Computer System Support for Scientific and Engineering Computation

Lecture 2 - May 5, 1988 (notes revised May 23, 1988)

1 More About Exceptions

At the end of lecture 1 we discussed how arithmetic exceptions can interfere with the process of searching for a suitable value in an equation solver. The problem was that a general equation solver cannot by itself know the domain of a function. We would have to tell it.

Equation solving is not the only situation in which premature termination caused by an exception can be very awkward. Searching a database presents a similar problem, since queries do not always take into account some peculiar aspect of the data organization or values.

Notice that again this involves an algorithm about which it is difficult to anticipate all contingencies, namely a searching process. Such a process must be robust in the face of "illegal" operations that might otherwise cause it to crash. How would you feel if every time you looked in the wrong drawer for your missing beige socks, you blew up? Do you think you would ever find all your socks scattered all over your house this way? Probably not.

Well, in a similar way, you could have an awfully tough time finding all the solutions of many poorly behaved equations, or all the people who are bald in a database of hair colors, unless you (or someone before you) took into account the existence of unpredictable vagaries in the equation or data. You may not find out how your program can get blown away until you look in places you shouldn't!

Floating point arithmetic is like the equation solver or a data base system in that it is a general solution to a class of programming problems. These things are all alike in another way, too. In practice, they are all incomplete, or rough, solutions. Pretty much inevitably, therefore, they are full of holes.

In general, you cannot conduct searches successfully where you do not know exactly what you are looking for, so long as the computer simply stops whenever you do something slightly invalid.

However, it is harmless to be allowed to continue a search after visiting a place that is off limits, as long as you are confident that what you seek couldn't have been there. It is sufficient therefore so to phrase your queries that an invalid query may safely be treated as false, and this usually happens automatically.

If you decide that the computer shouldn't stop, what happens when it does something plausible but invalid that should have stopped (as we discussed before)? Now you've got an answer that *is* wrong but perhaps looks right.

So you see, you have a dilemma, between getting caught without an answer when you encounter an exception, and getting stuck with an undetected bad answer when you ignore exceptions. Fortunately, a way to resolve the dilemma does exist. It is called *retrospective diagnostics,* and we will talk about it in a later lecture.

2 Examples of Aberrations

We mentioned in the last lecture that many of the computational problems introduced by floating point implementations are extremely rare. Nevertheless, they do occur from time to time, because of the ferocious rate at which computers perform calculations these days, and the large numbers of computers.

The trouble is that problems are usually presented in terms of symptoms and alleged inferences, from which we must infer underlying causes if we are to act competently.

For example, consider the hypothetical letter to a computer manufacturer in Appendix A1 and the analysis in Appendix A2. One must exercise good judgment to avoid introducing new hazards in an attempt to combat old ones.

Despite these hazards, there is a rationale that is useful in designing floating point systems. Setting forth that rationale is the main goal of these lectures. The next several sections present some more examples of these hazards.

2.1 What You See is Not Necessarily Exact

```
+------------------------ Edit ----------------------------+
|    q = 3.0/7.0                                            |
|    Print "  The value of  q = "; q                        |
|    Print "      but  3.0/7.0 = "; 3.0/7.0                 |
|    Print                                                  |
|    Print "  What You See Is Not Necessarily What You Get." |
|    End                                                    |
+-----------------------------------------------------------+
+------------------------ Run -----------------------------+
|    The value of  q =  .4285714328289032                   |
|       but  3.0/7.0 =  .4285714285714286                  |
|                                                           |
|    What You See Is Not Necessarily What You Get.          |
+---------------+-----------------------------+-------------+
```

Figure 2-1

The BASIC program in Figure 2-1 demonstrates that two different ways of using the result of 3.0/7.0 can result in two different real values when we come to print things out. The assignment $q = 3.0/7.0$ stores the value in the variable q which, for lack of an explicit declaration, has been declared *single precision.* But in the *Print* statement, the value 3.0/7.0

is evaluated in an internal format which is wider than double precision. Both expressions are then printed in double precision. So,

- the computed value of 3.0/7.0
- the double precision value sent to binary decimal conversion for printing
- the single precision value stored in q
- and the two values printed on the page

are five different values, and none of them is 3.0/7.0.

You see, then, it helps to be aware of what conversions, implicit, explicit and even hidden, a program will impose upon intermediate values, without telling you. And this can vary tremendously from machine to machine, from language to language, and even within a single implementation of a language.

There is another interesting point about the values printed out in Figure 2-1. None of the five values mentioned above are exactly $\frac{3}{7}$; yet neither are they arbitrary. Without knowing how the machine works, we could compute how they differ from 3.0/7.0.

If we expect the machine to work in a certain way, then those differences had better conform to our expectations, or else we will conclude that the machine is not working.

The point is, although the answer you get is not necessarily exact, it doesn't mean that what you get is necessarily unpredictable. You just have to think about it a little bit longer in terms of your floating point model.

2.2 Towards a Convention for Representing Zero

The program in Figure 2-2 divides floating point zero by floating point zero, and comes up with 100.000! At least, that's what the output tells us. When we look back at the program source, we see that we have merely chosen an output format that obscures what we computed. Only *one* of the values, z, is truly zero.

It would have been helpful had z been printed simply as "0". Then it would be distinguished in a natural way from the values x and y, which would obviously be non-zero values too tiny to show any non-zero digits in the output format we chose.

Is there any reason why we could not agree in all instances to print just 0, possibly signed, for a variable whose value is zero? Alas, some languages or implementations are very picky about type agreement. They refuse to print anything to a file that could not subsequently be read back. Perhaps that is why, when you wish to enter 0 for a floating point variable, you may be browbeaten into supplying "0.0" instead. The Pascal language is notorious in this regard.

The conclusion is that we could readily adopt helpful conventions that aid rather than hinder debugging.

2.3 What You Expect and What You Get

Figure 2-3 illustrates that what you get is not necessarily what you expect, and it might be hard to explain too.

Neither way of computing $(p+.1)-(p-1)$ yields the expected result, 2.0. The reason is that p is so large that adding 1.0 gets lost in rounding error.

```
+------------------------- Edit ------------------------------+
|    z = 0         : Print Using "    z = \#\#\#.\#\#\#"; z   |
|    y = 0.000123  : Print Using "    y = \#\#\#.\#\#\#"; y   |
|    x = y/100     : Print Using "    x = \#\#\#.\#\#\#"; x   |
|                    Print Using "  y/x = \#\#\#.\#\#\#"; y/x |
|    Print                                                    |
|    Print "  What You See Is Not Necessarily What You Get."  |
|    End                                                      |
+-------------------------------------------------------------+
+------------------------- Run -------------------------------+
|    z =   0.000                                              |
|    y =   0.000                                              |
|    x =   0.000                                              |
|  y/x = 100.000                                              |
|                                                             |
|  What You See Is Not Necessarily What You Get.              |
+----------------+----------------------------+---------------+
```

WHAT YOU SEE IS NOT NECESSARILY WHAT YOU GET.

Figure 2-2

```
+-------------------------- Edit ----------------------------+
|  p = 2 :  for i=1 to 6 :  p = p*p :  next i                 |
|  pp1 = p + 1 :   pm1 = p - 1                                |
|  d = pp1 - pm1                                              |
|  Print "  We expect d = 2 ,  but actually  d = "; d ; " ,"  |
|  Print "  although  (p+1) - (p-1) = "; (p+1) - (p-1) ; " . !"|
|  Print                                                      |
|  Print "  What you get isn't necessarily what you expected." |
|  End                                                        |
+-------------------------------------------------------------+
+-------------------------- Run ------------------------------+
|    We expect d = 2 ,  but actually  d =  0  ,               |
|    although  (p+1) - (p-1) =  1  . !                        |
|                                                             |
|    What you get isn't necessarily what you expected.        |
+------------------+-----------------------+------------------|
```

WHAT YOU GET ISN'T NECESSARILY WHAT YOU EXPECTED.

Figure 2-3

Because p is a power of two (chosen artfully), it's represented exactly in binary single precision, but $pp1$ rounds to p instead of $p+1$, and $pm1$ to p instead of $p-1$, in single precision. That explains why d comes out to zero. However, this compiler uses extra precision for evaluating subexpressions. That is why the expression $(p+1)-(p-1)$ comes out to 1.0 instead of zero. But why isn't it 2.0 ? p has been chosen to be just so large that in that extra precision, $p-1$ is computed correctly but $p+1$ rounds to p. Without knowing the precisions to which the compiler implicitly evaluates various expressions and variables, how would one explain what happened? In fact, it is possible to write a program that will determine the precisions that the compiler uses, provided they are not chosen too arbitrarily.

What appears to be a capricious difference between two ways of computing $(p+1)-(p-1)$ in this example, and 3.0/7.0 in a previous example, is caused by the internal use for subexpressions of extra precision beyond what can be carried by declared variables. This practice is good insofar as it normally delivers better accuracy but bad insofar as it creates unnecessary anomalies.

The anomalies arise because the compiler affords no way to declare variables with the same extra precision as it uses for subexpressions. The defect here is that it is using three types, but denying you direct access to the widest of them. If the compiler did the decent thing you could assign subexpressions to variables and look at them.

```
+-------------------------------- Edit --------------------------------+
| pOK  =  ((((2.0\^2)\^2)\^2)\^2)\^2                                     |
| pBAD = (((((2.0\^2)\^2)\^2)\^2)\^2)\^2 ' <<< Error 5: Illegal function call |
| Print                                                                |
| Print "This  Error  is really caused by misuse of the  8087's stack."  |
| End                                                                  |
+----------------------------------------------------------------------+
+-------------------------------- Run - -------------------------------+
|                                                                      |
|                 +----------- Message -----------+                    |
|                 | Error search:  SIXTH          |                    |
|                 | Time: 00:00                   |                    |
|                 | Line:    1  Stmt:    1  Free:  190k |               |
+-----------------+                               +--------------------+
                  +-------------------------------+
```

YOU ARE UNLIKELY EVER TO UNDERSTAND, MUCH LESS TO REMEDY EVERY ANOMALY.

Figure 2-4

Figure 2-4 presents a situation that most likely you would never be able to explain without very specialized knowledge (remember our third frightening fact: *if you were hurt by what you got, chances are you may never figure out how or why)*.

In this program, pOK gets 2 squared 5 times; pBAD should get 2 squared 6 times. If pBAD could have been computed it would have turned out to be 18446744073709551616, which is unexceptionable. Why did the compiler decline? The only thing illegal about this allegedly "illegal function call" is that the compiler doesn't like it.

To understand why, you have to know about the stack on the 8087 numeric coprocessor.

It turns out that the compiler is not using the stack in the way it was intended. This is a fault not so much in the compiler as in the way the 8087 was implemented (which is not quite the way it was intended either). Stack overflow on the 8087 is a mess that the compiler writers, understandably, decided to try to avoid. When the compiler encounters an expression sufficiently complicated to threaten stack overflow, it declines to compile it. Unfortunately, they were a little bit too simple-minded, even in this particular case where they could have got away with the computation; they begged the question. The result is the rather unilluminating error message, "Error 5: Illegal function call."

It's not entirely their fault. Kahan had a hand in it. The original design for the 8087 included a scheme that would have allowed the stack to overflow comfortably into memory, and to reload itself quickly on underflow. But the original design was altered by implementors who claimed that they had found a better way. Because Kahan did not follow through and check their claims, the implementation turned out to be horribly worse instead. Consequently, nowadays, hardly anybody takes the trouble to write the software that would handle stack over/underflow correctly. The only good example that comes to mind is a Modula-2 compiler due out late summer 1988.

This example illustrates the extreme lengths to which you might have to go to discover what happened. Could you figure out why it happened? What are your chances, without the anecdote above?

2.4 Don't Divide, and Be Conquered

The implementors of the 8087 are not the only ones to have turned a shortcut into a monster. A much more common kind of trouble arises because system designers are reluctant to implement division in hardware.

Rather than devise a complicated piece of hardware with a low duty cycle that slows down the rest of the machine by its very existence, designers have tried to implement division without a divider. The ways to do this are based upon Euler's formula (see lecture 1). Unfortunately, this method turns into an extremely expensive way to compute correctly rounded quotients, so expensive that most implementors are satisfied with quotients that are at best almost correctly rounded. They come so close to correct rounding that it would seem surly to quibble about the difference. Can the difference matter?

If no programmer could discover that division was slightly incorrect except with an extraordinarily devious program, then slightly incorrect rounding might well be acceptable. Unfortunately for those implementors, programs exist that are neither devious nor very long, yet which very likely will expose an incorrectly rounded quotient.

Figure 2-5 catches some of the problems that can occur in floating point division. This very simple program is certain to stop prematurely if the computer's floating-point arithmetic uses any other radix than 2 (binary), and almost certain to stop prematurely if division or multiplication is not rounded according to the IEEE standard. It stops prematurely on IBM 370s, DEC VAXs, CDC Cybers, CRAYs, and all decimal calculators. But it says "x = i ALWAYS !" on IBM PCs that use an 80x87 math coprocessor, on all Apples that use SANE, on all Sun-3's, on the ELXSI 6400, and on many other systems.

In this program, we do a divide and round, and then multiply by what we divided by. In the absence of error, this should get back where we started. But most floating point implementations are subject to two rounding errors, one on divide and one on multiply, with no guarantee that they will cancel. In fact, they often don't cancel; we don't get back where we started. However, on certain machines, with certain divisors d listed in the program, we always get back.

It should be an interesting exercise for the reader to figure out why this works on those machines that round in accordance with the IEEE standard 754. It is not at all obvious. Don't feel bad if you can't figure it out! Lots of people with fairly strong mathematical ability have been stumped by this simple program.

A program somewhat more complicated than this one uncovers with extremely high probability any incorrectly rounded quotients, practically regardless of how rare they are. It has discovered incorrectly rounded quotients quickly in division programs that had previously delivered billions of randomly generated quotients correctly rounded.

```
+------------------------ Edit -------------------------------+
| for i = 1 to 8000   :   tj = 1    '          j   k    d    |
| for j = 0 to 15     :   tk = 1    '         --- ---  ---   |
|    for k = 0 to j                 '          0   0    2    |
|       d = tj + tk  ' ... = 2^j + 2^k         1   0    3    |
|       q = i/d  :  x = q*d  '     <----<<     1   1    4    |
|       IF NOT( x=i ) THEN    '    <----<<     2   0    5    |
|          print "x = ";x;" NOT = ";i '<----<< 2   1    6    |
|          STOP :  END IF     '    <----<<     2   2    8    |
|       tk = tk+tk ' ... = 2^(k+1)             3   0    9    |
|       next k                '                3   1   10    |
|    tj`= tj+tj '    ... = 2^(j+1)             3   2   12    |
|    next j  :  next i        '                3   3   16    |
| print "  x = i  ALWAYS !"  '... and  d = 17, 18, 20, 24, 32,|
| end                        '...      33, 34, 36, 40, 48, ...|
+-------------------------------------------------------------+
+------------------------ Run --------------------------------+
|   x = i  ALWAYS !                                           |
+--------------------+--------------------+-------------------+
```

A Peculiar Property of Division and Multiplication
When Rounded According to IEEE Standard 754
Figure 2-5

3 The Cost of Precision

So far we have looked at some of the holes that can occur in floating point. We have not given very much thought yet to dealing with them. What about the cost, then? How much additional hardware or software effort is involved in getting everything as right as possible?

3.1 General Principles

Here we encounter that classic notion of economics and engineering, *The Law of Diminishing Returns*. That is, you reach a point where the cost of dealing with additional picayune details begins to become enormous, so that the practical benefits accrue much more slowly than the cost of achieving them.

At this point it is customary to throw up our hands, and say, "Well, it's just too costly. We can't afford to do it." The additional hardware cost begins to appear enormous, and just for the sake of correcting a few rare error cases, or adding a little more precision.

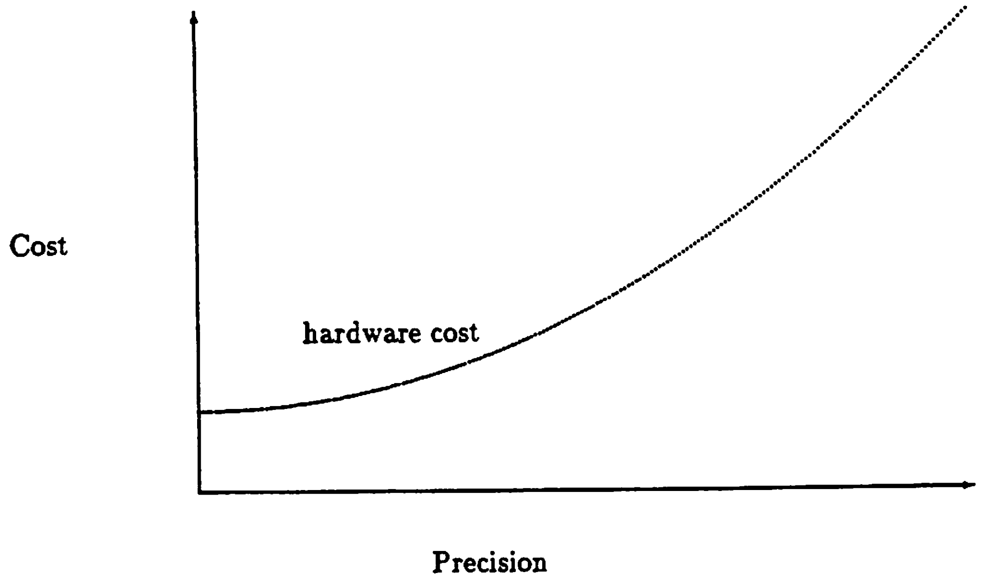


Figure 2-6

Appearances are deceiving. Figure 2-6, taken from some company planning presentations, shows how some people perceive the relationship between precision and cost. The cost can be thought of as the cost of floating point hardware running on some computer you would like to manufacture. Note that the cost of higher precision grows faster than linearly with the number of digits; it turns out that the cost goes up no faster than the square of the number of the digits.

Figure 2-7 plots the perceived cost of programming and running a desired computation: many users, naively and incorrectly, imagine their costs in time and effort will decrease as precision is added.

There is a lower threshold of precision below which the job can never get done. Then there is a sharp decrease, and suddenly, when you have got more than enough precision, it doesn't help to add any more. Once the error is negligible, making it smaller still leaves it

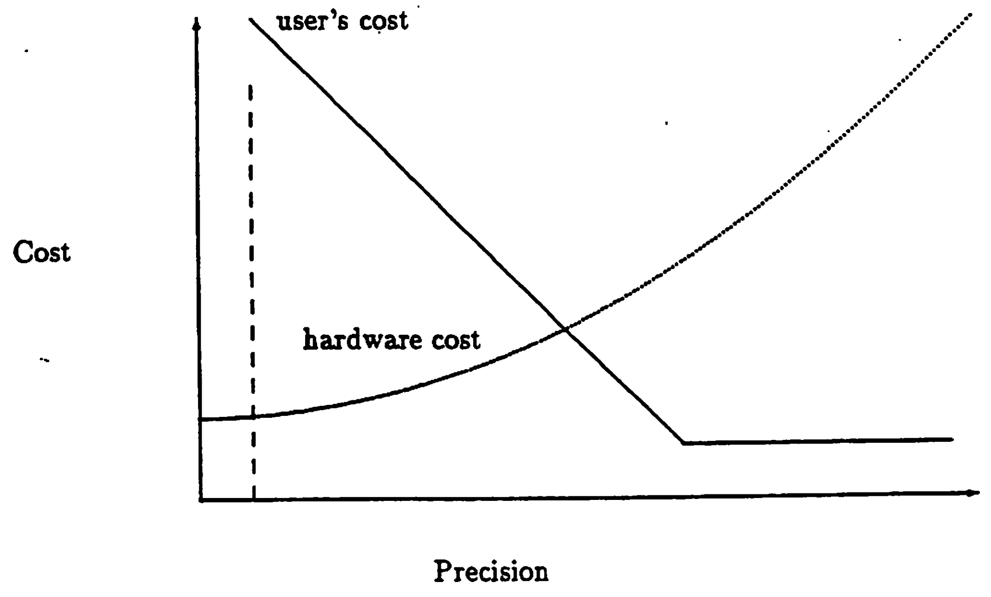


Figure 2-7

negligible.

So in some sense, the total cost is the sum of the two curves. It will be some sort of composite graph which says if you don't have enough precision the users can't use it; if you have barely enough they'll find it hard to use; and if you have much too much, they'll find it easy to use, but it will cost you too much to build. The trick is to find the optimum, somewhere in there.

But this is *wrong!* This curve in fact turns out to be a phantom. And the reason is a very strange theorem, which says that if you do arithmetic in a reasonable way (an IBM 360 is reasonable enough, a VAX is more than reasonable enough, and IEEE 754 is *abundantly* reasonable enough! A CDC Cyber, a CRAY, a Univac 1100, and most TI calculators are *not* reasonable enough), the only limit on accuracy is the over/underflow threshold. But that makes it hard to understand what this curve could mean.

The effect on the hardware cost curve is to make it step. There are places where the cost of increasing precision is pretty well level, and then all of a sudden, it takes off, up to a new plateau.

Let's now think about some arbitrary computational problem. You can consider your problem as a space with as many dimensions as you have input data coming into your problem. Every point in that space represents a collection of data you are going to submit to that problem. It turns out that for very many problems (matrix inversion, eigenvalue problems, and many others that are *transcendental*, there is a surface that runs through this space on which everything bad that *can* happen *does* happen. Figure 2-8 gives a crude 2-dimensional idea of what we are talking about.

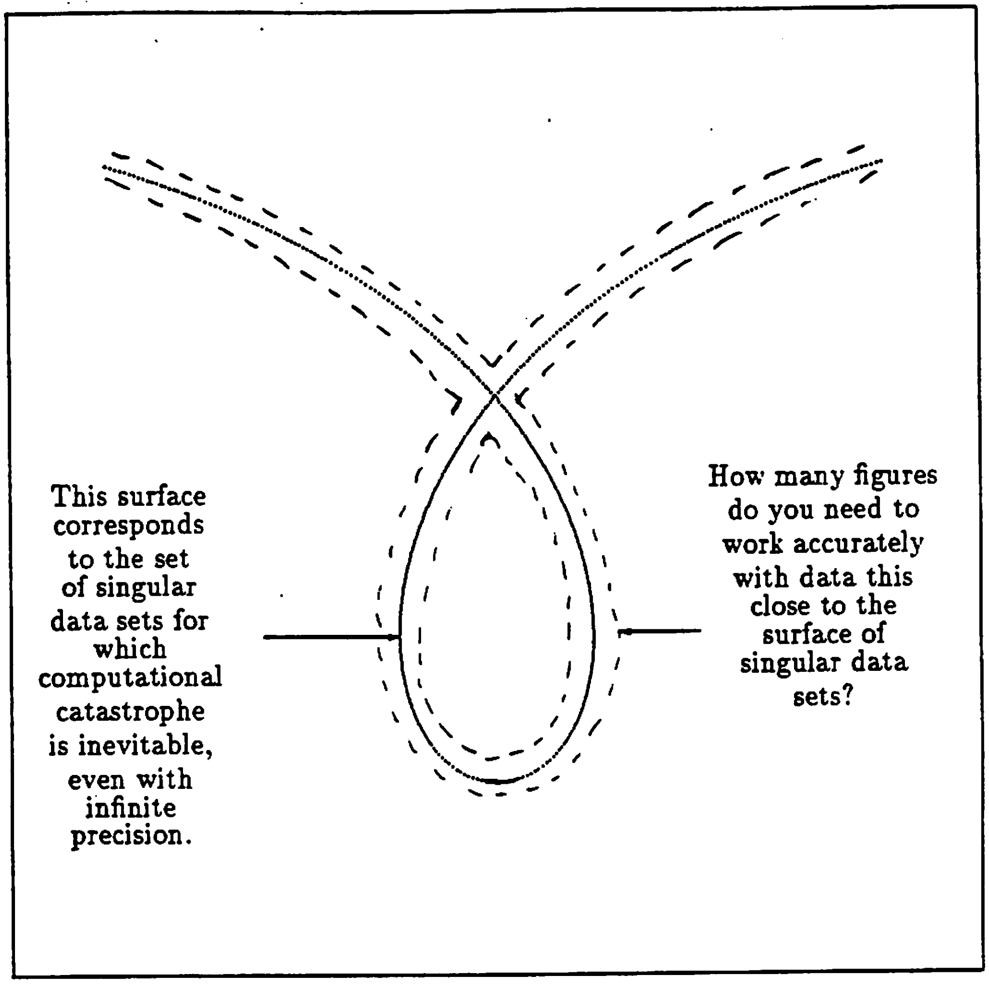

This surface corresponds to the set of singular data sets for which computational catastrophe is inevitable, even with infinite precision.
How many figures do you need to work accurately with data this close to the surface of singular data sets?

Figure 2-8

A polynomial of degree 10, having 11 coefficients, lies in a space of 11 dimensions. In particular, those polynomials having repeated zeroes lie on that troublesome surface in this 11-space, and they are the ones that cause loss of accuracy.

Well, the closer you are to the surface, the more figures you need to deal with the difficulties. The advantage of carrying extra accuracy is that it enables you to cope with the problems you couldn't handle adequately without.

So there is a sort of region of inadequacy around this nasty surface, whose width depends on how much accuracy you carry. For each bit of accuracy you add, you shrink the width of this region by a factor of two. Roughly speaking, you halve the number of problems you can't handle accurately.

So if you carry enough accuracy to put your data outside this region, then carrying more accuracy won't pay off any further. And this is a good rough description of what happens for a very wide class of problems. There *are*, exceptions, but this covers the majority of problems.

Figure 2-9 now illustrates how things really are. If you have too little accuracy, there's a vertical line, and if you have more than that, you're OK. The problem is that you don't

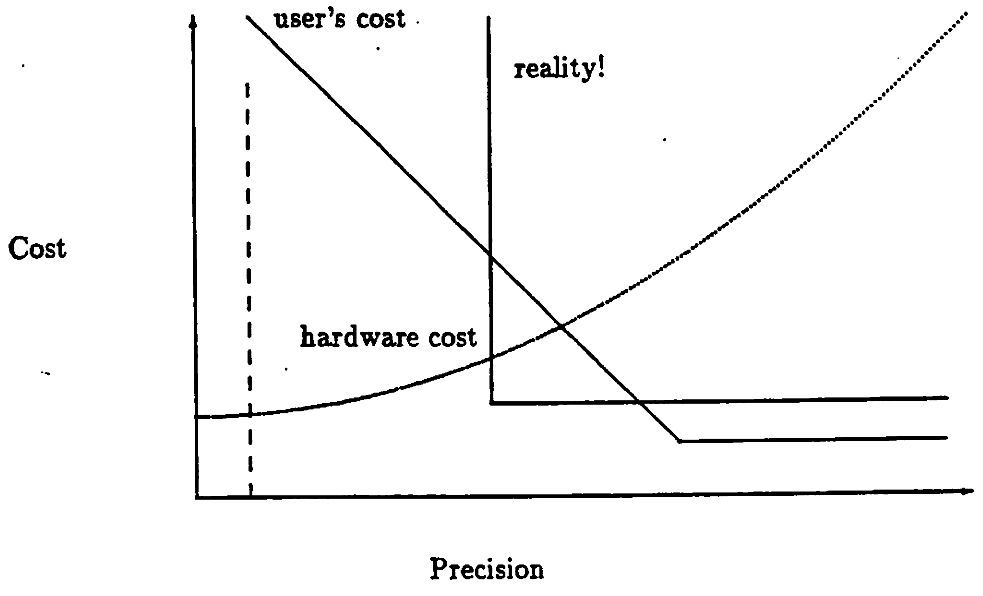


Figure 2-9

know where the line is. Because in practice, you don't know where the surface is. The cost of determining whether a matrix is singular is almost as great as inverting. You might just as well compute the inversion. It will not cost that much more.

So we don't know what the optimum is. There *is* a law of diminishing returns. If you add more precision to your machine, beyond a certain point you aren't going to be increasing the number of added customers by much. As you keep adding bits and halving the number of problems they can't solve, eventually they will believe they can solve all their problems. Adding more bits isn't going to make your machine more attractive to them beyond that point.

3.2 A Rule of Thumb for Working Precision

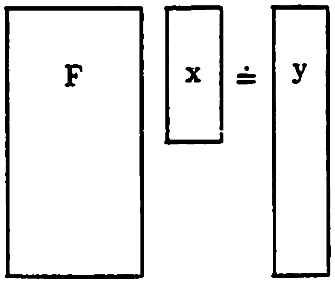
Just how much precision *should* you carry? The working precision you use should be at least twice as big as the accuracy you need, plus a few digits (say two or three). This is not a mathematical theorem, but there are some surprising theorems that could justify such a rule of thumb.

There are many interesting problems in which you lose about half the figures you carry. There are also some problems in which you lose almost all the figures you carry, if you do them in a certain way. In such cases, carrying more accuracy won't really help you, as the surface of incalculable problems is more likely to be an artifact of your algorithm than of the physical nature of the problem.

Interestingly, if you carry enough precision, you can squeeze the surface down to just a few points, which you can prove you can't get rid of as long as you carry only a finite amount of accuracy.

3.2.1 An Example Using Least Squares Techniques

Let's consider least squares problems. A least squares problem can be written thus:

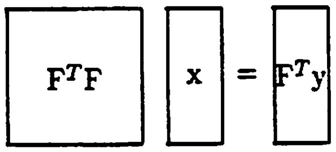


choose x to minimize $\| y - Fx \|$

where F contains independent variables, y the observed dependent variables, and x the unknown coefficients in a linear model. $\| y - Fx \|$ is the length of the vector of discrepancies between the the observations y and the prediction Fx.

Each element of y and row of F constitute an observation y_i made under certain circumstances represented by F_{ij}. If y is gross national product, then one column of F may contain weather observations; another may contain tons of coal extraction, another whether a Democrat or Republican is in the White House.

Now given some new combination of F's independent variables, what's the predicted y? To know that requires x. x is nominally given by:



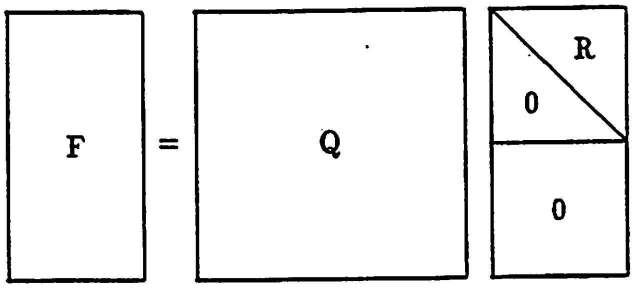
$$(F^T F)x = F^T y$$

if $F^T F$ is non-singular. Notice, however, that the elements of $F^T F$ are roughly squares of elements of F. Thus one can imagine that if the largest element of F is r times the smallest, then the largest element of $F^T F$ is roughly r^2 times the smallest element of $F^T F$. This is likely to cause trouble if some of the columns of F are not very independent: *e.g.*, Rainfall and Agricultural Production.

These imaginings can be made rigorous, but the essential conclusion is that it may take twice as many figures to represent $F^T F$ adequately accurately as F itself required.

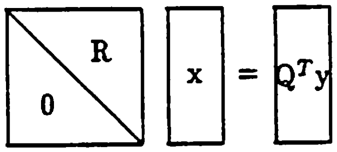
If F needs all the precision it's stored in, then $F^T F$ needs twice that precision, or else $F^T F$ will only be good to half the precision of F. Thus it is that the usual way of thinking about linear least squares supports the general proposition "carry twice as much precision as you need."

Interestingly enough, most reasonable linear least squares problems can be solved in a different way: factor F into an orthogonal matrix Q and an upper right triangular matrix R:



$$F = QR$$

then solve:



$$Rx = Q^T y$$

as a linear equation problem. Unlike F^TF, no squaring is inherent in this method. Provided the model fits the data well, you lose only the number of figures that the data deserves to lose. On the other hand, if the fit is very bad, you get back in trouble again, losing twice as many. But then the result is pretty meaningless anyway.

3.2.2 Thickening of a Graph of an Equation

Let's suppose you would like to solve an equation. For simplicity, let it be one equation in one unknown. We want to know for what values of x is $f(x) = 0$? (See Figure 2-11.) This is tantamount to saying that we'd like to know where the graph cuts the horizontal axis. Now if, in the course of computing, you generate a certain amount of noise because of roundoff, that noise is tantamount to thickening the graph somewhat. If the graph is very steep, you can tolerate a good deal of thickening. You can still find the crossover point, the root of the equation, fairly accurately. But see what happens where the graph is very shallow. Here we have two roots that are coincident, or very nearly so, and when you thicken the graph because of noise, you can see that that causes an astonishingly larger spread in the root. So if you can carry only a certain number of figures of accuracy, the number that you will lose depends on whether the graph cuts steeply or in a shallow way.

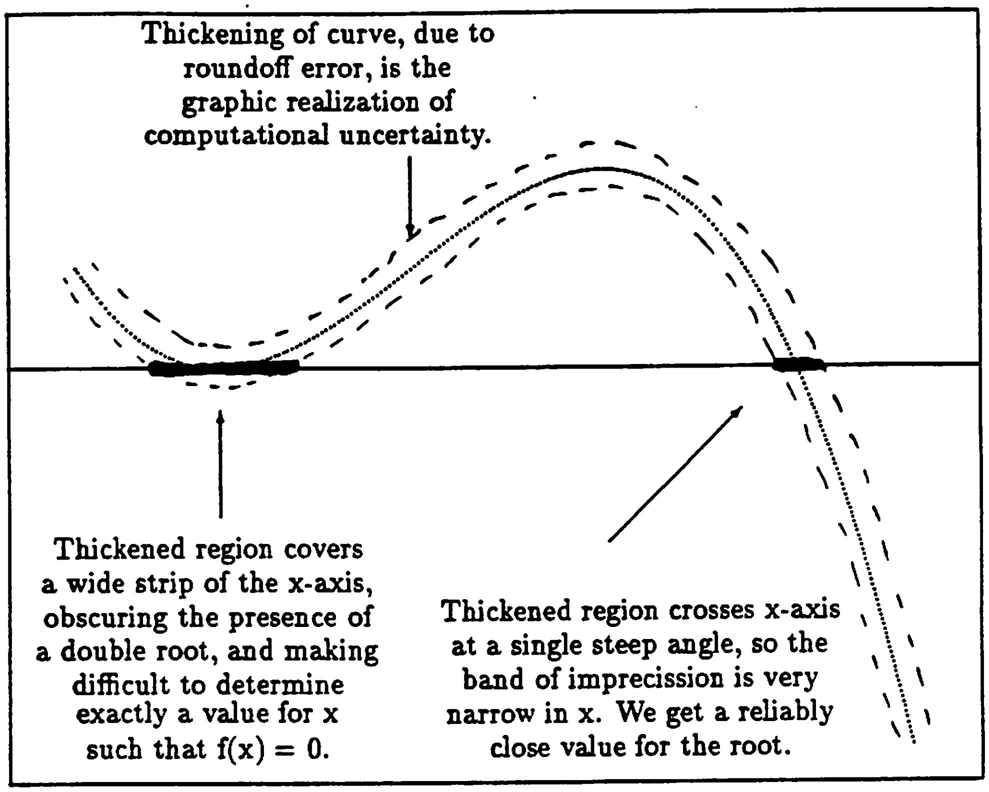

Thickening of curve, due to roundoff error, is the graphic realization of computational uncertainty.
Thickened region covers a wide strip of the x-axis, obscuring the presence of a double root, and making difficult to determine exactly a value for x such that f(x) = 0.
Thickened region crosses x-axis at a single steep angle, so the band of imprecission is very narrow in x. We get a reliably close value for the root.

Figure 2-11

It turns out that the amount of accuracy you will lose depends on the number of roots that fall within the limits of uncertainty. If you have a double root, you will lose about half your figures. If you have a triple root, you will lose two-thirds. And for a quadruple root, you will lose three-quarters. But a triple or quadruple root is extremely rare compared with a double root. So you see that usually you don't lose that many figures. Usually the roots are very simple, and the number of figures you lose depends mostly upon the difficulty of computing the function in the first place. But if there is going to be a pathological situation, it is very likely to be a double root, and then you will lose about half the figures you carry.

3.2.3 A Financial Example

Figure 2-12 illustrates another interesting situation where you lose half your figures. In fact, many textbooks will tell you that you will lose them all. But you don't. In this situation you compute a divided difference,

$$\frac{f(x)-f(y)}{x-y}.$$

This is the slope of a secant being used to approximate the tangent to the curve shown. A great many computations come out this way; in almost any situation involving extrapolation you can expect something like this to occur.

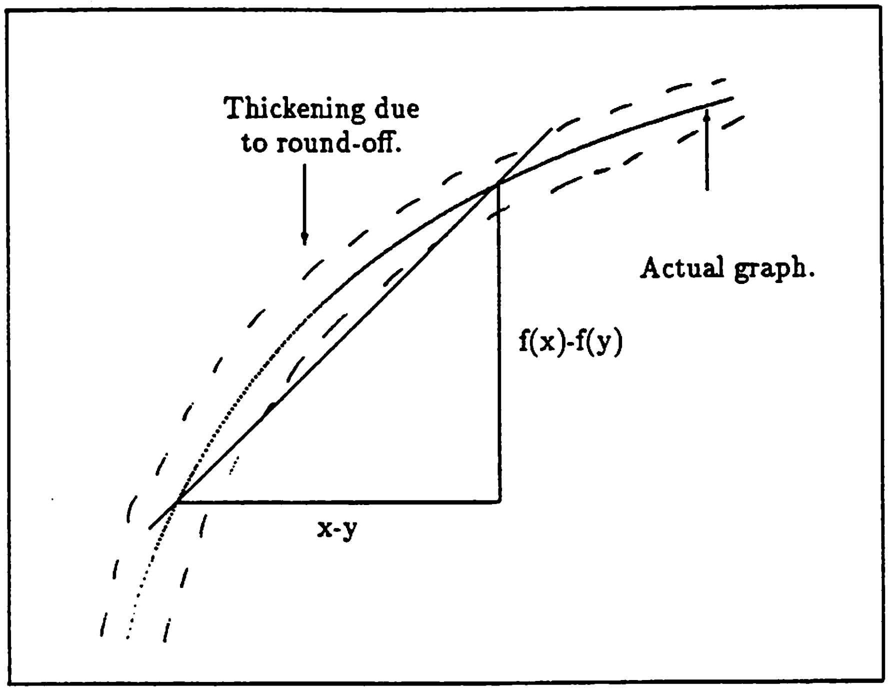


Figure 2-12

So for example, in a financial calculation, suppose you want to compute discounted cash flow, sometimes called the Present Value of Annuity, using the formula

$$PVA = 1 + (1+i)^1 + (1+i)^2 + \ \dots\ (1+i)^{n-1}$$

But rather than compute that literally, observe that it's a geometric series, so its sum

$$PVA = \frac{(1+i)^n - 1}{(1+i) - 1}$$

is a divided difference of $f(x) = (1+x)^n$ evaluated at $x = i$ and $y = 0$.

The books will tell you that if your x is very close to your y, you run the risk of losing all your figures. Why? When you compute $f(x)$ and $f(y)$ you only get a certain number of figures. After that they are just junk. Now when $f(x)$ and $f(y)$ get very close together, the difference starts to get lost down in the noise. So when you do the subtract, you will get a lot of zeroes. You lose most of your digits to cancellation.

Now many people blame the cancellation, thinking that cancellation is what caused the loss of accuracy. But the fact is that cancellation is just the messenger that brings you the bad news: $f(x)$ and $f(y)$ were contaminated by roundoff.

But now look at x and y . They don't have this problem, because you know *exactly* what they are. So when you compute their difference, it will be exactly right. So the problem is the uncertainty in the vertical values, not in the horizontal values. If you bring the two points close enough together, you could lose all your figures.

That's how it would appear. If you use the divided difference form for discounted cash flow, it would appear that you would be in trouble when i is very tiny. But if you are conscientious, you would say, "this formula is valid if i is not equal to zero." If i is equal to zero, or very nearly so, you just use the original series form, and add up all those terms. This is called a *removable singularity.* You can get rid of it by manipulation:

$$\text{if } i \neq 0,\ PVA = \frac{(1+i)^n - 1}{i}$$

$$\text{if } i = 0,\ PVA = n$$

This works for $i = 0$ but is not very accurate for i comparable in size to the least significant bit of 1, because the rounding error in computing $(1+i)$ subsequently swamps everything else. Instead compute

$$\text{if } ((1+i)-1) \neq 0,\ PVA = \frac{(1+i)^n - 1}{(1+i) - 1}$$

$$\text{if } ((1+i)-1) = 0,\ PVA = n$$

For small i, the rounding error in the numerator is compensated by the rounding error in the denominator! On any reasonable machine you will not lose all your figures, and in fact not more than half of them. This is another reason for the rule of thumb.

The reader is invited to try and figure out why you only lose half your figures. This is important, because tricks like this make it possible for many financial calculators to work.

It is important to remember that it is only a rule of thumb to carry slightly more than twice as many figures of accuracy as you need in your result. It is not true of computation in all of its generality. It is merely something you can use to get a rough idea of what kind of accuracy you should supply if you want to do certain kinds of engineering computations.

3.3 Calculating Companies in Competition

An interesting side note is that certain calculators, manufactured by company H, have very pleasant properties when you perform a particular operation on them and then invert it. You can do this any number of times and always get the exact answer back (except possibly for *one* change the first time). However, calculators manufactured by company T gradually lose precision, and after repeating the operation several times, you get back an answer increasingly different from what you started with.

A cute example of this is to enter a phone number into the calculator, take its natural logarithm, and then take the exponent of the result. Company T asserted in its advertisements that you would get the same number back. However, it turns out that if you did this seven times, you would lose the least significant digit of the "phone number." With company H's calculators, on the other hand, this loss did not occur except, occasionally, the first time.

Both companies, H and T, accepted and displayed 10 digits, but used 13 internally. The difference is that company H used careful algorithms and rounding, and after computing, *the 13 digit internal value was always rounded to 10 digits to match the display exactly.*

Company T, however, carried 13 digits throughout, which should give better results than 10. However, their display had a window wide enough to show only 10 out of the 13 digits carried; perhaps that's why they were somewhat careless with the least 3 digits. Consequently, their calculator behaved in anomalous ways at times. Specifically, for certain data their $\exp(\ln(x))$ appeared to produce x, and to do so even six times :

$$\exp(\ln(\exp(\ln(\exp(\ln(\exp(\ln(\exp(\ln(\exp(\ln(x))))))))))))$$

appeared to be x. But the seventh time, it changed, and further repetition induced a prolonged downward drift. That's hard for naive customers to explain. This fact was omitted from company T's advertisements.

In contrast, company H's algorithms might change the least significant bit displayed once, but subsequent $\ln(x)$ and $\exp(x)$ had no effect, because there was no hidden accumulation of error.

In truth, company H put a great deal of care into their arithmetic in other ways as well, and these factors also contributed to their calculator's accuracy. Consequently they sold many calculators, a rare instance of virtue being rewarded. In most cases in floating point, a virtuous feeling is the only reward of virtue. The virtuous floating point processor less often attracts the notice of its user, because it is less often wrong.

3.4 The Cost of Not Being Virtuous

Finally, not only are there rewards for virtue, but there are tremendous costs also for someone if you are not sufficiently careful in designing the machine you sell. For instance, when IBM changed from the old 7090 machines to the 360 series, there was a severe loss of accuracy, much greater than the apparent loss of precision of the new floating point format.

It turned out that a lot of people had data good to no more than 13 bits, so 27 bits on the 7094 worked acceptably according to the rule "carry twice as much as you need." But the 6 hex digits on the 360 were not quite enough, so most single-precision programs were eventually converted to double precision, a practice that continues to this day.

Once converted to double precision, a number or programs still failed to work correctly. IBM customer support spent a great deal to debug these programs, only to discover that some were failing due to double precision multiplication and subtraction lacking a guard digit. Subsequently IBM spent tens of millions of dollars to retrofit all 360's with proper double precision multiplication and division. But the greatest cost to IBM was that people who previously believed IBM equipment to be infallible now knew better, and often blamed it for their own mistakes, so that IBM customer support had to thoroughly debug many customers' programs just to prove that the failures were not IBM's fault.

So you see, if you are virtuous, you may save yourself and others a great deal of headache and expense, though you may never be noticed for it. But if you are careless, you can count on someone having to pay for it.

Appendix A1

An Exercise in Technical Support for Scientific Computation

Floating-point computation is beset by truths, half-truths and mistakes to an extent little appreciated by the world at large, as this exercise will illustrate. Imagine that you work for CRAY providing technical support for its salesmen and customers, and you have been passed the following letter. This letter is based upon actual events embellished only slightly for didactic effect.

```
Dear xxxx,
The Fortran function  AMOD  in both the  CFT and CFT77 compilers on a
CRAY  can give wrong results for certain arguments.  Here is an example:
PROGRAM:
    1       format(2z16)
    2       format(3z18)
    3       format( ..... )
    4       format( ..... )
            read 1, x
            read 1, y
            r = amod(x, y)
            print 4, "x", "y", "r"
            print 2, x, y, r
            print 3, x, y, r
            end

INPUT:
            4009f9ffffffffff
            4009fa0000000000
OUTPUT:
                 x                  y                  r
            4009F9FFFFFFFFFF   4009FA0000000000   BDFA800000000000
            499.99999999999818  500.00000000000000  -1.8189894035458565e-12

This violates the definition of  AMOD(x, y) = x - AINT(x/y)*y ,  because when
x and y  satisfy  0 < x < y ,  as they do here,  then  AMOD(x, y)  should give
a positive number  x  instead of the negative number  x - y .  This violation
crashed a long benchmark code that had worked perfectly well on an  IBM 3090,
DEC VAX  and  Sun-3.  The code calculates  f(x,y) = SQRT(AMOD(x,y))*EXP(-x)
among other things for innocuous values  x and  y  that are always positive.

The  CRAY  has very peculiar division;  240.0/3.0  does not give exactly  80.0
and  x/y  above  yields  1.0  exactly instead of  0.99999...  as on all other
computers and calculators that I have tried.  I can tolerate small errors in
quotients,  but  negative  values for  AMOD(positive, positive)  is too bizarre
to tolerate in an environment where we must share standard Fortran codes that
run unexceptionably on all our other machines.  Negative values cannot occur on
the  IBM machines because they chop quotients,  so  AINT(x/y)  cannot be wrong.
The Sun-3 conforms to the  IEEE standard 754,  which prescribes an exact
remainder,  so it cannot malfunction.  The  DEC VAX  has an  EMOD  instruction
in its architecture,  which may explain why its  AMOD  is always correct.

The  CRAY  is the odd man out here;  if you can't fix it,  we don't want it.

Yours .....
```

What do you recommend that CRAY do?

Appendix A2

Discussion of

An Exercise in Technical Support for Scientific Computation

The allegations about the CRAY's peculiar division are correct as of this date; almost all other computers and calculators do always deliver a quotient $x/y < 1$ despite roundoff whenever $0 < x < y$, so they can guarantee that AMOD(x,y) = x exactly in this case. Indeed, on almost all machines with binary floating- point, AMOD(x,y) is just fine so long as x lies between 0 and about 2.9999*y ; on non-binary machines, between 0 and about 1.9999*y. After that, what happens is not easy to predict.

The trouble starts with the definition of AMOD(x,y) . Ideally, this is the remainder r you would get after you carried out long division (as you learned it in school) to compute x/y , but stopped as soon as all the digits of the quotient n that precede the decimal point had been generated. Then $r = x - n*y$ where $0 \leq r/y < 1$ and the quotient n is the integer nearest x/y on the same side as zero; n would equal AINT(x/y) in the absence of roundoff. Ideally r can be computed and represented exactly (unless it underflows, but let's skip that for now) in the same floating-point format as x and y provided all the digits of n are generated correctly; that is a challenge because there can be so many of them when $|x/y|$ is very big, so many that most must be rounded away by AINT(n) to fit into the same floating-point format. Now you see where the trouble begins; the definition

$$\mathrm{AMOD}(x, y) = x - \mathrm{AINT}(x/y)*y$$

could mean the ideal remainder r , or it could mean the result of computing the rounded values x/y, AINT(x/y), AINT(x/y)*y and x - AINT(x/y)*y in turn. Which is the correct AMOD ?

Originally, when Fortran was young, all computers computed the version of AMOD contaminated by a few rounding errors; CRAYs still do it that way and they are not alone. Unfortunately, the contaminated AMOD(x,y) can fail to lie where most users expect it, namely between 0 and y . Only if AINT(x/y) is computed too big in magnitude, as could happen on machines like DEC VAXs that round quotients correctly, can AMOD have the wrong sign; but that cannot happen on machines that chop quotients as did all the old IBM 7094s, the old CDC 6400 and 6600 (but not 7600) and many other machines, and as do all IBM 370's and Amdahls nowadays. Perhaps that explains why the wrong sign for AMOD is such a surprise for old-timers and their programs. On the other hand, a contaminated AMOD(x,y) can easily be bigger than y , but not likely by so much as would be obvious.

All machines that conform to IEEE 754/854 should be able to derive an ideal AMOD quickly from the standards' mandatory *remainder* operation. Machines that lack a hardware *remainder* can compute it in software like that supplied in the C Math library distributed with 4.3 BSD Berkeley UNIX . Similar but proprietary software resides in DEC's VAX VMS Fortran library; it doesn't use an EMOD instruction, which is a peculiar *multiply.*

What should CRAY do? CRAY's floating-point hardware is too inaccurate to support an ideal AMOD at a tolerable cost. The least intrusive alternative may be a loop to test AMOD(x,y) for the correct sign (the same as y's), although only rarely will that test have to correct AMOD (by adding y to it).