi = 1.297 \times 10^{-4} < \varepsilon \equiv \frac{10^{1-4}}{2} = 5 \times 10^{-4}.$$

This model can be applied more widely than to correctly rounded arithmetic. If you perform any numeric operation, the results will have to be rounded. Thus, the model can be stated as:

$$\begin{aligned} \text{Program statement: } X &= Y \odot Z \\ \text{Actually computed: } x &= (y \odot z)\text{rounded correctly to } p \text{ significant digits} \\ &= (y \odot z) \cdot (1+\xi) \text{ where } |\xi| \le 1/2\beta^{1-p} \end{aligned}$$

This model can often be applied to arithmetic that is not correctly rounded by increasing ε. For example, the IBM 370, with $\beta = 16$, and $p = 6, 14$, or 28, depending on whether you are using single, double, or extended precision, uses

$$x = (y \odot z) \cdot (1-\xi).$$

Most of the operations - division, subtraction and multiplication - chop the result; and the relative error, ξ, is provably less than one $ulp(x)$. On the IBM, as well as any others that carry a "guard digit" for subtraction, the usual model is adequate. Thus, the relative error for subtraction can be expressed as $-\beta^{-p} < \xi < \beta^{1-p}$.

A slightly different model works for machines that lack a guard digit for subtraction:

$$\begin{aligned} \text{Program statement : } X &= Y - Z \\ \text{Actually computed : } x &= y \cdot (1+\zeta) - z \cdot (1+\varsigma) \\ &\quad \text{where} |\zeta| < \beta^{1-p}, |\varsigma| < \beta^{1-p}, \zeta\varsigma = 0 \end{aligned}$$

2.1 What good is a guard digit?

A guard digit enables you to calculate an exact result for $v - u$. More specifically stated:

Theorem 1 *If u and v are floating-point numbers in the same conventional format, and if*

$$\frac{1}{2} \le u/v \le 2,$$

then $v - u$ is representable exactly in that format. (Unless it underflows, which IEEE 754/854 can't.).

To construct a proof, suppose

$$0 < u \le v \le 2u.$$

(The upper bound contrains the difference so that the difference in the exponents can't be more than 1.) It follows that

$$0 \le v - u \le u.$$

Without a guard digit, the value $v - u$, that could be represented exactly, will not be computed exactly in some cases.

2.2 Assignment

Consider the triangle, with sides a, b, and c, where necessarily:

$$\begin{aligned} a+b &\geq c &\geq 0 \\ b+c &\geq a &\geq 0 \\ c+a &\geq b &\geq 0 \end{aligned}$$

Or else the "triangle" is not really a triangle.

Its area can be calculated using a formula attributed to Heron of Alexandria (who is thought to have lived sometime between 200BC and 200AD).

$$Area = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)},$$

where $s = (a+b+c)/2$.

Unfortunately, that formula is numerically unstable when the triangle is needle shaped, and the area approaches zero.

Revise Heron's formula (rearrange the order of operations) to give a result correct to within a few ulp's (presuming no underflow/overflow); you may assume a guard digit is carried for subtraction. What happens if there's no guard digit?

Appendix A

EXERCISE :

Exhibit a program that starts from any three given floating-point numbers x, y and z, and computes $p := x \cdot y \cdot z$ in some order that avoids undeserved over/underflow. Do likewise for $q := x \cdot y/z$.

SOLUTIONS: The proofs that these programs work correctly depend upon the properties of three *Environmental Constants* associated with the floating-point formats in which x, y, z, p and q are represented, regardless of whether those constants appear in the programs. The *Overflow threshold* Ω is the biggest finite number in that format; the *Underflow threshold* η is the smallest *normalized* positive number. The magnitudes of x, y and z are presumed to lie between Ω and $\varepsilon\eta$ inclusive where $\varepsilon\eta$ is the smallest *nonzero* magnitude and may be far tinier than η if underflow is *gradual*; on machines that underflow abruptly to zero $\varepsilon\eta = \eta$ except for CDC Cyber 17x's. $\varepsilon\eta = 2\eta$ for these Cybers to cope with "partially underflowed" numbers between η and $\varepsilon\eta$ that behave normally in *add*, *subtract* and *compare* but behave like zero in *multiply* and *divide*. Little is presumed about the product $\eta\Omega$, which lies very far from 1 on some machines.

An obvious program to compute p and q would first obtain their magnitudes using logarithms; $|p| = \exp(\ln|x| + \ln|y| + \ln|z|)$ and $|q| = \exp(\ln|x| + \ln|y| - \ln|z|)$. But these formulas lose accuracy badly when the data are very big or very small; the loss is caused by rounding each logarithm to working precision, and can be observed by comparing the computed values of $\exp(\ln|x|)$ and $|x|$ when x lies near Ω or η. And computing logarithms and exponentials wastes time. Our programs waste neither accuracy nor time.

Both programs start by sorting $|x|, |y|$ and $|z|$ so that $|x| \le |y| \le |z|$ and continue thus:

Program for p :
Compute $x \cdot z$ first and then $p := (x \cdot z) \cdot y$, except on a machine with gradual underflow; on such a machine, if $(x \cdot z)$ underflows, recompute $p := (z \cdot y) \cdot x$.

Proof that p is correct.
If $x \cdot z$ overflowed, then $1 < |x| \le |y| \le \Omega < |x \cdot z| < |(x \cdot z) \cdot y|$ so p deserves to overflow too (except perhaps on a CRAY, which can overflow in certain cases when a product lies between $\Omega/2$ and Ω; but that is too perverse to consider here). Similarly if $x \cdot z$ underflowed on a machine that underflows abruptly to zero, then

$$1 > |z| \ge |y| \ge |x| \ge \eta > |x \cdot z| > |(x \cdot z) \cdot y|$$

so p must underflow too. On a machine that underflows gradually, conformity with IEEE standards 754/854 requires also the ability to detect underflow, and this should be exploited if any of the data can be subnormal (i.e., between $\varepsilon\eta$ and η in magnitude). Then $x \cdot z$ underflows only when $1/\varepsilon \ge |z| \ge |y| \ge |x| \ge \varepsilon\eta$ and $\eta > |x \cdot z|$; since $\Omega > 1/\varepsilon^2$ on those machines, $\Omega > z \cdot y$ so $z \cdot y$ cannot overflow, and if it underflows too, then either $|z| > 1$ and then $|x \cdot y \cdot z| = |(x \cdot z)(z \cdot y)/z| < \eta^2/|z| < \eta$, or else $|z| \le 1$ and then $|x \cdot y \cdot z| < |x|\,\eta \le \eta$, and p deserves to underflow either way.

Programs for q :
If we could treat q as a product $x \cdot y \cdot (1/z)$, we could compute it safely using the program for p; but the risk that $1/z$ may over/underflow precludes that option. A safe and simple program works on machines that allow programs to branch on over/underflow:

```
First swap $x$ and $y$ if necessary to establish $|x| \le |y|$;
  next compute $p := x \cdot y$; subsequently
    if ($p$ overflowed and $|z| > 1$) then $q := (y/z) \cdot x$
      else if ($p$ underflowed and $|z| < 1$) then $q := (((x/\varepsilon)/z) \cdot y) \cdot \varepsilon$
        else $q := p/z$. (For Cybers use $\varepsilon = 1$ here, not 2.)
```

The validity of this program is easy to establish provided we may presume that $\sqrt{(\eta)}/\varepsilon^2 < \eta\Omega < \sqrt{\Omega}$, as appears to be true for all machines I know. But the ability to test for over/underflow and continue is not so common; what if over/underflow is silent? In the absence of a (portable) way to branch on over/underflow, we must produce a spaghetti-like code with branches that preclude spurious over/underflows. Such a program follows.

Two constants are needed. One is λ, the smallest power of the machine's radix no smaller than $\max\{1, 1/(\varepsilon\eta\Omega)\}$. The other is μ, the biggest power of the radix not exceeding $\min\{1, 1/(\eta\Omega)\}$. Multiplication by λ or μ is exact, so it cannot cause underflow on a machine that conforms to IEEE 754/854.

First sort $|x|, |y|$ and $|z|$, keeping track of z. This reduces the situation to one of three cases, depending upon whether $|z|$ is minimal, maximal, or neither:

```
In case $|z|$ is > minimal, say $|z| \le |x| \le |y|$, test $|y|$;
   if $|y| > 1$ then $q := (x/z) \cdot y$
   else $q := (x/(\lambda z)) \cdot (\lambda y)$.
In case $|z|$ is maximal, say $|z| \le |y| \le |x|$, test $|x|$;
   if $|x| < 1$ then $q := (y/z) \cdot x$
   else $q := (y/(\mu z)) \cdot (\mu x)$.
In case $|z|$ is neither, say $|x| \le |z| \le |y|$, test both;
   if $|x| > 1$ then $q := (y/z) \cdot x$
   else if $|y| < 1$ then $q := (x/z) \cdot y$
     else $q := (x \cdot y)/z$.
```

The proof that this program is correct is a tedious exercise in elementary inequalities, and is left to the reader.

The "CLASSICAL" model of Roundoff

see - J.H. Wilkinson (1961) "Rounding Errors in Algebraic Processes" Prentice-Hall / HMSO.
- HP-15C Advanced Applications Handbook (1982

If an arbitrary real no. X is rounded to the nearest Floating-Point number x then, to p sig. digits of radix β,

$$|x - X| \le \tfrac{1}{2}\,\mathrm{ulp}(X)$$

where $\beta^{-p} \le \mathrm{ulp}(X)/|X| \le \beta^{1-p}$

(Ignore the possibility of over/underflow for now.)

"true" value

$\therefore$ rounded value $x = X\cdot(1 \pm \xi)$ relative error

where $|\xi| \le \varepsilon := \beta^{1-p}/2$

e.g. 4 sig. dec. ($p=4$, $\beta=10$) : Round π to pi ...

True $\pi = 3.141\,59\,265$
pi $= 3.142$
$= \pi + 0.00041735\ldots = \pi\left(1 + \dfrac{0.00041735}{\pi}\right)$

Rounded pi $=$ (true π)$\cdot$(1 + 0.0001328...)
$x = X \cdot (1 + \xi)$

$\xi = 1.328\cdot10^{-4} < \varepsilon = \tfrac{1}{2}\cdot10^{-3}$.

Kahan
19 ~~[illegible]~~ May

" $X := Y \circledast Z$ "

x = $y \circledast z$ rounded correct'y to p sig. digits (β)

= $(y \circledast z)\cdot(1+\xi)$ where $|\xi| \le \frac{1}{2}\beta^{1-p}$.

This model is applicable in many cases that are NOT correctly rounded.

			SINGLE	DOUBLE	EXT'D.
e.g.	IBM 370,	$\beta = 16$,	$p = 6$	or 14	or 28

$x = (y \circledast z)\cdot(1 - \xi)$, $0 \le \xi < \beta^{1-p}$

$\uparrow$ CHOPPED if $\circledast \in \{+, \times, \div\}$

or $x = (y - z)\cdot(1 - \xi)$, $-\beta^{-p} \le \xi < \beta^{1-p}$.

$\uparrow$ Magnitude subtraction is NOT chopped.

A similar model works for machines that lack a GUARD DIGIT for SUBTRACTION:

$X := Y - Z$

$x = y\cdot(1+\eta) - z\cdot(1+\zeta)$

where $|\eta| < \beta^{1-p}$, $|\zeta| < \beta^{1-p}$, $\eta\zeta = 0$.

How come?

$$y = 1.000\ldots \qquad 1.000 \qquad 1.0009$$

$$-z = -0.9999 \quad \text{vs} \quad -0.99\cancel{9}9 \qquad -0.9999$$

$$0.0001 \qquad 0.001 \qquad 0.001$$

$$\swarrow \qquad \swarrow \qquad \swarrow$$

$$1.000 \times 10^{-4} \qquad 1.000 \times 10^{-3} \qquad 1.000 \times 10^{-3}$$

WHAT GOOD IS A GUARD DIGIT?

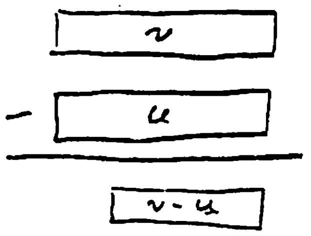
Theorem: If u and v are floating-point numbers in the same conventional format, and if

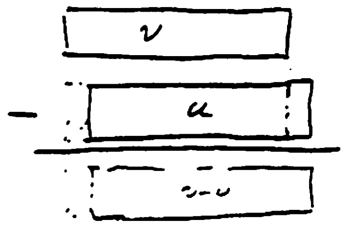
$$\tfrac{1}{2} \le u/v \le 2 ,$$

then $v-u$ is representable EXACTLY in that format (UNLESS IT UNDERFLOWS! IEEE 754/854 can't.).

Proof: Suppose $0 < u \le v \le 2u$ (or swap)

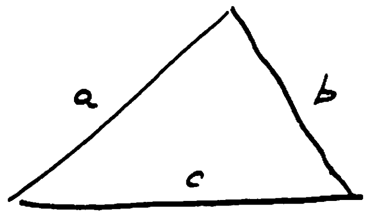
$\therefore \quad 0 \le v-u \le u$





WITHOUT A GUARD DIGIT, THE VALUE $v-u$ THAT COULD BE REPRESENTED EXACTLY WILL NOT BE COMPUTED EXACTLY IN SOME CASES.

ASSIGNMENT



$a+b \geq c$
$b+c \geq a$
$c+a \geq b$

$$\text{Area} = \sqrt{s\cdot(s-a)\cdot(s-b)\cdot(s-c)}$$

where $s := (a+b+c)/2$.

This formula, due to Heron of Alexandria (200 BC – 200 AD ?) is NUMERICALLY UNSTABLE.

REVISE HERON'S FORMULA TO GIVE A RESULT CORRECT TO WITHIN A FEW ULPS (presuming no over/underflow) ONLY IF A GUARD DIGIT IS CARRIED FOR SUBTRACTION.

```
Welcome to Cyanamids ARD Computer Center VAX 8650 VAX/VMS 4.7




$ type precis.for
        REAL*8 X,Y
        X=0.
        Y=0.01
        DO 10 I=1,30000
        X=X+Y
10      CONTINUE
        WRITE(1,20)X
20      FORMAT(1X,'X= ',D30.20)
        END

$ run precis
$ type for001.dat
X=       0.30000000000000888178D+03




Welcome to Cyanamids ARD Computer Center VAX 11/780 VAX/VMS 4.7


$ run precis
$ type for001.dat
X=       0.29999963378907138178D+03
$
```

PROBLEM for CS 179 : by Prof. W. Kahan

Exhibit a program that starts from any three given floating-point numbers x, y and z, and computes $p := x \cdot y \cdot z$ in some order that avoids undeserved over/underflow. Do likewise for $q := x \cdot y/z$.

SOLUTIONS: The proofs that these programs work correctly depend upon the properties of three *Environmental Constants* associated with the floating-point formats in which x, y, z, p and q are represented, regardless of whether those constants appear in the programs. The *Overflow threshold* Ω is the biggest finite number in that format; the *Underflow threshold* η is the smallest *normalized* positive number. The magnitudes of x, y and z are presumed to lie between Ω and $\varepsilon\eta$ inclusive where $\varepsilon\eta$ is the smallest *nonzero* magnitude and may be far tinier than η if underflow is *gradual*; on machines that underflow abruptly to zero $\varepsilon\eta = \eta$ except for CDC Cyber 17x's. $\varepsilon\eta = 2\eta$ for these Cybers to cope with "partially underflowed" numbers between η and $\varepsilon\eta$ that behave normally in *add*, *subtract* and *compare* but behave like zero in *multiply* and *divide*. Little is presumed about the product $\eta\Omega$, which lies very far from 1 on some machines.

An obvious program to compute p and q would first obtain their magnitudes using logarithms; $|p| = \exp(\ln|x| + \ln|y| + \ln|z|)$ and $|q| = \exp(\ln|x| + \ln|y| - \ln|z|)$. But these formulas lose accuracy badly when the data are very big or very small; the loss is caused by rounding each logarithm to working precision, and can be observed by comparing the computed values of $\exp(\ln|x|)$ and $|x|$ when it lies near Ω or η . And computing logarithms and exponentials wastes time. Our programs waste neither accuracy nor time.

Both programs start by Sorting $|x|$, $|y|$ and $|z|$ and continue thus:

Program for p :
Assume now that sorted $|x| \le |y| \le |z|$. Compute $x \cdot z$ first and then $p := (x \cdot z) \cdot y$ except on a machine with gradual underflow; on such a machine if $(x \cdot z)$ underflows recompute $p := (z \cdot y) \cdot x$.

Proof that p is correct.
If $x \cdot z$ overflowed, then $1 < |x| \le |y| \le \Omega < |x \cdot z| < |(x \cdot z) \cdot y|$ so p deserves to overflow too (except perhaps on a CRAY, which can overflow in certain cases when a product lies between $\Omega/2$ and Ω; but that is too perverse to consider here). Similarly if $x \cdot z$ underflowed on a machine that underflows abruptly to zero, then
$$1 > |z| \ge |y| \ge |x| \ge \eta > |x \cdot z| > |(x \cdot z) \cdot y|$$
so p must underflow too. On a machine that underflows gradually conformity with IEEE standards 754/854 requires also the ability to detect underflow, and this should be exploited if any of the data can be subnormal (i.e., between $\varepsilon\eta$ and η in magnitude). Then $x \cdot z$ underflows only when $1/\varepsilon \ge |z| \ge |y| \ge |x| \ge \varepsilon\eta$ and $\eta > |x \cdot z|$; since $\Omega > 1/\varepsilon^2$ on those machines, $\Omega > z \cdot y$ so $z \cdot y$

cannot overflow and if it underflows too then either $|z| > 1$ and then $|x \cdot y \cdot z| = |(x \cdot z)(z \cdot y)/z| < \eta^2/|z| < \eta$, or else $|z| \le 1$ and then $|x \cdot y \cdot z| < |x|\eta \le \eta$, and p deserves to underflow either way.

Programs for q :
If we could treat q as a product $x \cdot y \cdot (1/z)$, we could compute it safely using the program for p ; but the risk that $1/z$ may over/underflow precludes that option. A safe and simple program works on machines that allow programs to branch on over/underflow:

```
 First swap  $x$ and $y$  if necessary to establish  $|x| \le |y|$ ;
  next compute  $p := x \cdot y$ ;  subsequently
   if  ( $p$ overflowed  and  $|z| > 1$ )  then  $q := (y/z) \cdot x$
    else if  ( $p$ underflowed  and  $|z| < 1$ )  then
                                                  $q := (((x/\varepsilon)/z) \cdot y) \cdot \varepsilon$
     else  $q := p/z$ .    ( For Cybers use  $\varepsilon = 1$  here,  not 2 .)
```

The validity of this program is easy to establish provided we may presume that $\sqrt{(\eta)}/\varepsilon^2 < \eta\Omega < \sqrt{\Omega}$, as appears to be true for all machines I know. But the ability to test for over/underflow and continue is not so common; what if over/underflow is silent? In the absence of a (portable) way to branch on over/underflow, we must produce a spaghetti-like code with branches that preclude spurious over/underflows. Such a program follows.

Two constants are needed. One is λ , the smallest power of the machine's radix no smaller than $\max\{1,\ 1/(\varepsilon\eta\Omega)\}$. The other is μ , the biggest power of the radix not exceeding $\min\{1,\ 1/(\eta\Omega)\}$. Multiplication by λ or μ is exact, so it cannot cause underflow on a machine that conforms to IEEE 754/854.

First sort $|x|$, $|y|$ and $|z|$, keeping track of z . This reduces the situation to one of three cases, depending upon whether $|z|$ is minimal, maximal, or neither:

```
In case  $|z|$  is minimal,  say  $|z| \le |x| \le |y|$ ,  test  $|y|$ ;
  if  $|y| > 1$  then  $q := (x/z) \cdot y$
                  else  $q := (x/(\lambda z)) \cdot (\lambda y)$ .
In case  $|z|$  is maximal,  say  $|z| \ge |y| \ge |x|$ ,  test  $|x|$ ;
  if  $|x| < 1$  then  $q := (y/z) \cdot x$
                  else  $q := (y/(\mu z)) \cdot (\mu x)$ .
In case  $|z|$  is neither,  say  $|x| \le |z| \le |y|$ ,  test both;
  if  $|x| > 1$  then  $q := (y/z) \cdot x$
   else if  $|y| < 1$  then  $q := (x/z) \cdot y$
    else  $q := (x \cdot y)/z$ .
```

The proof that this program is correct is a tedious exercise in elementary inequalities, and is left to the reader.

Computer System Support for Scientific and Engineering Computation

Lecture 6b - May 19, 1988 (notes revised June 24, 1988)

1 Horner's Recurrence: Applying Wilkinson's Round-off Error Model

How do we use Wilkinson's model of round-off error?[1] Given the coefficients a_j of the polynomial $A(x) = \sum_0^N a_j x^{N-j} = a_0x^N + a_1x^{N-1} + \ldots + a_{N-1}x + a_N$ and a numerical value z, we can compute both $p := A(z)$ and the derivative $q := A'(z)$ by means of Horner's recurrence. Such problems arise often enough: to approximate interest rate payments, or to calculate sine or cosine. Also, many equations are solved by first casting them as polynomial equations, then solving the polynomials.

1.1 Horner's Recurrence

Horner's Recurrence is defined:

$$A(x) = (\ldots(((a_0x + a_1)x + a_2)x + a_3)x + \ldots + a_{N-1})x + a_N.$$

Horner's Recurrence is written this way, in a program, only for short polynomials. For longer polynomials, it is expressed in a loop; such a loop even allows simultaneous computation of both the polynomial and its derivative.

```
$q := 0$;
$p := a_0$;
for $j := 1$ to $N$ do
    { $q := z * q + p$;
      $p := z * p + a_j$;}
```

... now $p = A(z)$ and $q = A'(z)$.

This feature is useful, for example, in Newton's method for solving an equation, which uses both the polynomial and its derivative:

[1] Refer to "Roundoff in Polynomial Evaluation", W. Kahan, Class Notes, October 1986

$z^{n+1} = z^n - \frac{A(z^n)}{A'(z^n)}$

It is unnecessary to calculate the derivative separately; instead, augment the recurrence as above, where $p = A(z)$ and $q = A'(z)$, except for rounding errors.

More formally, we can substitute for a_j in the above equation, and so establish that for all x we have $A(x) = p_N + (x-z)(q_{N-1} + (x-z)\sum_o^{N-2} q_j x^{N-2-j})$. It soon follows that the final values of p and q are $p_N = A(z)$ and $q_{N-1} = A'(z)$, respectively.

1.2 Round-off error in Horner's Recurrence

Rounding errors that occur in these sorts of recurrences can be attacked remarkably better than most people think. By using Wilkinson's model of round-off error, we introduce algebraic relationships that relate the things we actually compute to the things we wanted to compute, or to the data. The computed values and the desired values are related in such a way that, despite the fact that the values of the round-off errors - Greek letters - are unknown, it is not hard td propagate the inequalities and then obtain bounds for the final error, or then perform backward error analysis, the technique made famous by Wilkinson.

To get a handle on rounding error, it is important to understand the difference between the program variable p and the mathematical value p_j. The program statements

```
$p := a_0$;
for $j := 1$ to $N$ do $p := z * p + a_j$;
```

are analyzed as

```
$p_0 = a_0$;
for $j = 1$ to $N$ do $p_j = z \cdot p_{j-1} + a_j$; (except for round-off)
```

The value p_j is the name of the contents of the register whose name is p when j is about to be incremented. The value of the program variable p is properly analyzed in terms of p_j, although p_j is not explicitly defined in the program. The point is that we should be sensitive to where and how the value of a program variable changes, for then we can manageably express round-off errors:

```
$p_0 = a_j$;
for $j = 1$ to $N$ do $p_j = \frac{z \cdot p_{j-1} \cdot (1+\zeta_{j-1}) + a_j}{1-\pi_j}$;
```

Each arithmetic operation introduces a round-off error; ζ_j and π_j represent the total effect of all errors introduced during the j^{th} iteration of the loop.

All we know about an individual rounding error is that its magnitude is bounded by some value ϵ, which depends, among other things, on the machine's radix, arithmetic, and rounding. (A machine that does not use guard digits in arithmetic operations introduces yet another "Greek letter" into every arithmetic calculation, but otherwise does not disturb the computation. In particular, in evaluating polynomials, machines that operate without guard digits do not introduce serious disasters, just added complication in the error analysis.)

To take account of all rounding errors in the loop:

```
$q := 0$;
$p := a_0$;
for $j := 1$ to $N$ do
        { $q := z * q + p$;
          $p := z * p + a_j$; }
```

we work out the *perturbed recurrence:*

```
$p_{-1} = q_{-1} = 0$;
$p_0 = a_0$;
$\pi_0 = \kappa_0 = 0$;
for $j = 1$ to $N$ do { $q_{j-1} = \frac{z \cdot q_{j-2}(1+\eta_{j-2}+p_{j-1})}{(1+\kappa_{j-1})}$;
                        $p_j = \frac{(z \cdot p_{j-1}(1+\zeta_{j-1})+a_j)}{(1+\pi_j)}$; }
```

... and $\eta_{N-1} = \zeta_N = 0$.

1.3 Analyzing round-off error

What do we do about these Greek letters? There are two strategies: backward error analysis, and running error analysis. In the course of computation, we find that we have calculated *exactly* a slightly different polynomial. This is the beauty of backward error analysis: we infer our result is *no worse* than if we miraculously accomplished perfect computation on the same problem except that somebody we do not know came along and changed a_0 by something on the order of n units in the last place, a_1 by something a bit less than that, and so on. This explanation is satisfactory especially when the coefficients are not known exactly. If the coefficients were calculated by means that introduced errors bigger than n units in the last place, then the error in calculating the polynomial may not make things appreciably worse. This is the sort of argument Wilkinson used to point out that, in many situations, this much error is quite satisfactory, despite the fact that the computed function value and zero may be utterly wrong (see Figure 1). If the zero is utterly wrong, at least we know that it is the correct answer for a polynomial with only slightly different coefficients. Theorem: Only if the given polynomial is very close to a polynomial with double, triple or higher order zeros can the zeros change drastically.

What else can we do about perturbation? How wrong is the perturbed result? Rather than saying the computed zero is "right" for some polynomial whose coefficients we do not know, how can we estimate how wrong the zero of the polynomial is? If we do not know how wrong a computation may be, then there is some chance the result is entirely wrong - in which case, why bother to perform the calculation in the first place?

Our goal is to calculate the uncertainty of the zero shown in Figure 2.

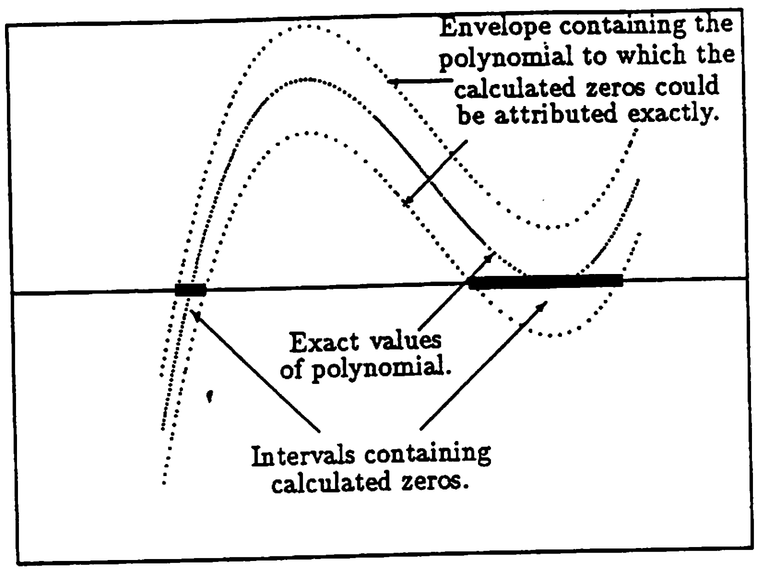

Envelope containing the polynomial to which the calculated zeros could be attributed exactly.
Exact values of polynomial.
Intervals containing calculated zeros.

Figure 1: Calculated zeros may be exactly those of a nearby polynomial.

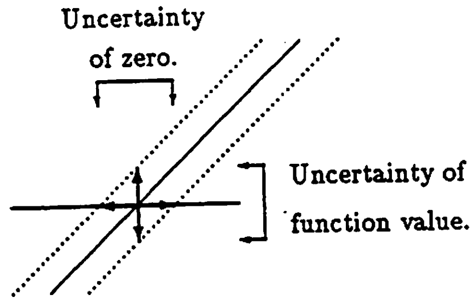


Figure 2: Uncertainty of computed zero due to roundoff in function value.

1.3.1 When to quit?

We calculate error bounds in order to answer one of life's big questions : how do we know when to quit? At some point in an iterative process, dithering starts. That is, suppose we are solving some equation and we do not have a nice, neat closed formula for the solution. Most books supply an arbitrary stopping point: quit when the difference between two computed iterates is less than some threshold. But how is the threshold chosen? If the threshold is too large, the result may be greatly in error; if the threshold is too small for a given equation, rounding error may force the computed results always to lie outside the ribbon of acceptable results, and so the program chugs on forever (see Figure 3).

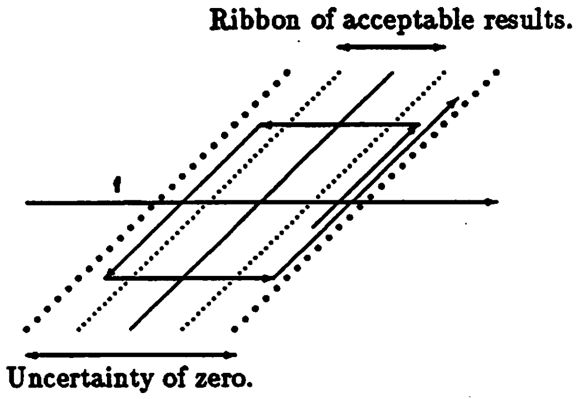


Figure 3: Computed values dither.

How does a programmer deal with this problem? He or she often passes it to the user of the program. However, users are often less well equipped than the programmer to deal with this issue.

Another approach is to terminate the computation when it starts to dither. On the average, this yields a better answer *if* errors are random. But what if the errors are not random? Such examples are easily constructed, especially from the exponential function. A very bad approximation from a numerical point of view may be physically plausible; thus, it is easy to terminate a computation much too soon.

For example, suppose we want to compute the zeros of $f(x) = e^{x^2} - 1$, and we choose Newton's method. Then, $x_{n+1} = x_n - \frac{1-e^{-x^2}}{2\cdot x_n} \approx x_n - \frac{1}{2\cdot x_n}$. For large values of x, we have $x_{n+1} \approx x_n$. If x_{n+1} is indistinguishable from x_n, which happens when the stopping criterion is less than their difference, the computation may terminate, even though the current "best" estimate of the zero is still much further than the stopping criterion from the true value of the zero (see Figure 4).

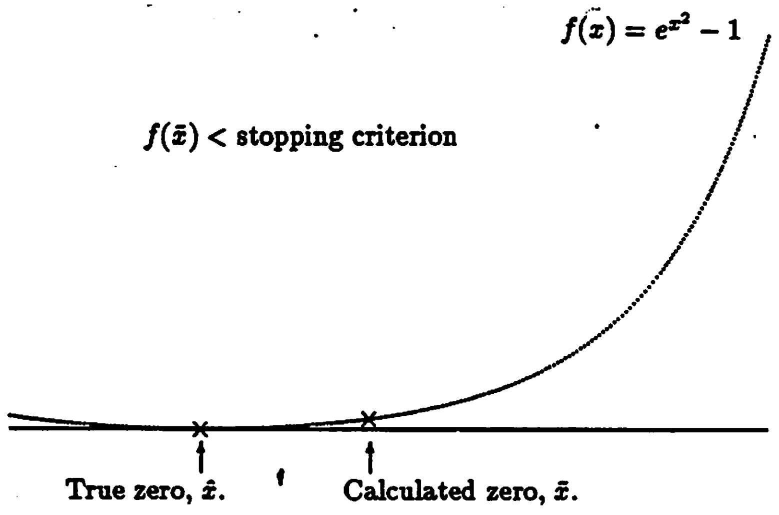


Figure 4: Program stops, but $\bar{x}$ could be utterly wrong.

1.3.2 Running Error Analysis

A running error analysis provides a robust stopping criterion. The following excerpt from "Roundoff in Polynomial Evaluation" demonstrates how to apply this mode of thinking to account for the rounding errors introduced by all the arithmetic operations:

> In our example, p_N is an estimate for $A(z)$, the desired quantity. To estimate the difference, and thus to obtain an upper bound on the error, from the perturbed recurrence substitute for for a_j in the definition of $A(x)$. Then, for all x,
>
> $$A(x) = p_N + \textstyle\sum_0^N(\pi_j x - \zeta_j z)p_j x^{N-1-j}$$
> $$+(x-z)(q_{N-1} + \textstyle\sum_0^{N-1}(\kappa_j x - \eta_j z)q_j x^{N-2-j} + (x-z)\sum_0^{N-2} q_j x^{N-2-j}).$$
>
> From this it follows that $A(z) = p_N + \sum_0^N(\pi_j - \zeta_j)p_j z^{N-j}$
>
> and
>
> $$A'(z) = q_{N-1} + \textstyle\sum_0^{N-1}((\kappa_j - \eta_j)q_j + ((N-j)\pi_j - (N-1-j)\zeta_j)p_j)z^{N-1-j}$$
>
> Since no rounding error appears more than once in each of these formulas, the nonzero Greek letters can be replaced by $\pm\epsilon$ to get best-possible bounds for the accumulated effect of roundoff:

$$\frac{|A(z)-p_N|}{\epsilon} < |\,p_N\,| + 2\sum_1^{N-1} |\,p_j\,|\, r^{N-j} + |\,p_0\,|\, r^N \qquad \text{where } r := |\,z\,|;$$

$$\frac{|A'(z)-q_{N-1}|}{\epsilon} < |\,q_{N-1}\,| + 2\sum_1^{N-2} |\,q_j\,|\, r^{N-1-j} + |\,q_0\,|\, r^{N-1}$$
$$+ \sum_1^{N-1}(2N-2j-1)\,|\,p_j\,|\, r^{N-1-j} + (N-1)\,|\,p_0\,|\, r^{N-1}.$$

The right-hand side of these inequalities are polynomials in r with coefficients derived from $|\,p_j\,|$ and $|\,q_j\,|$, so they can be computed by recurrence too. To do that, here is an *augmented recurrence*:

```
$r := |\,z\,|;$
$q := 0;$
$p := a_0;$
$e := |\,p\,|; d := -\frac{e}{r};$

for $j := 1$ to $N$ do
        { $q := z \cdot q + p;$
          $d := r \cdot d + e + |\,q + q\,| - |\,p\,|;$
          $p := z \cdot p + a_j;$
          $e := r \cdot e + |\,p + p\,|;$ }
$e := e - |\,p\,|;$
$d := d - |\,q\,|;$
```

Now $\frac{|A(z)-p|}{\epsilon} < e$ and $\frac{|A'(z)-q|}{\epsilon} < d$ except for over/underflow and ignorable roundoff incurred during the calculation. Verifying that the last two inequalities do follow from the previous two is a challenging exercise in algebraic manipulation; that verification will confirm that the two sides of each inequality could approach each other arbitrarily closely in the event, albeit unlikely, that all the rounding errors had magnitudes ϵ and appropriate signs.

The final error bound is not enormously pessimistic; in fact, it is a good estimate for machines that chop figures, such as in the IBM architectures. The degree of pessimism is of order $\sqrt{n}$. In the worst case, the error estimate is only off by a factor of two. To see this, consider the average error, and the sum of variances between the roundoff error and the average error.

Such an augmented recurrence is an inexpensive way to compute the width of the ribbon. Thus, we have a thoughtful way to decide when to quit the iteration. We compute, simultaneously, an estimate of the polynomial, and some approximation to the width of the ribbon. Given this knowledge about the width of the ribbon, we can stop the iteration naturally, in places where the polynomial is being computed in a ragged fashion. [2]

[2]See Laguerre's Theorem, stated in "Roundoff in Polynomial Computation", and "A Stopping Criterion for Polynomial Root Finding," Duane Adams, Communications of the ACM, Vol, 10, No. 10, October 1967.

1.3.3 Comments on running error analysis

A good trick to use in the augmented recurrence is to compute p in extended precision, and compute e in single precision. There is some economy in that the recurrence is the same, but the constants we multiply by are smaller.

$p := zp + a_j$;
$e := |z|e + |p|$;

There is a Perverse Theorem, which states that is it possible to write a vanilla program in Fortran, C, etc., and achieve an accuracy that is limited only by the exponent range of the machine. Such techniques would be useful here, and will be discussed further when we talk about Kulisch's methods. These methods amount to an awkward way to implement extended precision.

Note that we have ignored overflow and underflow. It is possible to accommodate overflow and underflow, and this subject will arise again when we justify gradual underflow.

1.4 Conclusions

There are systematic ways to estimate round-off errors; there are good reasons to do it; the only prerequisite is a trustworthy model. Furthermore, even on machines that lack guard digits, in this particular computation, the only effect is that the actual recurrence is a bit messier to compute - there is an extra "Greek letter". Even for the Cray, this method allows us to obtain error bounds and to decide when to terminate the iteration.

Note: If you're extremely careful about computing the function value and the error bound, you may find that the function is never smaller than the error bound.

This leads us to One Final Theorem: If the stopping criterion is set to double the error bound, then eventually the computed value will come out less than the error bound.

2 A more delicate approach to error analysis

Wilkinson's model is not adequate to explain all rounding errors. Due to wobbling precision, the bound is frequently too large by a factor of the radix, β. In general, no such inequality handles error modeling exactly, since we're dealing with a discrete set. That is, there are things that happen that do not fit into the model - the model could never be used to prove anything about these events.

Recall that Cray was confronted with an angry letter: "AMOD on the Cray doesn't work! Fix it!" None of us will sell our Cray stock: we know Cray won't change AMOD, but we also know that it won't deflect the current of world affairs.

How much *can* one do, on a Cray, to make it right?

The Fortran standard defines $\text{AMOD}(x, y) := x - \lfloor x/y \rfloor \cdot y$

Claim: On a reasonable machine, if $0 < x < y$, then $\lfloor x/y \rfloor < 1.0$

Un-proof: There is no way, using Wilkinson's model alone, to prove or disprove this claim.

Reason: Choose x and y to be adjacent numbers, so that y is a power of the radix, and x is just slightly smaller. If all we know about quotients of p-significant digit numbers with radix β is that the computed value of $\frac{x}{y} = (\text{true } \frac{x}{y}) \cdot (1 \pm 3\beta^{1-p})$ (and that is all we know when we use the standard $(1 \pm \epsilon) \cdot (a \odot b)$ model), then we cannot prove that $0 < x < y \Rightarrow$(computed $\frac{x}{y}$) < 1.

For, if $y = 1$ and $x = 1 - \beta^{-p}$, then the computed value of $\frac{x}{y} = (1 - \beta^{-p}) \cdot (1 \pm 3\beta^{1-p})$. This lies in $[1 - (3\beta + 1)\beta^{-p} + \ldots, 1 + (3\beta - 1)\beta^{-p}]$. The lower bound is less than 1, but the upper bound is greater than $1 + \beta^{1-p}$.

3 Summary

We have seen that this model of error analysis is very useful in most situations, yet not universally applicable. In the next lecture, we'll see a truly delicate analysis using a universally applicable model.