Floating-Point Arithmetic Besieged by "Business Decisions"

A Keynote Address, prepared for the
IEEE-Sponsored ARITH 17 Symposium on Computer Arithmetic,
delivered on Mon. 27 June 2005 in Hyannis, Massachussets

Abstract:

Daunting technical impediments to productivity in scientific and engineering computation have been exacerbated by a lack of adequate support from most programming languages for features of IEEE Standard 754 for Binary Floating-Point Arithmetic. **C**99 and perhaps Fortran 2003 are exceptions. Revisions to IEEE 754 are being contemplated to mitigate some of the impediments. Among the innumerable issues being addressed are …

- Decimal Arithmetic merging commercial Fixed- into Floating-Point.
- Flexible Floating-Point Exception Handling WITHOUT requiring trap-handlers to be programmed by applications programmers.
- Extended and Quadruple Precision to lift floating-point roundoff analysis from the conscience of almost every programmer.
- Aids to the localization of software modules perhaps responsible for contaminating results suspected of undue influence by roundoff.

But "Decision-Makers" ill-informed about our industry's history underestimate how costly are consequences of short-sighted "Business Decisions".

Auxiliary Reading

One purpose of this presentation is to tempt you to read lengthy documents that supply the mathematical reasoning underlying the assertions presented here.

Concerning the cases for Decimal arithmetic, and for dynamically redirected rounding to aid debugging, see my web page's …

`<www.cs.berkeley.edu/~wkahan/`**`Mindless.pdf`**`>`

In these pages see pp. 7–10, 13–14, 16, 20–22.

Concerning the necessity for extended and quadruple precision, see as well …

<…/JAVAhurt.pdf>, <…/MxMulEps.pdf>, <…/Qdrtcs.pdf>, <…/Triangle.pdf>, <…/MktgMath.pdf>, <…/refineig.pdf>, *etc*.

In these pages see pp. 16, 18, 22.

Concerning presubstitution as a way to cope better with certain exceptions …

<…/Grail.pdf>, <…/ARITH_17U>,

<…/ieee754status/IEEE754.PDF> (out of date)

In these pages see pp. 3–4.

This presentation will be updated and posted at

`<www.cs.berkeley.edu/~wkahan/`**`ARITH_17.pdf`**`>`

Floating-Point Exception-Handling is a topic too big to be addressed here informatively in the available time and space. See my web page for …

“A Brief Tutorial on Gradual Underflow” in `<…/ARITH_17U.pdf>` .

“A Demonstration of Presubstitution for ∞/∞ ” in `<…/Grail.pdf>` .

This last discusses ways an elegant short loop shown on the next page can be turned into an error-prone monster, like the one shown on the next page, when a programmer is denied linguistic support for non-default presubstitution.

The traditional treatment of floating-point exceptions as errors is a short-sighted policy:

In June 1996 the *Ariane V* rocket turned cartwheels and blew up half a billion dollars worth of instruments intended for European science in space. The proximate cause was the programming language ADA’s policy of aborting computation when an *Arithmetic Error*, in this case an irrelevant Floating-Point → Integer Overflow, occurred. See `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html` and p. 22 of `http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf` .

In Sept. 1997 the *Aegis* missile-cruiser *Yorktown* spent almost three hours adrift off Cape Charles VA, its software-controlled propulsion and steering disabled, waiting for Microsoft *Windows NT* 4.0 to be rebooted after a division-by-zero unexpectedly trapped into it from a data-base program that had interpreted an accidentally blank field as zero. See `http://www.gcn.com/archives/gcn/1998/july13/cov2.htm` .

Exceptions become Errors only when mishandled.

= =

Here is an elegant short loop intended for a program that computes eigensystems of huge dimensions accurately and efficiently on a network of computers:

```
 k := 0 ;   y := –x ;  …  x, y, ƒ  and  k  are arrays upon which arithmetic is performed elementwise.
{ Presubstitute  1.0  for  ∞/∞ ;  … For an explanation see  Grail.pdf .
  For  j = 1  up to  n  do
   {  ƒ := D(j) + y ;
      y := (y/ƒ)·LLD(j) – x ;  …  If  ƒ = ±0  then  y = ±∞  and on the next pass  y  becomes  LLD(j) – x .
      k := k + SignBit(ƒ) ;
    };}… .
```

Here is what it can become if the command "Presubstitute …" is unavailable:

```
 k := 0 ;   y := –x ;   N := (some presumably near-optimal integer discussed in the text of  Grail.pdf ) ;
 For  i = 0  up to  floor(n/N)  do  …  a batch of loops:
 {  ko := k ;   yo := y ;
    For  j = 1 + i·N  up to  min(n, (1+i)·N)  do  … the faster loop:
       {  ƒ := D(j) + y ;
          y := (y/ƒ)·LLD(j) – x ;  …  If  ƒ = 0  then  y = ±∞  and on the next pass  y  becomes  NaN .
          k := k + SignBit(ƒ) ;
        } ;
    If  any(isNaN(y))  then  do   ... a batch of slower loops:
       {  k := ko ;  y := yo ;
          For  j = 1 + i·N  up to  min(n, (i+1)·N)  do
           {    ƒ := D(j) + y ;
                q := if  isInfinite(y)  then  1.0  else  y/ƒ ;
                y := q·LLD(j) – x ;
                k := k + SignBit(ƒ) ;
             };};}… .          …                         For an explanation see  Grail.pdf .
```

= =

Historical “Business Decisions” bad for scientific and engineering computation

1982: Wm. Gates Jr. declared that the IBM PCs’ sockets for an 8087 numeric coprocessor “will almost all stay empty”, so Microsoft need not go to the trouble of supporting the 8087 fully in compilers. Microsoft still disdains double-extended arithmetic; and, alas, its lead is still followed by all but a handful of competing compilers.

1992: John Scully decided to abandon Motorola’s 680x0 and other CPUs, and to put Apple in bed with IBM and its Power PC processor, mistakenly believing RISC to be incompatible with Apple’s superb SANE (Standard Apple Numeric Environment) and discarding it.

Will Apple revive SANE after switching to Intel CPUs that can support it ?

1995: Bill Joy and James Gosling avoided getting advice from Sun’s expert Numerics group when designing Java to support only the parts of IEEE 754 they could understand, thus omitting exception-flags and extended precision, and adopting the semantics of ANSI **C** instead of the earlier serendipitous Kernighan-Ritchie **C** floating-point.
See “How Java’s Floating-Point Hurts Everyone Everywhere” `<…/JAVAhurt.pdf>` .

In 1973 IBM decided not to build Decimal floating-point into its CPUs.

Recently IBM reversed this decision, perhaps for the wrong reasons.

We could argue about them with Mike Cowlishaw, Mark Erle and Eric Schwarz.

Why is Decimal floating-point hardware a good idea anyway ?

Because it can help our industry avoid

Errors Designed Not To Be Found

like these in Microsoft’s Excel spreadsheet …

Parentheses in Microsoft’s *Excel 2000* spreadsheet can have 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. Consequently *Excel* confers supernatural powers upon some (not all) parentheses and induces other inconsistencies.

11 floating-point numbers X between $1 – 5/2^{53}$ and $1 – 13/2^{53}$ 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

27 distinct floating-point numbers X between $1 – 4/25^3$ and $1 + 22/2^{52}$ all look the same displayed.

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!

45 distinct values X between $1 + 23/2^{52}$ and $1 + 67/2^{52}$ all look the same displayed as Z does.

45 Consecutive Distinct Values X Display Like Z = 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

How could a user of *Excel* debug his work without knowing which operations depend not upon the value of their arguments but upon how they are displayed?

How can Microsoft cure the anomalies of *Excel* exhibited here?

- Switch *Excel*'s floating-point to honest decimal floating-point conforming to IEEE Standard 854 even if it has to be simulated in software.

Maybe after IBM's *Lotus 123* switches to Decimal ?

Decimal's great advantage is that, if enough digits are displayed,

What **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et.

This cuts out most of the calls to Help desks from bewildered users.

Uncertain Business Decisions

The market for decimal arithmetic is mainly commercial, mainly fixed-point.

Why support features of IEEE 854, valuable perhaps for scientific and engineering computation, if they serve probably no commercial needs?

…

Because the volume of commercial applications may make fast Decimal cheap enough for use by penurious scientists and engineers who also appreciate its

WYSIWYG

even if error-analysts tell them that Binary is mathematically superior. And then Decimal will ultimately supplant Binary in all but a few special applications.

BUT

Will Decimal Floating-Point *hardware* build up enough volume?

Not if software-simulated Decimal arithmetic is fast enough for commercial applications though maybe too slow for scientific and engineering applications.

The future for Decimal hardware seems very hard to predict.

You have succeeded too well in building Binary Floating-Point Hardware.

Floating-point computation is now so ubiquitous, so fast, and so cheap that almost none of it is worth debugging if it is wrong, if anybody notices.

By far the overwhelming majority of floating-point computations occur in entertainment and games.

IBM’s Cell Architecture: “The floating-point operation is presently geared for throughput of media and 3D objects. That means … that IEEE correctness is sacrificed for speed and simplicity. … A small display glitch in one display frame is tolerable; …” Kevin Krewell’s *Microprocessor Report* for Feb. 14 2005.

A larger glitch might turn into a feature propagated through a Blog thus:

“There is no need to find and sacrifice a virgin to the Gorgon who guards the gate to level 17. She will go catatonic if offered exactly $13.785 .”

How often does a harmful loss of accuracy to roundoff go undiagnosed?

Nobody knows. Nobody keeps score.

And when numerical anomalies are noticed they are routinely misdiagnosed.
Re *EXCEL*, see DavidEinstein's column on p. E2 of the *San Francisco Chronicle* for 16 and 30 May 2005.

Consider MATLAB, used daily by hundreds of thousands. How often do any of them notice roundoff-induced anomalies? Not often. Bugs can persist for
Decades.

e.g., log2(…) has lost as many as 48 of its 53 sig. bits at some arguments
since 1994 .

PC MATLAB's acos(…) and acosh(…) lost about half their 53 sig. bits at some arguments for several years.

MATLAB's subspace(X, Y) still loses half its sig bits at some arguments,
as it has been doing since 1988.

Nevertheless, as such systems go, MATLAB is among the best.

Why are roundoff-induced numerical anomalies so hard to find and diagnose?

Rounding errors, ostensibly negligible when committed, can propagate into substantial discrepancies only if amplified by a ***Singularity*** too near the data.

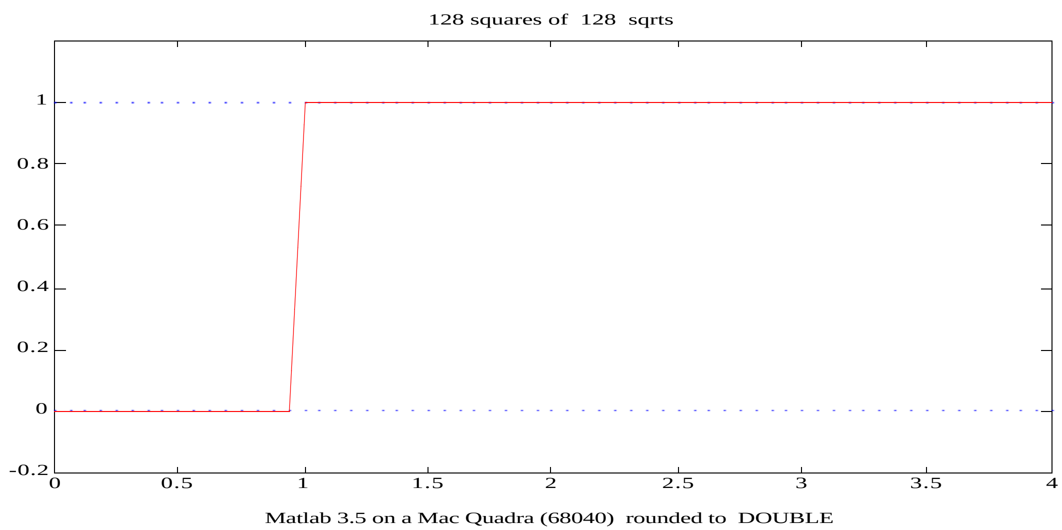
But mathematical singularities are so diverse that they defy classification. No economical way exists to detect singularities in the text of a source-code.

Division-by-Zero? Subtractive Cancellation? Vastly Accumulated Roundoff?

See §10 of `Mindless.pdf` for an example of a computation that goes utterly wrong with …

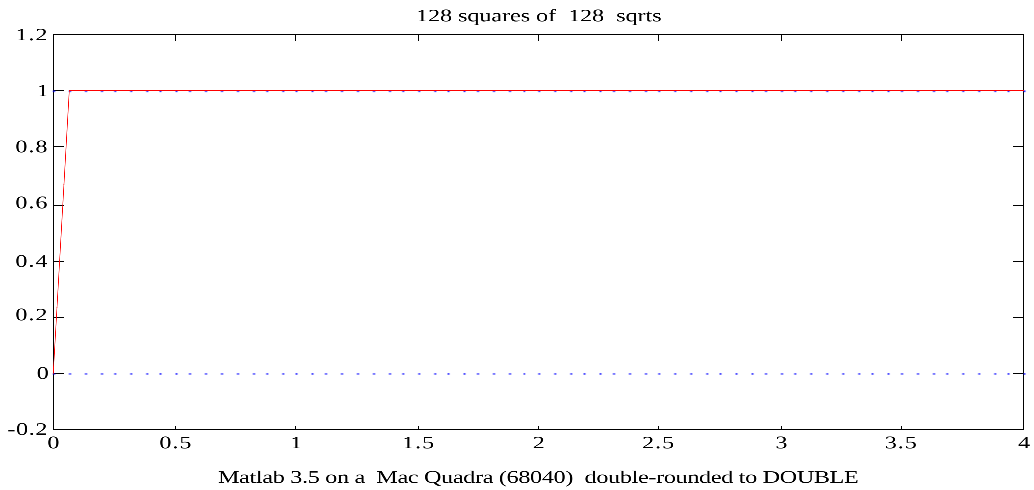
No divisions. No subtractions. Only 256 arithmetic operations:

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



This graph of computed H(X) is not the graph of H(X) = X without roundoff.

Here is a graph of the same *expression* for H(X) rounded slightly differently:



Without access to different roundings, how could someome tell that roundoff has spoiled the computation of H(X) ? Recomputation in higher precision might help, but a roughly correct graph would require at least about 40 sig. dec.

Sufficiently high precision usually suppresses roundoff, but not always. …

See §6 of `Mindless.pdf` for an example G(x) = 1 for all real arguments x . G(x) is defined by a program that produces the correct output when evaluated in infinite precision. But when G(n) is evaluated for n = 1, 2, 3, 4, …, 9999 in floating-point arithmetic of any large constant finite precision, the computed value of G(n) is 0 instead of 1 for all or almost all of those integers n .

How do nearby singularities cause damage?

We have a program F(x) intended to compute $ƒ(x)$.

Actually our program F(x) computes f(x, r) in which rounding errors are represented by r , an unknown known only to be very tiny. If rounding errors were all zero, we would get $f(x, 0) = ƒ(x)$, as desired.

Therefore the error in F(x) is $f(x, r) – f(x, 0) \approx \frac{\partial}{\partial r}f(x, 0)\cdot r$.

This error can be big despite that r is tiny only if $\frac{\partial}{\partial r}f(x, 0)$ is gargantuan, which can happen only if the data x is very near a *singularity* (where the derivative would be infinite or nonexistent) of the program's formula f(x, r) .

It may or may not be also a singularity of the desired $ƒ(x)$.

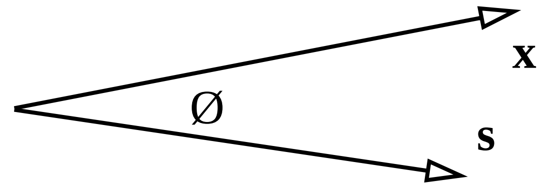
Typically the magnitude of $\frac{\partial}{\partial r}f(x, 0)$ grows like some small inverse power of the distance from the data x to the singularity, unless the singularity is a jump discontinuity as occurs in some geometrical computations. (*Inside* vs. *Outside*)

Therefore, typically, roundoff causes numerical embarrassment only at data x sufficiently near or at some singularity of the program's function f(x, 0) .

… typically … embarrassment … data … near or at … singularity of … f(x, 0) .

E.g.: What is the (smaller) angle between two intersecting lines, one parallel to column vector $\mathbf{x}$, the other to $\mathbf{s}$, in a Euclidean space of dimension ≥ 2 ?

A very simple Geometrical Computation



The usual formula: $0 \le Ø(\mathbf{x}, \mathbf{s}) := \arccos(\ |\mathbf{x}'\mathbf{s}|/(||\mathbf{x}||\cdot||\mathbf{s}||)\ ) \le \pi/2$.

But this formula errs by about $\sqrt{\text{roundoff threshold}}$ when Ø is very tiny, thus losing half the sig. digits carried by the arithmetic. This unobvious loss of accuracy is due to the $\sqrt{\ }$-like singularity of arccos(…) at arguments too near 1 .

Note that the singularity in arccos(…) is NOT a singularity of Ø(…) !

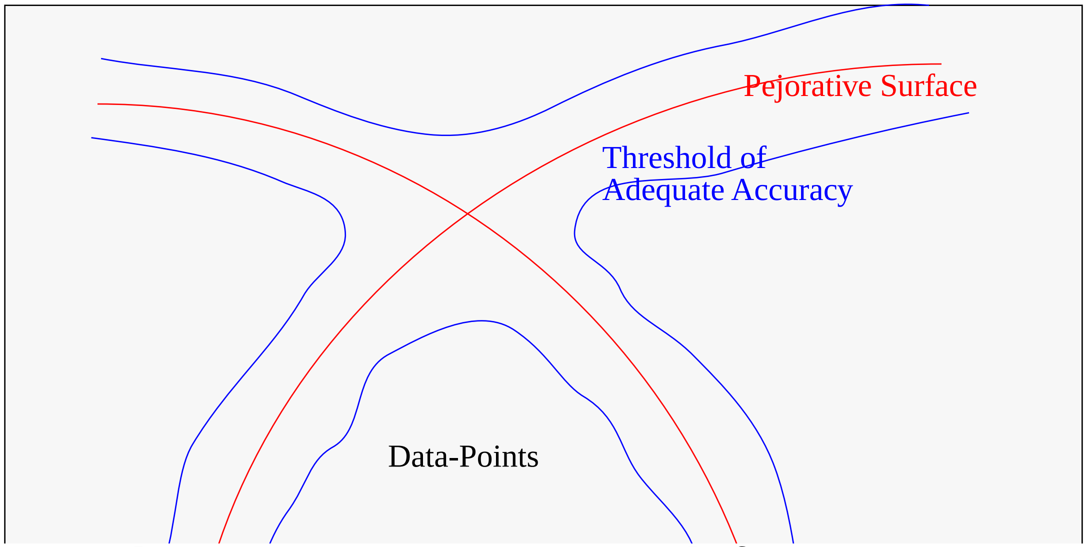
Another formula is $Ø(\mathbf{x}, \mathbf{s}) := \arcsin(\ ||\mathbf{x}\times\mathbf{s}||/(||\mathbf{x}||\cdot||\mathbf{s}||)\ )$ in three dimensions. In higher dimensions replace $||\mathbf{x}\times\mathbf{s}||$ by the root-sum-squares of the upper triangle of the matrix $\mathbf{xs}' - \mathbf{sx}'$. (*Cf*. Lagrange's Identity.) The arcsin(…) formula costs too much to compute. Worse, at some angles it loses about half the sig. digits carried; can you see why? Can you find a neat formula always fully accurate?

See §12 of `Mindless.pdf`.

Either formula for Ø is satisfactory if computed in at least twice the precision to which data is stored and results desired, provided that wider precision is not too slow. One precaution is necessary:

Use $Ø(\mathbf{x}, \mathbf{s}) := \arccos(\min\{\ |\mathbf{x}'\mathbf{s}|/(||\mathbf{x}||\cdot||\mathbf{s}||),\ +1\})$. Trials of random near-parallel 3-vectors $\mathbf{x}$ and $\mathbf{s}$, say $\mathbf{s} := \pm\pi\cdot\mathbf{x}$ rounded, encounter quotients $|\mathbf{x}'\mathbf{s}|/(||\mathbf{x}||\cdot||\mathbf{s}||) > 1$, invalid arguments for arccos(…) , with probability at least about 1/5 even with doubled-precision arithmetic**!** … trouble *at* the singularity**!**

In general, All Accuracy is Lost if Data Lie on a *Pejorative Surface*:



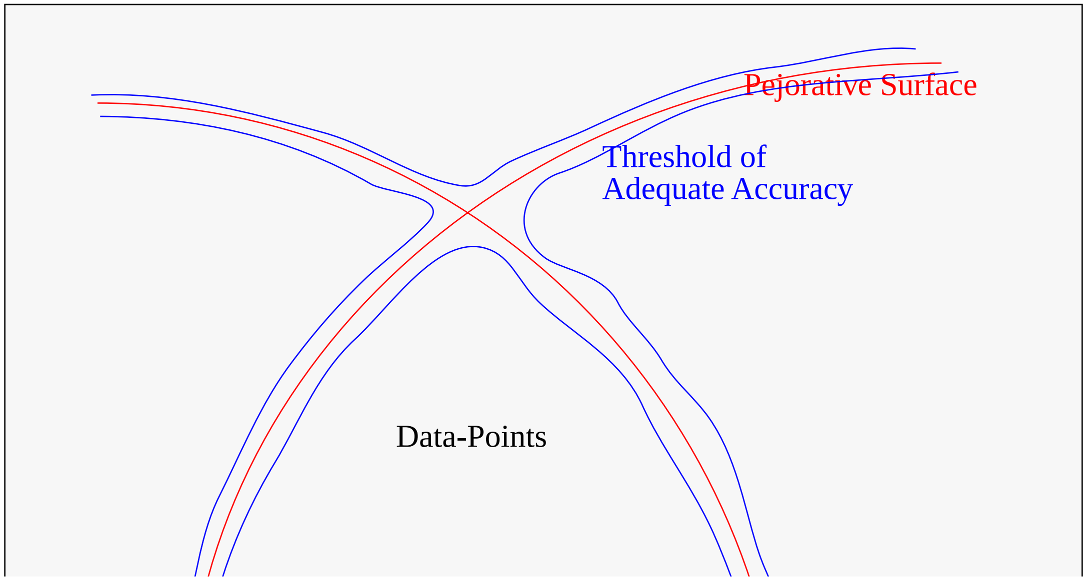
Accuracy is Adequate at Data Far Enough from Pejorative Surfaces.

e.g.:

Data Points	Computed Result	Pejorative Data	Threshold Data
Matrices	Inverse	Singular Matrices	Not too ill-conditioned
Matrices	Eigensystem	Degenerate Eigensystems	Not too near degeneracy
Polynomials	Zeros	Repeated Zeros	Not too near repeated
4 Vertices	Tetrahedron’s Volume	Collapsed Tetrahedra	Not too near collapse
Diff’l Equ’n	Trajectory	Boundary-Layer Singularity	Not too “Stiff”

Numerically Unstable Algorithms introduce additional Undeserved Pejorative Surfaces often so “narrow” that they can be almost impossible to find by uninformed testing.

Carrying Extra Precision Usually Squeezes Thresholds of Adequate Accuracy Towards Pejorative Surfaces.



11 bits of extra precision (beyond the data's) for *all* intermediate calculations usually diminishes the incidence of embarrassment due to roundoff by a factor typically smaller than 1/2000 . But Microsoft's compilers deny access to Pentiums' extra precision.

To detect unsuspected numerical instability in software, we must test it on data closer to a pejorative surface than the threshold of adequate accuracy. Whether due to an intrinsically ill-conditioned problem or to the choice of an unstable algorithm that malfunctions for otherwise innocuous data, mistreated data nearer a pejorative surface than the threshold of adequate accuracy can be ubiquitous and yet so sparse as almost surely not to be found by random testing**!** A recent instance is the 1994 Pentium FDIV bug. Lots of stories about it are on the web, my web page included. *Cf*. my `MktgMath.pdf` and `Mindless.pdf`.

Even if each proposed scheme for automated error-analysis has some applications upon which it works well, it cannot succeed in general because

singularities of functions of several variables are too diverse to classify,

so no routine way can exist to locate singularities of which you are unaware.

Their spikes are too often too narrow to be located by hill-climbing.
See `Mindless.pdf` for striking examples.

The best we can hope for are schemes that help us diagnose likely sources of a numerical malfunction when data brings it to light, usually accidentally.

The question about computer-assisted error-analysis worth considering is not

Which scheme(s) will work?

(none works universally, and a few computations defy them all)

but

Which scheme(s) can offer a lot of benefit at a tolerable cost?

Hypothetical Case Study: Bits Lost in Space

Imagine plans for unmanned astronomical observatories in outer space. For details see §11 of `Mindless.pdf` .

Directions to planets and 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

Angles must satisfy $-\pi \le \theta \le \pi$ and $-\pi/2 \le \phi \le \pi/2$, and similarly for Θ and Φ .

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 $|\theta-\Theta| \bmod 2\pi$. Three implementations of function ψ have been compared; they are called u, v and w . Of millions of tests, here are the six that aroused suspicion:

$\theta-\Theta$:	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 :	0.0015*8221229*	*0.0*	0.000*345266977*	0.0005*98019978*	3.1408*2050*	3.14055*681*
$\psi \approx$ v :	0.00159324*868*	0.00024414061*0*	0.00017263348*9*	0.00056223187*1*	3.140*61618*	3.140*61618*
$\psi \approx$ w :	0.00159324*868*	0.00024414061*0*	0.00017263348*9*	0.00056223187*1*	3.140780*44*	3.140548*47*

Subprograms u, v and w agree otherwise. Which if any of them dare we trust?

Which if any of subprograms u, v and w dare we trust? They were rerun on the suspect data in different rounding modes mandated by IEEE Standard 754. Fortunately, they were rerun on a system that permitted redirections of all default roundings (to nearest) without recompilation of the subprograms. Here are some results:

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

Subprogram w seems practically indifferent to changes in rounding's direction. In fact, it uses an unobvious formula stable for all admissible data. Subprogram u uses a formula easy to derive but numerically unstable for subtended angles too near 0 or π . Subprogram v uses a formula familiar to astronomers though it loses half the digits carried when the subtended angle is too near π , where astronomers are most unlikely to have tried it. Formulas are in `Mindless.pdf`.

Without access to source code, nor to another subprogram known to be reliable, how else might anyone decide which program(s) to distrust first?

Rerunning with redirected roundings is the only practicable way here.

The ability to redirect rounding is mandated by IEEE Standard 754 (1985) for floating-point arithmetic.

Some compilers have supported dynamically redirected rounding, but almost no programming languages support it. The exceptions are a few C99 compilers.

Java outlaws directed rounding.

The lack of use of this capability will lead to its atrophy. Use it or lose it.

For other desirable debugging tools we may wish were provided by programming development systems, using high-precision floating-point and interval arithmetic combined (they are not helpful enough by themselves), see §14 of `Mindless.pdf` .

Quadruple precision is an alternative to error analysis:

Perform the computation in extravagantly more precision than seems necessary. It reduces the incidence of embarrassment due to roundoff below a level anyone cares about unless the data lies upon or along a pejorative surface.

… though nothing is infallible.

More Business Decisions

If Quadruple Precision runs too slowly, it will not be used routinely, and then will not obviate the need for almost all error-analysis.

But no computer's sales have ever been influenced significantly by the speed of its quadruple-precision arithmetic. *E.g.*, …

Hardware: IBM /360-85, /370, /390. DEC VAX.
Software: SUN SPARCs. HP/Intel Itanium.

So, *fast* Quadruple Precision is most unlikely to become commonplace soon.

So, we need aids to error-analysis.

… Aids like those discussed in §14 of `Mindless.pdf`.

Who can pay for the introduction of aids to error-analysis into programming development systems? Not error-analysts by themselves. The beneficiaries of error-analysis should pay, but almost none are aware of benefits nor hazards.

So, aids to error-analysis ought to be ubiquitous, required by standards for hardware, programming languages and compilers. As a matter of, say, …

National Security ?

Epilogue

In 1953, when I began to use the Ferranti-Manchester Mk. I electronic computer at the University of Toronto, the general consensus was that

"Error Analysis is feasible for Fixed-Point Arithmetic,
but not for Floating-point."

At that time a Numerical Analyst could try to achieve immortality by having his name attached to a numerical method that might work when others failed.

e.g., Danilewski's method, Milne's method, Frame-Souriau-Faddeev method, …

The situation began to change in the late 1950s …

By 1957 successful approaches to Floating-Point Error-Analysis had been developed by

Wallace Givens at Argonne National Labs near Chigago
James H. Wilkinson at the National Physical Labs, Teddington, near London
Fritz Bauer at the Technische Hochschule in Munich
W. K. at the University of Toronto

"Backward Error-Analysis" was just the most widely mentioned among those approaches.

Now a large number of widely used and valuable numerical programs
-- accurate, robust and fast, with few, rare, and mostly known failure modes --
owe their development as well as their validity to modern error-analyses about which their users know nothing.

This is as it should be:

The essence of civilization is that we benefit from the experience of others without having to relive it.

By 1973, qualities that make for a good floating-point arithmetic had become apparent:
Call them "Mathematical Integrity".
See books by P. Sterbenz, D.E. Knuth, N.J. Higham, ...

By 1980, IEEE 754 hardware designs that preserved both Mathematical Integrity and high performance were produced: *e.g.*, George Taylor's for the ELXSI 6400.

Now a large number of widely used and valuable numerical programs
-- accurate, robust and fast, with few, rare, and mostly known failure modes --
owe their development as well as their validity to IEEE 754 without obliging their users to know anything about it. This too is as it should be.

The foregoing advances in civilization could be set back by either of two
"Business Decisions":

“Business Decision” #1: Jettison “features” of IEEE 754 not needed for a large market from an arithmetic engine targeted to that market initially.

If that engine succeeds in one market, it will inevitably find its way into other markets not contemplated initially. Some of these markets will rely upon software that implicitly relies rarely on one of the jettisoned features.

Will occasional malfunctions matter?
If so, who will debug them? How?

IBM is reported to intend its Cell architecture, initially developed for games, to be sold also for supercomputers and medical imaging.

“Business Decision” #2: Compared with games, entertainment, commerce and communications, the market for scientific and engineering computation is picayune, so nothing much can be earned by supporting its peculiar needs in compilers, debuggers and programming development systems. How shall programs upon which we rely heavily be debugged?

It is **futile** to single-step through floating-point programs intended to run at **gigaflops**.

It is **futile** to single-step through floating-point programs intended to run at **gigaflops**.

What to do instead? Techniques discussed in §14 of `Mindless.pdf`.

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.