How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?

§0: Abstract

Redesigning computers costs less than retraining people, so it behooves us to adapt computers to the way people have evolved rather than try to adapt people to the way computers have evolved. As the population of computer programmers has grown, proficiency in rounding-error analysis has dwindled. To compensate, better diagnostic aids should be incorporated into hardware, into program development environments, and into programming languages; but this is not happening. Schemes to assist roundoff analysis are beset by failure modes; no scheme is foolproof; only two or three are worth trying. Alas, these few rely upon hardware features built into IEEE Standard 754 for binary floating-point but now atrophying for lack of adequate linguistic support. Here extensive analyses of the genesis of embarrassment due to roundoff, and of the failure modes of all schemes devised so far to avert it, point clearly to what needs doing next.

Contents :

§1: Introduction

Numerical data piles up and numerical programs grow ever more ambitious and complicated while their users become, on average, far less knowlegeable about numerical error-analysis, though no less clever than their predecessors about subjects they care to learn. Consequently numerical anomalies go mostly unobserved or, if observed, routinely misdiagnosed. Fortunately most of them don't matter. Most computations don't matter.

Floating-point computation has become so cheap that it's often not worth much. Vastly expanding multitudes of mostly unwitting users use it mostly for entertainment and games. Their anomalies induced by roundoff flicker in and out of sight too briefly to be noticed or, if noticed, they might merely be promoted to "features" described perhaps in some chat-room on the web like this:

> "No virgin need be found and sacrificed to the gorgon who guards the gate to level seventeen; she will go catatonic if offered exactly \$13.875 ."

While not increasing so fast as games, increasingly many computations do matter -- a lot. But ever fewer of their programmers and users are enabled adequately by education and experience to

debug numerical anomalies. Rounding errors are especially refractory. They are invisible in a program's text; if they weren't their names would drown everything else. They exist only in the mind's eye, and there in a model of computation framed for the purposes of roundoff analysis.

Error-analysis attracts few students and affords fewer career paths. Therefore almost all users and programmers of floating-point computations require help not so much to perform error-analyses (they won't) as to determine whether roundoff is the cause of their distress, and where. That will be followed by an assignment of blame and the task of relieving the distress, if possible.

Several schemes have been advocated as substitutes for or aids to error-analysis by non-experts. None can be trusted fully if used as advertised, which is usually "Mindless", *i.e.* without a penetrating analysis of the program in question. Two or three schemes work well enough often enough to justify the expense of their incorporation in full-featured Programming Development Systems. One scheme is so cheap and so effective that every debugger can support it: It reruns precompiled subprograms in the three redirected rounding modes mandated by IEEE Standard 754 (1985), and thus almost always reveals whether a subprogram is hypersensitive to roundoff. This scheme will be applied frequently to the examples analyzed in this work.

The several "Mindless" schemes in question are surveyed very briefly in §4. They include *Interval Arithmetic*, and recomputation with increasing precision, or with redirected rounding, or with randomized rounding, or with randomly perturbed input data. The few schemes I think worth considering are explained in §14, to which systems programmers and language designers and implementers can jump right now to avoid reading mathematical error-analyses of examples intended to disparage the other schemes.

The examples in §5 and §6 frustrate all schemes that attempt to assess the effects of roundoff without at least breaking a program into smaller subprograms to be assessed individually. Both examples malfunction because of infinitesimally narrow spikes, one deserved, another not. More spikes, but now broad enough to be detectable during debugging, appear in §7 along with a bug that has persisted in MATLAB's log2(…) for over a decade. Two more such bugs appear in §8 along with an attempt to explain their ominous persistence as a consequence of false doctrine. A fatal flaw in recomputation with randomized rounding is illustrated in §9. Glib diagnoses that attribute numerical distress to cancellation, to division by small divisors, or to accumulations of hordes of rounding errors are contradicted by an example in §10 that is more nearly typical of how roundoff causes numerical distress. The user's point of view is illustratred by a case study in §11. A superfluous inaccuracy persistent in a MATLAB function since 1988 is discussed in §12. Terse geometrical arguments in §13 explain how Interval Arithmetic used naively too often deprives this costly and valuable scheme of its value. §14 describes the debugging tools I think worth having, and §15 is my concluding *Jeremiad* predicting doom if they are not to be had.

Of all the many ways in which floating-point computation can go astray, only roundoff, which should rarely have to be taken seriously, is considered seriously in what follows. Over/underflow and other diversions will have to be discussed some other time. Neither shall we consider here computations so numerically unstable that their faults become obvious after any testing adequate to satisfy requirements of *Due Diligence*. Our main concerns hereunder are errors hard to find. Some errors are designed not to be found; these will be considered next in §2 and §3.

§2: Errors Designed Not To Be Found

Some parentheses in Microsoft's *Excel 2000* spreadsheet possess uncanny powers:

Values *Excel 2000* Displays for Several Expressions

Expression	1.23456789012345000E+00	<– Entered to help count digits
V = 4/3 displays ...	1.33333333333333000E+00	Does *Excel* carry 15 sig. dec.?
W = V - 1	3.33333333333333000E-01	Whence comes the 15th 3 ?
X = W*3	1.00000000000000000E+00	Where went all 15 of the 9s ?
Y = X - 1	0.00000000000000000E+00	They all went away **!**
Z = Y*2^52	0.00000000000000000E+00	Really all gone.
(4/3 - 1)*3 - 1	0.00000000000000000E+00	Yes, gone.
((4/3 - 1)*3 - 1)	-2.22044604925031000E-16	(But not *ENTIRELY* gone **!**)
((4/3 - 1)*3 - 1)*2^52	-1.00000000000000000E+00	Excel's arithmetic is weird.

Besides generating an extra digit "3" and rounding away 15 "9"s, *Excel* changed the value of an expression placed between parentheses from zero to something else. Why?

Apparently *Excel* rounds *Cosmetically* in a futile attempt to make Binary floating-point appear to be Decimal. This is why *Excel* confers supernatural powers upon some (not all) parentheses.

Suppose Binary-to-Decimal conversion always leaves enough uncertainty in the last displayed decimal digit of a floating-point variable X that its display cannot distinguish it from several adjacent floating-point numbers. Should the order predicates (X < Y), (X = Y) and (X > Y) distinguish values that display the same? If not, how can they stay consistent with discontinuous functions like SIGN(X) , CEILING(X) and FLOOR(X) , and with functions like SQRT(X) and ACOS(X) whose domains have finite boundaries? Attempts to conceal these conundrums merely make their irrepressible manifestations harder to debug. Here is what happens to the 11 floating-point numbers X between $1 - 5/2^{53}$ and $1 - 13/2^{53}$ that all look the same displayed:

11 Consecutive DistinctValues X Displayed as " 0.99999999999999000…"

#	(X–1)	SIGN(X–1)	FLOOR(X)	(X < 1)	(X = 1)	ACOS(X)	X–1
8	… < 0	–1	0	TRUE	FALSE	… > 0	… < 0
3	… < 0	–1	0	TRUE	FALSE	… > 0	0

The three largest of these 11 values X display an inconsistent 0 for unparenthesized X–1 .

27 Consecutive Distinct Values X Displayed as " 1.00000000000000000… "

#	CEIL(X)	FLOOR(X)	(X < 1)	(X = 1)	X–1	(X–1)	SIGN(X–1)	ACOS(X)
4	1	1	FALSE	TRUE	0	… < 0	–1	… > 0
1	1	1	FALSE	TRUE	0	0	0	0
7	1	1	FALSE	TRUE	0	… > 0	+1	#NUM**!**
15	1	1	FALSE	TRUE	… > 0	… > 0	+1	#NUM**!**

Excel displays the 27 distinct floating-point numbers X between $1 - 4/2^{53}$ and $1 + 22/2^{52}$ as just " 1.000000000000000000… ", which is consistent with CEIL(X) = FLOOR(X) = 1 and the order predicates (X < 1) = FALSE and (X = 1) = TRUE. These contradict 15 values displayed for X–1 and 26 values displayed for (X–1), SIGN(X–1) and ACOS(X). The latter produces the error-indicator #NUM! when actually X > 1 .

Assign Z = 1.00000000000001 . Which of the 45 distinct values X between $1 + 23/2^{52}$ and $1 + 67/2^{52}$ that all display the same " 1.000000000000010000… " as Z actually equals Z ? All 45 computed values of predicate (X = Z) = TRUE , but 30 contradict the displayed X–Z and 44 contradict (X–1) and SIGN(X–Z) . The value stored for Z is the middle value $1 + 45/2^{52}$.

45 Consecutive Distinct Values X Displayed as " 1.00000000000001000… "

#	Displayed X	(X = Z)	X – Z	(X – Z)	SIGN(X – Z)
15	1.00000000000001000…	TRUE	… < 0	… < 0	–1
7	1.00000000000001000…	TRUE	0	… < 0	–1
1	1.00000000000001000…	TRUE	0	0	0
7	1.00000000000001000…	TRUE	0	… > 0	+1
15	1.00000000000001000…	TRUE	… > 0	… > 0	+1

43 Consecutive Distinct Values Y Displayed as " 1024.5000000000… "

#	Displayed Y	ROUND(Y)	ROUND(Y–25)	ROUND(Y–925)
19	1024.500000000…	1025	999	99
2	1024.500000000…	1025	1000	99
22	1024.500000000…	1025	1000	100

All 43 consecutive floating-point numbers Y display the same 1024.5000000000000000… and all the nearest integers ROUND(Y) are the same 1025 when halfway cases round away from zero. However ROUND(Y–25) produces 999 in 19 cases, 1000 in 24, though Y–25 must get computed with no rounding error in both binary and decimal arithmetics. ROUND(Y–925) produces 99 in 21 cases, 100 in 22, again with no roundoff during the subtraction. Why does the 19:24 split change to 21:22 ? Because ROUND is one of *Excel*'s functions that acts upon the displayed value of its argument, unlike functions like ACOS that act upon the true value.

How can a user of *Excel* predict which functions act upon displayed instead of actual values? Which expressions get rounded cosmetically before being displayed? The user's program cannot be debugged without an awareness of these questions, and an aware user ends up debugging Microsoft's pious fraud instead of just a malfunctioning *Excel* spreadsheet.

"Against stupidity even the gods struggle in vain." F. von Schiller (1759-1805)

What's so special about 15 sig. dec.?

Displaying at most 15 sig. dec., as *Excel* does, ensures that a number entered with at most 15 sig. dec., converted to binary floating-point rounded correctly to 53 sig. bits (which is what *Excel*'s arithmetic carries), and then displayed converted back to decimal floating-point rounded

correctly to at least as many sig. dec. as were entered but no more than 15, will always display exactly the same number as was entered. The decision to make *Excel*'s arithmetic seem to be Decimal instead of Binary restricted *Excel*'s display to at most 15 sig. dec., thus hiding the deception well enough to reduce greatly the number of calls upon *Excel*'s technical help-desk. When symptoms of the deception are perceived they are routinely misdiagnosed; *e.g.*, see David Einstein's column on p. E2 of the *San Francisco Chronicle* for 16 and 30 May 2005.

A host of nearly undebuggable anomalies would go away if *Excel*'s floating-point arithmetic were not binary but decimal implemented in software conforming to IEEE Standard 854 (1987) albeit slower than the built-in binary hardware. Decimal has the great advantage that, if enough digits are displayed, *What You See is What You Get*. Some day, perhaps, IBM's *LOTUS 123* spreadsheet may come out with decimal floating-point carrying 34 sig. dec.; if then Microsoft's *Excel* imitates (instead of "innovates"), its mysteries will become vastly fewer.

Meanwhile, if distinct 53 sig. bit binary floating-point numbers are converted to decimal and displayed correctly rounded to 17 sig. dec., they will always display differently. And if the displayed numbers are converted back to binary and rounded correctly to 53 sig. bits, they will reproduce the original binary floating-point numbers. Therefore, so long as binary floating-point persists in *Excel*, its users should be allowed to display as many as 17 sig. dec. instead of just 15, and *Excel* should eschew cosmetic rounding. These simple amendments would eliminate gratuitous anomalies, leaving only those anomalies intrinsic in rounded binary floating-point.

Excel's HELP files should advise users that its floating-point arithmetic is binary to explain why a value entered as " 8.04 ", for example, displays as " 8.0399 9999 9999 9991 " when displayed to all of 17 sig. dec. Roundof will still generate surprises like (4/3 – 1)*3 – 1 ≈ –2.22E–16 instead of 0 . Some surprises that do not occur with decimal arithmetic will continue to afflict binary; for example, both predicates (0.4*10 = 4) and (0.7*10 = 7) are TRUE although (0.4*7 = 0.7*4) is FALSE. And if u := 1/10 = 0.1000… and t := 3/10 = 0.3000… , why does (3*u – t)/(2*u – t + u) display 2.000… instead of a "#DIV/0!" warning? Don't just mumble "Roundoff"; it wouldn't occur here if arithmetic were decimal instead of binary floating-point.

This is no place to list all the corrections *Excel* needs. It was cited here only to exemplify
Errors Designed Not To Be Found.

The moral of *Excel*'s story is …

The bitter truth up front obviates obscurantist lies later.

What's in YOUR spreadsheet?

And now for something entirely different …

§3: Inscrutable Errors from Fanatical Compiler "Optimization"

The struggle to excel in benchmarks induces compiler writers and others to "optimize" floating-point computations in ways that too often sacrifice mathematical integrity. Usually this sacrifice is unintended. "Optimizations" could be debugged more easily were they reflected in revised listings of the source-code emitted by the compiler, but such revelatory listings cannot come from optimizers in a compiler's "back end" shared with different "front ends" for diverse languages. Some compilers and linker/loaders emit listings of the "optimized" code in a pseudo-assembly language. Most applications programmers despair of reading these partly because of their volume and partly because the compiler mixes every source-code line's machine instructions with other lines' in the course of exploiting whatever concurrency can be extracted from multiple pipelines and, nowadays increasingly, multiple processor cores on one chip.

Perhaps the only way to inhibit floating-point pejorations in the guise of "optimizations" is via education of the programming language community and its clientele. To that end two nasty kinds of pejoration are exhibited hereunder. One involves an over-zealous application of arithmetic's associative laws despite parentheses inserted to deter it. The second pejoration, embedded in a recent version of MATLAB, is a matrix analog of a register-spill anomaly that arises when a wide register is spilled temporarily to a narrow location in storage and then reloaded after having lost, presumably inadvertently, some of the bits originally generated in the wide register.

Optimization of matrix multiplication exploits the associativity of addition and properly so in all but a minuscule family of special situations. Still, because roundoff and over/underflow violate floating-point's associativity, it should not be exploited without a programmer's explicit licence, and must not be exploited if parentheses get in the way, lest programs crafted carefully and copied scrupulously, though perhaps uncomprehendingly, be spoiled.

Spoilage will be illustrated by *Compensated Summation*. This is today's name for a technique discovered in fixed point six decades ago, rediscovered in floating-point four decades ago, and rediscovered repeatedly since. It helps defend against roundoff's degradation of amortization schedules, frequently updated averages, slowly convergent infinite series, numerical quadrature and trajectory calculations (ordinary differential equations), among other things.

Our example's task is to approximate an infinite series

Ideal infinite $\text{sum} := \sum_{k\geq 1} \text{term}(k)$

by

Computed $\text{Sum} := \sum_{1}^{N} \text{Term}(k) + \text{Tail}(N)$

in which Tail(N) approximates $\sum_{k>N} \text{term}(k)$ ever better as N increases. But we shall not know N in advance. It may mount into billions.

Billions of rounding errors can degrade severely a sum computed naively :

```
[xxxxxx... Old Sum ...xxxxxx]
+               [xxxxxx... New Term ...xxxxxx]
----------------------------------------
[xxxxxx... New Sum ...xxxxxx] […lost digits…]
```

The lost digits affect the Computed Sum about as much as if those digits had first been discarded

from each New Term. The effect is severe if N is gargantuan. The following program compensates for those lost digits. For simplicity's sake it has been written assuming every Term(k) > Term(k+1) > Term(k+2) > … > 0 .

```
Sum := 0.0 ;  Oldsum := –1 ;  comp := 0.0 ;  k := 0 ;
While  Sum > Oldsum  do …
        k := 1+k ;  Oldsum := Sum ;  comp := comp + Term(k) ;
        Sum := comp + Oldsum ;
        comp := (Oldsum – Sum) + comp ;
   End While Loop;
Sum := Sum + ( Tail(k) + comp ) .  …  This is the final compensated  Sum.
```

However, an over-zealously "optimizing" compiler deduces that the statement

comp := (Oldsum – Sum) + comp ;

is merely an elaborate way to recompute comp := 0.0 , and thereupon scrubs out all references to comp, thus simplifying and slightly speeding up the Loop thus:

```
Sum := 0.0 ;  Oldsum := –1 ;  k := 0 ;
While  Sum > Oldsum  do …
        k := 1+k ;  Oldsum := Sum ;
        Sum := Term(k) + Oldsum ;
   End While Loop;
Sum := Sum + Tail(k) .  …  This is the final  "Optimized"  Sum.
```

Now let us assign formulas for the terms of the series:

$$\text{Term}(k) := 3465/(k^2 - 1/16) + 3465/((k + 1/2)^2 - 1/16) ,$$

$$\text{Tail}(k) := 3465/(k + 1/2) + 3465/(k + 1) ,$$

and then compute

$$\text{Sum} := \sum_1^N \text{Term}(k) + \text{Tail}(N)$$

using each of the foregoing programs, one compensated, the other "optimized".

Of course, a little mathematical analysis might render the programs unnecessary, but programming a computer is easier and running it is cheaper than analysis.

Here are the results from a Fortran program run on an IBM T21 Laptop:

Final Sums from Two Programs

Program:	Compensated	"Optimized"
Final Sum :	9240.000000000000	9240.000001147523
Time :	13.7 sec.	17.8 sec.
Loop-count K :	61,728,404	87,290,410
Time per Loop :	2.22E–7 sec.	2.04E–7 sec.

Even though the "Optimized" program's Loop runs almost 10% faster, this program took about 25% longer to get a result substantially worse than the program run as originally written.

Do you see why? If someone doesn't, would you like him to "optimize" your floating-point?

In general the over-zealously "optimized" Sum can be wrong in the worst way: Occasionally its error will be too small to be obvious but not small enough to be inconsequential.

How can a programmer unaware of the "optimization" debug that?

There is a way: Rerun both programs in different rounding modes afforded by IEEE Standard 754 on fully conforming systems. Currently the only fully conforming standard programming language is C99, and only on a very few machines, but let's not dwell on that now. On my machines each program can be rerun first rounding every arithmetic operation Down (towards $-\infty$) and again rounding Up (towards $+\infty$) without recompilation. Here are the results:

Final Sums from Two Programs Rounded Differently

Program:	Compensated	"Optimized"
Rounded to Nearest :	9240.000000000000	9240.000001147523
Rounded Down :	9239.999999999998	9239.999994834162
Rounded Up :	9240.000000000002	Ran almost forever

These results leave no doubt that "optimization" has actually made the program much worse.

Do you see why? If someone doesn't, would you like him to "optimize" your floating-point?

The second example of numerical pejoration caused by ill-advised "optimization" exposes the damage done when a compiler *spills* wide registers to and reloads them from narrow destinations, thus losing (presumably inadvertently) the wide registers' accuracy. We shall explore *Iterative Refinement* of computed eigensystems of real symmetric matrices. Nowadays software like MATLAB's `eig`, though nearly bulletproof, can still lose accuracy to roundoff in several ways:

- Losses worsen as dimensions (degrees of freedom) increase.
- Eigenvectors lose accuracy as their eigenvalues approach coincidence.
- Severe losses can occur if the data's structural symmetries are lost to roundoff.
- Severe losses can occur if software mishandles systematically wide-ranging data.

  Example: A flea atop a dog atop an elephant atop the Eiffel tower.
  The flea's vibrational frequencies so dominate the tower's that the tower's can be lost to roundoff unless appropriate special methods are used.

Iterative Refinement is a scheme that usually attenuates those losses without requiring that their cause(s) be identified. The scheme starts by computing a *Residual* that measures how badly the solution computed so far dissatisfies its defining equations. Then the residual guides refinement of that solution. My MATLAB program `refiheig` does that in a way too complicated to describe here other than to say that its accuracy is limited principally by the accuracy to which matrix products can be accumulated while computing residuals.

A first illustrative example is the n-by-n *Reversed Pascal* matrix. When n = 6 it looks like this:

```
          252   126    56    21     6     1
          126    70    35    15     5     1
P =        56    35    20    10     4     1
           21    15    10     6     3     1
            6     5     4     3     2     1
            1     1     1     1     1     1
```

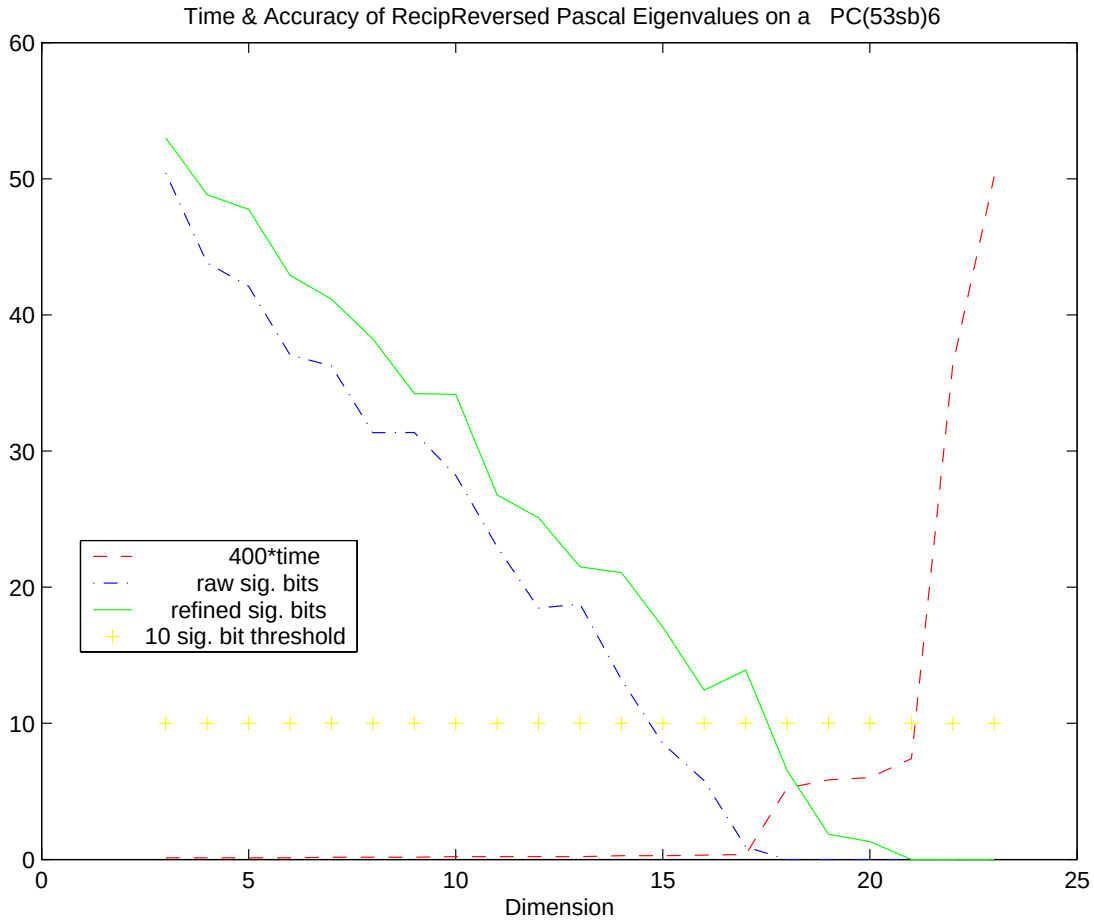
Its elements range ever more wildly as dimension n increases. Though no simple formulas for its n eigenvalues are known, they are known to be positive and come in reciprocal pairs:

If λ is an eigenvalue, so is $1/\lambda$.

Consequently the accuracy of computed eigenvalues will be gauged by how close products of appropriate pairs come to 1 .

Because the ratio (biggest eigenvalue)/(smallest) grows like $2^{4n}/(n\pi)$, we can expect smaller computed eigenvalues to lose almost 4n sig. bits to `eig`'s roundoff as the dimension n gets big. It can't get very big without losing all 53 sig. bits carried by `eig`'s arithmetic. What is the biggest dimension n for which `eig` yields at least 10 sig. bits (3 sig. dec.) of accuracy? Since much of the lost accuracy is lost to `eig`'s disregard of the systematically wild variation in the magnitudes of the elements of the Reversed Pascal matrix P , we hope that `refiheig` can recover some of the lost accuracy and thus increase the dimension n for which we get at least 10 sig. bits. If necessary we may iterate `refiheig` by calling it again up to twice, though repeated calls hardly ever improve accuracy much.

With Residuals Accumulated to 53 sig. bits



MATLAB v. 6.5 on a Wintel PC accumulating matrix products to 53 sig. bits:
Refinement boosts successful dimensions n from $n \le 14$ to $n \le 17$ in a tolerable time.

The results above were obtained by running `eig` and my `refiheig` under MATLAB 6.5 and Windows 2000 on an IBM T21 laptop. (My `refiheig` also runs under MATLAB versions 4.2

and 5.2 on Macintoshes as well as Wintel machines. This will become significant later.) Similar results are obtained on Sun SPARCs, SGS MIPS, HP PA-RISC, IBM Power PCs and Apple Power Macs; on all of them …

Iterative Refinement increases from n = 14 to n = 17 the
largest dimension for which at least 10 sig. bits are achieved.

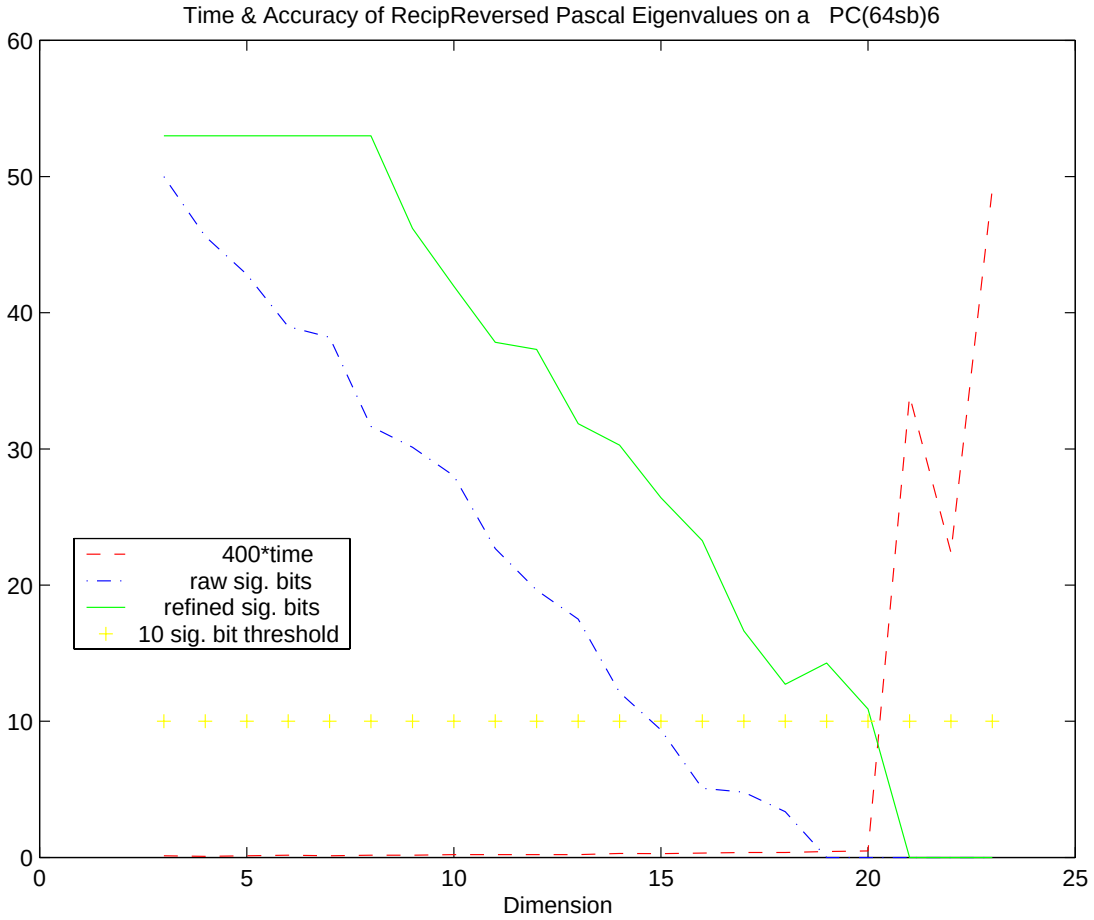
At dimensions n > 17 computation time rises steeply mainly to issue warnings of possibly severe loss of accuracy. For n ≤ 17 refined accuracy takes less than three times as long as `eig` takes.

However, Wintel machines can get better results in the same time running *exactly* the same MATLAB programs on the same version 6.5 of MATLAB after invoking the prefatory command

```
system_dependent('setprecision', 64)
```

(or on version 4.2 without that command) to accumulate matrix products to 64 sig. bits before storing them back to 53. This is how Intel's floating-point was originally (back in 1978) designed to be used, and 95% of the computers on and under desks still have this capability. Here are the better results:

With Residuals Accumulated to 64 sig. bits



MATLAB v. 6.5 on a Wintel PC accumulating matrix products to 64 sig. bits:
Refinement boosts successful dimensions n from n ≤ 14 to n ≤ 20 in a tolerable time.

With that extra-precise accumulation, Iterative Refinement increases from n = 14 to n = 20 (instead of just 17) the largest dimension for which 10 sig. bits are achieved, and with no significant increase in running time.

> We conclude that iterative refinement of eigenvalues is worthwhile without extra-precise accumulation but worth at least about ten more sig. bits with it.

Next let's see how well iterative refinement enhances the accuracies of eigenvectors of an n-by-n test matrix devised by Wallace Givens; it looks like this when n = 6 :

```
              22    18    14    10     6     2
              18    18    14    10     6     2
W :=          14    14    14    10     6     2
              10    10    10    10     6     2
               6     6     6     6     6     2
               2     2     2     2     2     2
```

Givens' matrix W can be derived from a discretization of an integral equation. Its eigenvalues and eigenvectors can be computed almost correctly rounded from simple formulas that we shall use only to check the accuracy of MATLAB's and my eigensystem software.

The smallest eigenvalues cluster just above 1 ; the biggest reach over $(4n/\pi)^2$. The eigenvectors have a special structure: Every eigenvector's elements can be obtained from any other's by permuting elements and reversing some signs. The accuracy of computed eigenvectors belonging to small clustered eigenvalues can be degraded by roundoff to an extent that grows about as fast as n^4 when the dimension n is huge. How much of that degradation can be undone by iterative refinement when, say, n = 1000 ? In the tabulations below the row marked "MxM" shows how many sig. bits were accumulated during matrix multiplication. The "near-minimal" residual was computed from the almost correctly rounded eigensystem.

Execution Times to Compute Givens' Eigenvectors

MATLAB version:	v. 6.5	v. 6.5	v. 4.2
MxM sig. bits	53 s.b.	64 s.b.	64 s.b.
eig	52.5 sec.	52.9 sec.	122 sec.
refiheig	67.1 sec.	66.7 sec.	1171 sec.

Residuals vs. near-minimal 2.3E-11

MATLAB version:	v. 6.5	v. 6.5	v. 4.2
MxM sig. bits	53 s.b.	64 s.b.	64 s.b.
eig	2.1E-9	1.2E-10	3.1E-9
refiheig	1.2E-10	2.9E-11	7.4E-12

Eigenvector Accuracies in Sig. Bits

MATLAB version:	v. 6.5	v. 6.5	v. 4.2
MxM sig. bits	53 s.b.	64 s.b.	64 s.b.
eig	18.4 s.b.	23.4 s.b.	18.6 s.b.
refiheig	25.9 s,b.	30.2 s.b.	40.7 s.b.

Alas, something has gone awry.

Why is MATLAB version 6.5 so much (20 times) faster than version 4.2 ?

Why is v. 6.5's refinement so much (10 sig. bits ≈ 3 sig. dec.) less accurate than v. 4.2's ?

V. 6.5 splits big matrices into small blocks to incur fewer cache misses during its blocked-matrix multiplications. These can run enormously faster than v. 4.2's ordinary matrix multiplications.

But v. 6.5 spills individual block products, each accumulated to 64 sig. bits, into memory holding only 53. This squanders almost all advantages of extra-precise accumulation, obscuring residuals while adding negligibly to speed. The consequent loss of 10 sig. bits of ultimate accuracy could not have been detected if we had compared only computed residuals instead of comparing computed with correct eigenvectors. Has anybody else noticed the spill anomaly ?

The anomaly should not be blamed entirely upon MATLAB. It uses matrix-multiply subprograms (BLAS 3) "optimized" by Intel for its *Pentium* architecture taking account of cache line-sizes and management. If the subprograms stored sums of block products retaining all 64 sig. bits accumulated, instead of just 53, the extra time and memory required would be practically inconsequential.

Thus does fanatical optimization for a little more speed induce a subtle but severe pejoration of accuracy made almost impossible to debug, if noticed, by lack of access to the optimized program as actually executed. How likely is the loss of accuracy to be noticed? Without comparisons of computed results with true results (rarely available) or with previously computed results (who else keeps old versions of MATLAB around?), suspicion could fall upon dubious results were they recomputed with redirected roundings. Among those who could do that, who would think to do it?

In response to the foregoing complaints the following advice has come out of the programming language community:

"If you dislike the effect upon your program of a compiler's optimization, turn it off."

This choice is unavailable to a user of MATLAB. The choice is impractical for a programmer of would-be portable code, like MATLAB and LAPACK and many others, for three reasons:

First, control over optimization is effected not from his program's text but from a command line that invokes the compiler. Optimization commands are not standardized; they vary in effect, often obscurely, among different compilers, sometimes for the same hardware.

Second, for arcane reasons having to do with benchmarking practices, compilers often "bundle" desirable along with pernicious optimizations, unnecessarily forcing knowledgeable programmers to choose between speed and accuracy knowing that programs that run too slowly won't get run.

Third, a programmer knowledgeable enough to choose the right command-line options for each of today's compilers cannot know what tomorrow will bring, nor who else will incorporate part or all of his source-code into theirs. He can only recall ruefully this line from *Hamlet*:

"There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy."

There has to be a meeting of currently disparate minds from two communities — Programming Languages and Portable Numerical Software — lest petty "optimizations" change the meanings of ostensibly portable programs in hitherto unimagined ways that practically stymie debugging.

§4: Five Plausible Schemes

Can the effects of roundoff upon a floating-point computation be assessed without submitting it to a mathematically rigorous and (if feasible at all) time-consuming error-analysis? In general, *No*.

This mathematical fact of computational life has not deterred advocates of schemes like these:

- •1 Repeat the computation in arithmetics of increasing precision, increasing it until as many as desired of the results' digits agree.
- •2 Repeat the computation in arithmetic of the same precision but rounded differently, say *Down*, and then *Up*, and maybe *Towards Zero* too, besides *To Nearest*, and compare three or four results.
- •3 Repeat the computation a few times in arithmetic of the same precision rounding operations randomly, some *Up*, some *Down*, and treat results statistically.
- •4 Repeat the computation a few times in arithmetic of the same precision but with slightly different input data each time, and see how widely results spread.
- •5 Perform the computation in *Significance Arithmetic*, or in *Interval Arithmetic*.

Here are brief summaries of the respective schemes' prospects:

- •1 Though not foolproof, increasing precision is extremely likely to work well provided the manner in which rounding is performed is the same for all precisions; but this scheme is costly to provide and may run intolerably slowly. For that price we may be served better by almost foolproof extendable-precision Interval Arithmetic.
- •2 Though far from foolproof, rounding every inexact arithmetic operation (but not constants) in the same direction for each of two or three directions besides the default *To Nearest* is very likely to confirm accidentally exposed hypersensitivity to roundoff. When feasible, this scheme offers the best *Benefit/Cost* ratio.
- •3 Repeated recomputation with randomly redirected roundings is far more likely than the previous non-random redirected roundings to mislead users in these two ways:
  - A few subtle programs that compensate for their own rounding errors may be thwarted and thus unnecessarily produce excessively inaccurate results.
  - Many numerically fragile programs, Gaussian Elimination among them, can be sent far astray by just one or two among their myriad rounding errors. Those one or two are too likely to be perturbed the same way at random, thus producing repeatedly almost identical but utterly wrong results, unless randomly rounded recomputation is repeated at least several times.

  Random rounding is costly to implement, runs slowly and ought to be rerun often.
- •4 Only if *Backward Error-Analysis* has succeeded in *proving* that a program's rounding errors alter its results about as much as do *all* end-figure perturbations of its input data may such perturbations have diagnostic value. Even then such perturbations can all produce the same utterly wrong result. Or else slightly perturbed data may produce wildly different but correct results like tan(x) at the two floating-point arguments adjacent to $\pi/2$ (which is not a floating-point number).
- •5 Significance Arithmetic attempts to retain, for every intermediate and final result, only those digits deemed uncontaminated by previous rounding or other errors. It

is a coarse kind of Interval Arithmetic doomed to overestimate or underestimate (or both) the number of significant digits to an extent proportional to the number of arithmetic operations between input data and output results.

Interval Arithmetic approximates every variable by an interval whose ends straddle the variable's true value. Used naively, this scheme is cursed by excessively wide intervals that undermine its credibility when wide intervals are deserved. Swollen intervals can often be curbed by combining Interval Arithmetic with ordinarily rounded arithmetic in a computation artfully recast as the determination of the fixed-point of a sufficiently contractive mapping. "Artful" is far from "Mindless". Far less art may coax success from extendable-precision Interval Arithmetic, though its price may be high and its performance slow.

Citations for the schemes mentioned above have been omitted for the time being because these notes are not intended to attack the schemes' advocates. Each scheme has its advocates, so it must have worked on at least one example. That's not the point of these notes. The point is …

Which schemes will work on *your* computation without requiring you to error-analyze it?

In general, *none*. But one or two of these schemes may be worth trying anyway.

I propose to collect examples each of which defeats some scheme(s) mentioned above, and which collectively defeat all those schemes, including the ones I favor. The collection will grow as time permits, including perhaps accretions from subsequent contributors.

§5: J-M. Muller's Recurrence

The futility of all mindless assessments of roundoff's effect is exposed by a recurrence contrived by Jean-Michel Muller around 1980 and modified slightly here. Given the function

$$Œ(y, z) := 108 - (815 - 1500/z)/y$$

and initial values $x_0 := 4$ and $x_1 := 4.25$, define $x_{n+1} := Œ(x_n, x_{n-1})$ for $n = 1, 2, 3, \ldots$ in turn. Our task is to compute x_N for some moderately big preassigned integer N, say $N = 80$.

The sequence $\{x_n\}$ does tend to a limit; $x_n \to L$ as $n \to +\infty$. In the absence of an analysis that finds L exactly it can be approximated by computing the sequence $\{x_n\}$ until, say, x_{N-1} differs negligibly from x_N; then this x_N approximates L.

Try to compute x_{80} or L before reading what follows.

All fast floating-point hardware, all Randomized Arithmetic, and most implementations of Significance Arithmetic will allege $x_{80} \approx L = 100$ very convincingly. The correct limit $L = 5$.

The correct $x_n = 5 - 2/\left(1 + (5/3)^n\right) = 8 - 15/x_{n-1}$, so $x_{80} = 4.99999999999999964263\ldots$.

Tabulated below are values x_n computed first exactly, then by a FORTRAN program carrying 64 sig. bits and a MATLAB program carrying 53 sig. bits on an Intel 302 (i386/387 IBM PC clone).

The Recurrence Exactly, then in 64 Sig, Bits, and then in 53 Sig. Bits

n	True x_n	FORTRAN's X_n	X_n's γ_n	MATLAB's x_n	x_n's γ_n
0	4	4	0	4	0
1	4.25	4.25	0	4.25	0
2	4.4705882352941…	4.4705882352941	1.4527240931E-23	4.4705882352941	-5.95035788e-20
3	4.6447368421052…	4.6447368421052	9.3144142261E-24	4.6447368421052	-7.27269462e-20
4	4.7705382436260…	4.7705382436260	9.3879254811E-24	4.7705382436250	-7.26081334e-20
5	4.8557007125890…	4.8557007125890	9.4011127174E-24	4.8557007125685	-7.26054934e-20
6	4.9108474990827…	4.9108474990828	9.4016062483E-24	4.9108474986606	-7.26062074e-20
7	4.9455374041239…	4.9455374041250	9.4016485474E-24	4.9455373955305	-7.26061505e-20
8	4.9669625817627…	4.9669625817851	9.4016502826E-24	4.9669624080410	-7.26061478e-20
9	4.9800457013556…	4.9800457018084	9.4016502839E-24	4.9800422042930	-7.26061478e-20
10	4.9879794484783…	4.9879794575704	9.4016502819E-24	4.9879092327957	-7.26061478e-20
11	4.9927702880620…	4.9927704703332	9.4016502815E-24	4.9913626413145	-7.26061478e-20
12	4.9956558915066…	4.9956595420973	9.4016502814E-24	4.9674550955522	-7.26061478e-20
13	4.9973912683813…	4.9974643422978	9.4016502814E-24	4.4296904983088	-7.26061478e-20
14	4.9984339439448…	4.9998961477637	9.4016502814E-24	-7.8172365784593	-7.26061478e-20
15	4.9990600719708…	5.0283045630311	9.4016502814E-24	168.93916767106	-7.26061478e-20
16	4.9994359371468…	5.5810310849684	9.4016502814E-24	102.03996315205	-7.26061478e-20
17	4.9996615241037…	15.420563287948	9.4016502814E-24	100.09994751625	-7.26061478e-20
18	4.9997969007134…	72.577658482982	9.4016502814E-24	100.00499204097	-7.26061478e-20
19	4.9998781354779…	98.110905976394	9.4016502814E-24	100.00024957923	-7.26061478e-20
20	4.9999268795046…	99.903728999705	9.4016502814E-24	100.00001247862	-7.26061479e-20
21	4.9999561270611…	99.995181883411	9.4016502814E-24	100.00000062392	-7.26061486e-20
22	4.9999736760057…	99.999759084721	9.4016502814E-24	100.00000003119	-7.26061591e-20
23	4.9999842055202…	99.999987954271	9.4016502815E-24	100.00000000156	-7.26060665e-20
24	4.9999905232822…	99.999999397715	9.4016502814E-24	100.00000000007	-7.26058323e-20
25	4.9999943139585…	99.999999969885	9.4016502814E-24	100.00000000000	-7.27116855e-20
26	4.9999965883712…	99.999999998494	9.4016502728E-24	100.00000000000	-7.11534953e-20
27	4.9999979530213…	99.999999999924	9.4016499619E-24	100.00000000000	-4.98074120e-20
28	4.9999987718123…	99.999999999996	9.4016399662E-24	100	Infinity
29	4.9999992630872…	99.999999999999	9.4017762549E-24	100	Infinity
30	4.9999995578522…	99.999999999999	9.4031615325E-24	100	Infinity
31	4.9999997347113…	100.00000000000	9.3755043286E-24	100	Infinity
32	4.9999998408267…	100.00000000000	7.9691782475E-24	100	Infinity
33	4.9999999044960…	100	Infinity	100	Infinity
34	4.9999999426976…	100	Infinity	100	Infinity
…	…	…	…	…	…
74	4.9999999999999…	100	Infinity	100	Infinity
75	4.9999999999999…	100	Infinity	100	Infinity
…	…	…	…	…	…

γ_n will be explained in a moment.

Evidently a few intermediate results change when the arithmetic's precision changes; in general *such intermediate changes need not imply incorrect final results*, as we shall see soon. Interval Arithmetic delivers a narrow interval around $L \approx 5$ instead of a worthless wide interval only if, as with ordinary arithmetic, extravagant precision rather beyond $4.3 \cdot N$ sig. bits is carried.

Why do so many different calculations produce the same wrong result $x_{80} \approx 100$?

To analyze the recurrence ignore x_0 and x_1 momentarily and substitute $x_n = y_{n+1}/y_n$ into the original recurrence $x_{n+1} := Œ(x_n, x_{n-1})$ to get $y_{n+2} = 108y_{n+1} - 815y_n + 1500y_{n-1}$. This linear recurrence can be solved in closed form with the aid of the zeros of its *Characteristic Polynomial*

$$z^3 - 108z^2 + 815z - 1500 \;=\; (z-3)(z-5)(z-100) \;.$$

Consequently the general solution x_n of the original recurrence is

$$x_n = (\alpha \cdot 3^{n+1} + \beta \cdot 5^{n+1} + \gamma \cdot 100^{n+1})/(\alpha \cdot 3^n + \beta \cdot 5^n + \gamma \cdot 100^n) \quad \text{for} \quad n = 0, 1, 2, 3, \ldots$$

in which constants α, β, γ are not all zero. They may be chosen to match any two prescribed values x_0 and x_1 ; choices $\alpha = \beta = 1$ and $\gamma = 0$ match our prescribed $x_0 := 4$ and $x_1 := 4.25$, and then would yield $x_n = (3^{n+1} + 5^{n+1})/(3^n + 5^n)$ if no rounding errors were committed. But roundoff perturbs computed values x_n . Then they are closely approximated at least initially by

$$x_n \approx (3^{n+1} + 5^{n+1} + \gamma_n \cdot 100^{n+1})/(3^n + 5^n + \gamma_n \cdot 100^n) \quad \text{for} \quad n = 3, 4, 5, \ldots$$

$$= \; 100 - (95 + 97 \cdot (3/5)^n)/(\, 20^n \cdot \gamma_n + 1 + (3/5)^n)$$

in which γ_n is a tiny nonzero near-constant resembling a rounding error in a number near 0.001 . This changes the limit $x_n \to L$ from $L = 5$ to $L = 100$. In the foregoing tabulation γ_n was obtained from the formula $\gamma_n := (\, (x_n - 3) \cdot 3^n + (x_n - 5) \cdot 5^n \,)/(\, (100 - x_n) \cdot 100^n \,)$.

What if the recurrence started at $x_1 := 4.25$ and $x_2 := 8 - 15/x_0 = 76/17$? At first sight neither x_{80} nor limit L should change. However, $76/17 = 4.4705882352941\ldots$ cannot be represented exactly as a floating-point number, so it must be rounded off, thus changing x_{80} and L to 100 . Though not what was intended, *this is the correct result* for the initial x_2 stored in the computer.

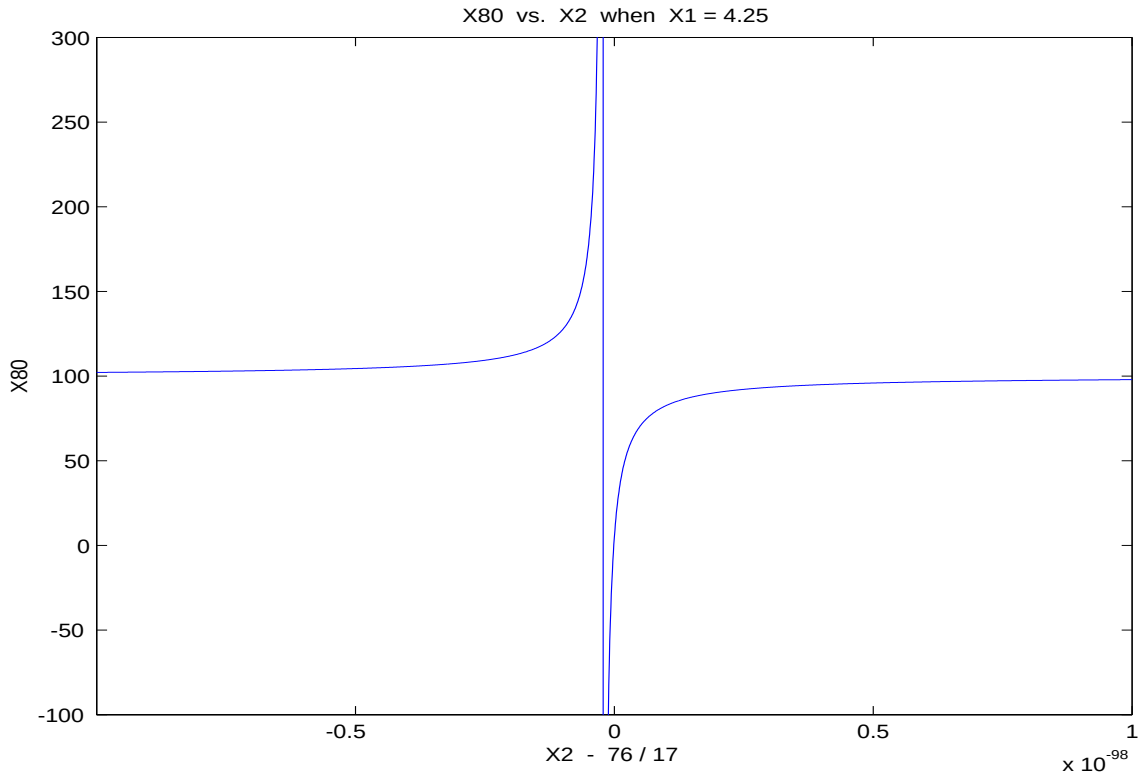
Exact rational arithmetic can compute x_{80} perfectly, getting

$$20679515313825691893956541713900959836557784303479467296 4/4135903062765138381747484931067110433633221064823559411 3$$

if enough digits are carried. Arithmetic must carry more than $2.33 \cdot N$ bits, or $0.7 \cdot N$ decimal digits, to compute x_N exactly as a quotient of integers solely from the recurrence; the time taken grows like $N^{ç}$ for some constant ç between roughly 2.5 and 3 depending upon how multi-word multiplication is implemented.

In the absence of roundoff the sequence $\{x_n\}$ generated by the recurrence $x_{n+1} := Œ(x_n, x_{n-1})$ would approach a limit $L(x_n, x_{n-1})$ that is a discontinuous function of any two consecutive members of the sequence. We find $L(3, 3) = 3$; otherwise $L(x, y) = 5$ on the hyperbola whose equation is $(x - 8)y + 15 = 0$; and $L(x, y) = 100$ everywhere else. Although x_N for any fixed large N is a nearly-constant bilinear rational function of (x_1, x_0) , it spikes violently as (x_1, x_0) moves across the hyperbola very near which the function takes both 0 and ∞ as values. Without knowing in advance where to look, a random search for such a spike will almost never find it.

MATLAB Plot of x_{80} as a function of x_2 near 76/17 for fixed $x_1 = 17/4$:



The horizontal axis runs over $-1e{-}98 \le x_2 - 76/17 \le +1e{-}98$. No floating-point numbers x_2 lie in that interval.

$x_{80} = 5$ when $x_2 - 76/17 = 0$, and $x_{80} = \pm\infty$ when $x_2 - 76/17 = -2.241902748434\ldots e{-}100$.

§6: A Smooth Surprise

Examples like J-M. Muller's seem pathological and thus largely irrelevant to people who intend to compute only well-behaved smooth functions of their data, not spiky functions like x_{80} nor discontinuous functions like limit L(x, y) above. The next example may surprise those people.

It is a relatively simple function G(x) which takes the value 1 for all real arguments x . That $G(x) \equiv 1$ has been confirmed instantly by an automated algebra system DERIVE 4.1 from the Soft Warehouse Inc., Honolulu HI, run under DOS on an Intel i386-based PC (25 MHz., 15 MB of DRAM), so the confirmation cannot be very complicated. However, this confirmation assumes arithmetic with real numbers to be performed always exactly. If arithmetic is performed approximately but sufficiently accurately, the computed value of G(x) is almost always zero instead of 1 ! This happens for all sufficiently large arithmetic precisions, and not because gargantuan numbers cancel; none of these need arise during the computation.

When G(n) is evaluated at $n = 1, 2, 3, \ldots, 9999$, say, in floating-point arithmetic of any ample preassigned finite precision, the computed values of G(n) are almost always zero. There are exceptions. When the arithmetic rounds every operation to 24 sig. bits in conformity with IEEE Standard 754 (corresponding to Java's `float` arithmetic) then $G(1) = G(7) = G(2048) = 1$; but otherwise 9996 computed values $G(n) = 0$. All 9999 computed values $G(n) = 0$ when arithmetic is rounded to 53 sig. bits (Java's `double`) or to 64 sig. bits (IEEE 754's Double-Extended) on a Pentium. The HP-28S and other Hewlett-Packard programmable calculators that round their decimal floating-point arithmetic correctly to 12 sig. dec. get $G(2) = G(42) = 1$ but otherwise compute 9997 values $G(n) = 0$. DERIVE's approximate arithmetic is neither binary nor decimal floating-point but a kind of rational arithmetic whose "Precision", though specified roughly in sig. dec., is enforced by truncating continued fractions somehow. When requested to compute G(n) with 64 sig. dec. of Precision, DERIVE got 9998 values $G(n) = 0$ and $G(159) = 1$. A request for 72 sig. dec. got $G(133) = G(4733) = G(4862) = G(4888) = 1$ and only 9996 values $G(n) = 0$. A request for 84 sig. dec. got all 9999 values $G(n) = 0$.

Why does G(n) behave so perversely? G(x) is defined by a short program like the following:

Real variables x, y, z ;
Real Function T(z) := { If z = 0 then 1 else (exp(z) – 1)/z } ;
Real Function Q(y) := | y – √(y^2 + 1) | – 1/(y + √(y^2 + 1)) ;
Real Function G(x) := T(Q(x)^2) ;
For Integer n = 1 to 9999 do Display{ n , G(n) } end.

Absent roundoff, $Q(y) \equiv 0$ for all real (but not all complex) numbers y . If $y \geq 1$, roundoff bestows upon Q(y) a tiny value of the order of a rounding error in y . It is hardly ever zero.

Absent roundoff, T(z) is a smooth infinitely differentiable function of z ; in fact

$$T(z) = \int_0^1 e^{zw}\, dw = \sum_{m\geq 1} z^{m-1}/m! \; .$$

But roundoff causes the one-line program for T(z) to malfunction when z is tiny. *In extremis*, when z is tinier than a rounding error in 1 but not zero, the computed exp(z) rounds to 1 and then the computed T(z) vanishes, as does G(n) . Unless n vastly exceeds 9999, inaccuracy in

program G comes entirely from its inaccurate subprogram T ; and increasing arithmetic's precision uniformly everywhere in the program almost never cures G's inaccuracy.

The trouble with T(z) and G(x) is not their intended behavior but rather the unfortunate (*i.e.*, numerically unstable) way they have been computed from expressions programmed correctly for exact arithmetic though incorrectly for rounded arithmetic. Two questions are brought to mind:

How can distress caused by roundoff be diagnosed reliably? How can it be cured?

"Aha!" says an observer; "The distress is caused by massive cancellation." No; cancellation *never* causes numerical inaccuracy. After cancellation Q(y) is rightly tiny unless y is huge; and soon we shall see a cure for inaccuracy in T(z) despite massive cancellation in (exp(z) – 1) .

In general, cancellation is at worst the Bearer of Bad Tidings, namely that prior rounding errors discarded digits whose absence now is regretted. Some computations, like root-finding, succeed because of massive cancellation. Other computations can go utterly awry with no subtraction, no cancellation, as we shall see in §10's example. Cancellation needn't signify numerical distress.

"Aha!" says another observer; "The distress is caused by a tiny divisor." Not necessarily, though T(z) does suffer from a tiny divisor z because it is the wrong tiny divisor, as we shall see soon. Tiny divisors bode ill only if, in producing huge quotients that later mostly cancel, they make us wish divisors and quotients had been computed more accurately. No huge quotients occur in T . Other computations can go utterly awry with no divisions, no small divisors, as we shall see in §10's example. Small divisors needn't signify numerical distress.

Someone without access to the formula for T(z) may try to narrow suspicion to it by rerunning the program with roundoff redirected Up and again redirected Down , and then comparing the three results. The outcome depends upon how exp(…) is implemented. If exp(z) rounds Up to 1.000…001 , computed values of T(z) and G(n) will diverge enough to arouse suspicion, and this usually happens when n is big enough: 2 is big enough for 24 sig. bits, 3028891 for 53, 5e9 for 64. Otherwise, if the implementation of exp(z) begins as many do with a test like …

If |z| < RoundoffThreshold/4 then Return(1.0) else … ,

then redirected roundings may change nothing, and then miscomputed values G(n) = 0 must be almost always too consistent to arouse suspicion about their accuracy.

There are ways to compute T(z) = (exp(z) – 1)/z accurately enough. They figure in financial calculations. Here is an easy way, albeit tricky:

```
Real Function  T(Real  z) :
      T := exp(z) ;                          …  rounded,  of course.
      If  (T = 1)  Return( T ) ;             …  when  |z|  is very tiny.
      If  (T = 0)  Return( T := –1/z ) ;     …  when  exp(z)  underflows.
      Return( T := ( T – 1 )/log(T) ) ;      …  in all other cases.
      End  T .
```

This way works because the computed value of `exp(z)` is actually exp(z+ß) where |ß| amounts to at most a rounding error or two in values near 1 . Consequently the value computed for `T(z)` is actually very nearly T(z+ß) rounded off; its relative error is roughly ß·T'(z)/T(z) , which can

easily be proved to lie between 0 and ß by differentiating the formula $T(z) = \int_0^1 e^{zw}\,dw$. In effect the program's possibly tiny divisor `log(T)` compensates for the rounding error in `exp(z)` preceding a possibly massive cancellation in (exp(z) – 1) provided the arithmetic, regardless of its precision, rounds the program's difference `(T – 1)` properly and delivers its `log(T)` to near-full working relative accuracy. Then substituting this program `T(z)` for the one-line expression given initially to define T(z) produces the correct G(n) = 1 for all n not too enormous.

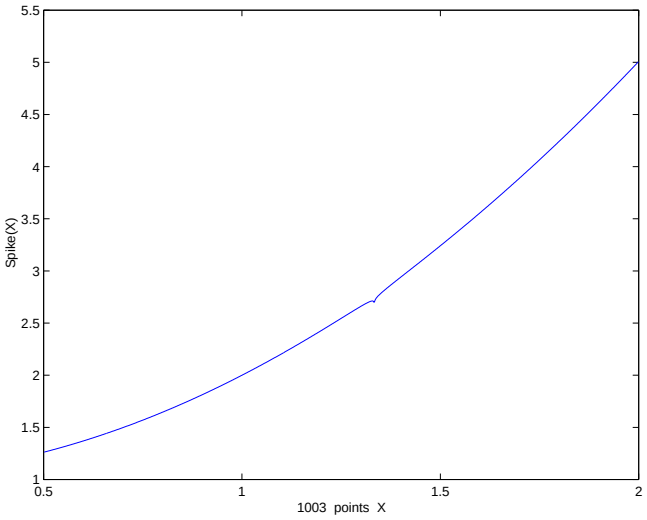
Ironically, if multi-precision Interval Arithmetic were used naively to compute G(n) either from its initial formula or from its accurate program, the results at every precision would be intervals so excessively wide as could not distinguish the accurate program from the inaccurate one.

This chillingly simple example G(n) undermines confidence in all five of the mindless schemes to which these notes are devoted, and casts deserved doubt upon oft-uttered glib diagnoses of "Cancellation" and "Small Divisors" as concomitants of numerical distress. Still, fairness requires an admission that this example is atypical. It was contrived to thwart the first and fifth schemes, namely repeated recomputation with ordinary or Interval Arithmetic of ever increasing precision. Numerical distress due solely to roundoff is relieved too often by increased precision for its use when available to be deterred by this example despite its worrisome simplicity.

§7: Some More Spikes, and MATLAB's log2

Some spikes are deserved; others are accidents of roundoff. Both kinds have been difficult to detect. Here is a deceptively simple looking function whose graph deserves a spike:

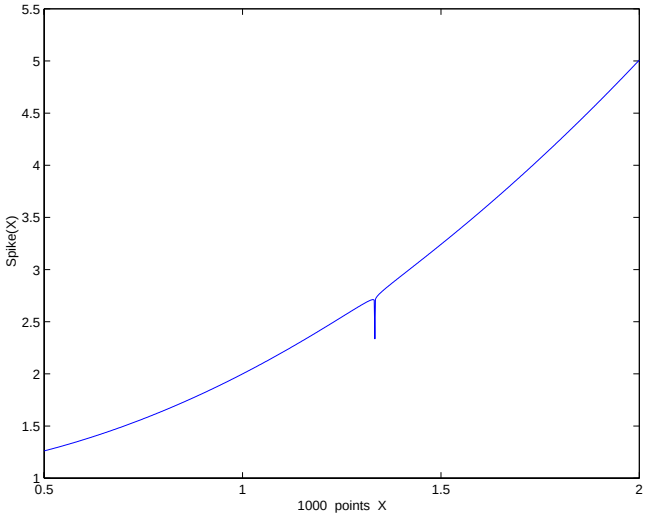
$$\text{Spike}(x) := 1 + x^2 + \log(|\, 1 + 3\cdot(1-x)\,|)/80\ .$$



But where is it?

The plot above was obtained from 1003 points $x = 1/2 + n/669$ for $n = 1, 2, 3, \ldots, 1003$. The plot below was obtained from 1000 nearby points $x = 1/2 + n/666$ for $n = 0, 1, 2, \ldots, 999$:

$$\text{Spike}(x) := 1 + x^2 + \log(|\, 1 + 3\cdot(1-x)\,|)/80\ .$$

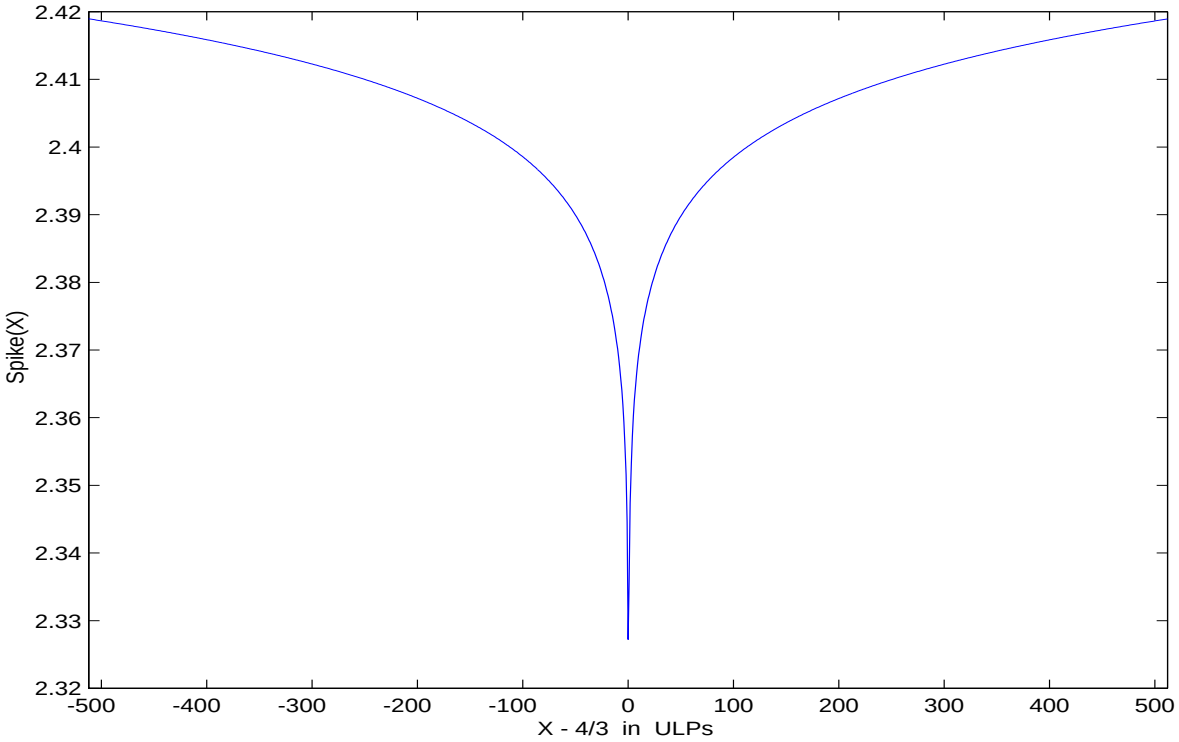


Why is the spike so short?

Since $\text{Spike}(4/3) = -\infty$ we expect the spike to plunge down into an abyss, but it doesn't. Below is a plot of Spike(x) at the 1025 8-byte floating-point arguments x adjacent to $x = 4/3$, which is not one of them. The argument nearest 4/3 differs from it by (1/3)ulp ; an ulp is a Unit in the Last Place the arithmetic carries. Thus Spike(x)'s computed values are always finite. Its spike is too thin and shallow to be discovered by uninformed random search unaided by luck.

Ancient Greeks used to say " Better to be lucky than clever."

Spike(x) := 1 + x^2 + log(| 1 + 3·(1–x) |)/80 .



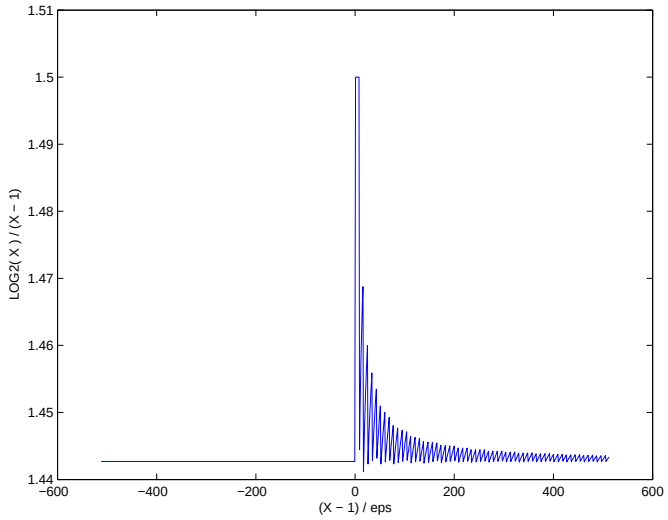
Here 1.333333333333220 ≤ x ≤ 1.333333333333447 . The spike is barely discernible much farther from x = 4/3 .

Some undeserved and unwanted spikes, accidents of roundoff in software or firmware, have eluded discovery and diagnosis for many years. For example, no spike should mar the graph of

$$\log_2(x)/(x-1) = \left(1 - (x-1)/2 + (x-1)^2/3 - (x-1)^3/4 + \dots \right)/\log(2)$$

plotted at arguments x near but not 1 . However, here is a spike that has persisted since 1994 in three MATLAB versions 4.2 to 6.5 on all my computers (this graph came from an IBM PC) :

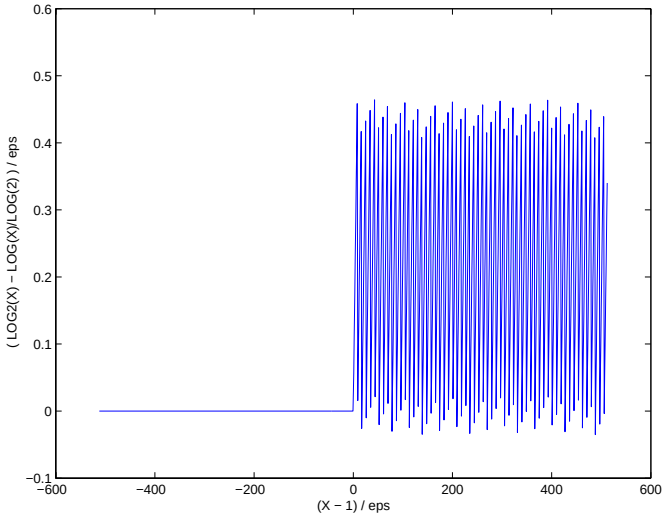
MATLAB's $\log_2(x)/(x-1)$



Here and in the next three graphs 0.9999999999998863 ≤ x ≤ 1.000000000000114 .

MATLAB's eps = $2^{-52} \approx$ 2e–16 is an ulp of 8-byte numbers between 1 and 2 . The graph plots log2(x)/(x–1) at 1536 consecutive floating-point arguments x straddling 1 . A spike exposes errors as big as 4% in MATLAB's log2(x) at arguments x very slightly bigger than 1 . (Thrice bigger errors occur on some other computers.) Why do 48 sig. bits get lost?

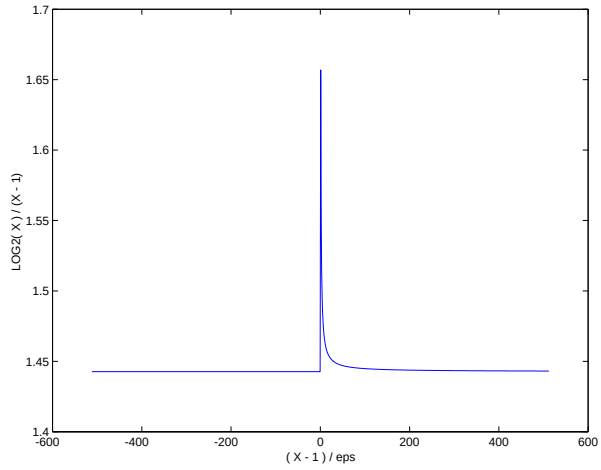
The next graph plots (log2(x) – log(x)/log(2))/eps . It would be zero in the absence of roundoff.

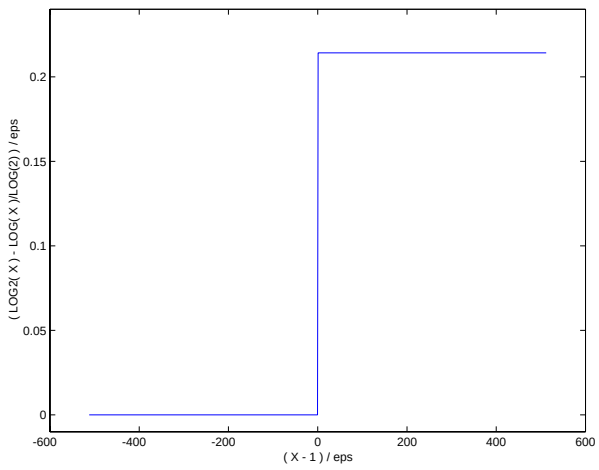


Since |log2(x)|/eps should not exceed 1478 , the errors plotted above should not exceed 3000·eps < 7e-13 . Instead huge errors' amplitudes suggest that MATLAB's log2(x) comes from a formula that approximates $\log_2(f\cdot\sqrt{2})$ over $1/2 \le f < 1$, after which $\log_2(x/\sqrt{2}) + 1/2$ was expected to deliver log2(x) for x slightly bigger than 1 . Delivered instead were the rounding errors in $\log_2(x/\sqrt{2})$ after the rest of it cancelled with +1/2 . Better results would be obtained from that formula if it were shifted to approximate $\log_2(f)$ over $1/\sqrt{2} \le f < \sqrt{2}$.

The foregoing graphs obtained from MATLAB 6.5 on a Wintel PC exhibit what appears at first sight to be a kind of raggedness often associated with misbehavior induced by roundoff. Closer inspection reveals regularities. In general, raggedness and roundoff do not always accompany each other. Here are plots of the same expressions at the same arguments as before but now by MATLAB 5.2 on an Apple iMac (Power PC G3 processor):

LOG2(X)/(X – 1) (LOG2(X) – LOG(X)/LOG(2))/eps





Instead of oscillation we see a smooth spike and a single jump. Are they likely to be attributed to roundoff by someone who is unsure about how the functions plotted are supposed to behave? Analogous graphs plotted on an old Apple Quadra 950 (Motorola 68040 processor) show the same smooth spike and single jump except for noticeably smaller amplitudes. These differences should suggest roundoff as the culprit to anyone who reran exactly the same computation with exactly the same data on those different computers. How common is such obsessive repetition?

Something else about all foregoing spiky examples is uncommon: We (think we) know which spikes are deserved and why. More often, albeit still too rarely, a numerical result comes under suspicion because of some anomaly discerned, perhaps faintly, before a spike's existence is suspected. An example of such an anomaly is the pimple in the first graph of Spike(x) plotted above,— the graph with no spike. Many an anomaly like that emanates from a program to whose source-text full access is denied. An example is MATLAB's log2(…) ; it is a "built-in function" whose algorithm cannot be displayed by MATLAB's user. And when a program's source text can be displayed, as can Spike(…)'s, "full access" may overstate how much of the program will be comprehended. Let's not embarrass the educational establishment by asking …

What percentage of college graduates
who have passed obligatory Math. courses
can supply correct values for log(1) and log(0) ?

Consider instead the predicament faced by the user of a partially opaque program after it produces a possibly dubious result from ostensibly innocuous data. What can this user do to dispel some of the fog of numerical uncertainty? If recompilation is not an option neither are multi-precision nor Interval arithmetic, nor randomized rounding on a typical PC. Two possible options remain:

One possibility is repeated execution with slightly altered input data. In general such alterations would pose a challenge: Alter too little and nothing would change; alter too much and results could change too much to convey information of diagnostic value. For our examples, after their spikes have been located, altering the data by an ulp or two will provide food for thought.

Another possibility is repeated execution with redirected rounding. This can be accomplished in MATLAB 6.5 on a PC by invoking the command "`system_dependent('setround', r#)`" with

`r#` = `+inf` to round *Up*, towards +∞ , or
= `-inf` to round *Down*, towards –∞ , or
= `0` to round *Towards Zero* , or
= `0.5` to round *To Nearest* , the default.

Let's try all possibilities. Here are some results computed by MATLAB 6.5 on a Wintel PC :

Spike(x) := 1 + x^2 + log(| 1 + 3·(1–x) |)/80

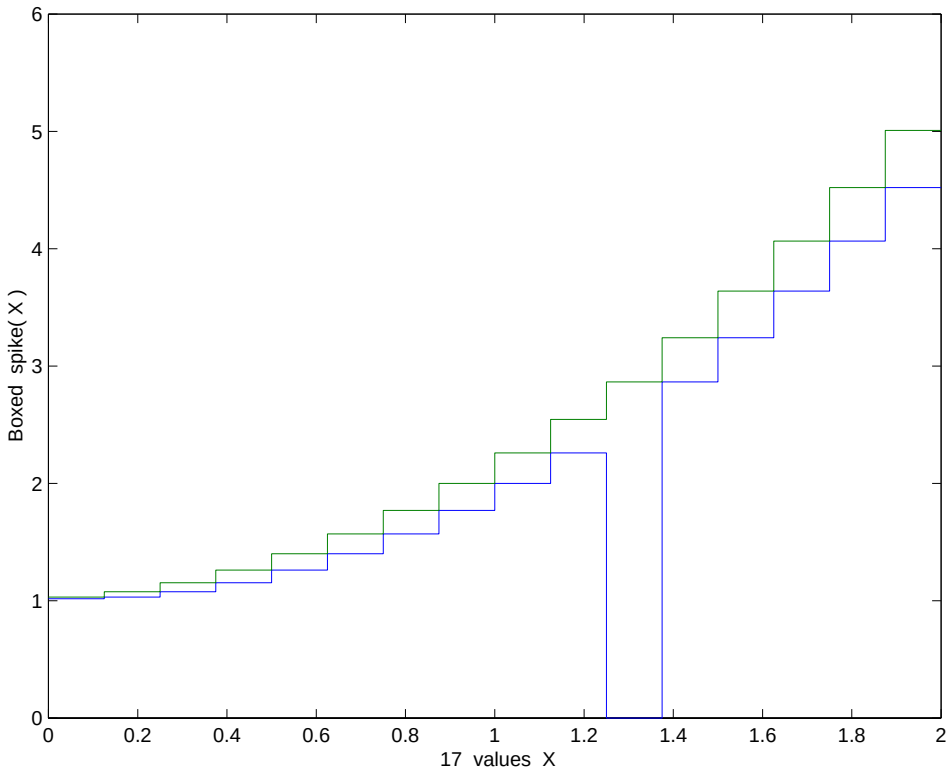
Rounding …	x = 1.333333333333333**7**	1.333333333333333**2**	x = 1.333333333333333**4**
To Nearest	2.**344560789927811**	2.**327232110413813**	2.**335896450170813**
Towards +∞	2.**344560789927812**	2.**327232110413813**	2.**335896450170813**
Towards –∞	2.**344560789927811**	2.**327232110413813**	2.**335896450170813**
Towards 0	2.**344560789927811**	2.**327232110413813**	2.**335896450170813**

Arguments x specified as x = 4/3 – `eps` , 4/3 and 4/3 + `eps` came out as shown because "4/3" is (4 – `eps`)/3 .

Redirected roundings inside Spike(…) have almost no effect upon its computed value. This *corroborates* (it doesn't *prove*) that Spike(…)'s spike at $x \approx 4/3$ is deserved. Changing the 17th sig. dec. of x changed the 3rd sig. dec. of Spike(x) , so its spike must be pretty sharp. However, too few numerical samples were plotted to hint at the spike's infinite depth.

Guy Steele has pointed out that Interval Arithmetic, properly implemented, can reveal a spike's depth with little effort by its user. "Properly implemented" includes, among many other things, a library program that searches a given domain and finds *all* extrema of a function specified by an expression rather than just by a program that can be executed but not read. Otherwise the function's range may be vastly overestimated. Here, simulating an implementation whose interval LOG(…) is perfect, is a plot of boxes that surely enclose the graph of spike(x) :

Boxed Spike(x) = $1 + x^2 + \log(| 1 + 3\cdot(1-x) |)/80$



Relatively few plotted points suffice to reveal the spike's existence. Is the spike's apparently infinite depth an artifact of Interval Arithmetic's pessimism discussed in §13 ? Locating so narrow a spike sharply enough to plumb its infinite depth persuasively requires a sufficiently dense plot feasible only if Interval Arithmetic is integrated with floating-point arithmetic of arbitrarily high run-time precision.

Whether Interval Arithmetic could reveal the anomaly in `log2(x)/(x-1)` to MATLAB's *user* is hard to say. The anomaly is a bug. Would it be inherited by Interval Arithmetic? More likely is the *programmer* of MATLAB's `log2(…)` to be aided in validating his program or exposing its defects with the aid of Interval Arithmetic, and then only if the specifications for his program's tolerable error are bug free. Validation is an interesting topic for another day.

This document concerns a software user's ability to track down a probable cause for suspicious behavior. Initially the user would not know that `log2(…)` is anomalous. The user will decide to test it only after a process of elimination has brought suspicion upon it. What test can he try?

MATLAB 6.5's `log2(x)/(x-1)`

Rounding …	x = 1 – eps/2	x = 1 + eps
To Nearest	1.442695040888963	1.5
Towards +∞	1.442695040888964	2
Towards –∞	1.442695040888963	0.5
Towards 0	1.442695040888963	2

Redirected roundings testify to a bug in MATLAB 6.5's `log2(…)` : Its rounding errors ruin all but the first few sig. bits of its value at arguments x barely bigger than 1 . Arguments x barely less than 1 produce values near 1/log(2) = 1.4426950408889634… as they should. This table can be sent (it has been) as convincing evidence of a bug to MATLAB's author. While awaiting a helpful response, MATLAB's user can substitute " `log(x)/log(2)` " for " `log2(x)` " in his arithmetic expressions unless he expects " `log2(2^n)` " to reproduce every integer n exactly unless over/underflow interferes. A slower more accurate program is my `lg2(x)` posted at
http://www.cs.berkeley.edu/~wkahan/LOG10HAF.TXT .

§8: An Old Hand Accuses Division

Many an Old Hand at floating-point computation will point to what causes trouble in §6's Smooth Surprise immediately; he will blame the tiny divisor z in

$$T(z) := \{ \text{If } z = 0 \text{ then } 1 \text{ else } (\exp(z) - 1)/z \} .$$

It is an instance of a hazard the Old Hand remembers well, namely conditional statements like

$$\dots \text{ If } x = y \text{ then } \dots \text{ else } \dots/(x-y) .$$

These used to malfunction routinely when the two predicates " x = y " and " x–y = 0 " had inconsistent boolean values on many computers and/or with some compilers in the 1970s. On CDC 6x00s division-by-zero could thwart T(z) , logic notwithstanding, unless " If z = 0.0 " were replaced by " If z·1.0 = 0.0 " . None of that happens now, at least not on machines that conform fully to IEEE Standard 754 (1985) as almost all do now. Many algorithms that used to malfunction mysteriously or dramatically, depending upon the hardware and/or compiler, now work about as well as they deserve. How well do they deserve to work, and who decides?

"Use every man after his desert, and who should 'scape whipping?" *Hamlet*, act II sc. ii .

The nearly universal adoption of IEEE 754 in the 1980s replaced previous fuzzy mental models of floating-point arithmetic by a sharper mathematical model from which reasonble expectations of computational behavior could more easily be inferred and proved, at least in principle. By enhancing computed results' predictability, IEEE 754 enhanced also their achievable quality.

For example take the revised program for T(z) above, which can be rewritten in one line thus:

$$T(z) := \{ \text{If } \exp(z) = 0 \text{ then } -1/z \text{ else if } \exp(z) = 1 \text{ then } 1 \text{ else } (\exp(z) - 1)/\log(\exp(z)) \} .$$

This version has the same division as before except for two extra rounding errors which, when z is too tiny, turn the quotient into *Roundoff*/*Roundoff* in the eyes of the Old Hand. And Interval Arithmetic evaluation of that quotient would confirm his fear of its indeterminacy. But we know now that its indeterminacy is illusory. Instead a rounding error in the numerator's exp(z) is offset by the same one in the denominator to produce an almost fully accurate quotient provided

$$\log(x) = (x-1)\cdot(1 - (x-1)/2 + (x-1)^2/3 - (x-1)^3/4 + \dots) , \quad \text{when } |x-1| < 1 ,$$

is computed as accurately as we have every right to expect nowadays, namely well within a unit in the last digit carried by the arithmetic.

Of course, the revised version of §6's T(z) is a trick.

"A trick used three times becomes a standard technique" (G. Pólyà).

A similar trick figures often in financial calculations involving mortgages, bonds, leases and loans. Frequently they entail computation of the *Future Value* function

$$FV(N, x) := \{ \text{If } x = 0 \text{ then } N \text{ else } ((1+x)^N - 1)/x \}$$

in which the number of payment periods |N| is a moderately big integer, and the periodic interest or discount rate x = i/100 , expressed as a fraction instead of a percentage i , is fairly small in magnitude unless usury is in force. If |x| is too tiny the foregoing expression for FV can lose all its sig. digits to roundoff as did the original T(z) above, and in the same way obvious to the Old Hand. This is not the place to explain how the trick rescues FV . Instead, to tantalize the Old Hand, here is a simpler revised (unless the compiler "optimizes" parentheses away) expression for the same function:

$$FV(N, x) := \{ \text{If } (1+x) = 1 \text{ then } N \text{ else } ((1+x)^N - 1)/((1+x) - 1) \} .$$

To Old Hands an expression with two extra rounding errors in its divisor seems more likely than the original to lose all digits carried when |x| is tiny, yet nowadays it can be proved to lose at most half the sig. digits carried by arithmetic provided integer |N| is not immoderately big.

This phenomenon, losing at most half the digits carried to roundoff, occurs surprisingly often. Half full, or half empty? Some applications cannot tolerate so great a loss when it carries away anticipated properties like smoothness, monotonicity and symmetry too. Other applications, like least-squares linear regression to statistical data in the life and social sciences, need no more than seven sig. dec. in their results and achieve that accuracy fastest by carrying over twice as many sig. dec. during their arithmetic. Either way, the phenomenon raises doubts about glib diagnoses of "Small Divisors" and "Cancellation" as invariable concomitants of numerical distress.

Programmers who still fear division can compute FV well for moderately big positive integers N without any division at all, and with about twice as much work as would be required to compute $(1+x)^N$ alone by means solely of multiplications and additions. The algorithm's derivation via *Divided Differences* is left as an exercise.

There is just one reason to fear floating-point division: It can be slow. A hardware designer, after noticing how many fewer divisions occur than multiplications and add/subtractions, may have "optimized" his design in a way that causes divisions to run too slowly.

Unless a programmer loses his nerve, he need no longer fear that Division-by-Zero will derail his program. For instance, *Secant Iteration* solves a real or complex equation $f(z) = 0$ for a real or complex scalar unknown z by generating a sequence of presumably improving guesses

$$x_{n+1} := x_n - (x_n - x_{n-1})\cdot(f(x_n)/(f(x_n) - f(x_{n-1}))) .$$

The program reacts to an ∞ produced by Division-by-Zero (since 0/0 and $0\cdot\infty$ are ruled out by prior tests) the same way it reacts to a wildly aberrant x_{n+1} caused any other way: Replace an aberrant x_{n+1} by another guess moderated by the history of recent iterations.

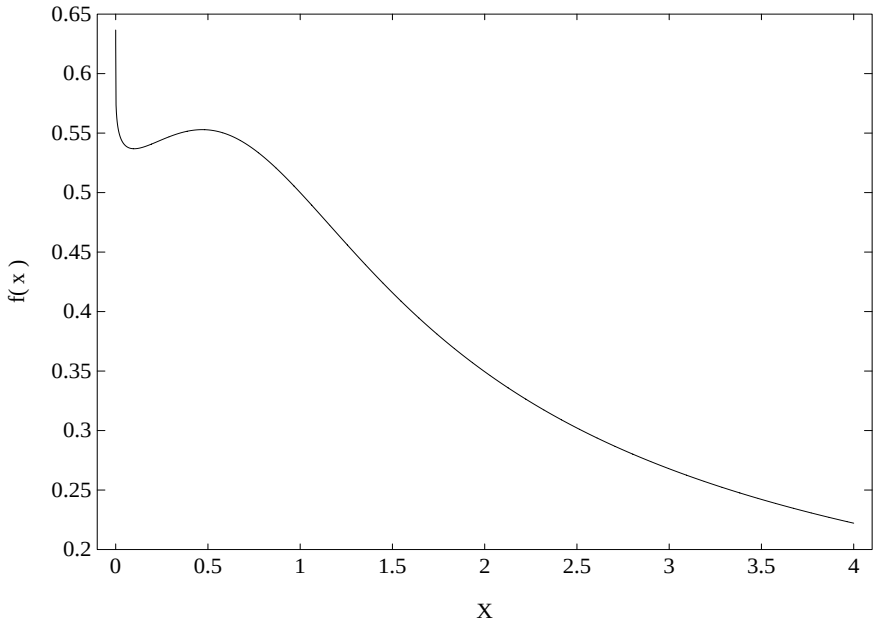
Some divisions by zero must be averted. The ways we did that three decades ago are no longer the only ways. Moreover, newer ways can produce better results more easily than older ways did.

Here is an example: For *all finite* $x > 0$ consider the function

$$f(x) := \{ \text{ if } x < 1 \text{ then } -\arctan(\log(x))/\arccos(x)^2$$
$$\text{else if } x = 1 \text{ then } 1/2$$
$$\text{else } \arctan(\log(x))/\operatorname{arccosh}(x)^2 \ \} .$$

It has a smooth graph. It is smooth as x passes through 1 because this $f(x)$ has a convergent Taylor Series there that will be exhibited in a moment. As $x \to 0+$ the graph rises to $f(0) = 2/\pi$ sharply because $f'(0) = -\infty$. A graph computed from the foregoing formula for $f(x)$ appears below. It looks perfectly smooth as x passes through 1 , but appearances deceive. Actually, old 386-MATLAB running on a PC lost 26 of the 53 sig. bits it carried during the computation of $f(x)$ at arguments x next to 1 . The graph's resolution is too coarse to reveal the loss.

f(x) plotted by 386-MATLAB v. 3.5m (1992)



To expose 386-MATLAB's errors we must first compute $f(x)$ correctly around $x = 1$ using its series. A brief look at nine terms of its Taylor series

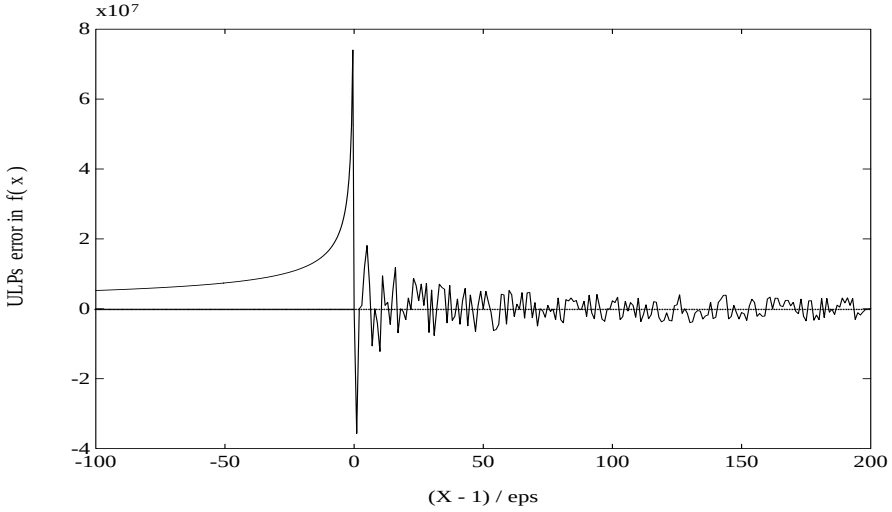
$$f(1+z) = \frac{1}{2} - \frac{1}{6}z - \frac{1}{20}z^2 + \frac{124}{945}z^3 - \frac{8221}{113400}z^4 - \frac{46969}{1247400}z^5 + \frac{948249251}{10216206000}z^6 - \frac{208838923}{3831077250}z^7 - \frac{14025530287}{521026506000}z^8 + \ldots$$

computed by *MAPLE*® persuades us of two things:

- At most a few leading terms involve integers small enough to be computed by hand.
- The series converges slowly; its radius of convergence is 1 since $f'(1 + (-1)) = -\infty$.

Thus the series serves to check any other program's accuracy only in a narrow neighborhood of $f(x)$'s removable singularity at $x = 1+z = 1$, where $f(1) = 1/2$. Below is a graph of the error, the difference between 386-MATLAB's $f(x)$ and its series, plotted at 401 consecutive floating-point arguments x running from $1 - 100\cdot\texttt{eps}$ to $1 + 200\cdot\texttt{eps}$, where $\texttt{eps} = 2^{-52} \approx 2.2_{10}{-}16$.

ULPs of Error in 386-MATLAB 3.5's f(x)



The error is measured in ULPs (Units in the Last Place) of the values of $f(x)$, whence an ULP = { if $x > 1$ then $\texttt{eps}/2 \approx 1.1/10^{16}$ else $\texttt{eps}/4$ }. Note the scale ($\times 10^7$ ULPs) of the vertical axis. The worst error is 74055679.7 ULPs at $x = 1 - \texttt{eps}/2$.

What caused those errors? To assist diagnosis, we reran the computation of 386-MATLAB's $f(x)$ in Directed Rounding Modes (Scheme 2 in §4). Results tabulated here expose hypersensitivity to roundoff enough to arouse suspicion but not yet enough for conviction.

386-MATLAB's $f(x)$ computed with Directed Roundings

Direction	$f(1-\texttt{eps}/2)$	$f(1+\texttt{eps})$
To Nearest :	0.5000000041109161	0.4999999960408469
To Zero :	0.5000000041036400	0.5000000065775587
To $+\infty$:	0.5001221042336121	0.9999999552965182
To $-\infty$:	0.5000000041036401	0.5000000065775587

To sharpen the focus of diagnosis, we reran separately the subprograms used in 386-MATLAB's $f(x)$ with the same inputs as revealed the hypersensitivities just exposed above. 386-MATLAB's `log` and `atan` were almost indifferent to directions of rounding, but the table below shows how its `acos` and `acosh` turned out both hypersensitive and wrong in almost half their sig. bits. Suspicions aroused by evidence of hypersensitivity were confirmed by comparison with correctly computed values of arccos(1–eps/2) and arccosh(1+eps). Note that the errors in MATLAB's `acos` and `acosh` were far tinier than the variations caused by redirected roundings. Actual errors had to be determined the hard way: Compute correct values somehow and then compare.

386-MATLAB's `acos` and `acosh` with Redirected Roundings

Direction	$10^8 \cdot \texttt{acos}(1-\texttt{eps}/2)$	$10^8 \cdot \texttt{acosh}(1+\texttt{eps})$
To Nearest :	1.490116113259023	2.107342433887993
To Zero :	1.490116113269865	2.107342411683533
To $+\infty$:	1.489934203216439	1.490116152691456
To $-\infty$:	1.490116113269865	2.107342411683533
Correct value:	1.49011611938476564	2.10734242554470155

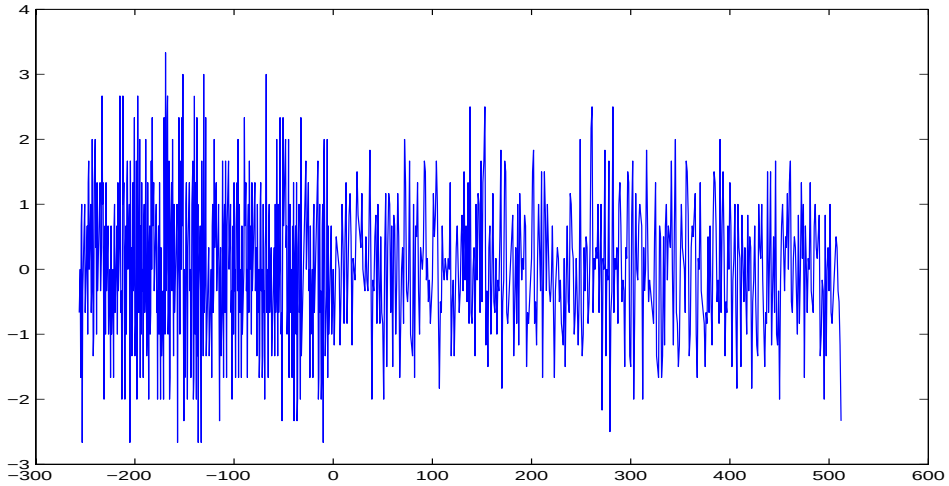
MATLAB's `acos` is a "built in" function whose source-text has been inaccessible to MATLAB's users for decades. MATLAB 3.5 (1991) on 680x0-based Macintoshes, and MATLAB versions 4.2 (1997) and later on Macs and PCs have enjoyed accurate implementations of `acos`.

MATLAB's `acosh` has been implemented inaccurately as an .m file, and therefore accessible and alterable, from early versions in 1984 until version 5.2 (1998). Accurate implementations of `acosh` were built into versions 5.3 (1999) and later on PCs.

When `acos`, `acosh`, `atan` and `log` are each accurate within less than an ulp, the formula for $f(x)$ given above transliterates into a program whose error can reasonably be expected never to

exceed a few ulps. Such expectations are consistent with this graph of the program's error plotted at 1025 consecutive floating-point arguments x between 1 – 256·eps and 1 + 512·eps :

ULPs of Error in PC MATLAB 6.5's *f*(x)



Compare this graph's and the previous graph's vertical scales. Here satisfactorily small ragged rounding errors confirm that division by tiny divisors need not cause numerical distress if they are correlated properly with their numerators. But now this and previous graphs raise worrisome questions about the diagnosis and persistence of erroneous numerical software. …

Why have erroneous implementations of fundamental functions like `acos`, `acosh` and `log2` persisted in MATLAB for so many years? Have their errors escaped notice by MATLAB's many myriads of users? It's possible. I noticed these errors only after slightly excessive discrepancies among results from old and new versions of MATLAB on PCs, Power Macs and my old Quadra aroused my curiosity during the preparation of numerical exercises for students. Not everyone gets an opportunity to compare numerical results from so many sources. Not everyone wants one.

With one thermometer you always know the temperature; with two of them you rarely know it.

The longevity of inaccuracies in numerical software by and for numerical adepts has ominous implications: Numerical software does not have to be very complicated to be difficult to debug by experts, practically impossible to debug by amateurs. Numerical software from numerically naive programmers, no matter how competent they are in other fields, must often be much less accurate than programmers and users believe. How often? How much? How would we know?

Another possibility, *Unnecessarily Low Expectations*, may explain the persistence of erroneous numerical software. Old Hands at numerical computation may recall that in the 1950s floating-point arithmetic's errors were generally deemed impossible to analyze. John von Neumann had recommended against building floating-point into computers in 1947. But by 1960 numerical analysts, particularly James H. Wilkinson, were promulgating explanations for floating-point errors under the heading of *Backward Error-Analysis*. It went like this:

Many numerical programs are hypersensitive to roundoff for at least some data if not all. Some are deemed *Numerically Stable* when their results are scarcely worse than if their data had been perturbed by a few ulps first and then computation had been performed exactly without roundoff. For example, the solution of matrix equations $A·Z = B$ by *Gaussian Elimination with Pivotal Exchanges* is numerically stable in this sense except for pathological cases. Eigensystems of all symmetric matrices and of all but pathological nonsymmetric matrices can be computed by stable algorithms replacing a plethora of unstable algorithms advocated in the literature before 1960. A small *Residual* is typical of algorithms stable in the backward sense: Even if wrong, the matrix X by which Gaussian Elimination approximates the solution $Z = A^{-1}·B$ almost always has a small residual $A·X – B$ satisfying an inequality like

$$||A·(X – Z)|| \;=\; ||A·X – B|| \;\le\; \gamma·n^{3/2}·\varepsilon·(\,||A||·||X|| + ||B||\,)$$

wherein γ is a moderate constant, n is the dimension of A, and ε is a rounding error threshold like MATLAB's `eps`. "Almost always" allows for pathological exceptions like matrices A so nearly singular that Gaussian Elimination may well be thwarted for lack of a nonzero pivot.

The success of Backward Error-Analysis at *explaining* floating-point errors has been mistaken by many an Old Hand as an *excuse* to do and expect no better. Since MATLAB's `log2(x)` computes $\log_2(x·(1 + æ))$ for some unknown $|æ| < \mathtt{eps}$ he could deem it numerically stable in the sense of Backward Error-Analysis. Likewise for old `acos(x)` $\approx \arccos(x·(1 + æ'))$ and old `acosh(x)` $\approx \operatorname{arccosh}(x·(1 + æ''))$. To tolerate such errors for doctrinal reasons would be illogical, unnecessary, and pernicious. Illogical because Backward Error-Analysis explains but does not excuse. Unnecessary because accurate implementations of those functions for IEEE 754 have been available for decades on Macs and PCs; other computers could use the Math. library released in the 1980s with 4.3 BSD Berkeley UNIX and now refined and promulgated for use with Java as *fdlibm*, the freely distributed math. library maintained by a few U.C. Berkeley graduates now working for Sun Microsystems. MATLAB uses fdlibm nowadays.

Tolerating unnecessary backward errors in the math. library is pernicious in so far as it obstructs the numerical removal of mathematically removable singularities. Our example $f(x)$ above will illustrate what goes wrong. Suppose log, arccos and arccosh were implemented no better than a mistaken Old Hand might expect; suppose (oversimplified) that their implementations were $\mathrm{LOG}(x) = \log(x·(1 + æ))$, $\mathrm{ACOS}(x) = \arccos(x·(1 + æ'))$ and $\mathrm{ACOSH}(x) = \operatorname{arccosh}(x·(1 + æ''))$ wherein $|æ| < \varepsilon$, $|æ'| < \varepsilon$ and $|æ''| < \varepsilon$:= (roundoff threshold). The tiny perturbations æ, æ' and æ'' are accidents of roundoff and therefore uncorrelated for all we know. They would induce uncertainties (bounds upon errors) amounting roughly (oversimplified) to …

$$|\,\mathrm{LOG}(x) – \log(x)\,| \;\le\; \varepsilon\;,$$

$$|\,\mathrm{ACOSH}(x) – \operatorname{arccosh}(x)\,| \;\le\; \varepsilon·x/\sqrt{(x^2 – 1)}\;, \qquad \text{and}$$

$$|\,\mathrm{ACOS}(x) – \arccos(x)\,| \;\le\; \varepsilon·|x|/\sqrt{(1 – x^2)}\;.$$

The oversimplifications affect the first two inequalities when $|\log(x)|$ is huge, and the last two when $|x – 1|$ is not much bigger than ε, but neither of these cases will matter to what follows.

The obvious implementation of function $f(x)$ as a program f(x) looks like this:

$$\begin{aligned} \mathrm{f}(x) := \{\ &\text{if } x < 1 \text{ then } –\mathrm{ATAN}(\mathrm{LOG}(x))/\mathrm{ACOS}(x)^2 \\ &\text{else if } x = 1 \text{ then } 1/2 \\ &\text{else } \mathrm{ATAN}(\mathrm{LOG}(x))/\mathrm{ACOSH}(x)^2\ \}\;. \end{aligned}$$

And this program produced all the graphs of f(x) displayed above. But if the math, library were so inaccurate as an Old Hand might mistakenly expect, the program's relative uncertainty at arguments x near 1.0 (but not so near that $|x - 1|$ is not much bigger than ε) would be roughly

$$| f(x) - f(x) |/f(x) \le \varepsilon\cdot(3 + 2/|x-1|) .$$

Thus, program f(x) could lose all but a few sig. bits, for all the Old Hand knew.

But the Old Hand knew how to avoid most of that loss by using N terms of the Taylor series

$$f(x) = \tfrac{1}{2} - \tfrac{1}{6}(x-1) - \tfrac{1}{20}(x-1)^2 + \tfrac{124}{945}(x-1)^3 - \tfrac{8221}{113400}(x-1)^4 - \tfrac{46969}{1247400}(x-1)^5 + \tfrac{948249251}{10216206000}(x-1)^6 + \dots$$

when $|x-1| < \Theta_N$ for some suitably chosen small integer N and threshold Θ_N. His program *f*(x) looked like this:

$$f(x) := \{ \text{ if } x \le 1-\Theta_N \text{ then } -\mathrm{ATAN}(\mathrm{LOG}(x))/\mathrm{ACOS}(x)^2$$
$$\text{else if } x \ge 1+\Theta_N \text{ then } \mathrm{ATAN}(\mathrm{LOG}(x))/\mathrm{ACOSH}(x)^2$$
$$\text{else (N terms of the series for } f(x) \text{ around } x = 1) \} .$$

By chosing threshold Θ_N properly he got his program's relative uncertainty down to roughly

$$| f(x) - f(x) |/f(x) \le \min\{ \varepsilon\cdot(3 + 2/|x-1|) , \; 2\varepsilon + 2\mu_N\cdot|x-1|^N \}$$

wherein μ_N is the magnitude of the coefficient of the first omitted term $\pm\mu_N\cdot(x-1)^N$ in the series. A rough estimate adequate for our purposes is $\mu_N \approx 1/20$, correct within an order of magnitude unless N is 37 or 81. Given N, a properly chosen Θ_N makes $\varepsilon\cdot(3 + 2/\Theta_N) \approx 2\varepsilon + 2\mu_N\cdot{\Theta_N}^N$, which happens nearly enough when $\Theta_N \approx (\varepsilon/\mu_N)^{1/(N+1)}$, and then the program's uncertainty peaks at roughly $2\varepsilon\cdot(1 + (\varepsilon/\mu_N)^{-1/(N+1)})$ when $x = 1 \pm \Theta_N$. The bottom line is this:

> The Old Hand's program *f*(x) lost almost 1/(N+1) of the sig. bits carried, whereas the obvious program f(x) loses just a few sig. bits nowadays. For similar accuracy, N had to be a substantial fraction of the number of sig. bits carried by the arithmetic.

Worse than this extra work is that the Old Hand's old ways imposed a superfluous burden upon the conscientious programmer, the one who tries to achieve fully accurate results over as wide a range of valid inputs as possible. This is the kind of person we hope is programming the design, construction and control of our transportation, our bridges and buildings, our chemical and pharmaceutical processes, *etc*. To burn such programmers out prematurely seems perverse.

§9: Repeated Randomized Rounding

Roundoff may be accidental but never random. A few rounding errors, probably one or two, did most of the damage to MATLAB's `log2(x)/(x-1)` tabulated in §7; and plots of the scaled error (log2(x) – log(x)/log(2))/`eps` exhibit regular rather than random behavior as `x` increases past 1 .

Roundoff is not random, yet mathematical models that pretend roundoff is random have their uses, and abuses. Such a model can be exploited by a numerical analyst during an error-analysis of her program which then she can test upon randomly sampled data for which accurate results are known or computable by a (presumably) slower program. If her program's actual errors too far exceed what her analysis led her to expect, she will know something is wrong with her program or her error-analysis of it. Diagnosis and correction can ensue. This is a good use of statistics.

Statistics get abused when an engineer, economist or … using that program relies naively upon a probabilistic estimate of the error in the program's output for his particular input data. Results from slightly different randomly perturbed data can be interpreted properly only in the light of an adequate understanding of both the function desired and the function computed by the program. How much should the desired function vary when its data is varied? If the computed function varies not much more than that, has it been shifted, as §6's G(x) got shifted, by far more than the variations? Has all perturbed data fallen on the wrong side of a step like the one in §7's last graph? Only error-analysis of the program can answer these last two questions. It's not mindless.

To gauge how badly roundoff affects a computed result, recomputation with perturbed rounding errors makes sense. Lest a few such recomputations produce biased results, randomly perturbed rounding errors seem appropriate. The hope is that the recomputed results' mean approximate the "True Result" that would be obtained if all rounding errors assumed their mean value (zero presumably), and that the recomputed results' variance can be used to estimate the probability of too large a gap between their mean and that True Result. This hope is misplaced.

Alas, randomized rounding has a fatal flaw. It has had to be rediscovered the hard way by well-intentioned advocates of recrudescing proposals ever since randomized rounding was first (so far as I know) proposed for the IBM 7030 "Stretch" in the late 1950s. The fatal flaw arises out of conditions inadequate to sustain two bedrock principles of Statistics:

- *The Law of Large Numbers*: As ever more independent unbiased random samples are drawn from a population, the samples' mean and variance will approach the population's.

- *The Central Limit Theorem*: If sufficiently many independent random variates have variances not too dissimilar, the variates' sum will be a random variate distributed approximately *Normally* with mean the sum of their means, and variance the sum of their variances.

Regardless of whether a large number of rounding errors contribute to each recomputed result, a large number of these results (each is one sample) must be computed to satisfy the Law of Large Numbers. But nobody is eager to spend a lot of time on a large number of recomputations.

Regardless of whether a large number of rounding errors contribute to each result, they are far less likely than men to be "all … created equal and independent" as asserted in Jefferson's first draft of the *Declaration of Independence*. Quite often a computed result's error is dominated by

so few as one or two rounding errors, as is MATLAB's log2(…) . Even the solution of a huge system of linear equations by Gaussian Elimination, incurring millions of rounding errors, is often perturbed predominantly by two rounding errors incurred in the first pass of elimination, especially when the system of equations is hypersensitive to roundoff because of "ill-condition". An example is exhibited below. In general, the one or two most injurious rounding errors are no easier to distinguish from the others than are pickpockets in a crowd at the racetrack. In short,

> Without an error-analysis, the Central Limit Theorem cannot be relied upon to estimate from the variance of a few recomputed results how likely is their mean to differ vastly from the True Result.

Our example is drawn from a scheme called "CESTAC" patented in Europe by J. Vignes in the late 1970s. It added +1, –1 or 0 chosen randomly to the last bit of every arithmetic operation. A better scheme circumvents Vignes' patent by randomly toggling UP or DOWN the directed rounding, mandated by IEEE Standard 754 for Binary Floating-Point Arithmetic, before each arithmetic operation. Stephan G. Popovitch seems to have done that in his version of CESTAC called "ProSolveur". It attempts to solve small systems of equations on an IBM PC using three randomly rounded computations to assay the accuracies of results. Then ProSolveur displays only those figures it "believes" to be correct. Of the many ways ProSolveur can go astray, only one of those we believe characteristic of CESTAC is exposed by the simple example exhibited below. Here is Prosolveur's welcoming screen:

```
================================================================================

      (c) Copyright 1987 - LA COMMANDE ELECTRONIQUE - Tous droits réservés

                                PROSOLVEUR

                  »  ProSolveur Version 1.1  par Stephan G. POPOVITCH  «

                          Frappez une touche pour continuer

================================================================================
```

ProSolveur's user enters algebraic equations symbolically to be solved numerically, indicates which symbols represent data (parameters) and which are unknowns ("inconnus" in French), and supplies values for the data. Then ProSolveur displays its results and the user's data and equations in two panels under headings of which only the following need be explained:

st	Entry's Status, p = parameter (datum), i = "inconnu" (unknown).
entrée	Initially, user's guess, if any; afterwards, Prosolveur's "résultat".
±(%)	Percentage uncertainty ProSolveur attributes to entré or résultat.
unité	Unit ($, Km, Kg, sec., …) if one has been chosen by the user.
résultat	Prosolveur's result displayed to as many sig. dec. as Prosolveur deems correct.
id	Line number identifying an equation or a comment beginning with "*".
fichier	Name of the disk(ette) file containing the line identified.

2x2 Problem submitted thrice to ProSolveur :

```
====================== variables ======================================
st   entrée          ± (%)       nom          unité       résultat
p      4194304.000               A
i                                x
p      4194303.000               B
i                                y
p      4194302.000               C
p            3.000               p
i                                X
i                                Y
i                                µ
i                                ß
====================== équations ======================================
id    équation                                              fichier
(1)                                                         << ? >>
(2)   A*x + B*y = 0                                         2X2
(3)   B*x + C*y = p                                         2X2
(4)                A*X + B*Y = 0                            2X2
(5)                B*X + C*Y = p                            2X2
(6)                              A*µ + B*ß = 0              2X2
(7)                              B*µ + C*ß = p              2X2
=======================================================================
no des équations du système à résoudre : 2:7
```

The command line beneath the panel above displays the id numbers of equations ProSolveur has been asked to solve, and also its warning messages if any. Our example, the simplest of many, exposes a failure mode by asking ProSolveur to solve six repetitive linear equations:

Results delivered by ProSolveur :

```
====================== variables ======================================
st   entrée          ± (%)       nom          unité       résultat
p      4194304.000               A
i         1.3E+007   1           x                         1.3E+7
p      4194303.000               B
i        -1.3E+007   1           y                        -1.3E+7
p      4194302.000               C
p            3.000               p
i         1.2E+007   1           X                         1.2E+7
i        -1.2E+007   1           Y                        -1.2E+7
i     12509610.504               µ                         1.2509611E+7
i    -12509613.487               ß                        -1.2509613E+7
=======================================================================
```

Since the determinant of the equations is $A\cdot C - B\cdot B = -1$, the ideal results (with no rounding error) for this ill-conditioned (hypersensitive to roundoff) linear system should be

$x = X = \mu = 3B = 125\underline{82909}$ and $y = Y = ß = -3A = -125\underline{82912}$.

ProSolveur's awesomely optimistic claims for the accuracies of its computed µ and ß indicate that the three "random" samples drawn by ProSolveur are far too few because they were drawn from a nearly *discrete* rather than continuously distributed population. The only rounding errors that matter in this computation are the two committed during the computation of $B\cdot(B/A)$, after which $C - B\cdot(B/A)$ mostly cancels to a very rough approximation of $1/A$ without generating any more error. There are only two ways to perturb each of those two crucial rounding errors, so the probability that both would repeat in all three samples is $1/16$. This is the probability that

ProSolveur will say that its error is too small to estimate, below 0.00001% , when actually its error is about 57000 times bigger than that for our example's calculation. If the Central Limit Theorem applied, the probability of such a big error would be not 1/16 but below 10^{-70000} .

More instances of ProSolveur's naively excessive optimism have been posted at
http://www.cs.berkeley.edu/~wkahan/improberr.pdf .

Applied mindlessly, recomputation with randomized roundings provides no reliable estimate of the probability of rare errors far larger than were anticipated. And without knowing whether such gross errors have occurred, how can their cost be predicted? What good is a probabilistic error estimate that cannot support the calculation of a price worth paying for insurance against the possibly calamitous cost of intolerably large errors? Even if a procedure produces probabilistic estimates that have turned out about right in numerous test cases each susceptible to confirmation, these ostensibly successful tests are misleading without a fair appraisal of the incidence of failure, and also of the existence of failure modes overlooked by the tests' designers.

Imponderable probabilities multiplied by imponderable costs of calamitous errors should not be allowed to paralyze us. Life is too full of imponderables. Probabilistic error estimates deserve to be trusted for any computation whose error-analysis vindicates them. But these are not mindless.

We have yet to consider the possibility that probabilistic rounding may have ruined a subprogram that was designed to work and works well only if rounding is performed as specified or expected by the programmer. Many of the math. library's "built-in" functions like `pow(x, y)` = x^y and `floor` are like that. If presumed utterly trustworthy (not being debugged), such subprograms' innards must be sheltered from schemes that rerun distrusted programs in altered rounding modes.

Even with pivotal exchanges, Gaussian Elimination is *not* utterly trustworthy. Scattered results from redirected roundings can be due to an ill-conditioned (nearly singular) matrix like the one presented above to Prosolveur, or else due to poor scaling or other rare accidents which Backward Error-Analysis explains but does not excuse.

Let us not confuse randomized rounding during recomputation with systematically redirected rounding during recomputation as exemplified in §7's tabulation of `log2(x)/(x-1)`. The two recomputation schemes have different purposes. Systematically redirected rounding explores the behavior of a software module suspected of hypersensitivity to roundoff at a particular set of input data. Such exploration is unlikely to prove anything with mathematical certainty. Instead such exploration is highly likely to strengthen suspicion if it is deserved, or to allay suspicion and guide searches for the source of a numerical anomaly elsewhere.

§10: Cancellation is Not the Culprit

Diagnosis would be easier if a program's numerical instability in the face of roundoff were visible to the naked eye in the program's text, unlike its computed results erroneous for some perhaps infinitesimal range of data. The "Usual Suspects", subtractions susceptible to cancellations and divisions susceptible to small divisors, were nearly exonerated during the explanation of §6's Smooth Surprise, the function $G(x) \equiv 1$ for which zero is almost always computed. Neither cancellation nor small divisors need be concomitants of numerical distress. Another suspect, arithmetic operations so numerous that their hordes of rounding errors threaten to overwhelm the desired result, can hardly ever carry out such a threat. Instead, floating-point computation may go utterly awry without …

- Subtractions (hence no cancellation) ,
- Divisions (hence no small divisors) , nor
- Very many arithmetic operations (hence no hordes of rounding errors).

Next is an example with only 256 arithmetic operations, and yet it loses all the figures carried by every commercially significant computer's floating-point hardware no matter how many sig. dec. or bits are carried. (The current maximum is below 36 sig. dec., 120 sig. bits). Worse, …

> Most numerical computations that go awry because of roundoff behave more nearly like this next example than like our others.

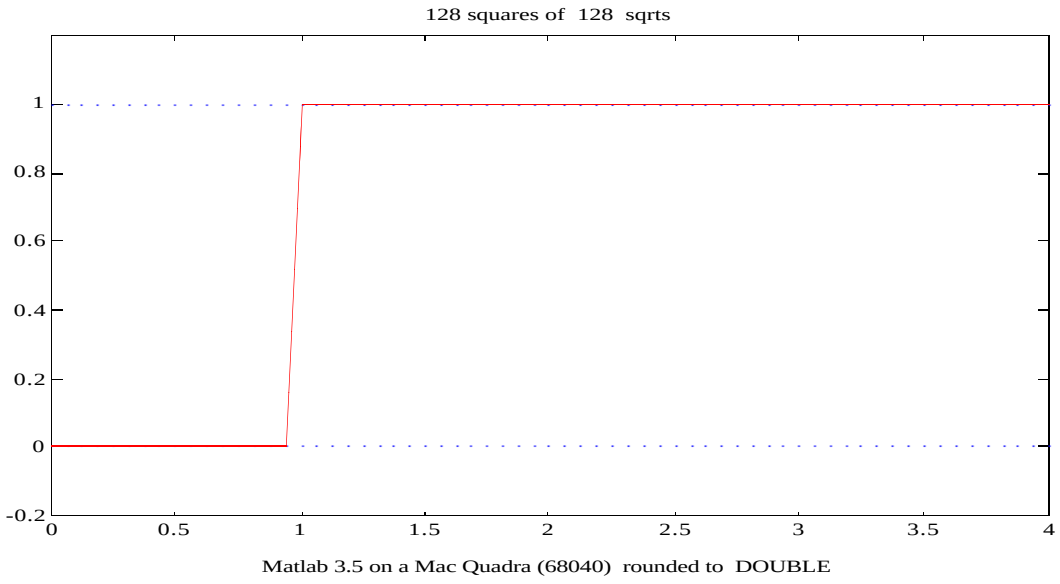
Define a floating-point-valued function H(X) for nonnegative floating-point arguments X thus:

$Y := \sqrt{\sqrt{\ldots\sqrt{\sqrt{\sqrt{X}}}}}$; … 128 consecutive square roots …

$H := ((\ldots((Y^2)^2)^2\ldots)^2)^2$. … 128 consecutive squares.

A naive expectation is that H(X) should match X except perhaps in its last three sig. dec. or last nine sig. bits. Something utterly else happens. What follows is a plot of H(X) versus X as computed by arithmetic rounded to 53 sig. bits:

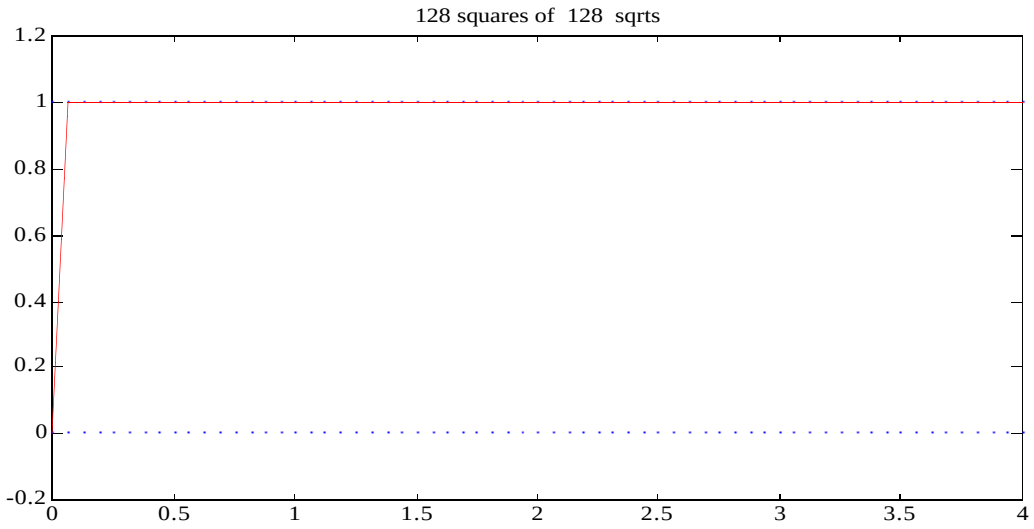
$H(X) := ((\ldots((Y(X)^2)^2)^2\ldots)^2)^2$ where $Y(X) := \sqrt{\sqrt{\ldots\sqrt{\sqrt{\sqrt{X}}}}}$, 128 times each



The same thing happens on Sun SPARCs, on recent (for 25 years) hp calculators, on PCs and recent (since 1995) Macintoshes using recent versions (5 or later) of MATLAB, and so on.

How can this graph be explained? Of course H(0.0) = 0.0 and H(1.0) = 1.0 because then H commits no rounding errors. Otherwise Y must be a rounded approximation to $X^{2^{-128}}$. Let's suppose that the computer rounds every square root correctly (error smaller than 0.5 in the last digit retained). If X > 1 then Y = 1 exactly; do you see why? And then H = 1 exactly too. On the other hand, if 0 < X < 1 then Y = 0.999…999 or the the arithmetic's binary floating-point number next less than 1 ; do you see why? And then raising that number Y to the power 2^{128} *Underflows* (do you see why?) to 0.0 , which is returned as H(X) .

However, some computers and calculators do something else; here is their graph of H(X) :



Matlab 3.5 on a Mac Quadra (68040) double-rounded to DOUBLE

Why? The previous page's analysis implies that sqrt(x) must sometimes return something else than ($\sqrt{x}$ correctly rounded) on computers that produce the last graph. Instead, if $s = 1 - \mu$ is the floating-point number next less than 1 , namely s = 1 – eps/2 in MATLAB, s = 0.9999…999 in decimal arithmetic, then sqrt(s) must return 1.0 instead of s on those computers. Actually $\sqrt{s} = 1 - \mu/2 - \mu^2/8 - \ldots$ falls so nearly halfway between s and 1 that sqrt(s) can be extremely nearly correctly rounded and yet be rounded wrongly up instead of down. Most computers did this until the late 1980s when IEEE Standard 754 for Binary Floating-Point became ubiquitous.

Something else again can happen on the old Apple Mac Quadra's 68040 and on Intel-based PCs and their clones though they conform to IEEE 754. It is a double rounding. First sqrt(s) is rounded correctly to 64 sig. bits in one of a set of floating-point registers intended to evaluate all subexpressions to this wider precision regardless of the narrower precisions, 53 or 24 sig. bits, of many operands. Normally this extra precision would be advantageous in so far as it attenuates roundoff in all subexpressions, literal constants and local variables before they are rounded back to 53 or 24 sig. bits in 8-byte or 4-byte variables stored in memory for subsequent display or communication. Alas, the programming language community and especially Bill Gates Jr. at Microsoft and Bill Joy at Sun Microsystems failed to appreciate the importance of that extra

precision and consequently declined to support it, so it is threatened with atrophy now. But all that is a story for another day; see my web page's "How Java's Floating-Point Hurts Everyone Everywhere", …/JAVAhurt.pdf , and "Marketing *vs.* Mathematics", …/MktgMath.pdf , and "MATLAB's Loss is Nobody's Gain", …/MxMulEps.pdf .

When older versions of MATLAB first round sqrt(1 – eps/2) to 64 sig. bits in one of those extra-precise but anonymous floating-point registers, the result is 1 – eps/4 correctly but temporarily. This result is then stored in an 8-byte memory cell rounded to 53 sig. bits; it rounds correctly to 1.0 , which explains the last graph with a step at zero.

Version 6.5 of MATLAB can set bits that control rounding precision in the PC's floating-point registers to mimic SPARCs and other workstations' 8-byte floating-point, thus rounding sqrt(8-byte) once to 53 sig. bits. This version gets the graph with the step at x = 1 . To benefit from extra precision during the multiplication of non-sparse matrices in MATLAB 6.5 on PCs, invoke "`system_dependent('setprecision', 64)`". Then sqrt(8-bytes) will get rounded correctly twice, producing the graph with a step at zero.

Why should we care that an extravagantly complicated computation of H(X) = X misbehaves in obscure ways because of roundoff? Because most computations deemed "numerically unstable" malfunction in a similar way, usually exposed by casual tests. Commonplace instances include differential equation solvers and eigensystem solvers. And because their malfunctions have so much in common, an explanation for them in general mathematical terms deserves our attention.

Suppose a floating-point program F(X) is intended to compute a function $f(x)$. The program F(X) you see is not the program you get. Instead you get a function f(x, r) in which r is a column of rounding errors, one for every arithmetic operation in F(X) susceptible to roundoff. Of course, r is unknown but tiny; and if F(X) is algebraically correct then f(x, o) = $f(x)$. Consequently, in most cases, $f(x, r) = f(x) + (\partial f/\partial r)_{r=0}\cdot r + O(r)^2$. Here ∂f/∂r is the Jacobian matrix of first partial derivatives of f(x, r) with respect to variables in r . If ∂f/∂r is not huge, the execution of program F(X) will produce f(x, r) with an error $f(x, r) - f(x) \approx (\partial f/\partial r)\cdot r$ that is tolerable because every elemement of r is so tiny. Otherwise, when the error f(x, r) – $f(x)$ is intolerably big, it must be so big because some elements of ∂f(x, r)/∂r are gargantuan.

How can ∂f/∂r become gargantuan? It can do so only if x comes close, in some sense, to a *Singularity* of f(x, r) where ∂f/∂r would become infinite. This singularity of f need not be a singularity of the function f , but instead an artifact of the formula chosen for the program F . For example, the program T(z) := { If z = 0 then 1 else (exp(z) – 1)/z } that figured in §6's Smooth Surprise has a division-by-zero singularity at z = 0 which, though ostensibly removed by the branch, can still exert a baleful influence if roundoff disconnects the numerator from the tiny divisor. Another example is an $\infty - \infty$ singularity approached when a program F computes an innocuous function f as the difference between two gargantuan numbers whose cancellation leaves only the ghosts of digits lost previously. Some singularities can turn out to be benign, as is the division by a tiny log(…) in the accurate but tricky version of T(z) in §6.

Whether malignant or benign, $\infty - \infty$ and …/0 are not the only kinds of singularities. On the contrary, singularities in general are far too diverse to be classified mathematically. This is why

"Neither cancellation nor small divisors need be concomitants of numerical distress"; many other kinds of singularities cause numerical distress more often. As in our example H :

Absent roundoff, our example $H(x) := (x^{2^{-N}})^{2^N}$ for $N = 128$ would compute $H(x) = x$. Only one rounding error r dominates the computation, and by ignoring the others we can approximate the computed value of H by the expression $h(x, r) = \left(e^r \cdot x^{2^{-N}}\right)^{2^N}$ whose $\partial h/\partial r = 2^N \cdot h(x, r)$. Therefore error $h(x, r) - h(x, 0) \approx 2^N \cdot r \cdot x$; and when $N = 128$ we find that the relative error in $h(x, r)$ is a rounding error r (perhaps not so big as $2^{-53} \approx 10^{-16}$) amplified by $2^{128} \approx 10^{38}$. The singularity occurs when parameter $N = 128$ (which figures in program H and expression h but not in $h(x, 0) = x$) is replaced by $N = +\infty$. This replacement seems drastic at first; actually it is a consequence of a singularity so strong that its effect is felt when N is big but not very big.

In general, singularities whose nearness amplifies roundoff intolerably tend to be unobvious. If they were always obvious, error-analysts would be mostly unemployed. Such is not the case.

How can somebody innocent of error-analysis at least detect if not correct miscalculation due to roundoff? One way is to study error-analysis; a good text on the subject is Nicholas J. Higham's book "Accuracy and Stability of Numerical Algorithms" 2d. ed. (2002, SIAM, Philadelphia), though it is about 700 pages long. Another way is to rerun a suspected subprogram under diverse rounding modes and compare results. Rerunning our example program H(X) with rounding directed *Down* reproduces the first graph with a step up from 0 to 1 at $X = 1$. Rounding directed *Up* produces a new graph that steps up from 1 to ∞ (due to Overflow) at $X > 1$. These graphs reveal the hypersensitivity of H(X) to roundoff unmistakably and with little effort.

§11: A Case Study of Bits Lost in Space

Imagine plans for unmanned astronomical observatories in outer space. They needed software to compute their locations relative to stars and planets whose positions are listed in a computerized ephemeris. Three vendors tendered programs for that purpose. To assess their accuracies without becoming bogged down in the messy mathematics of error-analyses, we have presented the same test data to the different vendors' programs and compared their results. Compared with what? Were we able ourselves to generate software that computed accurate results, we would not have to purchase one of these programs. Their three results matched nearly enough for almost all our millions of tests, but a few tests have exposed substantial disagreements. Now what shall we do?

Presented here is a case study that may shed light upon that question by focussing upon a small subprogram that computes subtended angles from spherical polar coordinates of pairs of celestial objects listed in the ephemeris. Computed angles will be compared with observed angles to help adjust or determine an observatory's location in space, but these procedures and corrections for the finite speed of light coming from the planets are all omitted here for the sake of simplicity.

First some notation. Directions to distant stars are specified by angles named as follows:

Names of Angles used for Spherical Polar Coordinates

Angle Symbols	Relative to Horizon	Relative to Ecliptic Plane	Relative to Equatorial Plane
θ, Θ	Azimuth	Right Ascension	Longitude
ϕ, Φ	Elevation	Declination	Latitude

These angles must satisfy $-\pi \le \theta \le \pi$ and $-\pi/2 \le \phi \le \pi/2$ in Radian measure, $-180^\circ \le \theta \le 180^\circ$ and $-90^\circ \le \phi \le 90^\circ$ in degrees. Similarly for Θ and Φ. Radians will be used in what follows because the observatories' instruments resolve angles in radians with 3 bits to the left and 24 bits to the right of the binary point; displayed in decimal they would look like "x.xxxxxxx".

Two stars whose coordinates are (θ, ϕ) and (Θ, Φ) subtend an angle ψ at the observer's eye. This ψ is a function $\psi(\theta-\Theta, \phi, \Phi)$ that depends upon θ and Θ only through their difference $|\theta-\Theta| \bmod 2\pi$. The three implementations of this function ψ to be compared are called u, v and w. They run at roughly the same speed. They perform all their computations in arithmetic conforming to IEEE Standard 754's specifications for single precision (4 bytes wide, 24 sig. bits worth more than six sig. dec.), the same precision as the data from the ephemeris, so the reader of this case study need not fear drowning in digits. Still, in order that anyone so inclined may recover all binary data and results exactly, a full nine sig. dec. will be displayed here. All results were computed on the same Intel Pentium processor as will be installed in the observatories.

Three Subprograms u, v and w Approximate Subtended Angle $\psi(\theta-\Theta, \phi, \Phi)$.

$\theta-\Theta$:	0.00123456784	0.000244140625	0.000244140625	1.92608738	2.58913445	3.14160085
ϕ :	0.300587952	0.000244140625	0.785398185	-1.57023454	1.57074428	1.10034931
Φ :	0.299516767	0.000244140654	0.785398245	-1.57079506	-1.56994033	-1.09930503
$\psi \approx$ u :	0.00158*221229*	*0.0*	0.000*345266977*	0.0005*98019978*	3.1408*2050*	3.14055*681*
$\psi \approx$ v :	0.0015932486*8*	0.000244140610	0.000172633489	0.000562231871	3.140*61618*	3.140*61618*
$\psi \approx$ w :	0.0015932486*8*	0.000244140610	0.000172633489	0.000562231871	3.1407804*4*	3.1405484*7*

This tabulation exhibits only the few atypical test results from u(θ–Θ, ϕ, Φ), v(θ–Θ, ϕ, Φ) and w(θ–Θ, ϕ, Φ) . They have agreed to at least six sig. dec. for almost all of millions of randomly generated test arguments. But the few atypical discrepancies are of the worst kind, intolerably bigger than the known uncertainties in the observatories' instruments and ephemeris, yet too small to be obvious. Which if any of subprograms u, v and w dare we trust?

Because the three subprograms under test agreed so closely for almost all inputs, we inferred that their different formulas were algebraically equivalent in the absence of roundoff to which their sensitivities differed. To assess these sensitivities we reran the subprograms in different directed rounding modes with exactly the same atypical data. The table below exhibits typical results for some of the atypical data. Results from redirected roundings resembled symptoms of numerical instability due to roundoff at the data tested on subprograms u and v . Subprogram w seemed stable. Could it be trusted? Unfortunately, our tests could not prove any of the subprograms correct. All that was proved was that at least two of the three seemed intolerably hypersensitive to rounding errors. This was worth knowing if only because it dropped the number of subprograms we thought worth further testing down to one.

Three Subprograms u, v and w Run with Redirected Roundings.

θ–Θ :	0.000244140625			2.58913445		
ϕ :	0.000244140625			1.57074428		
Φ :	0.000244140654			-1.56994033		
ψ ≈ u :	0.000598019920	NaN arccos(>1)	0.000598019920	3.14061594	3.14067936	3.14082050
ψ ≈ v :	0.000244140581	0.000244140683	0.000244140581	3.14039660	3.14159274	3.14039660
ψ ≈ w :	0.000244140610	0.000244140683	0.000244140610	3.14078045	3.14078069	3.14078045
Rounded:	To Zero	To +Infinity	To –Infinity	To Zero	To +Infinity	To –Infinity

When advised of our tests' results, all three vendors revised their subprograms to perform all floating-point arithmetic in some higher precision while keeping the subprograms' input data and output results in single precision (4 bytes wide) as before. Now all those tests find no significant differences among the three vendors' revised programs' results, though they all run a little slower than the original programs. And they all get results that agree to at least six sig. dec. with results from the original program w . Now what should we do?

Of course the foregoing story is imaginary. It is probably impossible because, alas, compilers and Programming Development Systems generally obstruct rather than aid attempts to diagnose a floating-point program's numerical distress by rerunning its subprograms in redirected rounding modes and/or in different precisions. Diagnostic proceedures that ought to be mindless aren't.

Still, if only to satisfy our curiosity, let us imagine what might come to light if the vendors were obliged to describe the algorithms used by their subprograms, or if these were reverse-engineered after disassembly. Here are the formulas that produced the foregoing tabulated results:

• **Subprogram u :**
$\psi(\theta-\Theta, \phi, \Phi) \approx u(\theta-\Theta, \phi, \Phi) := \arccos(\sin(\phi)\cdot\sin(\Phi) + \cos(\phi)\cdot\cos(\Phi)\cdot\cos(\theta-\Theta))$.
This formula, programmed by a computer science graduate who figured it out with the aid of his freshman Calculus text, can lose all figures the arithmetic carries when u nears zero, and can lose almost half the figures carried when u nears π . Should he have forseen these errors? How?

• **Subprogram v :**
$\psi(\theta-\Theta, \phi, \Phi) \approx v(\theta-\Theta, \phi, \Phi) := 2\cdot\arcsin(\sqrt{(\sin^2((\phi-\Phi)/2) + (\cos(\phi)\cdot\cos(\Phi))\cdot\sin^2((\theta-\Theta)/2))})$.
This formula from a text on Astronomy loses almost half the figures carried when v nears π . The loss is due to the singularity (infinite derivative) in arcsin(…) when its value is $\pi/2$.

• **Subprogram w :**
$\psi(\theta-\Theta, \phi, \Phi) \approx w(\theta-\Theta, \phi, \Phi) := 2\cdot\arctan(\sqrt{q/r})$ wherein
$t := \tan^2((\theta-\Theta)/2)$, $p := \tan^2((\phi-\Phi)/2)$, $P := \tan^2((\phi+\Phi)/2)$,
$q := (P + t + 1)\cdot p + t$, and $r := ((p+1)\cdot t + 1)\cdot P + 1$.
This formula, devised for the occasion, conserves almost all the arithmetic's accuracy for all valid angles input in radians, for which no tan(…) can be infinite. For angles in degrees use
$\psi(\theta-\Theta, \phi, \Phi) \approx w(\theta-\Theta, \phi, \Phi) :=$ { If $p+P+t = \infty$ then $180^\circ-\phi-\Phi$ else $2\cdot\arctan(\sqrt{q/r})$ } .

Only subprogram w should be accepted for use by an observatory whose position in outer space is often determinable most accurately when it lies in or very near a straight line segment joining a planet to a star, in which case the angle they subtend at the observatory will be π or very near it.

The foregoing case study is hypothetical. *Fictional*. The numerical results are true results. Truth is stranger than Fiction: Mathematically valid formulas, including some repeated in textx for centuries ("they have 'stood the test of Time"), can be numerically treacherous.

How can you separate numerically trustworthy formulas from the treacherous ones?

Without an error-analysis, you can't. And if you can't, the simplest way to evade numerical embarrassment is to perform computation carrying extravagantly more precision throughout than you think necessary, and pray that it is enough. Usually somewhat more than twice the precision you trust in the data and seek in the results is enough. If it isn't, or if it runs so slowly that you have had to choose some narrower precision because it is the widest that doesn't run too slowly, how can you tell which formula has betrayed you when some datum has aroused your suspicion?

Rerun each formula separately on its same input but with different directed roundings;
the first one to exhibit hypersensitivity to roundoff is the first to suspect.

This usually works. Nothing less than an order of magnitude more costly works better. And nothing at all works infallibly.

More examples of numerically unstable classical trigonometrical formulas and stable substitutes for them are posted on my web page. See "Miscalculating Area and Angles of a Needle-like Triangle", http://www.cs.berkwley.edu/~wkahan/Triangle.pdf , and "What has the Volume of a Tetrahedron to do with Computer Programming Languages?", …/VtetLang.pdf . The unstable formulas lose at least about half or almost all figures carried for data coming from geometrically near-degenerate configurations even when a configuration is numerically well-conditioned, in which case the loss of accuracy is due not to some geometrical instability (there is none) but to a

gratuitous near-singularity in the chosen classical formula. Every instance of those numerical instabilities is exposed by reruns in redirected rounding modes; such reruns affect only negligibly the stable formulas supplied on my web page to supplant the unstable formulas.

§12: Mangled Angles

Geometrical computations notoriously demand occasionally extravagant precision to resolve critical numerical questions in a way consistent with geometry. For instance, questions of incidence (Where do these figures intersect or touch?) involving several points and/or lines through them can require arbitrarily high precision to be answered consistently. Otherwise, inconsistent answers can derail computation. A relatively simple and striking example is exhibited in `http://www.cs.berkeley.edu/~jrs/meshpapers/robnotes.ps.gz` on Prof. Jonathan Shewchuk's web page, which also provides software to deal with such situations successfully by simulating arithmetic of precision as high as needed.

We shall consider now a simpler example chosen to support the thesis that numerical software is extremely difficult to debug. Evidence for this thesis is the longevity of inaccurate software in use by vast numbers of numerically active and, in most instances, sophisticated users of MATLAB. Once again we consider the angle between two directions x and y specified now in Cartesian coordinates instead of the spherical polar coordinates of §11's function $\psi(\theta-\Theta, \phi, \Phi)$. The usual formula for the unoriented angle $\angle(x, y)$ between two (column) vectors x and y in an Euclidean space of arbitrary dimension is $\angle(x, y) := \arccos(x'\cdot y/(\|x\|\cdot\|y\|))$ wherein the length $\|x\| := \sqrt{(x'\cdot x)}$. "Unoriented" means $0 \le \angle(x, y) = \angle(y, x) \le \pi$.

The usual formula is known to lose near half the sig. digits carried when x and y are almost (anti-)parallel. For example, if x chosen at random and $y := \pi\cdot x$ are both rounded to n sig. bits, $\angle(x, y)$ cannot exceed $1/2^{n-1}$ no matter how big the dimension. But if m sig. bits are carried during the computation of the usual formula, then with probability at least about 1/5 the computed $\angle(x, y)$ will err by at least roughly $1/2^{m/2}$ unless m exceeds 2n sufficiently. Both error and probability grow slowly with dimension. Similarly behavior afflicts $\pi - \angle(x, y)$ when $y := -\pi\cdot x$ rounded. These results conform to an ancient rule-of-thumb I inherited from an elderly computer J.C.P. Miller:

> During all intermediate computations carry at least somewhat more than twice as many sig. digits as have been stored in the data and are desired from the results.

This recipe protects against embarrassment due to roundoff except in direly pathological cases.

Strangely, the recipe fails to guard the usual formula for $\angle(x, y)$ against embarrassment. If random x and $y := \pm\pi\cdot x$ are rounded to n sig.bits, the probability of an arccos(…) invalid because $| x'\cdot y/(\|x\|\cdot\|y\|) | > 1$ exceeds about 1/5 unless computation carries rather more than 2n sig. bits. Most programmers who test the usual formula on nearly (anti-)parallel vectors learn to replace it by $\angle(x, y) := \arccos(\max\{\min\{x'\cdot y/(\|x\|\cdot\|y\|), +1\}, -1\})$ either without noticing or without caring that it can lose about half the sig. digits carried. Can these digits be saved?

Yes. If precision much greater than n sig. bits runs too slow, other formulas can be used. The best known for three dimensions is the cross-product formula

$\angle(x, y) :=$ if $x'\cdot y \ge 0$ then $\arcsin(\|x\times y\|/(\|x\|\cdot\|y\|))$ else $\pi - \arcsin(\|x\times y\|/(\|x\|\cdot\|y\|))$.

Analogous formulas exist for higher dimensions though they entail too much work at very large dimensions. No matter; these formulas lose about half the digits carried when $\angle(x, y) \approx \pi/2$. The loss is exposed by hypersensitivity to the direction of roundoff when these formulas or the usual formula are executed with vulnerable data in IEEE 754's four directed rounding modes.

Here is a better formula less well known than it deserves:

$$\angle(x, y) := 2 \arctan(\|x\cdot\|y\| - \|x\|\cdot y\| / \|x\cdot\|y\| + \|x\|\cdot y\|) .$$

When executed in arithmetic rounded to n sig. bits its absolute error never much exceeds $1/2^{n-1}$ unless the dimension is gargantuan. And redirected roundoff barely affects this formula.

Why have we looked at so many formulas for the angle $\angle(x, y)$?

The formulas lose accuracy at singularities simple enough to be obvious to anyone with a modest exposure to numerical methods. Infinite derivatives of arcsin(…) where $\arcsin(1) = \pi/2$ and of arccos(…) where $\arccos(1) = 0$ and $\arccos(-1) = \pi$ attract scrutiny to places where accuracy gets lost. Hardly any accuracy is lost at the singularities of $\arctan(\pm\infty)$ because the derivative vanishes there. It all looks too easy. It can't be that easy all the time.

Apparently, ostensibly obvious singularities and good ways around them can be concealed from numerical experts as well as nonexperts by surprisingly little complexity. What else can explain the persistence since 1988 of a defect that loses up to half the digits carried when a MATLAB program misnamed `subspace(X, Y)` computes the angle between two subspaces spanned by the columns of two given matrices X and Y ? Has no user traced his troubles to `subspace` ?

To simplify our exposition a nonessential restriction to subspaces with the same dimension will be imposed. Let the given matrices X and Y have the same dimensions with (usually many) more rows than columns. The columns can be orthogonalized quickly by MATLAB's `qr(…)` program, so we may assume that $X'\cdot X = Y'\cdot Y = I$; the columns of X constitute an orthonormal basis for the subspace they span. Y likewise. Then the usual formula for the angle between the subspaces is $\angle(X, Y) := \arccos(\max\{\min\{\|X'\cdot Y\|, +1\}, -1\})$ wherein norm ||…|| is the largest singular value and max{min{… is there for reasons noted above. After that discussion we expect this formula to lose about half the digits carried when X and Y are orthonormal bases for slightly different spaces or the same space. Can such a loss occur often?

From 1988 to 2002 this formula caused versions 3.5 - 5.3 of MATLAB's `subspace(X, X)` to produce angles greater than $1/10^8$ instead of 0.0 or the roundoff threshold `eps` $\approx 2.2/10^{16}$ for over 90% of random matrices X . Surely someone must have noticed and complained.

Better methods had been published as early as 1973. MATLAB 6.x adopted one in 2002; but it suffers from the same flaw as afflicts the cross-product formula arcsin(…) : For at least 1% of random orthonormal matrices X and Y satisfying $X'\cdot X \approx Y'\cdot Y \approx I$ and $X'\cdot Y \approx O$ within `eps`, the adopted `subspace(X, Y)` produces angles differing from $\pi/2$ by more than $1/10^8$ instead of a correct difference not much bigger than `eps` . For how long will this error go uncorrected?

See a paper by A.V. Knyazev and M.E. Argentati in pp. 2009-2041 of *SIAM. J. Sci. Comput.* **23** (2002) <http://www.siam.org/journal /sisc/23-6/37733.html> for a "comprehensive overview" of angles between subspaces including mention of applications to statistics, science and software testing, plus algorithms to compute angles accurately and an extensive bibliography. Their algorithms are uglier than necessary. A neat perhaps novel algorithm is outlined hereunder:

Starting with given X and Y with orthonormal columns, so X'·X = Y'·Y = I , compute X'·Y and then its nearest orthogonal matrix Q ; it can come quickly from MATLAB's `poldec(…)` or from `svd(…)` or faster from a few steps of an iteration if X'·Y is not too far from orthogonal, as happens when ∠(X, Y) is small. For more about Q see p. 385 *et seq*. of N.J. Higham's book *Accuracy and Stability of Numerical Algorithms* 2d. ed. (2002) *Soc. Indust. Appl. Math.*, Philadelphia. Then ∠(X, Y) = 2 arcsin(||X·Q – Y||) within a modest multiple of `eps` . This algorithm is almost indifferent to redirected rounding, unlike MATLAB's current `subspace` .

What do mangled angles teach us? The goal of error-analysis is not to find errors but to fix them. They have to be found first. The embarrassing longevity, over three decades, of inaccurate and/or ugly programs to compute a function so widely used as ∠(X, Y) says something bleak about the difficulty of floating-point error-analysis for experts and nonexperts: Without adequate aids like redirected roundings, diagnosis and cure are becoming practically impossible. Our failure to find errors long suspected or known to exist is too demoralizing. We may just give up.

§13: Bloated Coffins

Interval Arithmetic (IA) is a good thing if implemented properly and integrated properly into a popular programming language. IA aids searches for zeros and extrema of functions of vector arguments, and is an almost indispensible tool for coping with tolerances in the computer-aided design of manufactured devices. Occasionally IA facilitates a mathematical proof. If intended also to assess roundoff's degradation of computed results, IA should be integrated with multi-precision floating-point arithmetic. Then IA's error estimates can serve to predict how much extra precision will suffice to recompute a desired result at least as accurately as desired even if usually such predictions greatly overestimate the smallest adequate amount of extra precision.

IA always over-estimates errors' accrual, too often so extravagantly as to undermine its own credibility as did *The Little Boy Who Cried* "Wolf!". How this happens will be discussed below not to disparage IA but to explain why its users are so likely to be disappointed if they use it mindlessly. Insinuating IA successfully into a computation usually alters its algorithm for the purpose, perhaps recasting the computation with the aid of unobvious perturbation analyses into a self-correcting iteration. This is not mindless; it is a long story for another day. Today's story is a long sad account of over-optimistic expectations, disappointments and frustration.

First some notation: Lower-case letters like x, y, … will be used here to represent noninterval variables, sometimes called "points" be they scalars or vectors. Bold upper-case letters **X**, **Y**, … will be used here to represent regions over which the corresponding lower-case variables range. For instance, if a scalar interval $\mathbf{X} = [\underline{x}, \overline{x}]$ is constructed to contain the scalar variable x within the range $\underline{x} \le x \le \overline{x}$, we shall write " $x \in \mathbf{X}$ ". The same goes for a vector **X** of intervals when it contains a vector point $x \in \mathbf{X}$, but in this case we shall call **X** "a *coffin*" as an abbreviation for "a rectangular parallelepiped with edges parallel to the coordinate axes". The *diameter* $\leftrightarrow(\mathbf{X})$ is the diameter of the smallest circle, sphere or hypersphere that contains **X**; when $\mathbf{X} = [\underline{x}, \overline{x}]$ is a scalar interval its diameter is just its width: $\leftrightarrow(\mathbf{X}) = \overline{x} - \underline{x}$.

The range of a function $f(x)$ as x runs through **X** will be denoted by $f(\mathbf{X})$. This is what we wish IA would compute. Instead, if a program f(x) written to compute $f(x)$ is rewritten to produce an IA program F(**X**) it should, if rewritten *correctly*, satisfy a containment relation $F(\mathbf{X}) \supseteq f(\mathbf{X})$. A mindless but correct rewriting merely replaces every lower-case point variable in program f(x) by its upper-case interval analog, and replaces every arithmetic operation upon point variables by its analogous IA operation. This may be easier said than done. When done, $F(\mathbf{X}) \supseteq f(\mathbf{X})$; but all too often diameter $\leftrightarrow(F(\mathbf{X}))$ exceeds $\leftrightarrow(f(\mathbf{X}))$ by orders of magnitude.

For example, take $f(x) := 4 \cdot x \cdot (1-x)$. Rewriting a program $f(x) := 4 \cdot x \cdot (1-x)$ mindlessly turns it into $F(\mathbf{X}) := 4 \cdot \mathbf{X} \cdot (1-\mathbf{X})$. Since we care about IA's overestimates of roundoff's effects, let's consider an interval $\mathbf{X} = [x-h, x+h]$ whose width 2h amounts to several rounding errors in numbers near x. In particular take $x = 0.5$ and $h \le 2^{-20}$ so that $\mathbf{X} = [0.5 - h, 0.5 + h] = 1-\mathbf{X}$ and then $F(\mathbf{X}) = [1-4h+4h^2, 1+4h+4h^2]$ in the absence of additional roundoff that could only widen it. Now $\leftrightarrow(F(\mathbf{X})) = 8h$ is millions of times as big as $\leftrightarrow(f(\mathbf{X})) = \leftrightarrow([1-4h^2, 1]) = 4h^2$. Worse, $\arccos(f(\mathbf{X})) = [0, \arccos(1-4h^2)]$ but ACOS(F(**X**)) is thwarted by an arccos(>1).

This $F(\mathbf{X}) = 4 \cdot \mathbf{X} \cdot (1-\mathbf{X})$ is too wide because IA took no account of the anti-correlaton between the factors **X** and $1-\mathbf{X}$; they might as well be independent variables **X** and **Y** each with the

interval value **[**0.5 – h, 0.5 + h**]** . A different program f(x) := 1 – (2·x–1)^2 computes f(x) well when x is near 0.5 ; its mindlessly rewritten analog F(**X**) := 1 – (2·**X**–1)^2 = f(**X**) too provided subexpression (2·**X**–1)^2 compiles to a call to a proper IA implementation of (…)^2 , not to an uncorrelated product (…)·(…) . Now F(**[**0.5 – h, 0.5 + h**]**) = f(**[**0.5 – h, 0.5 + h**]**) . Good. But then computing this F(**[**0, h^6**]**) yields **[**0, f(h^6)+r**]** in which roundoff r is at least a unit in the last place of 1.0 , inflating ↔(F(**[**0, h^6**]**)) = f(h^6)+r = ↔(f(**[**0, h^6**]**)) + r by perhaps many orders of magnitude. If extravagant inflation is to be prevented for *every* interval argument **X** , the IA analog of f(x) = 4·x·(1–x) must employ a more complicated formula like …

F(**X**) := If (**X** is near enough to **[**0, 0**]**) then 4·**X**·(1 – **X**)
else if (**X** is near enough to **[**1, 1**]**) then F(1–**X**)
else 1 – (2·**X**–1)^2 .

In general, we must partition f(x)'s domain into subdomains over each of which an apt choice of expression F(**X**) can keep ↔(F(**X**)) from exceeding ↔(f(**X**)) excessively, we hope.

The expression Spike(x) := 1 + x^2 + log(| 1 + 3·(1–x) |)/80 explored in §7 suffers slightly from bloated width due to anticorrelated variation of subexpressions over the interval 0 < x < 4/3 in which x^2 increases while log(…) decreases. The bloat becomes severe for **X** including 4/3 when " log(| 1 + 3·(1–x) |)/80 " is replaced by " log((1 + 3·(1–x))·(x – 4·(x–1)))/160 " ; it is algebraically identical but gets NaN from log(negative) no matter how narrow **X** may be.

Another phenomenon bloats IA's intervals when they estimate functions of more than one real variable: Coffins have too few shapes. For example, consider multiplying a complex interval **Z** := **[**√2–h, √2+h**]** + ı**[**–h, h**]** by a complex constant c :=(1 + ı)/√2 . Even if roundoff during IA multiplication is negligible, IA produces a product **P** = **[**1–h√2, 1+h√2**]** + ı**[**1–h√2, 1+h√2**]** which barely contains c·**Z** but has diameter ↔(**P**) = 4h rather bigger than ↔(c·**Z**) = √8 h . This inflation occurs because the coffin **Z** (actually a square with sides of length 2h parallel to the real and imaginary axes) gets turned into a diamond c·**Z** of the same size. The smallest coffin that contains the diamond is a coffin **P** with bigger sides of length √8 h . Inflations like this become compounded during lengthy computations, producing coffins bloated by factors that can grow as fast as exponentially with the number of IA operations.

Soon after R.E. Moore introduced IA in the 1950s, P. Henrici sought a way to retard the inflations of coffins during complex IA; he replaced them by circles. Just as a real interval **X** := **[**x–h, x+h**]** can be rewritten **X** = x ± h in terms of a center-point x and half-width h , so can a circular disk **Ç** in the complex plane be written **Ç** := ç + μ● in terms of a center-point ç , radius μ , and the unit disk ● . So long as radii were nearly infinitesimal, as they should be if due solely to roundoff, and provided no singularity was approached too closely, complex circle-arithmetic attenuated excessive bloating far better than complex IA with coffins could during simple computations. But complicated complex computations continued to suffer from exponentially excessive bloating for reasons that will become apparent shortly.

While seeking IA bounds for solutions of differential equations in the 1960s, F. Krückeberg sought a way to retard inflations of coffins; he replaced them by more general parallelepipeds:

Write $\mathbf{P} = p + S\blacksquare$ to represent a parallelepiped centered at point p with shape determined by a linear map (matrix) S acting on the unit cube $\blacksquare$. Just when S is diagonal is $\mathbf{P}$ a coffin. If $S = \begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix}$ then $\mathbf{P}$ is a diamond-shaped parallelogram like $\blacklozenge$ but twice as high as wide.

IA becomes far more costly with parallelepipeds than with coffins. Given a program $f(x)$ that computes a vector-valued function $f(x)$ of a vector argument x, and given $\mathbf{P} = p + S\blacksquare$, the computation of a containing parallelepiped $F(\mathbf{P}) := f(p) + T\blacksquare \supseteq f(\mathbf{P}) = f(p + S\blacksquare)$ reduces to a determination of a matrix T via symbolic as well as numerical operations upon program f. In the simplest case, and provided $\leftrightarrow(S\blacksquare)$ is tiny enough, $T = f'(p)\cdot S\cdot V$ wherein the coffin $V\blacksquare \supseteq (f'(p)\cdot S)^{-1}\cdot(f(\mathbf{P})–f(p))$ and $f'(p)$ is the Jacobian matrix of first partial derivatives of program $f(x)$ at $x := p$. In other cases, where the inverse of $f'(p)\cdot S$ does not exist or when $\leftrightarrow(S\blacksquare)$ is not so tiny, T becomes slightly arbitrary and much more complicated to determine. The labor can be automated, at least in principle, and thus rendered mindless or very nearly so.

However the labor is worthwhile only in special cases because in general, in the absence of contraindications inferred from error-analyses, IA with parallelepipeds tends to bloat almost as badly as does IA exclusively with coffins. Bloating is due to geometrical oversimplification:

Three forces tend to inflate circumscribing regions computed by the foregoing IA schemes. The first force has already been discussed; it arises from regions restricted to shapes, like coffins', that are too simple. A second force is generated by regions' convexity if they are not tiny enough. The third force is generated when circumscribing regions are tiny enough but possess sharp edges or corners. The next example will illustrate how the latter two forces act.

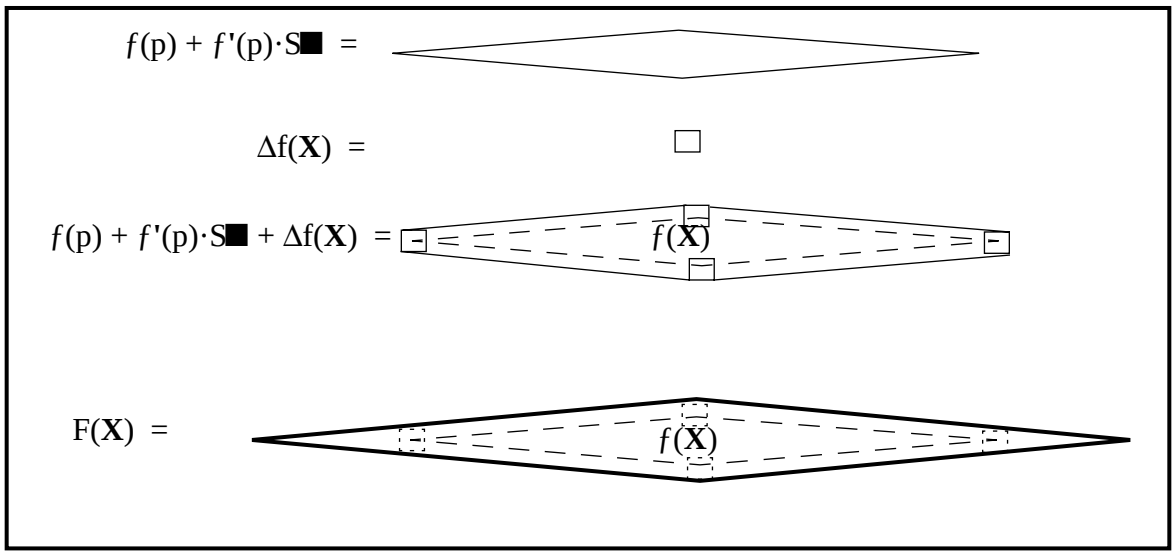
Suppose x represents the initial position and velocity of a planet in orbit about a star, and $f(x)$ is this planet's position and velocity after a year. This planet's position and velocity after K years is $f^{(K)}(x) := f(f(f(\ldots f(x)\ldots)))$ composed K times. If a small $\mathbf{X}$ is convex and roughly spherical, $f(\mathbf{X})$ is banana-shaped because planets slightly closer to the star orbit slightly faster. Then $f^{(K)}(\mathbf{X})$ tends to a spiral aligned along the orbit, ultimately (as $K \to \infty$) resembling a ring of Saturn. IA computes a convex circumscribing region $F(\mathbf{X}) \supseteq f(\mathbf{X})$. Some points $x \in F(\mathbf{X})$ lie closer to the star than any points in $f(\mathbf{X})$, and consequently travel faster than they should, thus exaggerating the length and curvature of $f(F(\mathbf{X}))$ compared with $f(f(\mathbf{X}))$. As K increases, $F^{(K)}(\mathbf{X})$ compounds that exaggeration. Soon the shape of $f(F^{(K)}(\mathbf{X}))$ so resembles the letter $\mathbf{C}$ that convex $F(F^{(K)}(\mathbf{X}))$ enclosing $f(F^{(K)}(\mathbf{X}))$ encloses the star too, whereupon $f(F^{(K+1)}(\mathbf{X}))$ explodes. This is how the mere convexity of F ultimately forces excessive bloat.

If $\leftrightarrow(\mathbf{X})$ is too tiny for mere convexity to bloat $F(\mathbf{X})$ much, a third force threatens to bloat it. Here is how: Suppose $\mathbf{X} = p + S\blacksquare$ is a parallelepiped and $f(x) = f(p) + f'(p)\cdot(x–p) + \ldots$, so that $f(\mathbf{X}) = f(p) + f'(p)\cdot S\blacksquare + \ldots$ is very nearly a parallelepiped too. However, parallepiped $F(\mathbf{X})$ must enclose the first two terms plus contributions from higher order terms "…" as well as computational errors. $F(\mathbf{X}) \supseteq f(p) + f'(p)\cdot S\blacksquare + \Delta f(\mathbf{X}) \supseteq f(\mathbf{X})$ in which $\Delta f(\mathbf{X})$ bounds those two contributions and " + " is the *Minkowsi Sum*: $\mathbf{Y}+\mathbf{Z} := \{y+z \text{ for all } y \in \mathbf{Y} \text{ and } z \in \mathbf{Z}\}$.

When $\leftrightarrow(S\blacksquare)$ is so tiny that $\leftrightarrow(\Delta f(\mathbf{X}))$ is rather smaller than $\leftrightarrow(f'(p)\cdot S\blacksquare)$ we might expect $F(\mathbf{X})$ to have nearly the same shape as $f'(p)\cdot S\blacksquare$ and a slightly bigger diameter, thus enclosing

$f(\mathbf{X})$ tightly. Something else happens because the orbit function $f(x)$ maps almost every near-infinitesimal parallelepiped $\mathbf{X}$ to a flattened parallelepiped $f(\mathbf{X})$ resembling a two-bladed axe-head, and as K increases $f^{(K)}(\mathbf{X})$ tends to a needle-shaped parallelepiped. When $\mathbf{X} = p + S\blacksquare$, though very tiny, is needle-shaped, so is $f(p) + f'(p)\cdot S\blacksquare$; but the error-term $\Delta f(\mathbf{X})$ is not needle-shaped though it is tinier again. Below is a picture showing how the addition of a relatively tiny error-term $\Delta f(\mathbf{X})$ to the needle $f(p) + f'(p)\cdot S\blacksquare$ thickens it enough to force $F(\mathbf{X})$, the sum's smallest enclosing parallelepiped, to extend too far beyond $f(\mathbf{X})$.

Enlarged Extension of a Slightly Thickened Needle



The narrower the needle, the greater is the extension, often amounting to orders of magnitude beyond $\leftrightarrow(\Delta f(\mathbf{X}))$. This is how sharp edges and corners force IA with general parallelepipeds to bloat excessively. This force has usually been strong enough to frustrate the application of IA that originally motivated it, namely error-bounds provably valid but not too excessive for trajectories and orbits obtained as solutions of differential equations with given initial conditions.

No simple (much less mindless) way is known to defeat all three geometrical forces that thwart applications of IA. In 1968 I replaced parallelepipeds by ellipsoids to get rid of sharp edges and corners, thereby suppressing inflationary forces enough that bloating grew by factors like $\sqrt{\text{number of arithmetic operations}}$ instead of exponentially so long as computed error-bounds stayed small enough not to be bloated by the force of mere convexity. Flattened and needle-shaped ellipsoids still occurred, and their associated ill-conditioned matrices required extra-precise arithmetic and other costly expedients, none of them remotely mindless. For a brief outline of ellipsoidal computations see "Ellipsoidal Error Bounds for Trajectory Calculations" posted at http://www.cs.berkeley.edu/~wkahan/Math128/Ellipsoi.pdf.

Even if IA's coffins are generalized, as they should be, to include figures like parallelepipeds and ellipsoids in an attempt to suppress excessive bloating, the attempt will fail too often on nontrivial computations unless augmented by considerable thought. It's not a mindless method.

§14: Desperate Debugging

Programming Development Systems offer programmers ways to insert break-points into their programs and specify conditions under which execution will pause there. Then the programmer can single-step through his program looking for the first step at which his program went astray. Though invaluable, these debugging aids often fail to help us diagnose bugs due to roundoff in floating-point software of typical complexity. The futility of single-stepping a long way into a program intended to run at gigaflops is not the only difficulty. Two kinds of ignorance interfere with accurate diagnosis: One is ignorance of the "correct" path from which the program strayed. Another is ignorance of how far the program should be allowed to stray, since it cannot follow the "correct" path perfectly. The two kinds of ignorance will be treated in turn hereunder.

Higher precision will very often estimate a "correct" path well enough. To this end, imagine a debugger that can transform a given subprogram p , whose literal constants and variables x, y, z, … have been declared by the programmer to have precisions thought adequate at the time, into an analogous subprogram P whose corresponding literal constants and variables X, Y, Z, … are declared by the debugger to have greater precisions, preferably about twice as great. Then the debugger can execute both programs p and P simultaneously (actually interleaved) with the same input (or copies of it if the subprogram will change it) and compare their progress to see where one of the variables x, y, z, … first departs excessively from its analog X, Y, Z, … .

To get all that to work properly, three technicalities must be addressed. First, if subprogram p invokes other subprograms the programmer must tell the debugger which of them to transform into higher precision analogs, leaving others unaltered. Second, if subprogram p includes tests-and-branches dependent upon its input, the programmer must tell the debugger which branches P must follow the same way p goes regardless of how the branch would otherwise go in P . When P will follow a branch differently than p does, the programmer must tell the debugger to pause, or else tell it where to resume comparisons of corresponding variables, or both. The necessity of these latter options is obvious for the subprogram T(z) that figured in §6's Smooth Surprise. Less obvious is the necessity for P to persist longer than p in a convergent equation-solving iteration so that P's solution will be computed more accurately than p's in accordance with P's higher precision. However, if the former option, namely forcing P to match p's branching despite contrary predicates, appears perverse, consider the following situation:

Gaussian Elimination with *Pivotal Exchanges* is the method by which most systems of linear equations are solved. It scans columns to choose an element of biggest magnitude to serve as the *Pivot* (divisor), and its row as the *Pivotal Row*, for the next pass of the elimination process. On rare occasions two of a column's biggest elements can have almost identical magnitudes, and then both are valid choices for pivot. The actual choice may be an accident of roundoff; usually it alters intermediate results a lot but final results inconsequentially. If the choice alters final results drastically, the equations' matrix may be nearly singular or else the equations and/or unknowns may have been scaled badly, perhaps because of inappropriate units like kilometers for both the length and width of glass fibers. Only forcing P to match p's choices of pivots will expose the consequences of these choices to scrutiny by the method's programmer or user. Otherwise he may blame discrepant results indiscriminately upon "ill-conditioning" and consequently embark upon a futile quest for algebraic redundancy (linear dependence). I've seen this happen often.

The third technicality runs into the second kind of ignorance: How far should the debugger allow

p's variables to deviate from their analogs among P's before bringing deviations to the attention of whoever is trying to debug p ? This question has no easy answer. Sometimes early end-figure deviations propagate into subsequent gross deviations that may or may not dwindle away later. And if they do dwindle away at the end as in §5's recurrence, still results may be as wrong as when the recurrence starts from $x_0 := 4$ and $x_1 := 4.25$. Or final results may be quite right as when the recurrence starts from $x_1 := 17/4$ and $x_2 := 76/17$ rounded. An example more representative than §5's is *QR Iteration* for computing matrix eigenvalues. Without branches like Gaussian Elimination's, QR Iteration routinely generates grossly deviant intermediate results and yet delivers final results in an array of fairly accurate eigenvalues differing at worst in their ordering from what would have been delivered had rounding errors been much smaller.

There is no easy way to decide when p's variables have deviated too far from their analogs in P . There is an onerous way, though it seems far-fetched at first. It resembles the computation of loop-invariants for programs that have nothing to do with floating-point. Here is the way to do it:

Mark a number of break-points in subprogram p and in the corresponding places in P . We shall call these break-points "stages". At each stage, copy the values of all p's variables onto P's and execute P completely starting from that stage. If that stage's final results differ too much from the previous stage's, something deleterious happened in p between these two stages. Insert more stages between them to narrow the search for an offending event if there is one. No such event need exist if successive stages' final results drift away slowly but ultimately too far, as happens with numerically unstable programs like H(X) = X whose graph in §10 was a step.

This scheme succeeds as well as it can as soon as two of its stages straddle the shortest piece of software (maybe all of p) hypersensitive to roundoff at the input data tested. The scheme costs lots of time and storage, and it can fail on some pathological programs like Muller's recurrence in §5 and the Smooth Surprise's program G(x) = 1 in §6 that almost always computes 0 . Such failures are rare. Of all comparably effective schemes I know about, none comes closer than this one to earning the epithet "mindless". I wish all Programming Development Systems provided it even if it runs too slowly to run on the lengthier floating-point programs.

What runs too slowly won't get run. Consequently, possibly aberrant subprograms p have first to be segregated from the others in a lengthier program by a diagnostic scheme that runs at least almost as fast as if the lengthier program were not being subjected to close scrutiny. Here speed matters because, with today's gigahertz clock-rates, trillions of floating-point operations and millions of subprogram invocations may have to elapse before a first observable anomaly occurs. Redirected rounding during repeated executions of parts of the program in question is the only scheme I know likely to expose hypersensitivity to roundoff in one of those parts, perhaps one whose source-code is inaccessible, and to do so at an acceptable speed and bearable cost.

Programming Development Systems and debuggers that support recomputation with redirected roundings must, as mentioned at the end of §9, expect their users to specify which subprograms' innards are to be sheltered from redirected roundings. By default, built-in library functions, including Fortran's "Intrinsic Functions", may well be sheltered that way except possibly for their last arithmetic operation whose result is the function's output. Directed rounding of this last result is appropriate when the function is intended to appear "atomic" like multiplication and addition. For example, if division is not built into the hardware but is composed from other

arithmetic operations, only the finally delivered quotient should be exposed to directed rounding. The same goes for sqrt(…), which is built into the hardware on some machines but not others. Exp(…), log(…), pow(…) and Fortran's **, and other math. library functions, as well as some others known by the programmer to require shelter from redirected roundings, pose a technical nuisance to a compiler that "inlines" such functions to gain a little extra speed. Then interleaved floating-point instructions will have to be marked, some to have their rounding redirectable, others not, in ways dependent upon how the hardware has implemented directed roundings mandated by IEEE Standard 754 (1985).

Redirected rounding is not so simple to support as it first appears. Debuggers can support it properly only by collaborating closely with the compiler and, in some systems, with dynamic linkers that can revise a subprogram as it is loaded into memory. A debugger that surmounts these technical obstacles offers its users a way easier, faster and more often successful than all other known ways to find sources of anomalies triggered by ostensibly innocuous data. Without such a tool such an anomaly becomes so nearly impossible to track down that the temptation to ignore it, and to pray that it is not the sole harbinger of an impending calamity, becomes irresistable.

§15: Conclusion

"Only Knowledge is purely Good, only Ignorance purely Evil."
Socrates, 470-399 BC.

We should be disappointed but not surprised by people's tendency to conceal errors instead of acknowledge and correct them. Only for baseball does anyone maintain a public record of errors. The journal *MTAC* (now *Math. of Computation*) used to publish errors in tables. Now nobody tracks errors in numerical software. Nor in other software, come to think of it. Who publishes how many Service Packs Microsoft issues to fix bugs in previous Service Packs for *Windows*? No wonder that so much software is reputed to be unreliable. How unreliable? Who knows?

If we publish no record of our mistakes, how shall we learn to avoid more of them?

Strangely, our culture is afflicted simultaneously with a fascination for bad news and an aversion to it. Cowed by the National Rifle Association, Congress has forbidden the Bureau of Alcohol, Tobacco and Firearms from spending money to collect statistics that might explain why guns kill almost 30000 civilians in the U.S.A. each year, but hardly any in Canada. Planeloads of American soldiers returning home in coffins used to land surreptitiously at night to obstruct a count of heroes each of whom would be celebrated at an isolated sad ceremony scattered around the country. A few years ago Californians almost passed a proposition to forbid collecting racial statistics lest they reveal how well or badly laws against racial discrimination are working.

Ignorance is Bliss for too many of us, Socrates notwithstanding.

Current computers' software systems provide practically no practicable assistance to diagnose numerical anomalies encountered occasionally by programmers and users of numerical software. Whatever is impracticable is unnecessary too to fulfill obligations of *Due Diligence*, so corporate lawyers may prefer the current situation to one in which widely available diagnostic tools made a merely difficult task out of one that is now almost impossible. However, engineers probably and

scientists certainly would prefer to be able with high probability to identify anomalous software modules promulgated in libraries and packages. Then these could be circumvented or avoided while their authors, notified of the evidence for an anomaly, sought a remedy. Or didn't.

At present only the second of the five schemes explored in these notes offers an economical way to diagnose anomalies caused by roundoff in precompiled software: Rerun the suspected module with exactly the same input but with default roundings (those not already directed by the author of the module) redirected. Though far from foolproof, this scheme has worked on over-zealous optimization in §3, on intermediate iterates x_n in §5, on T(§6), on MATLAB's log2(§7), acosh(§8) and acos(§8), on Gaussian Elimination (§9), on step H(§10) , on subtended angle ψ(§11), on `subspace`(§12) and on innumerable other examples upon which nothing else so inexpensive could possibly have worked so well.

In the near future I hope that programming languages will by default evaluate all constants and expressions in the hardware's widest floating-point format that does not run too slowly, as was the custom with old-fashioned Kernighan-Ritchie C . Of course, the language has to allow the programmer access to variables declared to have this widest format, not like C compilers offered nowadays by Microsoft and formerly by Sun Microsystems when they used MC68020/68882 processors. C99 tries to get its implementors and its users to do things right. Routine use of far more precision than deemed necessary by clever but numerically naive programmers, provided it does not run too slowly, is the best way available, with today's mixture of popular programming languages with overtaxed underfunded education, to diminish the incidence of roundoff-induced anomalies below any level of commercial significance even if we knew about every anomaly.

Farther in the future I hope that popular programming languages will support Interval Arithmetic of arbitrarily high (within limits) precision variable (coarsely) at run-time. Then programmers may use it to prove most of their numerical software free from roundoff-induced anomalies even if it runs sometimes slower than usual. "Sometimes slower" need not deter the majority of programmers if, as I expect, processor clock-rates and floating-point arithmetics continue to outpace memory speeds. The cost of moderate extra demands for processor cycles and memory cells will seem picayune compared with the cost of a numerically adept mathematician's time.

Speedy floating-point arithmetic is dangerous unless its design takes account of two requirements: One is the suppression of avoidable anomalies, each perhaps easily tolerable by itself, lest they accumulate to blight mathematical thought with a *Death of a Thousand Cuts*. Second, human thought is fallible, so computer systems must also help us both to find and fix our errors, and to render insignificant those we cannot find and fix. In particular, better floating-point debugging capabilities deserve high priority among computer system designers and implementors concerned with their own safety, since all of us depend upon the reliability of numerical computations that pervade our technology, from aircraft to antibiotics, from our MRI and PET images to seismic images of the Earth beneath us, from weather prediction to waste disposal and treatment.

At present, occasionally inaccurate floating-point software of moderate complexity is difficult verging on impossible to debug. If this state of affairs persists long enough to become generally accepted as inevitable, the obligations of *Due Diligence* will atrophy, and nobody will expect to be held accountable for unobvious numerical malfunctions. And nobody will be safe from them.