

Desperately Needed Remedies for the Undebuggability of Large Floating-Point Computations in Science and Engineering

W. Kahan, Prof. Emeritus
Math. Dept., and
Elect. Eng. & Computer Sci. Dept.
Univ. of Calif. @ Berkeley

Prepared for the
Seminar on Scientific and Statistical Computing
4:30 pm., Thurs. 10 April 2014, Eckart 133
University of Chicago

This document is posted at [<www.eecs.berkeley.edu/~wkahan/B0u1der.pdf>](http://www.eecs.berkeley.edu/~wkahan/B0u1der.pdf).
For most details omitted here see [<www.eecs.berkeley.edu/~wkahan/Bou1der.pdf>](http://www.eecs.berkeley.edu/~wkahan/Bou1der.pdf)
and [<www.eecs.berkeley.edu/~wkahan/Mindless.pdf>](http://www.eecs.berkeley.edu/~wkahan/Mindless.pdf)

Desperately Needed Remedies for the Undebuggability of Large Floating-Point Computations in Science and Engineering

Abstract:

How long does it take to either allay or confirm suspicions, should they arise, about the accuracy of a computed result? Often diagnosis has been overtaken by the end of a computing platform's service life. Diagnosis could be sped up by at least an order of magnitude if more users and developers of numerical software knew enough to demand the needed software tools. Almost all these have existed though not all of them together in one place at one time. These tools cope with vulnerabilities peculiar to Floating-Point, namely roundoff and arithmetic exceptions. Programming languages tend to turn exceptions into branches which are prone to error. In particular, unanticipated events deemed ERRORS are handled in obsolete ways inherited from the era of batch computing. There are better ways. They would have prevented the crash of Air France #447 in June 2009, among other things.

This document has been posted at www.eecs.berkeley.edu/~wkahan/B0u1der.pdf. More details appear at .../Bou1der.pdf, .../NeeDebug.pdf and .../Mindless.pdf. All these documents remain susceptible to revision.

“This ... paper, by its very length, defends itself against the risk of being read.”
... attributed to Winston S. Churchill

To fit into its allotted time,
this paper’s oral presentation skips over most of the details.
It is intended to induce you to investigate those details.

“A fanatic is one who can’t change his mind and won’t change the subject.”
... Winston S. Churchill (1874 - 1965)

Am I a fanatic?

If so, you have been warned.

What is the incidence of Floating-Point computations of the worst kind,
wrong enough to mislead but not so wrong as is obviously wrong ?

Nobody knows. Nobody is keeping score.

Evidence exists implying an incidence rather greater than is generally believed.

Two Kinds of Evidence:

- **Persistence** in Software and in Programming Texts of numerically flawed formulas that have *withstood* rather than *passed* the *Test of Time* . For example, ...
Naive solutions of quadratic equations; ... of discretized differential equations
- Occasional **Revelations** of gross inaccuracies, in widely used and respected packages like MATLAB and LAPACK, caused by bugs lying hidden for years. *E.g.*, ...
Over 40 years of occasional *underestimates*, some severe, of matrices' ranks.

Evidently, providers of numerical software need help to debug it; they need abundant assistance from users.

How much debugging of numerical software is included in a chemist's job-description?

Distinctions between users and providers of numerical software are blurred by developers who incorporate, into their own software, modules developed by others. *e.g.*, LAPACK

If providers expect users to help debug numerical software, they (and we) must find ways to reduce the costs in time and expertise of investigating numerical results that arouse suspicions.

Later we shall see why the **earliest symptoms** of hitherto unsuspected **gross inaccuracies** that will befall our software at some **unknown innocuous data** are highly likely to be inaccuracies, at other data, barely bad enough to arouse suspicions.

How much can investigation of a suspect Floating-Point computation's accuracy cost?
Often more than the computed result is worth.

Computers are now so cheap, most perform computations of which no one is worth very much:
Entertainment, Communications, Companionship, Embedded Controllers
are computers' most prevalent and most remunerative uses;
not our scientific and engineering computations.

A Problem of Misperception in the Marketplace:

The software tools needed to reduce by orders of magnitude the costs of debugging anomalous Floating-Point computations have almost all existed, but not all in the same package, and not in current software development systems.

Why not? Ignorance.

- The producers of software development systems are unaware that such tools could be produced, much less that there is a demand for them.
- The scientists and engineers who would benefit from such tools are hardly aware of them, much less that those tools should be requested.

Those tools have been described on my web pages. For more details about them see [<.../Boulder.pdf>](#) , [<.../NeeDebug.pdf>](#) and [<.../Mindless.pdf>](#).

Computer scientists worldwide are working hard on schemes to debug and verify software, especially in the context of parallel computation, but practically none for Floating-Point software.

What is it about Floating-Point that repels Computer Scientists?

Floating-Point arithmetic usually approximates *Real* arithmetic closely, but not always.

- What you see is not exactly what you get.

What you get is not exactly what your program commanded.

Consequently what you get can be *Utterly Wrong* without any of the usual suspects: *i.e.* no subtractive cancellation, no division, no vast number of rounded operations.

For a simple didactic example see www.eecs.berkeley.edu/~wkahan/WrongR.pdf

- Worse, unlike *Correctness* of non-numerical computer programs, *Accuracy* of Floating-Pt. programs is *Not Transitive* if composed.

This means that ...

If program $H(X)$ approximates function $h(x)$ in all digits but its last, and if program $G(Y)$ approximates function $g(y)$ in all digits but its last,

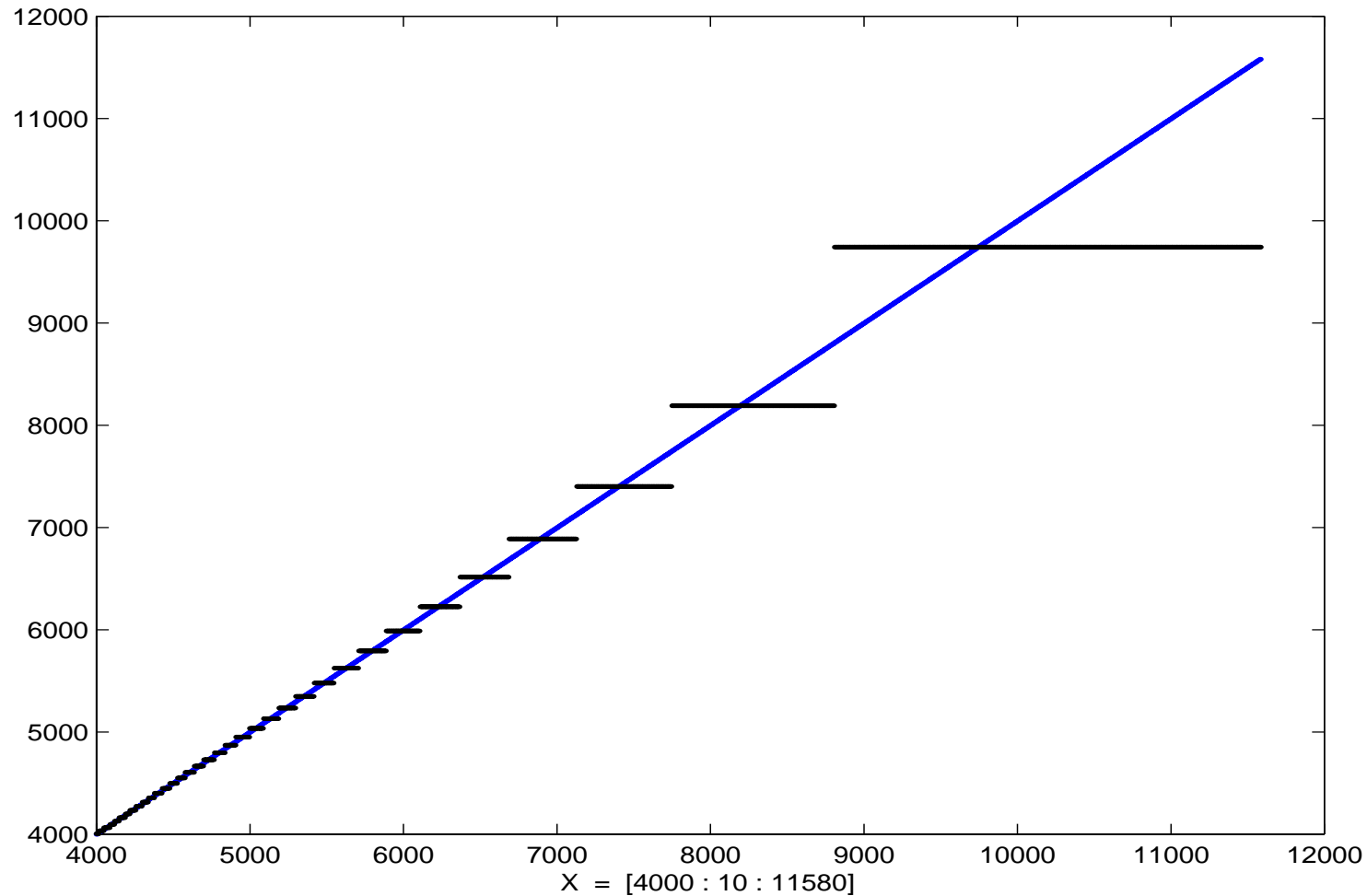
yet program $F(X) := G(H(X))$ may approximate function $f(x) := g(h(x))$

Utterly Wrongly over a large part of its domain.

Here is a simple didactic example, albeit contrived:

$h(x) := \exp(-x^{-4}) @ x > 1$; $g(y) := 1/4 \sqrt{-\log(y)} @ 0 < y < 1$; $f(x) := g(h(x)) = x @ x > 1$.

$f(x) = x$ vs. $G(H((x))) = (-\log(\exp(-x^{-4})))^{-1/4}$



This is explained in pp. 24 - 25 of my posting www.cs.berkeley.edu/~wkahan/MxMulEps.pdf.

- How high is the incidence of misleadingly inaccurate computed results?

We cannot know. Nobody is keeping score.

- What evidence suggests that it's higher than generally believed?

Two kinds of evidence, **Revelation** and **Persistence** :

- **Revelation**, after long use, that a widely trusted program produces, for otherwise innocuous input data, results significantly more inaccurate than previously believed.
- **Persistence** of numerically naive and thus vulnerable formulas in the source-code of some programs, and in some published papers and textbooks.

A typical example of naiveté too common in programming textbooks:

The zeros z of a real quadratic $\alpha \cdot z^2 - 2\beta \cdot z + \gamma$, assuming $\alpha \neq 0$ & $\gamma \neq 0$, are

$$z_1 := (\beta + \sqrt{(\beta^2 - \alpha \cdot \gamma)})/\alpha \quad \text{and} \quad z_2 := (\beta - \sqrt{(\beta^2 - \alpha \cdot \gamma)})/\alpha \quad \text{naively.}$$

Numerically more reliable (absent over/underflow) formulas for the zeros are

$$\delta := \beta^2 - \alpha \cdot \gamma; \text{ if } \delta < 0 \text{ then } \{ z_1 := \beta/\alpha + \mathbf{i}\sqrt{-\delta}/\alpha; z_2 := \beta/\alpha - \mathbf{i}\sqrt{-\delta}/\alpha \}$$

$$\text{else } \{ \zeta := \beta + \text{copysign}(\beta, \sqrt{\delta}); z_1 := \zeta/\alpha; z_2 := \gamma/\zeta \}.$$

Do you see why? Where are the formulas' singularities? What happens near them?

- After long use, a widely trusted program is discovered to have produced, for otherwise innocuous input data, results significantly more inaccurate than previously believed.



The Vancouver Stock Exchange maintained an index of (mainly mining) stock prices.

On Fri. evening 25 Nov. 1983 the index ended at 524.811 .

On Mon. morning 28 Nov. 1983 the index began at 1098.892 ; was it correct?

Stock prices didn't rise so much over a weekend. Roundoff had accumulated over years.



Given m -by- n matrix B and a small tolerance τ , we seek the least “rank” r for which

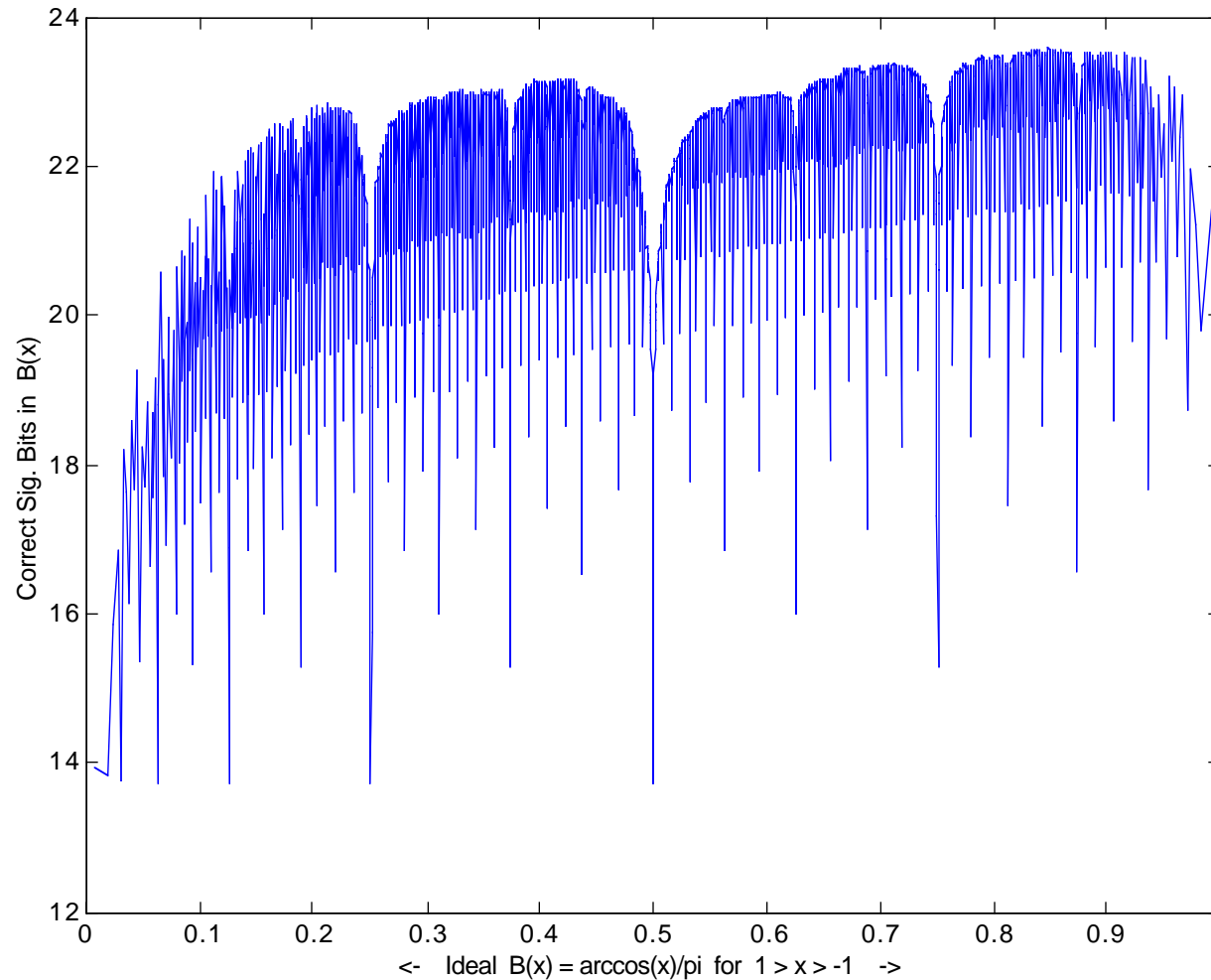
$$\begin{matrix} & n \\ & \boxed{B} \\ m & \end{matrix} \approx \begin{matrix} r \\ \boxed{Q} \\ m & \end{matrix} \cdot \begin{matrix} n \\ \boxed{R} \\ r & \end{matrix} \text{ within } \pm\tau .$$

When rank r is small, this factorization reveals crucial structural information used to analyze Big Data and to design control systems, *etc.*

A fast “Pivoting QR” factorization had been used widely for over [forty years](#) despite that it sometimes *over*-estimated r a little. Moderate *over*-estimates cause little harm.

In 2008 otherwise innocuous matrices B were discovered for which roundoff caused r to be *under*-estimated severely enough that significant data was missed, and some control systems misbehaved. Since then the program’s defect has been repaired, we hope.

Of 24 Sig. Bits Carried, How Many are Correct in EDSAC's $B(x)$?



Unnoticed for **two years**, accuracy spiked down wherever $B(x)$ came near (not exactly) a small odd integer multiple of a power of $1/2$. The smaller the integer, the wider and deeper the spike, down to near half the sig. bits lost. Such arguments x , common in practice, were missed in tests.

Roundoff-Induced Anomalies Evade Expert Searches for Too Long:

- PATRIOT Anti-Missile Missiles missed a SCUD that hit a barracks in the Gulf War.
- From 1988 to 1998, MATLAB's built-in function `round(x)`, that rounds x to a nearest integer-valued floating-point number, rounded all sufficiently big odd integers to the next bigger even integer in PC-MATLABs' 3.5 and 4.2. Not Macs.
- For more than a decade, MATLAB has been miscomputing $\text{gcd}(3, 2^{80}) = 3$, $\text{gcd}(28059810762433, 2^{15}) = 28059810762433$, $\text{lcm}(3, 2^{80}) = 2^{80}$, $\text{lcm}(28059810762433, 2^{15}) = 2^{15}$, and many others, with no warning.

Anomalies due to Over/Underflow can evade expert searches for too long too.

In 2010, excessive inaccuracies were discovered in LAPACK's programs `_LARFP` and traced to underflows caused by the steps taken to avoid overflows. Whether the revisions to those programs promulgated subsequently are fully satisfactory remains to be seen.

What exposes a misjudgment due to rounding errors ?

- A calamity severe enough to bring about an investigation, and investigators thorough and skilled enough to diagnose correctly that roundoff was the cause (if it was).
This *combination* appears to have occurred extremely rarely, if at all.
- Suspicions aroused by computed results different enough from one's expectations.
Someone would have to be exceptionally observant, experienced and diligent.
- Discordant results of recomputations using different arithmetics or different methods.
What would induce someone to go to the expense of such a recomputation?

In the mid 1990s a program written at NASA Ames predicted deflections under load of an airframe for a supersonic transport that turned out destined never to be built. Though intended for CRAY-I and CRAY-2 supercomputers, the program was developed on SGI Workstations serving as terminals.

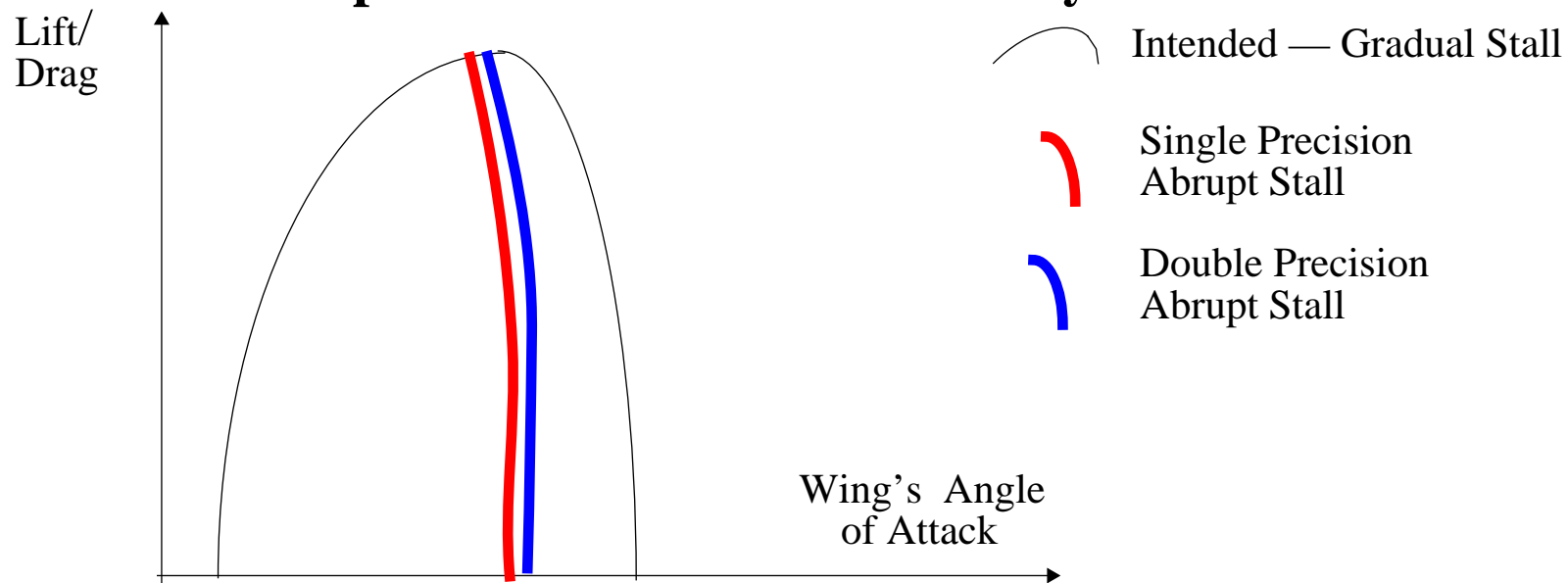
When a problem with a mesh coarse enough to fit in the workstation was run on all three machines, three results emerged disagreeing in their third sig. dec. This had ominous implications for the CRAYs' results from realistic problems with much finer meshes.

I traced the divergence to the CRAYs' idiosyncratic biased roundings. Adding iterative refinement to the program, a minor change, rendered the divergence tolerable. To rid the program of its worst errors would have required a major change; see my web page's .../Math128/FloTriK.pdf .

- What if the user of a widely trusted program doesn't know that its results, for some otherwise innocuous input data, are significantly more inaccurate than the user believes?

This almost happened to a graduate student of aeronautical engineering in the early 1960s when his scheme to enhance lift for wings of Short-Takeoff-and-Landing aircraft seemed to suffer from abrupt onset of stall, according to his computations on an IBM 7090.

Abrupt Stall of Lift Enhanced by Blown Slots ?



Abrupt stall “caused” by inaccurate LOG in **Single**, by lack of guard digit in **Double** precision. Only after his was one of several programs chosen to test a new LOG's accuracy did he learn that the abrupt stall was entirely an artifact of roundoff. He resuscitated his research. For details see pp. 23 - 26 of www.eecs.berkeley.edu/~wkahan/NeedDebug.pdf .

Why are roundoff-induced misjudgments, formerly rare, likely to become rather less rare?

Computers' memories have become HUGE because memory has become CHEAP, and more so are vast numbers of Graphics Processors produced & sold for entertainment.

But moving data through the memory system has become costly in TIME and ENERGY.

4-byte-wide floats cost half as much as 8-byte-wide doubles.

Graphics Processors are optimized for floats.

So computations formerly performed in double are being converted to float instead.

Why not ?

Arithmetic precision of double : 53 sig. bits ~ 16 sig.dec. $\epsilon \approx 2^{-52}$

of float : 24 sig. bits ~ 7 sig.dec. $\epsilon \approx 2^{-23}$

7 correct sig. dec. is more than adequate accuracy

for almost all computed results used by scientists and engineers.

But what you see is not always what you get.

A computation formerly carrying 16 sig.dec. could afford to lose 10 and still yield 6.

How many sig. dec. can that computation now carrying 7 afford to lose?

Most computational methods lose a number of sig.dec. independent of how many were carried.

What exposes a misjudgment due to rounding errors ?

It's unlikely to be exposed.

Why must such misjudgments be happening?

Programs that depend upon some Floating-Point computation are being written by far more people than take a course in Numerical Analysis with enough Error-Analysis to sensitize them to the risks inherent in roundoff, especially in `float` computations.

“Acquiescing to rounded arithmetic places you in a state of sin.” — D.H. Lehmer

People clever and knowledgeable in their own domains of science, engineering, statistics, finance, medicine, *etc.*, are naively using in their programs formulas mathematically correct but numerically vulnerable, instead of numerically robust but unobvious formulas.

Many such formulas are posted on my web pages; the next page exhibits a lengthy list taken mostly from p. 22 of `.../NeeDebug.pdf`.

We may depend unwittingly upon some of these clever people's programs via the world-wide-web, the cloud, medical equipment, navigational apparatus, *etc.* How can we defend ourselves against numerical naiveté, or at least enhance the likelihood that their programs' numerical vulnerabilities will be exposed, preferably before too late?

Additional relevant postings on www.cs.berkeley.edu/~wkahan/...
Textbook formulas withstand, not pass, the Test of Time: .../Triangle.pdf
Simple geometrical miscalculations with cross-products: .../MathH110/Cross.pdf
Bad solutions for good equations .../Math128/FailMode.pdf
Lots about *Iterative Refinement* .../p325-demmel.pdf
Eigensystem refinement .../Math128/Refineig.pdf
General symmetric eigensystem refinement .../Math128/GnSymEig.pdf
Refine finite-differenced boundary-value problem .../Math128/FloTriK.pdf
.../Cantilever.pdf
Discriminants of quadratics .../Qdrtc.pdf
Roundoff creates spurious roots .../Math128/SOLVEkey.pdf
Roundoff causes mysterious overflows .../CS279/DHBLNG.pdf
MATLAB's loss is nobody's gain .../MxMulEps.pdf
“Business Decisions” can undermine numerical integrity .../ARITH_17.pdf
The improbability of probabilistic assessments of roundoff .../improber.pdf
The futility of mindlessly automatic error-analysis .../Mindless.pdf

How necessary is the investigation of every suspicious computed result as possibly a harbinger of substantially worse to come?

... if not symptomatic of a failure of some physical theory — a potential *Nobel Prize* !

“Les doutes sont fâcheux plus que toute autre chose.”

(Doubts cause more trouble than the worst truths.)

Le Misanthrope III.v (1666) by Molière (1622 - 1673)

After we have seen the most likely causes of a catastrophic numerical inaccuracy, we shall see why its possibility is most likely to be exposed by incidents that raise suspicions about computed results.

That is why suspicious computed results must be investigated.

To justify this necessity, we must understand what can turn almost infinitesimal rounding errors into grossly wrong results:

Perturbations get Amplified by Singularities Near the Data.

How Singularities Near Data Amplify Perturbations of that Data.

Perturbed data $\mathbf{x} \rightarrow \mathbf{x} \pm \Delta\mathbf{x}$
 perturbs $f(\mathbf{x}) \rightarrow f(\mathbf{x} \pm \Delta\mathbf{x}) = f(\mathbf{x}) \pm \Delta f(\mathbf{x}) \approx f(\mathbf{x}) \pm f'(\mathbf{x}) \cdot \Delta\mathbf{x} .$

$\Delta f(\mathbf{x}) \approx f'(\mathbf{x}) \cdot \Delta\mathbf{x}$ can be huge when $\Delta\mathbf{x}$ is tiny only if derivative $f'(\mathbf{x})$ is gargantuan.

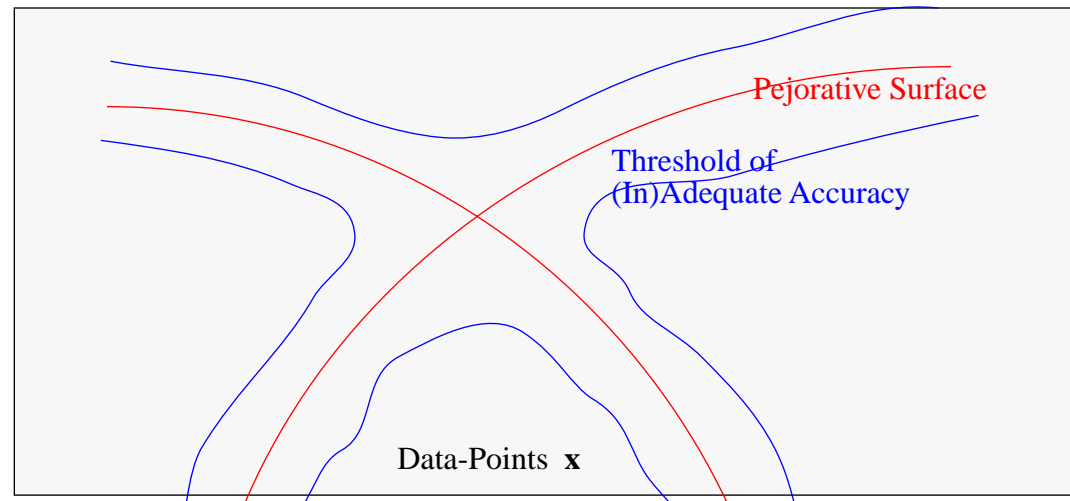
This can happen only if \mathbf{x} is near enough to a *Singularity* of f where its derivative $f' = \infty .$

Let's call the locus (point, curve, surface, hypersurface, ...) of data \mathbf{x} whereon $f'(\mathbf{x}) = \infty$ the “**Pejorative Surface**” of function f in its domain-space of data.

For example ...

Data Points	Computed Result	Data on a Pejorative Surface	Threshold Data
Matrices	Inverse	Cone of Singular Matrices	Not too “Ill-Conditioned”
Matrices	Eigensystem	... with Degenerate Eigensystems	Not too near Degenerate
Polynomials	Zeros	... with Repeated Zeros	Not too near repeated
4 Vertices	Tetrahedron's Volume	Collapsed Tetrahedra	Not too near collapse
Diff'l Equ'n	Trajectory	... with boundary-layer singularity	Not too “Stiff”

All Accuracy can be Lost at Uncertain Data on a Pejorative Surface



$f(\mathbf{x})$'s accuracy is adequate only at data \mathbf{x} far enough from Pejorative Surfaces.

Suppose the data's "Precision" bounds its tiny uncertainty $\Delta\mathbf{x}$ thus: $\delta\xi \geq \|\Delta\mathbf{x}\|$.

Then $f(\mathbf{x} \pm \Delta\mathbf{x})$ inherits uncertainty $\delta\xi \cdot \|f'(\mathbf{x})\| \geq \|\Delta f\|$, roughly, from uncertain data.

How fast does $\|f'(\mathbf{x})\| \rightarrow \infty$ as $\mathbf{x} \rightarrow$ (a Pejorative Surface) ?

Let $\delta\pi(\mathbf{x}) :=$ (distance from \mathbf{x} to a nearest Pejorative Surface). *Typically* (not always !)

$\|f'(\mathbf{x})\|$ is roughly proportional to $1/\delta\pi(\mathbf{x})$ while $\delta\pi(\mathbf{x})$ is small enough.

Uncertainty $\delta\xi \geq \|\Delta\mathbf{x}\|$ causes $f(\mathbf{x} \pm \Delta\mathbf{x})$ to "Lose" to the data's uncertainty roughly

Const. $- \log(\delta\pi(\mathbf{x})) + \log(\delta\xi)$ dec. digits.

Rounding Errors often resemble Uncertain Data

Suppose program $F(\mathbf{X})$ is intended to compute $f(\mathbf{x})$ but actually $F(\mathbf{X}) = f(\mathbf{X}, \mathbf{r})$ in which column \mathbf{r} represents the rounding errors in F and $f(\mathbf{x}, \mathbf{0}) = f(\mathbf{x})$. The precision of the arithmetic imposes a bound like $\rho > \|\mathbf{r}\|$ analogous to the uncertainty $\delta\xi$ used above. To simplify exposition, assume the data \mathbf{X} we have equals the data \mathbf{x} we wish we had.

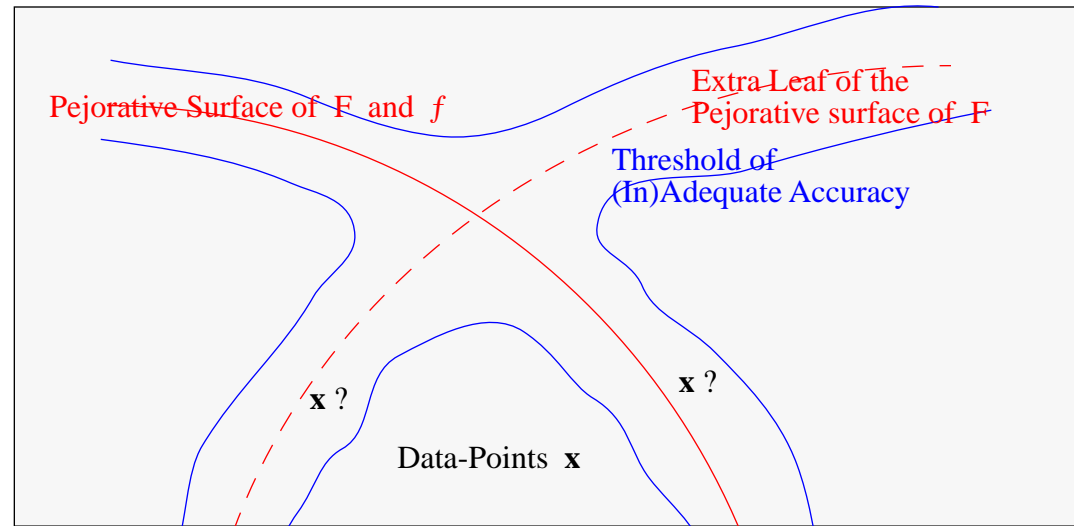
Let $f_r(\mathbf{x}) := \partial f(\mathbf{x}, \mathbf{r}) / \partial \mathbf{r} |_{\mathbf{r}=\mathbf{0}}$. Because ρ is so tiny, program $F(\mathbf{x})$ actually computes $f(\mathbf{x}, \mathbf{r}) \approx f(\mathbf{x}, \mathbf{0}) + f_r(\mathbf{x}) \cdot \mathbf{r} = f(\mathbf{x}) + f_r(\mathbf{x}) \cdot \mathbf{r}$, so $\|F(\mathbf{x}) - f(\mathbf{x})\| \approx \|f_r(\mathbf{x}) \cdot \mathbf{r}\| < \|f_r(\mathbf{x})\| \cdot \rho$.

Error $F(\mathbf{x}) - f(\mathbf{x})$ can be huge when \mathbf{r} is tiny only if derivative f_r is gargantuan, which can happen only if \mathbf{x} is near enough to a *Singularity* of f where its derivative $f_r = \infty$.

Let's call the locus (point, curve, surface, hypersurface, ...) of data \mathbf{x} whereon $f_r(\mathbf{x}) = \infty$ the "*Pejorative Surface*" of program F in its domain-space of data. Program F 's pejorative surface almost always contains function f 's.

Numerically bad things happen when the program's pejorative surface has an *Extra Leaf* extending beyond the function's. Then at innocuous data \mathbf{x} too near that Extra Leaf of Pejorative Surface the program $F(\mathbf{x})$ produces undeservedly badly inaccurate results though $f(\mathbf{x})$ is unexceptional.

All or Most Accuracy is Lost if Data lie on a “Pejorative” Surface



$F(\mathbf{x})$ is accurate enough only at data \mathbf{x} far enough from all pejorative surfaces.

An opportunity to discover whether the program’s pejorative surface has an Extra Leaf arises when $F(\mathbf{x})$ is inaccurate enough to arouse suspicion. Does $F(\mathbf{x})$ deserve its inaccuracy because \mathbf{x} is “Ill-Conditioned” — too close to the Pejorative Surface of f ? Or is the inaccuracy undeserved because innocuous data \mathbf{x} is unlucky — too close to a hitherto unsuspected Extra Leaf? These important questions are difficult to resolve.

Why is their resolution necessary?

A suspicious result may be the first and only warning that a defective program will produce a badly misleading result from otherwise innocuous data.

A computation has produced a suspicious result.

- Is it inaccurate because the data is “Ill-Conditioned” ? OR ...
- Is the data innocuous except that the program dislikes it?

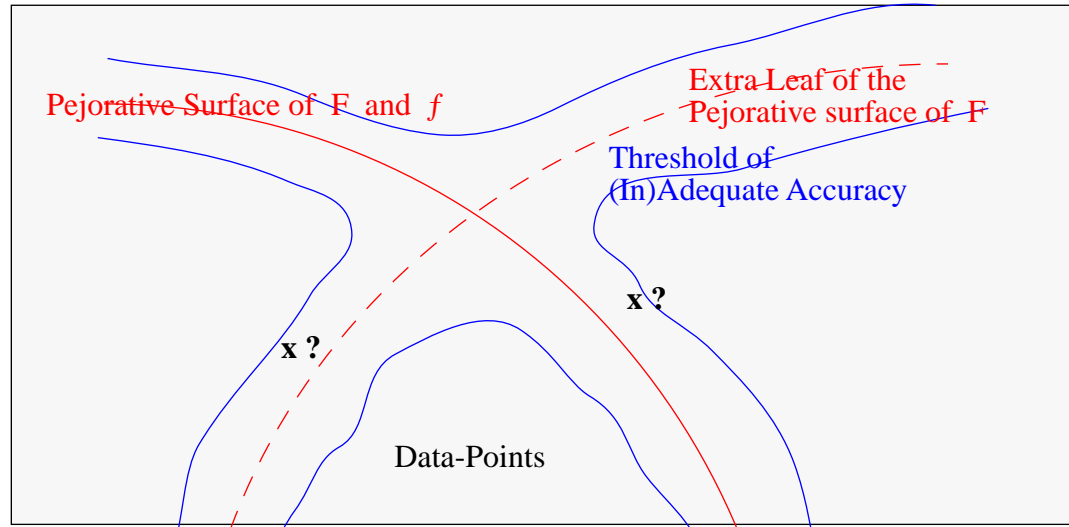
We must find out lest later we accept unwittingly an utterly inaccurate result at some other innocuous data much closer to the program’s Extra Leaf of its Pejorative Surface, of whose existence we had chosen to remain unaware.

Two choices present themselves:

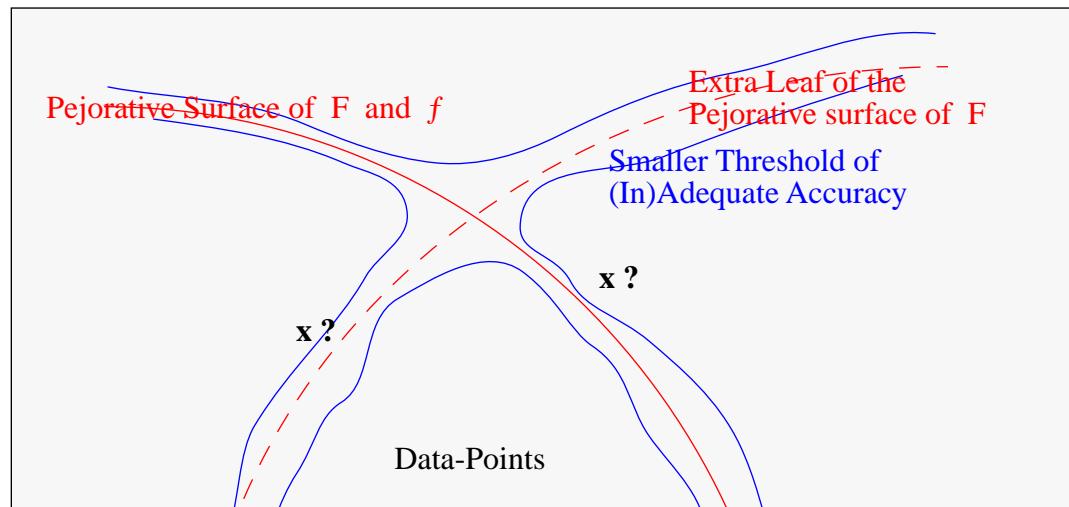
- Enhance the likelihood of these difficult questions’ resolution by supplying tools to reduce by orders of magnitude the cost in talent and time to resolve them. OR ...
- Reduce by orders of magnitude the likelihood that these questions will arise or matter.

If feasible, this latter choice is by far the more humane and more likely to succeed. It is accomplished by changing programming languages to carry *BY DEFAULT* (except where the program demands otherwise explicitly) extravagantly more Floating-Point precision than anyone is likely to think necessary. IEEE 754 (2008) *Quadruple* almost always suffices, as does COBOL’s *Comp* format, both with at least 33 sig.dec. of precision.

Higher precision \Rightarrow Smaller roundoff $\rho \Rightarrow$ smaller **volume** around any Extra Leaf, if there is one.



Higher Precision \Rightarrow Smaller $\rho \Rightarrow$ smaller **volume** around the Extra Leaf, if any:



Usually the hazardous **volume** around the Extra Leaf shrinks in proportion with ρ .

Why is 16-byte-wide IEEE 754 (2008) *Quadruple* most likely extravagant enough?

Although the foregoing relations among arithmetic precision (ρ), distance $\delta\pi(\mathbf{x})$ to a singularity, and consequent loss of perhaps all accuracy in $F(\mathbf{x})$ are *Typical*, the next most common relations predict a loss of at most about half the sig.dec. carried by the arithmetic no matter how near data \mathbf{x} comes to a Pejorative Surface.

Some Examples:

- Nearly redundant Least-Squares problems.
- Nearly double zeros of polynomials, like the quadratic mentioned above.
- Most locations of extrema.
- Small angles between subspaces; see my web page's .../Math128/NearstQ.pdf .
- EDSAC's arccos described above. (Its Pejorative Surface looks like coarse sandpaper.)
- The financial Future Value function $FV(n, i) := ((1 + i)^n - 1)/i$ for interest rate i as a fraction, and integer n compounding periods, but *only* if FV is computed thus:

Presubstitute n for $0/0$; $FV := ((1 + i)^n - 1)/((1 + i) - 1)$. **Preserve Parentheses!**

(Because FV is the divided difference of a polynomial, it can also be computed quickly but unobviously without a division, and without losing more than a few sig.dec.)

Ample experience (IBM mainframes, & with others' compilers) implies that arithmetic precision is usually extravagant enough if it is somewhat more than twice as wide as the data's and the desired result's. Often that shrunken hazardous **volume** contains no data.

16-byte *Quad* has 113 sig.bits; 8-byte *Double* has 53; 4-byte *Float* has 24 .

What earlier experience supports carrying somewhat more precision in the arithmetic than twice the precision carried in the data and available for the result to vastly reduce embarrassment due to roundoff-induced anomalies?

During the 1970s, the original Kernighan-Ritchie *C* language developed for the DEC PDP-11 evaluated all Floating-Point expressions in 8-byte wide *Double* (56 sig. bits) no matter whether variables were stored as *Doubles* or as 4-byte *Floats* (24 sig. bits). They did so because of peculiarities of the PDP-11 architecture. At the time, almost all data and results on “Minicomputers” like the PDP-11 were 4-byte *Floats*.

Serendipitously, all Floating-Point computations in *C* turned out much more accurate and reliable than when programmed in FORTRAN, which must round every arithmetic operation to the precision of its one or two operand(s), or the wider operand if different.

Alas, before this serendipity could be appreciated by any but a very few error-analysts, it was ended in the early 1980s by the *C*-standards committee (ANSI X3-J11) to placate vendors of CDC 7600 & Cybers, Cray X-MP/Y-MP, and CRAY I & II supercomputers. Now most *C* compilers evaluate Floating-Point FORTRANnishly and eschew *Quad*.

Experience also tells us that not everyone likes *Quad* to be the default. It can double (or worse) the computation’s cost in TIME and ENERGY.

Widespread practices resist change stubbornly. Default evaluation in *Quad*, the humane option, is unlikely to be adopted widely. In consequence, at least for the foreseeable future, the other option may be our only option:

- Enhance the likelihood of these difficult questions' resolution by supplying tools to reduce by orders of magnitude the cost in talent and time to resolve them.

What tools?

Given a program F and data \mathbf{x} at which $F(\mathbf{x})$ has aroused suspicions for some reason, we hope to find the smallest part (subprogram, block, statement, ...) of F that also arouses suspicions so that mathematical attention may be focussed upon it as a possible cause of the suspicious (mis)behavior of $F(\mathbf{x})$. Data \mathbf{x} is precious; our tools must not change data lest the change chase away the program's suspicious (mis)behavior.

Our tools would help modify program F so as to detect hypersensitivity to roundoff by rerunning $F(\mathbf{x})$ a few times with different roundings —

- different in Direction,
- different in Precision.

We hope a few reruns will expose a small part of F responsible for its misbehavior; this happens almost always. (It cannot happen in *all* cases; contrived exceptions exist.) I put such tools on my old computers; for details: <.../Boulder.pdf> & <.../Mindless.pdf>.

How Well does Recomputation with Redirected Rounding Work?

It works astonishingly well at exposing hypersensitivity to roundoff despite that no mindless tool can do so infallibly. Rerunning with Redirected Roundings works on ten examples in [<.../Mindless.pdf>](#), and on all the examples appearing in the lengthy list on p. 22 above. A typical example comes from the section titled “Difficult Eigenproblems” in [<www.eecs.berkeley.edu/~wkahan/MathH110/HilbMats.pdf>](#) .

The data consist of symmetric positive definite integer matrices A and H . Sought is a column \mathbf{v} of the eigenvalues λ that satisfy $A \cdot \mathbf{b} = \lambda \cdot H \cdot \mathbf{b}$ for some $\mathbf{b} \neq \mathbf{0}$. Three such columns get computed:

- One column $\mathbf{u} \approx \mathbf{v}$ is computed by MATLAB’s `eig(A, H)`.
- Another column $\mathbf{w} \approx \mathbf{v}$ is computed by MATLAB’s `eig(X*A*X, X*H*X)` where X is obtained from the identity matrix by reversing its rows.
- A third column \mathbf{v} is obtained from the squared singular values of a bidiagonal matrix derived unobviously from the given A and H because both are Hilbert matrices. (Rarely would a third accurate column \mathbf{v} be computable so quickly.)

In the absence of roundoff we should get $\mathbf{u} = \mathbf{v} = \mathbf{w}$, but the three computed (& sorted) columns disagree in their leading digits despite 8-byte *Double* precision arithmetic. ...

Columns \mathbf{u} , \mathbf{v} and \mathbf{w} were computed with arithmetic rounded the default way *To Nearest*. Column $\Delta\mathbf{u}_o = \mathbf{u}_o - \mathbf{u}$ shows how \mathbf{u} changed when computed with rounding redirected *Toward Zero*. Similarly $\Delta\mathbf{u}_\uparrow$ shows how rounding *Up* changed \mathbf{u} , and $\Delta\mathbf{u}_\downarrow$ is for rounding *Down*. Likewise for $\Delta\mathbf{v}_{\dots}$ and $\Delta\mathbf{w}_{\dots}$, *all computed at full speed*.

\mathbf{u}	$\Delta\mathbf{u}_o$	$\Delta\mathbf{u}_\uparrow$	$\Delta\mathbf{u}_\downarrow$	\mathbf{v}	$\Delta\mathbf{v}_o$	$\Delta\mathbf{v}_\uparrow$	$\Delta\mathbf{v}_\downarrow$	\mathbf{w}	$\Delta\mathbf{w}_o$	$\Delta\mathbf{w}_\uparrow$	$\Delta\mathbf{w}_\downarrow$
0.255	-0.007	-0.004	-0.389	0.2095058938478430	-3e-16	3e-16	-3e-16	0.247	-0.029	0.002	-0.001
0.386	-0.060	-0.006	-0.136	0.3239813175038243	-9e-16	7e-16	-9e-16	0.377	-0.101	0.001	-0.000
0.512	-0.133	-0.006	-0.133	0.4391226809250292	-12e-16	12e-16	-12e-16	0.502	-0.137	0.001	0.001
0.631	-0.126	-0.006	-0.126	0.5528261852845718	-19e-16	22e-16	-19e-16	0.622	-0.129	0.002	0.002
0.740	-0.114	-0.005	-0.115	0.6612493756197405	-22e-16	26e-16	-22e-16	0.731	-0.115	0.003	0.004
0.833	-0.098	-0.004	-0.099	0.7603044306722687	-26e-16	36e-16	-26e-16	0.825	-0.098	0.003	0.005
0.908	-0.078	-0.002	-0.079	0.8461150279850096	-33e-16	36e-16	-33e-16	0.903	-0.077	0.003	0.005
0.962	-0.056	-0.001	-0.056	0.9152685078254560	-39e-16	40e-16	-39e-16	0.959	-0.055	-0.052	0.003
0.993	-0.031	-0.000	-0.032	0.9649935940457747	-40e-16	42e-16	-40e-16	0.992	-0.032	-0.031	0.001
5.724	-4.732	-3.016	-4.732	0.9932996529571477	-41e-16	44e-16	-41e-16	1.151	-0.159	-0.159	-0.005

Rerunning each computation in three rounding modes revealed that \mathbf{v} is practically unperturbed by redirected roundoff, but it perturbs \mathbf{u} and \mathbf{w} by about as much as they differ from \mathbf{v} and each other. Afterwards an error-analysis confirmed \mathbf{v} 's accuracy and explained why \mathbf{u} and \mathbf{w} must be inaccurate. Big payoff for a small investment of time.

Redirected Rounding's Implementation Challenges

At first sight, Redirected Roundings appear to be implementable via a pre-processor that rewrites a chosen part of the text of the program being debugged and then recompiles it.

It's not always that easy.

Redirected Rounding is outlawed by JAVA and some other programming languages.

The most widespread computers redirect rounding, when they can, from a *Control Register* treated by most languages and compilers as a global variable, alas. Some other computers redirect roundings from op-code bits that must be reloaded to change. In consequence, the debugger must manage precompiled modules like DLLs appropriately.

Many optimizing compilers achieve concurrency by keeping pipelines filled; to do so they interleave instructions from otherwise disjoint blocks of source-code, and "Inline" the Math. Library's functions. Then the compiler must mark inlined operations so that the debugger can be told whether to redirect their roundings.

For more see §14 of www.eecs.berkeley.edu/~wkahan/Mindless.pdf .

Redirected Rounding's goal may be easier to reach with a different software tool:

Recomputation with Higher Precision

It doesn't have to be much higher.

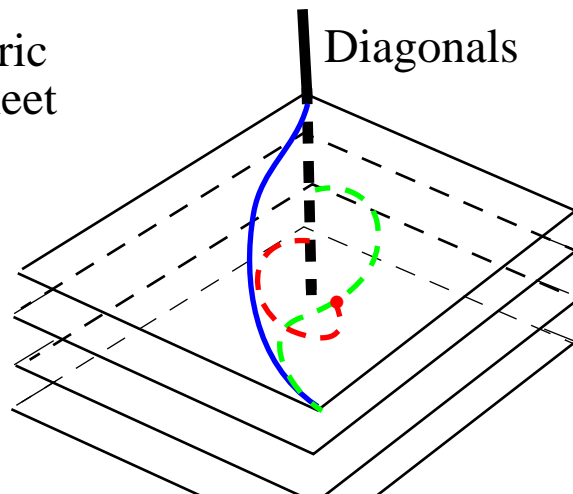
A Tool for (Slower) Recomputation with Higher Precision

This tool would ease the task of running two programs $F(\mathbf{x})$ and $\mathbb{F}(\mathbf{x})$ in lock-step. Here \mathbb{F} is derived from F by promoting all Floating-Point variables and some (probably not all) constants to a higher precision. Both programs could start with the same data \mathbf{x} .

The programs are NOT intended to be run forward in lock-step until they first diverge. That would be pointless because so many numerical processes are forward-unstable but backward-stable; this means that small perturbations like roundoff can deflect the path of a computation utterly without changing its destination significantly. For instance, the path of Gaussian Elimination with row-exchanges (“Pivoting”) can be deflected by an otherwise inconsequential rounding error if two candidates for pivots in the same column are almost equal. Deflection occurs often in eigensystem calculations; roundoff can change the order in which eigenvalues are revealed without much change to computed eigenvalues.

All the symmetric matrices in a sheet have the same eigenvalues.

Adjacent sheets differ by practically negligible roundoff.



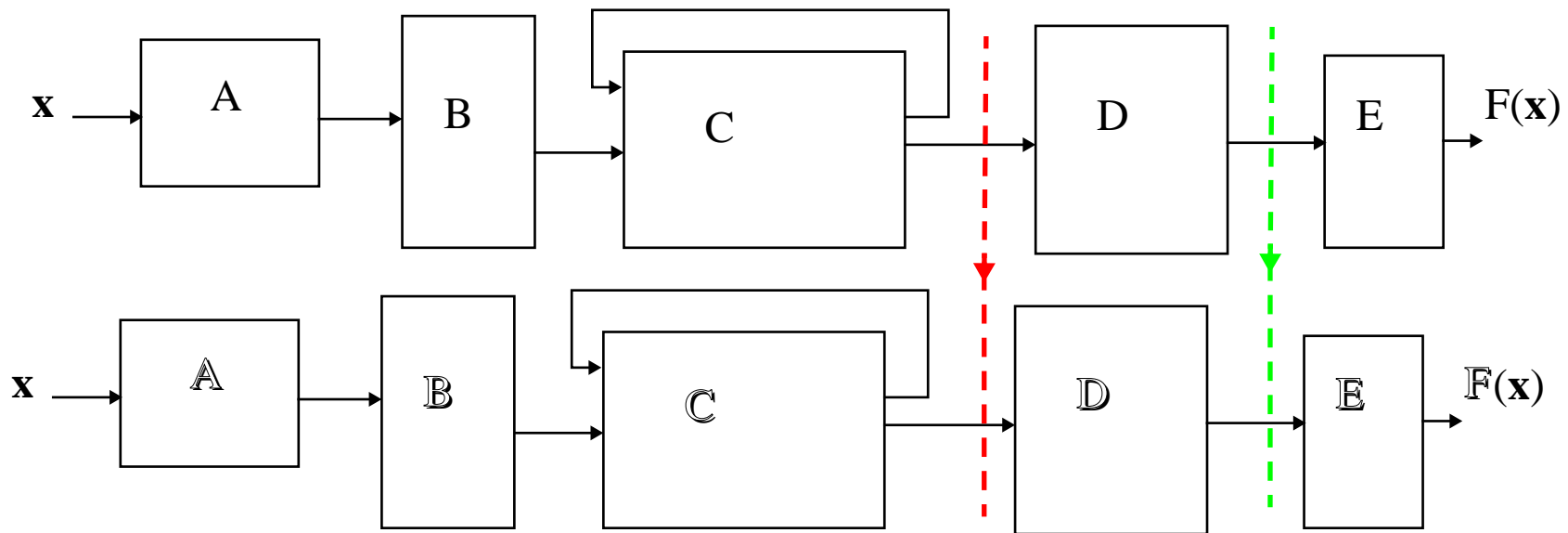
Paths followed during a program's computation of eigenvalues with ...

... no rounding errors

... the usual rounding errors

... and altered rounding errors

Instead of running F and \mathbb{F} in lock-step from their beginnings, the user of this tool will choose places in program F that I shall call “stages”. He will run $F(\mathbf{x})$ up to a chosen stage and then **copy the values of all the variables alive at that stage exactly** to their counterparts in \mathbb{F} ; then run \mathbb{F} to its end to see how much its result disagrees with $F(\mathbf{x})$. If they disagree too much, a later stage will be chosen; if they agree closely, an earlier stage will be chosen. With luck two adjacent stages will straddle a short section of F that causes $F(\mathbf{x})$ and $\mathbb{F}(\mathbf{x})$ to disagree too much. This section attracts focussed suspicion.



Keep in mind that *suspicion* is not yet *conviction*, which requires an error-analysis.

How Well does Recomputation with Higher Precision Work?

It almost always works, even if no short segment between stages of F can be blamed for a substantial disagreement between $F(x)$ and $\mathbb{F}(x)$, as is the case for Muller's Example. If all of program F has to be replaced by a better scheme, this fact is well worth knowing.

Copying to \mathbb{F} all the values of variables in F alive at a stage can be extremely tedious without help from a software tool. And help is needed to keep track of all the technical decisions that cannot be taken out of the tool-user's hands. For instance ...

- Which functions in F from its Math Library (*log*, *cos*, ...) should not be replaced in \mathbb{F} by their higher precision counterparts ?
- Which literal constants in F should not be replaced in \mathbb{F} by their higher precision counterparts ? Which tolerances for terminating iterations should be replaced?
- Which conditional branches in F should \mathbb{F} follow regardless of the condition?
- What is to be done for \mathbb{F} about software modules in F obtained from vendors pre-compiled without source-code ?

A tool to help recompute with higher precision is more interesting than first appears.

And after it works well it invites an error-analysis; learn how from N. Higham's book [2002].

What about other schemes like ...

- Interval Arithmetic
- Significance Arithmetic (used by *Mathematica* among others)
- Repeated runs with Random Rounding (cf. Vignes' *CESTAC*, *CADNA*)
- Searches for Singularities by Theorem Provers & Computerized Algebra
- ... ?

So far, all such schemes lack at least one of these three requirements ...

<1>: Almost certainly issues a warning when a computation is too inaccurate.
Otherwise the scheme is too dangerously deceptive to use routinely.

<2>: Issues undeserved warnings rarely enough to be tolerable.
Recall *The Little Boy who cried "Wolf!"* and was subsequently ignored.

<3>: Runs at most several times slower than the original program requiring diagnosis.
What runs too slowly will not get run.

Summary of the Story So Far:

I claim that scientists and engineers are almost all unaware ...

- ... of how high is the incidence of misleadingly inaccurate computed results.
- ... of how necessary is the **investigation** of every suspicious computed result as a potential harbinger of substantially worse to come.
- ... of the potential availability of software tools that would reduce those investigations' costs in expertise and time by orders of magnitude.
- ... that these tools will remain unavailable unless producers of software development systems (languages, compilers, debuggers) know these tools are in demand.
- What software tools would reduce those investigations' costs, in expertise and time, by *Orders of Magnitude* ? How do I know?
On a few ancient computers I implemented and enjoy most of the tools I describe.
- If almost nobody (but me) asks for such tools,
the demand for them will be presumed inadequate to pay for their development.

Computer scientists and programmers already have lots of other fish to fry.

USS Yorktown (CG-48) *Aegis* Guided Missile Cruiser, 1984 — 2004



And now for something entirely different ...

Floating-Point Exception-Handling

Conflicting Terminology:

Some programming languages, like *Java*, use “exception” for the policy, object or action, like a trap, that is generated by a perhaps unusual but usually anticipated event like a Time-Out, Division-by-Zero, End-of-File, or an attempt to Dereference a Null Pointer.

Here I follow IEEE 754’s slightly ambiguous use of “Floating-Point Exception” for a class of events or one of them. There are five classes:

INVALID OPERATION	like $\sqrt{-5.0}$ in a REAL arithmetic context
DIVISION-BY-ZERO	actually creation of $\pm\infty$ from finite operand(s)
OVERFLOW	an operation’s finite result is too big
UNDERFLOW	an operations nonzero result is too close to 0
INEXACT	an operation’s result has to be rounded or altered

Each exception generates, by *Default* (unless the program demands otherwise), a value *Presubstituted* for the exceptional operation’s result, continues the program’s execution and, as a side-effect, signals the event by raising a *flag* which the program can sense later, or (as happens most often) ignore.

When put forth in 1977, Presubstitution departed radically from previous practice.

Floating-Point Exceptions turn into Errors ONLY when they are Handled Badly.

Tradition has tended to conflate “Exception” with “Error” and handle both via disruptions of control, either aborting execution or jumping/trapping to a prescribed handler. ...

- FORTRAN:** Abort, showing an Error-Number and, perhaps, a traceback.
Since 1990, FORTRAN has offered a little support for IEEE 754’s defaults and flags.
- BASIC:** ON ERROR GOTO ... ; ON ERROR GOSUB to a handler.
- C :** setjmp/longjmp ... to a handler; ERRNO; abort.
C99 has let compiler writers choose whether to support IEEE 754’s defaults and flags.
- ADA:** *Arithmetic Error* Falls Through to a handler or the caller, or aborts.
- JAVA:** try/throw/catch/finally; abort showing error-message and traceback.
JAVA has incorporated IEEE 754’s defaults but outlawed its flags; this is *dangerous* !

These disruptions of control are appropriate when a programmer is debugging his own code into which no other provision to handle the exception has been introduced yet. Then the occurrence of the exception may well be an error; an eventuality may have been overlooked.

Otherwise IEEE Standard 754 disallows these disruptions unless a program(mer) asks for one explicitly. They must *not be the default* for any Floating-Point Exception-class.

Why *not* ?

Why must a Floating-Point Exception's default not disrupt control?

As we shall see, ...

- Disruptions of control are *Error-Prone* when they may have more than one cause.
- Disruptions of control hinder techniques for formal validations of programs.
- IEEE 754's presubstitutions and flags seem easier (although not easy) ways to cope with Floating-point Exceptions, especially by programmers who incorporate other programmers' subprograms into their own programs.
- Disruptions of control can be perilous; but so can continued execution after some exceptions. The mitigation of this dilemma requires *Retrospective Diagnostics*.

Error-Prone?

Prof. Westley Weimer's PhD. thesis, composed at U.C. Berkeley, exposed hundreds of erroneous uses of try/throw/catch/finally in a few million lines of non-numerical code. Mistakes were likeliest in scopes where two or more kinds of exceptions may be thrown.

See <www.cs.virginia.edu/~weimer>.

Floating-Point is probably more prone to error because every operation is susceptible, unless proved otherwise, to more than one kind of Exception.

Every Floating-Point operation is susceptible, unless proved otherwise, to more than one kind of exception. A program with many operations could enter a handler from any one of them, and for any of a few kinds of exception, and quite possibly unanticipatedly.

A program that handles Floating-point Exceptions by disruptions of control resembles a game ...

Snakes-and-Ladders

End	98	97	96	95	94	93	92	91	90
80	81	82	83	84	85	86	87	88	89
79	78	77	76	75	74	73	72	71	70
60	61	62	63	64	65	66	67	68	69
59	58	57	56	55	54	53	52	51	50
40	41	42	43	44	45	46	47	48	49
39	38	37	36	35	34	33	32	31	30
20	21	22	23	24	25	26	27	28	29
19	18	17	16	15	14	13	12	11	10
Start	1	2	3	4	5	6	7	8	9

... with an important difference ...

... with an important difference, for Floating-point Exceptions, ...

Invisible Snakes-and-Ladders

End	98	97	96	95	94	93	92	91	90
80	81	82	83	84	85	86	87	88	89
79	78	77	76	75	74	73	72	71	70
60	61	62	63	64	65	66	67	68	69
59	58	57	56	55	54	53	52	51	50
40	41	42	43	44	45	46	47	48	49
39	38	37	36	35	34	33	32	31	30
20	21	22	23	24	25	26	27	28	29
19	18	17	16	15	14	13	12	11	10
Start	1	2	3	4	5	6	7	8	9

None or else too many of the origins of jumps into an Exception handler are visible in the program's source-text. This hinders its formal validation.

Among programming languages, the predominant policy for handling exceptions, including Floating-Point exceptions, either disrupts control or else ignores them.

UNDERFLOW, INEXACT are almost always ignored.

INVALID OPERATION, DIVIDE-BY-ZERO, OVERFLOW would usually disrupt control.

A policy that predisposes every unanticipated Exception to disrupt control can have very bad consequences. e.g. ...

- Numerical searches for roots or extrema **abandoned** prematurely
- The missile-cruiser USS Yorktown paralyzed for $2\frac{3}{4}$ hrs. in 1997
- The Ariane 5 rocket blown up in 1996
- Air France #447 **crashed** in 2009

Let's look into two examples ...

The others are discussed in <.../Boulder.pdf>.

A policy that aborts execution as soon as a severe Exception occurs can also

Prematurely Abort a Search :

Suppose a program seaches for an object Z that satisfies some condition upon $f(Z)$.

e.g.,

- Locate a Zero Z of $f(x)$, where $f(Z) = 0$, or
- Locate a Maximum Z of $f(x)$, where $f(Z) = \max_x f(x)$.

How can the search's trial-arguments x be restricted to the domain of f if its boundary is unknown? Is this boundary easier to find than whatever Z about f is to be sought?

Example:

$$\text{shoe}(x) := (\tan(x) - \arcsin(x)) / (x \cdot |x|^3) \quad \text{except} \quad \text{shoe}(0) := +\infty .$$

We seek a root $Z > 0$ of the equation $\text{shoe}(Z) = 0$ if such a root exists. (We don't know.) We know $x = 0.5$ lies in shoe's domain, but (pretend) we don't know its boundary.

Does your rootfinder find Z ? Or does it persuade you that Z probably does not exist ?

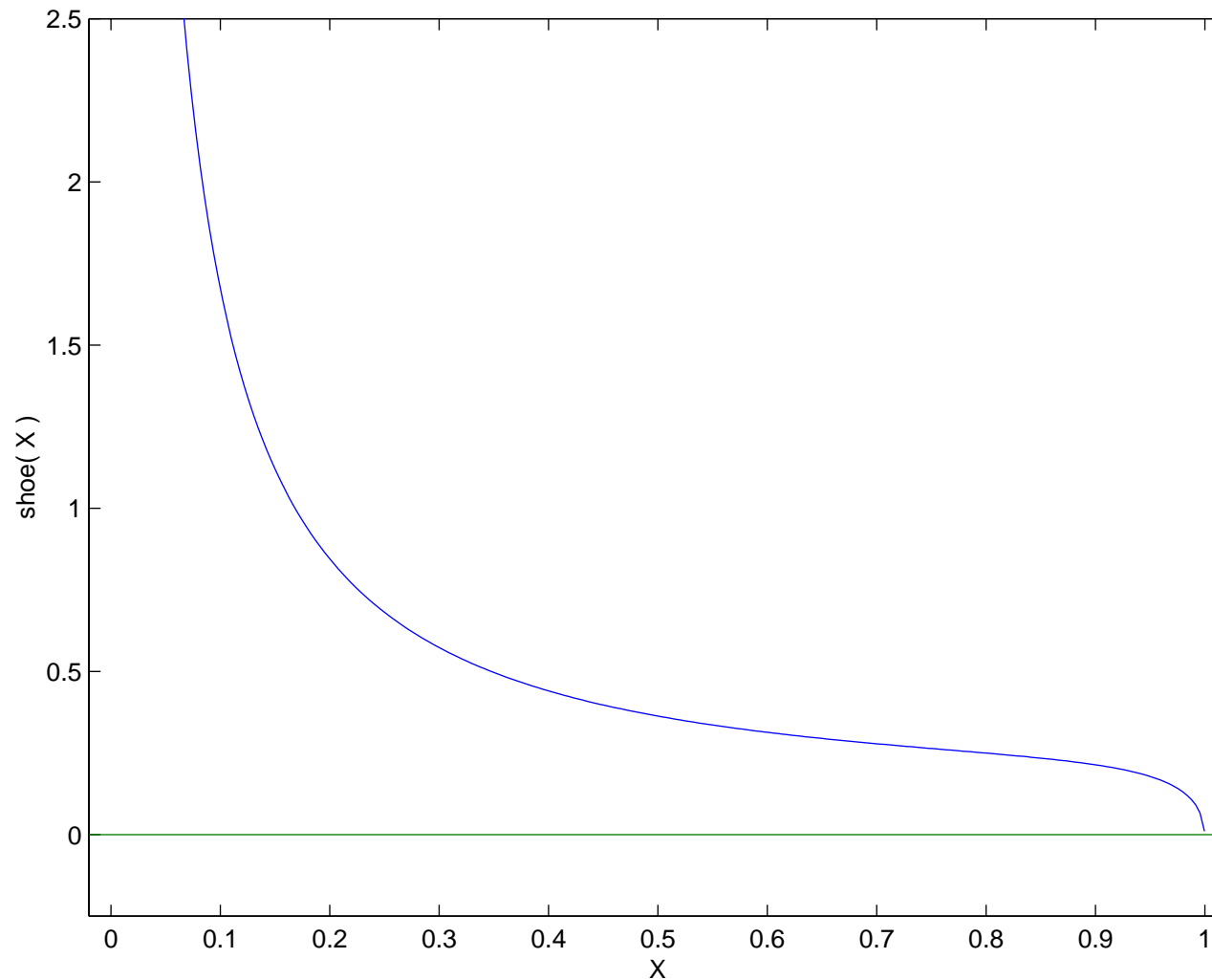
Try, say, each of 19 initial guesses $x = 0.05, 0.1, 0.15, 0.2, \dots, 0.5, \dots, 0.9, 0.95$.

`fzero` in MATLAB 6.5 on a PC said it cannot find a root near any one of them.

`root` in MathCAD 3.11 on an old Mac diverged, or converged to a huge *complex* no.

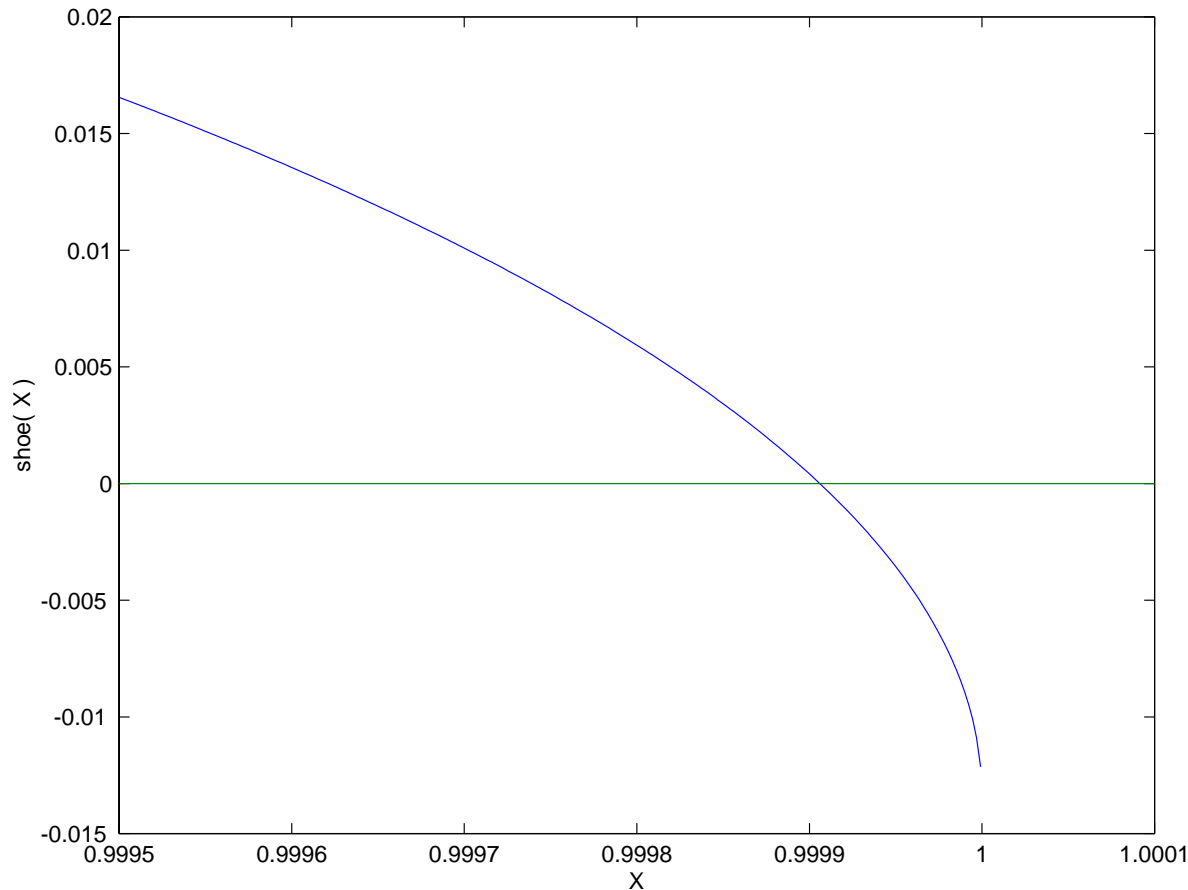
Why did [SOLV] on HP-18C, 19C and 28C handheld calculators find what they didn't ?

$$\text{shoe}(x) := (\tan(x) - \arcsin(x)) / (x \cdot |x|^3)$$



If no positive Z in $\text{shoe}(x)$'s domain satisfied $\text{shoe}(Z) = 0$,
then the SHOE would leak at its toe.

$$\text{shoe}(x) := (\tan(x) - \arcsin(x)) / (x \cdot |x|^3)$$



Notice the 1000-fold change in the scale of the x - axis.

The HP-28C found the root $Z = 0.999906012413$ from each of those 19 first guesses. What did the calculator know/do that the computers didn't? ... **Defer Judgment** .

See P.J. McClellan [1987] I think some Casio calculators too may know how to do it.

Air France #447 (Airbus 330) lost 1 June 2009

Modern commercial and military jet aircraft achieve their efficiencies only because they fly under control of computers that manage control surfaces (ailerons, elevators, rudder) and throttle. Only auto-pilot computers have the stamina to stay “on the razor’s edge” of optimal altitude, speed, and an angle of attack barely short of an *Abrupt Stall*. cf. p.14



35000 ft. over the Atlantic about 1000 mi. NE of Rio de Janeiro, AF#447 flew through a mild thunderstorm into one so violent that its super-cooled moisture condensed on and blocked all three *Pitot Tubes*. They could no longer sense airspeed. Bereft of consistent airspeed data, the computers relinquished command of throttles and control surfaces to the pilots with a notice that *did not explain why*. The three pilots struggled for perhaps ten seconds too long to understand why the computers had disengaged, so the aircraft stalled at too steep an angle of attack before they could institute the standard recovery procedure. Three minutes later, AF#447 pancaked into the ocean killing all 228 aboard. The computers had abandoned AF#447 too soon.

See <www.bea.aero/fr/enquetes/vol.a.point.enquete.af447.27mai2011.en.pdf>, NOVA6207 from PBS, and <www.aviationweek.com/aw/jsp_includes/articlePrint.jsp?headLine=High-Altitude%20Upset%20Recovery&storyID=news/bca0711p2.xml>

A Board of Inquiry has blamed the crash posthumously upon the younger co-pilot.
The contribution of the autopilot's software to the crash has been overlooked.

When the auto-pilot disengaged, its error-message to the co-pilots said "Invalid Data". It should have said "Airspeed Inconsistent with Maintenance of Altitude", but didn't. With this crucial information, the co-pilots would have deduced what to do immediately. Instead, they didn't know which instruments to (dis)trust. Flying in pitch-black rough air, they could see no external references, could not feel whether the aircraft was falling. They could not know whether to trust repeated loud **STALL!** warnings. Unable to trust the altimeters, the younger co-pilot thought trying to climb was better than allowing descent.

He was mistaken. Raising the aircraft's nose caused the stall.

After about a minute, as AF #447 fell through 20000 ft., the ice melted and the pitot tubes delivered correct airspeeds. But the disengaged autopilot's software was no longer monitoring the diverse sensors of airspeed, altitude, attitude, *etc.*, so the co-pilots were not notified that the "Invalid Data" condition had lapsed. Had they been so notified, they would have regained trust in their instruments, heeded the **STALL!** warning, and saved the aircraft. Instead, just as they were emerging from the thrall of confusion, they crashed.

Can you deduce what conventions programming languages should impose to reduce the risk of calamities like AF #447's crash?

<.../Bou1der.pdf> offers some suggestions.

Naval embarrassment (Yorktown).
Half a billion dollars lost (Ariane V).
228 lives lost (AF #447).

What more will it take to persuade the computing industry
and particularly the arbiters of taste and fashion in programming languages
to reconsider whether an abortion policy inherited from the 1960s
era of Batch Computing should be the only default response to
unanticipated exceptions ?

Though a policy of continued execution after them may well pose
a difficult question for the programmer,
especially where *Embedded Systems* are concerned,
who else is better equipped to incur the obligation to answer it?

No program should be declared complete until it specifies what it will [return to its caller](#) by
default if an unanticipated event deemed an ERROR causes the program's termination.

Damned if you do and damned if you don't Defer Judgment

Choosing a *default* policy for handling an Exception-class runs into a ...

Dangerous Dilemma:

- Disrupting the path of a program's control can be dangerous.
- Continuing execution to a perhaps misleading result can be dangerous.

Computer systems need 3 things to *mitigate* the dilemma :

- 1• An *Algebraically Completed* number system for *Default Presubstitutions*.
- 2• Sticky flags to Memorialize *Leading Exceptions* in each Exception-class.
- 3• *Retrospective Diagnostics* to help the program's **User** debug it.
The program's **User** may be another program composed by maybe a different programmer.

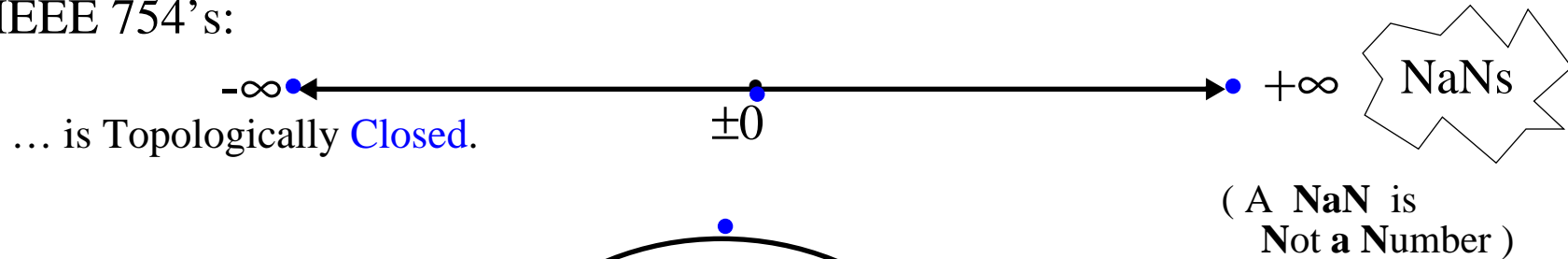
These, explained in <.../Bou1der.pdf>, are intended for Floating-Point computations.

How well they suit other kinds of computations too is for someone else to decide.

Mathematicians do not need these 3 things for their symbolic and algebraic manipulations on paper.

Three Proper Algebraic Completions of the Real Numbers

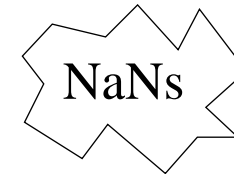
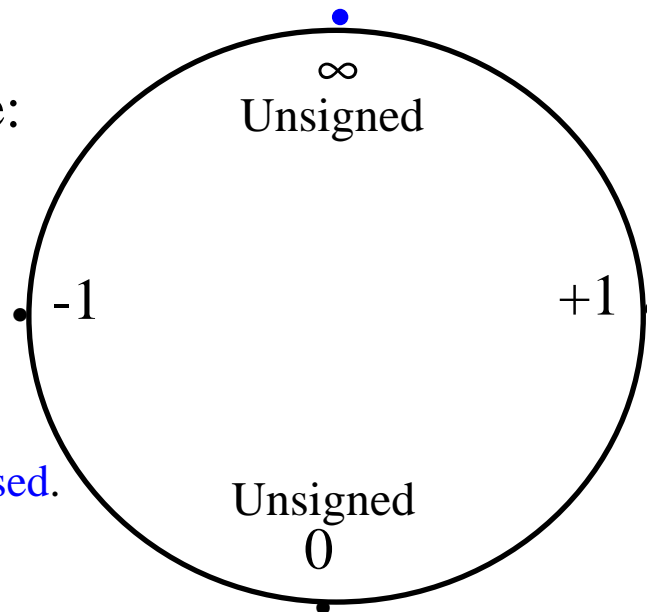
IEEE 754's:



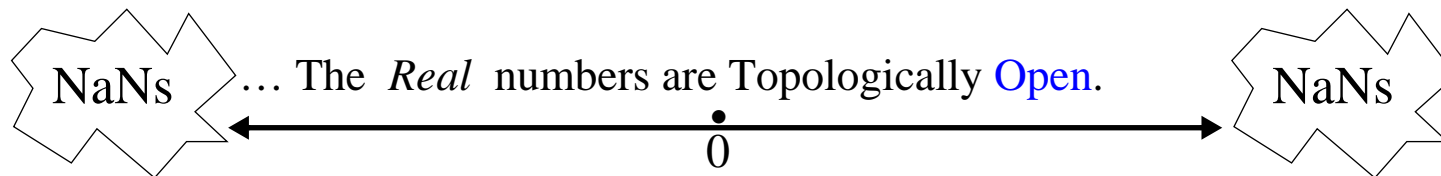
Projective Closure:

(Stereographic Projection, like the Riemann Sphere of the Complex Plane)

... is Topologically **Closed**.



For more about NaNs see p. 56 of <.../NeedDebug>



Proper *Algebraic Completion* maintains *Algebraic Integrity* while providing a result for *every* operation.

Algebraic Integrity: *Non-Exceptional* evaluations of algebraically equivalent expressions over the Real Numbers produce the same values.

To conserve Algebraic Integrity as much as possible, every Proper Algebraic Completion must ensure that, if Exceptions cause evaluations of algebraically equivalent expressions over the Algebraically Completed Real Numbers to produce more than one value, they can produce at most two, and if these are not $+\infty$ and $-\infty$ then at least one is NaN.

Among a few others, the Completion chosen by IEEE Standard 754 does this.

Other Completions, like APL's $0/0 := 1$ and MathCAD's $0/0 := 0$, destroy Algebraic Integrity.

For example, compare evaluations of three algebraically equivalent expressions:

x	$2/(1 + 1/x)$	$2 \cdot x/(1 + x)$	$2 + (2/x)/(-1 - 1/x)$
-1	$+\infty$!	$-\infty$!	$-\infty$!
0	0 !	0	NaN !
$\pm\infty$	2	NaN !	2

Unlike Real, Floating-Point evaluations usually conserve Algebraic Integrity at best approximately after the occurrence of roundoff and over/underflow, so some algebraically equivalent expressions evaluate more accurately than others.

For more about Algebraic Completion and Algebraic Integrity see pp. 51 - 53 of <.../NeedDebug> .

1• Presubstitution ...

... provides, within its scope, each Exception-class with a short process that supplies a value for any Floating-Point Exception that occurs, instead of aborting execution.

IEEE Standard 754 provides five presubstitutions by default for ...

INVALID OPERATION	defaults to NaN	Not-a-Number
OVERFLOW	defaults to $\pm\infty$	
DIVIDE-BY-ZERO (∞ from finite operands)	defaults to $\pm\infty$	
INEXACT RESULT	defaults to a rounded value	
UNDERFLOW is GRADUAL	and ultimately glides down to zero by default.	

These presubstitutions descend partly from the chosen Algebraic Completion of the Reals, partly from greater risks other presubstitutions may pose if their Exceptions are ignored.

Untrapped Exceptions are too likely to be overlooked and/or ignored.

- From past experience, INEXACT RESULT and UNDERFLOW are almost always ignored regardless of their presubstitutions if these are at all plausible. Ignored underflow is deemed least risky if GRADUAL.
- DIVIDE-BY-ZERO might as well be ignored because ∞ either goes away quietly ($\text{finite}/\infty = 0$) or else almost always turns into NaN during an INVALID OPERATION, which raises *its* flag.
- INVALID OPERATION should not but will be ignored inadvertently. Its NaN is harder to ignore.

Consequently, each default presubstitution has a side-effect;– it raises a *flag*. (See later.)

Ideally, a program should be allowed to choose different presubstitutions of its own.

Ideally, (on some computers today this ideal may be beyond reach)
 a program should be allowed to choose different presubstitutions of its own.

INEXACT RESULT's default presubstitution is *Round-to-Nearest* .

- IEEE 754 offers three non-default *Directed Roundings* (Up, Down, to Zero) that a program can invoke to replace or over-ride (only) the default rounding.
 ... useful for debugging as discussed previously, and for *Interval Arithmetic*.

UNDERFLOW's default presubstitution is *Gradual Underflow*, deemed most likely ignorable.

- IEEE 754 (2008) allows a kind of *Flush-to Zero* (almost), but not as the default.
 ... useful for some few iterative schemes that converge to zero very quickly, and on some hardware whose builders did not know how to make Gradual Underflow go fast.
 See www.cs.berkeley.edu/~wkahan/ARITH_17U.pdf for details.

OVERFLOW's and DIVIDE-BY-ZERO's default presubstitution is $\pm\infty$.

- Sometimes *Saturation* to \pm (Biggest finite Floating-point number) works better.

INVALID OPERATIONS' default presubstitutions are all NaN .

- Better presubstitutions must distinguish among $0/0$, ∞/∞ , $0\cdot\infty$, $\infty-\infty$, ...
- The scope of a presubstitution, like that of any variable, respects block structure.
- Hardware implementation is easiest with *Lightweight Traps*, each at a cost very like the cost of a rare conditional invocation of a function from the Math. library.

For examples of non-default presubstitutions see www.cs.berkeley.edu/~wkahan/Grail.pdf ,
 its pp. 1-8 explain the urgent need to implement them, and how to do it in pp. 8-10.

2• Flags

IEEE Standard 754 mandates a *Sticky flag* for each Exception-class to memorialize its every Exception that has occurred since *its flag* was last clear. Programs may raise, clear, sense, save and restore each *flag*, but not too often lest the program be slowed.

The flag of an Exception-class may be raised as a by-product of arithmetic.

The flag is a *function*, *a flag* a *variable* of data-type FLAG in memory like other variables.

The flag is not a bit in hardware's *Status Register*. Such a bit serves to update *its flag* when the program senses or saves it, perhaps after waiting for the bit to stabilize.

Any flag's data-type gets coerced to LOGICAL in conditional and LOGICAL expressions.

Any flag may also serve *Retrospective Diagnostics* by pointing to where it was raised.

An Exception that raises *its flag* need not overwrite it if it's already raised; ... faster !

Three frequent operations upon flags are ...

- Swap a saved flag with *the* current one to restore the old and sense the new.
- Merge a saved flag into *the* current *flag* (like a logical OR) to propagate one.
- Save, clear and restore all (IEEE 754's five) *flags* at once.

Reference to *the flag* is a Floating-Point operation the optimizing compiler must not swap with a prior or subsequent Floating-Point operation lest *the flag* be corrupted. This constraint upon code movement is another reason to reference *flags* sparingly.

Flags' Scopes

Variables of data-type FLAG are scoped like other variables, in so far as they respect block structure, except for *the* five Exception-classes' five *flags* which, if supported at all, have usually been treated as GLOBAL variables. Why?

By mistake; they have been conflated with bits in a status register.

The Exception-classes' five *flags* can implicitly be inherited and exported by every Floating-point operation or subprogram (or *Java* "method") unless it can specify otherwise in a language-supplied initial *Signature*.

The least annoying scheme I know for managing *flags*' inheritance and export is *APL*'s for *System Variables* []*CT* (Comparison tolerance) and []*IO* (Index Origin):

An *APL* function always inherits system variables and, if it changes one, exports the change unless this variable has been *Localized* by redeclaration at the function's start. If augmented by a command to merge a changed flag with *the flag*, this scheme works well.

Still, because they are side-effects, ...

flags are Nuisances !

flags are Nuisances. Why bother with them?

Because every known alternative can be worse :

Execution continued oblivious to Exceptions can be dangerous,
and is reckless.

Java forbids *flags*, forcing a conscientious programmer to test for an Exceptional result after every liable operation.

So many tests-and-branches are tedious and error-prone.

Recall pp. 23-4 of www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf . Similarly for ...

C's single flag `ERRNO` must be sensed immediately lest another Exception overwrite it.

What can *flags* do that `try/throw/catch/finally` cannot ?

If a `throw` is hidden in a subprogram invoked more than once in the `try` clause, the `catch` clause can't know the state of variables perhaps altered between those invocations.

Recall W. Weimer's discovery that `try/throw/catch/finally` is **error-prone**.

A Floating-Point Exception *flag* costs relatively little unless the program references it.

- Apt Presubstitutions render most (not all) Exceptions and their *flags* ignorable.
- Apt non-default presubstitutions render more Exceptions and *flags* ignorable.

We should try not to burn out conscientious programmers prematurely.

Their task is difficult enough with presubstitutions and *flags*; too difficult without.

And *flags* let overlooked Exceptions be caught by *Retrospective Diagnostics*

3• Retrospective Diagnostics

We are not gods.

Sometimes some of us overlook something.

At any point in a program's execution, usually when it ends, its *Unrequited Exceptions* are those overlooked or ignored so far.

Evidence of one's existence is *its flag* still standing raised.

Retrospective Diagnostics help a program's user debug Unrequited Exceptions by facilitating interrogation of NaNs and raised *flags* now interpreted as pointers (indirectly, and perhaps only approximately) to relevant sites in the program.

Earliest Retrospective Diagnostics

See my web page's .../7094II.pdf

In the early 1960s, programs on the IBM 7090/7094 were run in batches. Each program was swept from the computer either after delivering its output, be it lines of print or card images or compile-time error-messages, or upon using up its allotment of computer time.

Often the only output was a cryptic run-time error-message and a 5-digit octal address.

I put a LOGICAL FUNCTION KICKED(...) into FORTRAN's Math. library, and altered the accounting system's summary of time used *etc.* appended to each job's output. Then ...

```
IF (KICKED(OFF)) ... executable statement ...
```

in a FORTRAN program would do nothing but record its location when executed. If later the program's execution was aborted, a few extra seconds were allotted to execute the executable statement (GO TO ..., PRINT ..., CALL ..., or REWIND ...) after the last executed invocation of KICKED . Any subsequent abortion was final.

.....

IBM's presubstitution for UNDERFLOW was 0.0 , and its other presubstitutions for ...

- DIVISION-BY-ZERO a quotient of 0.0 , or 0 for integers,
- OVERFLOW ±(biggest floating-point number),

... were defaults a programmer could override only by a demand for abortion instead.

I added options for Gradual Underflow, and for Division-by-Zero to produce a hugest number, and for an extended exponent upon Over/Underflow. I added sticky *flags* for a program to test *etc.* any time after the Exceptions, and added Retrospective Diagnostics.

Earliest Retrospective Diagnostics continued

Each raised *flag* held the nonzero 5-digit octal address of the 7090/7094 program's site that first raised the *flag* after it had last been clear. I added tests for raised *flag* to the accounting system's summary of time used *etc.* appended to each job's output; and for each *flag* still raised at the job's end I appended a message to the job's output saying ...

“You have an unrequited ... name of Exception ... at ... octal address ... ”

This is the only change to IBM's system on the 7094 for which I was ever thanked.

... by a mathematician whose results invalidated by a DIVIDE-BY-ZERO would have embarrassed him had he announced them to the world.

My other alterations to IBM's system were taken for granted as if IBM had granted them.

Attempts over the period 1964-7 to insinuate similar facilities, all endorsed by a SHARE committee, into IBM's subsequent systems were thwarted by ...

... that's a long story for another occasion.

END OF REMINISCENCES.



Note how NaNs, *flags* and Retrospective Diagnostics differ from a system's event-log:

- The system's event-log records events *chronologically*, by time of occurrence.
- NaNs and *flags* point (indirectly) to (earliest) sites (hashed) in the program.

If Exceptions were logged chronologically, they could slow the program badly, overflow the disk, and exhaust our patience even if we attempt data-mining.

Retrospective Diagnostics' Annunciator and Interrogator

How shall a program's Unrequited Exceptions be brought to the attention of its user?

- If the program's user is another program denied access to the former's *flags* by the operating system, retrospective diagnostics are thwarted.
- If the program's user is another program with access to the former's *flags*, the latter program determines their use or may pass them through to the next user.
- If the program's user is human, the program can annotate its output in a way that makes the user ...
 - *Aware* that Unrequited Exceptions exist, and then
 - *Able* to investigate them if so inclined.

“Aware” :

- Don't do it this way:

On my MS-Windows machines, some error-messages display for fractions of a second.

- Do do it this way:

On my Macs, an icon can blink or jiggle to attract my attention until I click on it.

The Math. library needs a subprogram that creates an *Annunciator*, an icon that attracts a user's attention by blinks or jiggles, which a program can invoke to annotate its output.

Clicking on an *Annunciator* should open an *Interrogator*, dropping a menu that lists unrequited Exceptions and allows displayed NaNs to be clicked-and-dragged into the list. Clicking on an item in the list should reveal (roughly) whence in the program it came.

Retrospective Diagnostics can Annoy ...

They can annoy the programmer with an implicit obligation to annotate output upon whose validity doubt may be cast deservedly by Unrequited Exceptions. This obligation is one of **Due Diligence**.

Is programming a *Profession* ? If so, one of its obligations is *Due Diligence* .

Retrospective Diagnostics can annoy a program’s user if the Annunciator resembles
The little boy who cried “Wolf !”

by calling the user’s attention to Unrequited Exceptions that seem never to matter. This may happen because the programmer decided to “Play it Safe”, actually too safe.

My IBM 7094’s retrospective diagnostics were usually torn off the end of a program’s output and discarded.

To warn or not to warn. The dilemma is intrinsic in approximate computation by one person to serve an unknown other. They share the risk. And the ***Law of Torts*** assigns to each a share of blame in proportion to his expertise, should occasion for blame arise.

.....

Retrospective Diagnostics may function better on some platforms than on others, and not at all on yet others. Debugging may be easier on some platforms than on others. Numerical software may be developed and/or run more reliably on some platforms than on others.

What Constellation of Competencies must be Collected to develop the Diagnostic Tools described herein?

Languages must be altered to support Quad by Default unless a program refuses it, and to enforce ERROR-exit to the caller unless a program specifies a different destination.

Languages must be altered to support ...

- Scopes for (re)directed roundings, and
- Scopes for non-default Presubstitutions, and for *flags*.

Compilers must be altered to augment Symbol Tables and other information attached to object modules to help debuggers (and the loaders on some architectures) implement rerunning with redirected roundings or with higher precision.

Operating Systems must be altered to support Lightweight Traps for handling non-default Presubstitutions, and *flags*' and NaNs' Retrospective Diagnostics.

Debuggers must be augmented to support users of the foregoing capabilities.

Retrospective Diagnostics may function better on some platforms than on others, and not at all on yet others. Debugging may be easier on some platforms than on others. Numerical software may be developed and/or run more reliably on some platforms than on others.

“This ... paper, by its very length, defends itself against the risk of being read.”
... attributed to Winston S. Churchill

If there be better ideas about it,
and if the reader is kind enough to pass some on to me,
this is not the subject's
Last Word.