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

for Computer Science Dept. Colloquia at …
Univ. of Calif. @ Berkeley, 7 Oct 2009
Univ. of Calif. @ Davis, 17 Nov. 2009
Southern Methodist University, Dallas, 18 Oct. 2010
Lawrence Berkeley National Laboratory, 15 Feb. 2011

by
W. Kahan, Prof. Emeritus,
Math. Dept., and E.E. & C. S. Dept., Univ. of Calif. @ Berkeley

to be posted at `<www.cs.berkeley.edu/~wkahan/NeeDebug.pdf>`

Collateral Reading:

`<www.cs.berkeley.edu/~wkahan/Mindless.pdf>`, especially section 14
`<www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf>`, only a little outdated

Contents

Page to be supplied later

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

Abstract:

Despite almost universal conformity to IEEE Standard 754, Floating-Point Arithmetic still teems with mysteries and misconceptions, some still enshrined in programming languages. Roundoff, invisible in programs’ texts, occasionally causes the worst anomalies:—

Results unobviously wrong enough to mislead and almost always misdiagnosed.

Also misdiagnosed more often than not is misbehavior precipitated by arithmetic Exceptions, like over/underflow and division-by-zero, treated as programmers’ errors deserving disruption of the program's intended path of control. Ample instances of misdiagnoses will be presented.

Must computing professionals acquiesce to a resurgent superstition that numerical software is inevitably buggy, like Microsoft’s *Windows* ?

Developers and users of numerical software must demand but cannot by themselves produce the peculiar tools needed to debug floating-point anomalies whenever these are suspected. The tools must come from Computer Science departments; nobody else in industry and academia has motive and opportunity. Help is needed from designers and implementers of hardware, of programming languages, and of the debuggers in software development environments to collaborate on features that will help to localize an anomaly’s cause to a comparatively short segment of code, when possible. These features will be explained. Some have existed in hardware for decades but are atrophying for lack of employment.

Collateral reading: `<www.cs.berkeley.edu/~wkahan/Mindless.pdf>` and `<…/JAVAhurt.pdf>`

Most Software is already Hard Enough to Debug.
How does Floating-Point arithmetic make it Harder?

- Roundoff: …absent from strictly integer and character computation

  What you see is Not Entirely what you get.
  What you get is Not Exactly what you wanted or expected.
  “Not Exactly” can be “Far From” and yet Not Wrong**!**
  Single-Stepping through or past Gigaflops is usually Futile.

- Floating-Point Exceptions: …unlike other exceptions

  Invalid Op’n, “Div’n-by-Zero”, Overflow, Underflow, *Inexact Op’n*
  They are not Errors unless they are handled badly.
  To handle them well requires the Option to Defer Judgement.
  Deferred Judgement is Too Dangerous without well-supported
  Presubstitutions, Flags, NaNs, & Retrospective Diagnostics.

Has Floating-Point become Too Hard to Debug?

Questions to be addressed in what follows:

Nobody is keeping score of incorrect numerical results, so …

- What evidence exposes extraordinarily inadequate debugging of software engaged in floating-point?

- Who should care enough about it to do something?

- Who can do something helpful about it?

- What can be done about it that will adequately repay the substantial efforts expended?

. . . .

You deserve to be warned of certain dangers …

DANGER !

> “ A fanatic is one who can’t change his mind
> and won’t change the subject. ”
>
> … Winston S. Churchill

Am I a fanatic?
Can you trust what I’ll say? … the Whole Truth and Nothing But?
How much of my concerns should you share?

Perhaps some history will help to stimulate a timely judgement …

Some History:

I started programming electronic computers in 1953.
Then their *raison d’être* was still Scientific and Engineering computation.

Despite disparagement by John von Neumann, floating-point was used widely; but it was regarded as refractory to error-analysis and intrinsically “fuzzy”.

Why disparagement? See pp. 3 - 4 of <www.eecs.berkeley.edu/~wkahan/SIAMjvnl.pdf>

Computed results were prudently distrusted unless corroborated by experiment, by independent recomputation, or by extra computation of tests or residuals.

For every numerical task, several methods were proffered, each named after someone hoping for immortality: Jordan, Milne, Graeffe, Danilewski, … .

As an old Alka-Seltzer ad on TV said, “Try it**!** You’ll like it.” And if not, you could try another method. Picked at random, it was as likely as not to fail.

A *Polyalgorithm* assembled several algorithms, all for the same numerical task, intended to be tried one after the other until enough of them agreed closely enough with what was presumed to be nearly enough the desired result.

Anomalies suspected due to roundoff were rarely diagnosed correctly. Instead, a desperate or haphazard change to the program moved them away sometimes.

Some History, continued …

By 1957 a few of us — W. Givens, J.H.Wilkinson, F.L.Bauer, myself — had tumbled to a hint dropped by A.M. Turing in 1949 that spawned what has been mislabelled “Backward Error-Analysis”, which works for many (not all) floating-point computations, especially with matrices.

In the 1960s and 1970s, successful error-analyses spurred a proliferation of numerical algorithms that worked at least almost always on at least almost all commercially significant computers. Maybe not Seymour Cray’s.

J. Sethian’s example of an expression that would malfunction only on CRAYs in the 1980s is on p. 30 of `<www.eecs.berkeley.edu/~wkahan/CS279/CrayUG.pdf>` . Its validity on every other computer required a different proof for each.

Hardware’s near ubiquitous adoption of IEEE Standard 754 (1985) for Binary Floating-Point greatly enhanced the reliability and portability of numerical software even though programming languages have been disinclined to support also the diagnostic capabilities mandated by the standard.

Blame Microsoft’s Bill Gates (1982), Apple’s John Scully (1994), Sun’s Bill Joy (1997)

Those unsupported capabilities of IEEE 754 have been atrophying in hardware.

Some History, brought up to date:

Over the span of my computing career the population of computers, their speeds and their storage capacities at every level have grown by several orders of magnitude. Also grown by orders of magnitude are the speeds of some numerical methods, the productivity of programming environments, and the ratio **(**Random memory-access time**)/(**Fltg. Pt. Arith. time**)** .

“*Quantity* has a *Quality* all of its own.”

Attributed to V.I. Lenin, later to J. Stalin.

The foregoing massive increases should have been enough to instigate profound reassessments and reforms in the way the computing “profession” conducts its practice. But human habits change far slower than our technology; the computing industry cites *Compatibility* to justify its way of

“… visiting the iniquity of the fathers upon the children unto the third and fourth generation …”

Exodus 20:5

Two further massive changes have immediate and overwhelming consequences:

- Computers have become several orders of magnitude cheaper than they were.
- Massive Parallelism is becoming ubiquitous in computer hardware. …

Massive Parallelism is becoming ubiquitous in computer hardware.

No other way is available to speed up computation a lot.

My colleagues are fully engaged in attempts to redevelop numerical algorithms that will exploit parallelism as fully as possible. Like the Red Queen, we have to run as fast as we can to keep up. In consequence …

- Error-analysis has not yet caught up with these algorithms. They may have yet unrevealed numerical instabilities and failure modes.

  For example, B.N. Parlett’s “Holy Grail” scheme to compute orthogonal eigenvectors independently.

- Scant resources are left for my colleagues to devise debugging techniques enhanced enough to meet the challenges of massively parallel methods.

Computers have become amazingly cheap.

They are so cheap, their most ubiquitous and remunerative uses are for Entertainment, Companionship, and Embedded Controllers.

Computers are so cheap, they can be and are used to compute results of which no one is worth very much.

Computers are now so cheap that most computers' floating-point results are worth less than today's probable cost of attempting and perhaps failing to assess each suspicious result's reliability.

Therefore suspicions about floating-point results are likely to remain unexplored without easy availability of software debugging tools so well suited to that task as to bring its cost down by orders of magnitude.

Is the demand for such debugging tools so ample as to offer their developers attractive profits? Not yet.

The scientists and engineers and statisticians *etc*. who would use and benefit from tools peculiarly suited to debug floating-point have failed to demand them for lack of awareness that such tools are feasible, helpful and have all existed in at least rudimentary forms, though never all together.

The development and promulgation of such peculiar tools would require the aggregation of Computer Science expertise capable of modifying everything, …

Hardware and Operating Systems
Programming Languages and Compilers
Linkers and Loaders
Debuggers and Program Development Environments

It’s a lot to ask, especially now that Computer Science and Numerical Analysis are so nearly everywhere separated if not divorced.

Why?

Why do most Computer Scientists regard Numerical Analysis as if it were a sliver under the fingernail?

Besides its difficulty to debug, Floating-Point computation seems so perverse.

A program whose text is punctiliously correct can produce utterly wrong results though its every operation was performed correctly. Worse, results are often too wrong to tolerate but not wrong enough to be obvious.

Floating-Point can go utterly wrong though none of "the usual suspects" have occurred: No subtractions (no cancellation), no divisions (no tiny divisors), few arithmetic operations (no horde of rounding errors); see

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

"Once you acquiesce to rounding errors, you place yourself into a state of sin."

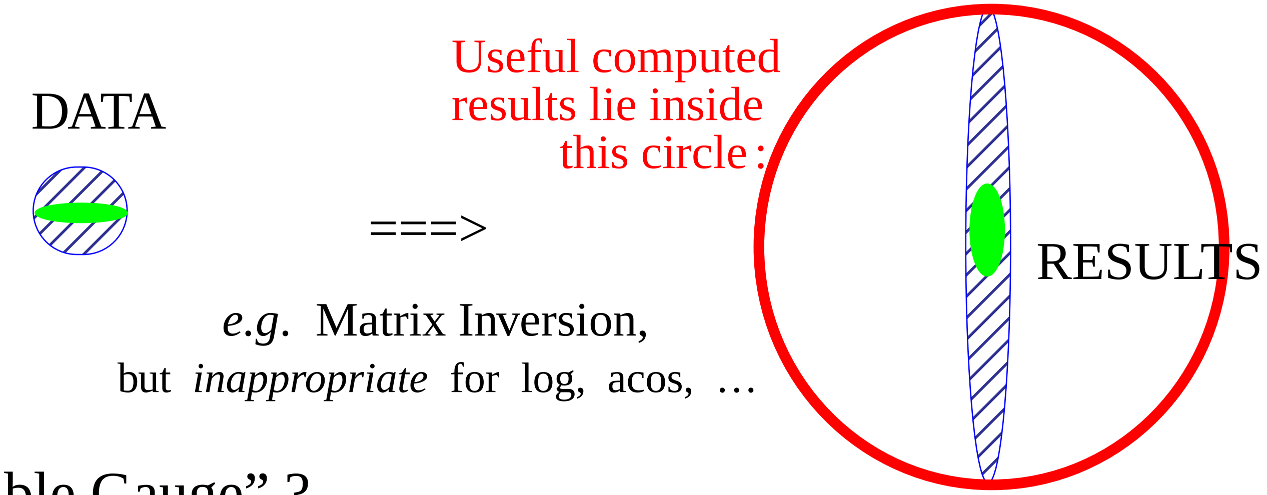
D.H. Lehmer, ~ 1971

And floating-point error-analysis is so ugly. Consider backward error-analyses:

Not all of them are useful. And when one is useful, it may be only for a norm (way of gauging error) that does not suit your data. When is one useful?

A useful backward error-analysis says when, despite roundoff, a program's
computed result differs from the desired mathematically correct result
negligibly (as measured by a suitable gauge)
more than that correct result differs from all other mathematically correct results
obtainable from perturbed input data differing from the actual input data
negligibly (as measured by a suitable gauge). Not every program admits one.

Our chosen gauge may exaggerate uncertainties in actual input data and computed results whose correlations the gauge disregards.



Useful computed results lie inside this circle:

e.g. Matrix Inversion,
but *inappropriate* for log, acos, …

What is "a Suitable Gauge" ?

It suits the analyst and/or the programmer. If not the program's user too then he must find another program with a more suitable gauge. It might not exist yet.

A hot area of research is the development of algorithms for mathematical problems whose structures the usual gauges serve badly. See work at U.C. B. by J.W. Demmel, M. Gu, J.R. Shewchuk, … , and Plamen Koev at San Jose State Univ.

A useful backward error-analysis is an explanation, not an excuse, for what may turn out to be an extremely incorrect result. The explanation seems at times merely a way to blame a bad result upon the data regardless of whether the data "deserves" a good result.

Floating-Point error-analysis is too ugly for inclusion in the syllabus for Computational Engineering;— no rôle for N.J. Higham's book *Accuracy & Stability of Numerical Algorithms* 2nd ed., 700 pp., S.I.A.M Philadelphia.

"…numerical analysis… There's a credibility gap:
We don't know how much of the computer's answers to believe."

in §4.2.2 of D.E. Knuth's *The Art of Computer Programming* Vol. **2** *Seminumerical Algorithms* 3rd ed. (1998) Addison-Wesley

Perhaps what most offends Computer Scientists about Numerical Analysis is its defeat of the way they compose ever more elaborate software from aggregates of well-regarded and/or well-tested correct modules. Unfortunately, though correctness is transitive,

Floating-point Accuracy is not *Transitive* . …

Floating-Point Accuracy is not Transitive :

Suppose `g(y)` is a program that computes a mathematical function $G(y)$ accurately,
and `h(x)` is a program that computes a mathematical function $H(x)$ accurately,
each as accurately as possible in floating-point arithmetic.

Nevertheless, `f(x)` := `g(h(x))` may compute $F(x) := G(H(x))$
utterly inaccurately !

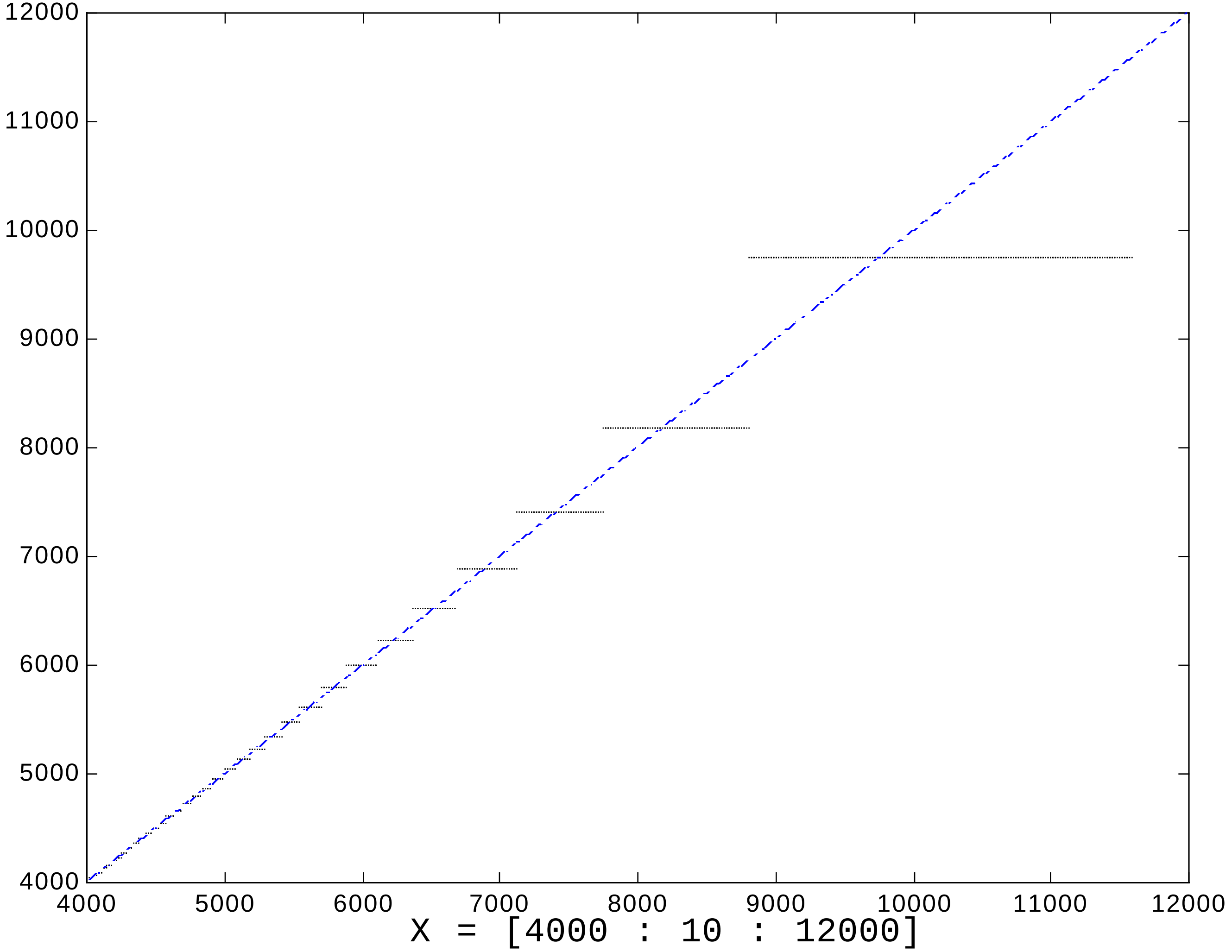
Here is an example contrived to be stark:

- $G(y) := 1/\sqrt[4]{-\log(y)}$ for $0 < y < 1$. $\mathtt{g(y)\ :=\ (-log(y))^{-1/4}}$
- $H(x) := \exp(-1/x^4)$ for $x > 1$. $\mathtt{h(x)\ :=\ exp(-x^{-4})}$
- $F(x) := G(\,H(x)\,) = x$ for $x > 1$. $\mathtt{f(x)\ :=\ g(\ h(x)\ )}$

Therefore $\mathtt{f(x)\ =\ (-log(exp(-x^{-4})))^{-1/4}}$ for $\mathtt{x\ >\ 1}$.

Each of `g(y)` and `h(x)` is accurate in every digit but its last delivered.
How accurate is `f(x) := g( h(x) )` ? Let’s see …

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



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

Only rarely is accuracy lost so severely to intransitivity; otherwise numerical software would be impossible. Some kinds of accuracy are more vulnerable than others to this kind of loss; most matrix operations fall into the more vulnerable category for subtle reasons.

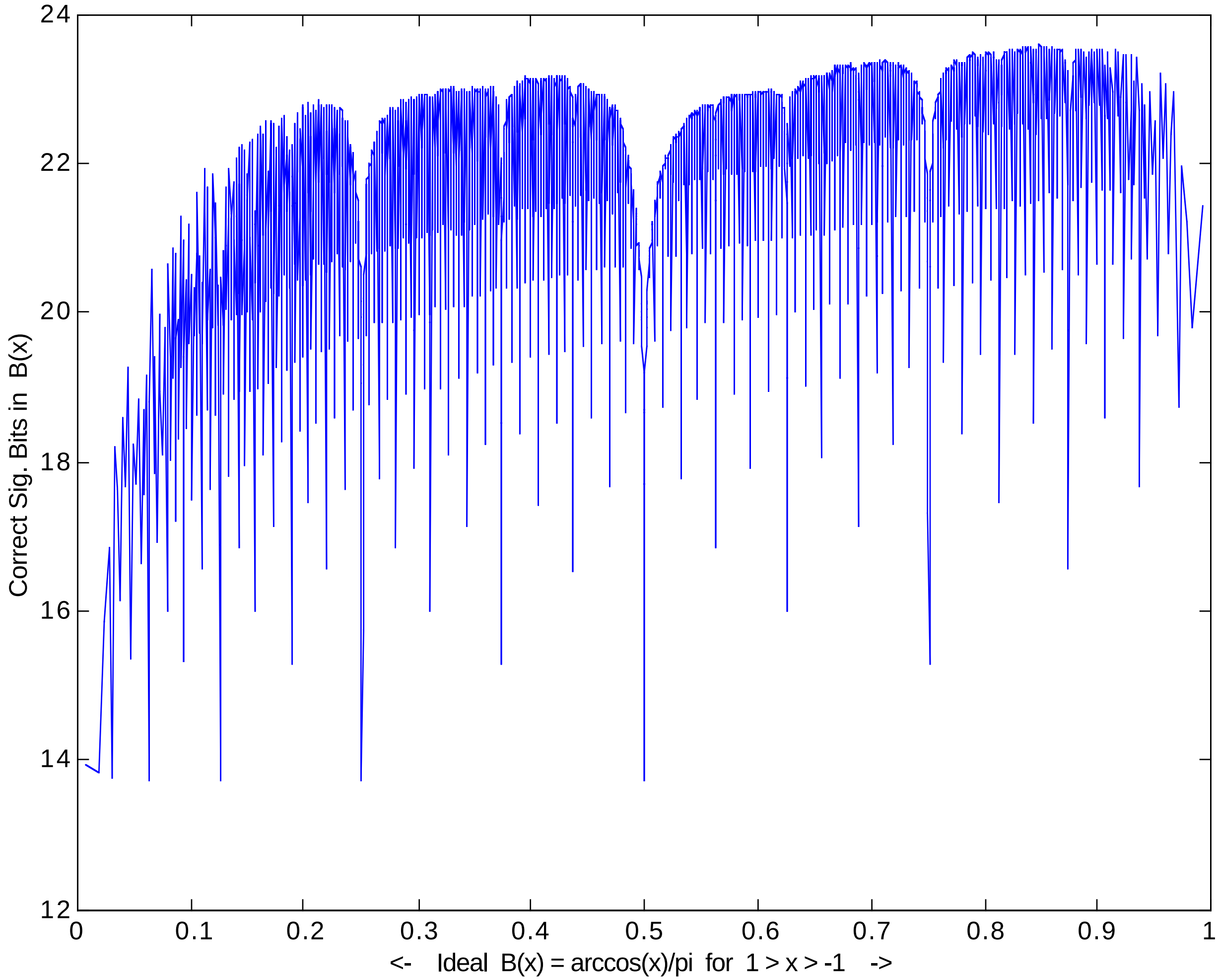
. . . .

In conscientiously tested numerical software,
the rarity of roundoff-induced anomalies makes them
extremely difficult to find by analysis and/or testing.

Worse, the anomalies can be simultaneously
rare, hard to find, and dense in the data.

An instance that stayed hidden from 1949 to 1951 is `arccos` on the computer EDSAC at Cambridge. The program had passed tests on at least a hundred arguments, so it was trusted fully until A. van Wijngaarden noticed and explained its anomalous results. See pp. 36-41 of `<www.cs.berkeley.edu/~wkahan/MktgMath.pdf>` for the ingenious short algorithm that computed EDSAC's `arccos` erroneously as plotted below:

Of 24 Sig. Bits Carried, how Many are Correct in EDSAC’s `arccos(x)` ?



Accuracy spikes downward wherever `arccos(x)`/π is very near (but not exactly) a small odd integer multiple of a power of 1/2 . The smaller that integer, the wider and deeper the spike, down to nearly half the sig. bits carried. Such arguments x are common in practice.

Inadequate accuracy can be ubiquitous and yet so sparse as hardly likely to be found by random testing**!** A recent instance is the 1994 Pentium FDIV bug; lots of stories about it are on the WWW. Anomalous losses of accuracy can defy detection for far too long :

- PATRIOT Anti-Missile missiles missed a SCUD that fell on a barracks in the Gulf War. The miss was traced to several hours' accumulation of roundoff. A hit would not have helped**!**
- Over a weekend in Nov. 1983 the Vancouver Stock Exchange index of mostly mining stocks jumped from 524**.**811 to a more accurate 1098**.**892 after a years-long roundoff bug was "repaired" in a way likely to inflate the index very slowly.
- From 1988 to 1998, MATLAB's built-in function `round(x)` that rounds `x` to a nearest integer-valued floating-point number malfunctioned in 386-MATLAB 3.5 and PC-MATLAB 4.2 by rounding every sufficiently big *odd* integer to the next bigger *even* integer. Mac MATLAB was O.K. thanks to Apple's S.A.N.E. on the M68040.
- For more than a decade, MATLAB has been miscomputing `gcd(3, 2^80) = 3` , `gcd(28059810762433, 2^15) = 28059810762433` , `lcm(3, 2^80) = 2^80` , `lcm(28059810762433, 2^15) = 2^15` , and many others with no warning. See `<www.cs.berkeley.edu/~wkahan/MathH110/GCD5.pdf>` for corrected programs and `<.../HilbMats.pdf>` for their application to the exact construction of Hilbert matrices and their inverses that are used to test numerical linear algebra software.

When roundoff corrupts a computation badly enough to mislead, its error is hardly ever obvious.

Here is an incident more nearly typical than those cited so far:

In the 1990s, engineers at NASA Ames in Mountain View, Calif., were developing a program to predict the deflections under loads of the structures of proposed super-sonic transports intended to compete with the French-British *Concorde*. (None proposed were built.)

Though designed to run on CRAY-1 and CRAY-2 supercomputers, the program was first debugged on SGI workstations used as terminals for the expensive supercomputers.

For a structure about as big as would fit in the workstation, it and the two CRAYs got three sets of results each disagreeing with both others in the third sig. dec. despite that all machines' floating-pt. arithmetics carried at least 48 sig. bits,— worth at least 14 sig. dec.

Could either CRAY's results for far bigger realistic structures be trusted?

In very slow doubled precision on the CRAY-2, the program got results that agreed to several more sig. dec. with the workstation's, whose arithmetic conforms to IEEE 754.

I traced the CRAYs' aberration to bias in their idiosyncratic roundings. Today's CRAYs conform to IEEE 754.

After the program was revised to use *Iterative Refinement* it got good results on CRAYs.

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>`
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 severe are the consequences of roundoff-induced numerical anomalies?

Nobody keeps score, so nobody knows how often scientific and engineering computations in floating-point suffer embarrassing (if noticed) anomalies due to roundoff. They can't be negligible; some known examples like those listed above make us uneasy.

Hereunder is an example to make graduate students uneasy:

In 1961, though an assistant professor at the Univ. of Toronto trying to solve differential equations numerically, I was trusted enough to be allowed to alter IBM's software on the IBM 7090. SHARE (IBM mainframes users' group) accepted most of my alterations. My accounting system alterations were copied by the Univ. of Maryland among others.

IF KICKED(OFF) .… , Retrospective diagnostics, …

An ODE's aberration was traced to an inaccurate LOG(X) in IBM's math. library. I replaced it with a faster and more accurate LOG(X). Before substituting it for IBM's in the math. library everyone used, I tested it on a few days' of their batched jobs, just the ones that used LOG, taken off tapes on the IBM 1401. In those days jobs' outputs included accounts of both time used and math. library functions (like DLLs) used.

Did IBM anticipate that royalties would be charged for DLLs' use?

Of about two dozen jobs' outputs, only two were affected noticeably by LOG's revision. One was a psychologist's, angry because he had already sent results to be published.

" Why couldn't you get LOG right the first time ? "

The second affected job belonged to a grad. student of Aeronautical Engineering who had an idea for the wings of STOL aircraft. He proposed to expel air, bled off the turbo-prop engine's compressor, through slots in the wing to enhance lift for shorter take-offs and landings. Achieving the same objective by slats in the front and flaps in the rear of a wing imposes narrowed limits upon an aircraft's attitude lest it stall. Worse, stall's onset can be so abrupt as to give pilots no warning. The student hoped his scheme would cure this.

cf. model aircraft's wing's "Washout"

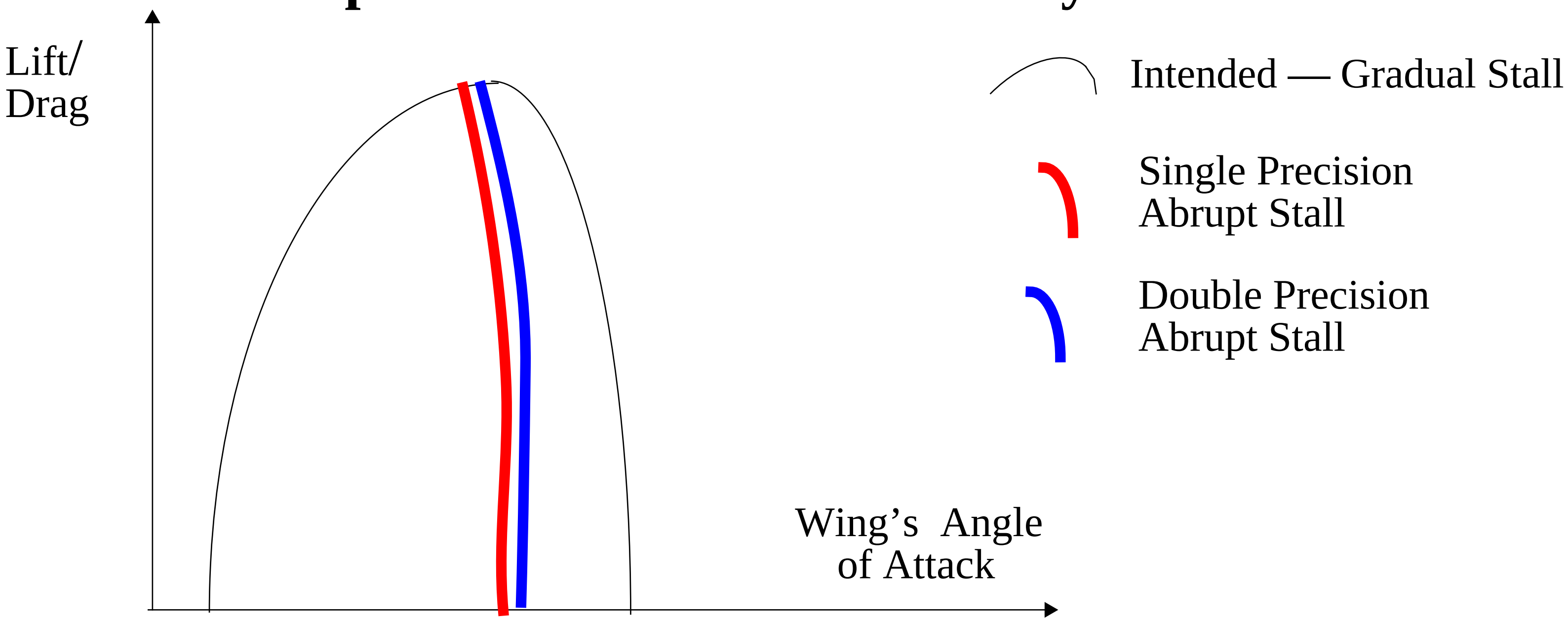
When I showed him his job's two sets of results, the old set from IBM's LOG and a new set from my LOG, he said sadly

> "Your new results show a gradual stall. I wish they were correct, but they cannot be.
> My old results, showing abrupt stall, have been corroborated in Double precision."

Chastened, I re-examined my new LOG and ran it through more exhaustive tests. A week or two later, IBM released new support software for the 7090, and abrupt stall went away from both Double and Single precision, the latter with my new LOG. The grad. student and his advisor were delighted to have an industrially significant thesis.

For a while.

Abrupt Stall of Lift Enhanced by Blown Slots ?



Abrupt stall “caused” by inaccurate LOG in Single, by lack of guard digit in Double precision.

In 1963 the U. of Toronto’s 7090 was replaced by a faster IBM 7094 with Double precision hardware, and abrupt stall came back only in Double precision. Abrupt stall was traced to lack of a *guard digit* in the 7094’s double-precision hardware. Abrupt stall went away when “(X – 1.0)” was replaced by “((X – 0.5) – 0.5)”. Can you see why?

After graduation the student went to work for De Havilland of Canada where his program encountered abrupt stall again when run on a Univac 1108. The cause turned out to be an “optimizing” FORTRAN compiler that put “((X – 0.5) – 0.5)” back to “(X – 1.0)”.

I took years after the abrupt stall episode to appreciate its relevance to a question:

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).
  Apparently this combination has been extremely rare, perhaps fortunately.
- Discordant results of recomputations using different arithmetics or different methods.
  What would induce someone to go to the expense of such a recomputation?
- Suspicions aroused by computed results different enough from one's expectations.
  Someone would have to be extraordinarily observant and experienced.

We don't know, because nobody has been keeping score.

Suppose roundoff falsifies a computed simulation of a proposed design. Then …

- If success is predicted but a trial implementation of the design malfunctions, and if the malfunction is traced back to a miscomputation, will this mistake become public?
- If failure is predicted and the design remains unimplemented, who'll know truly why? If later a realization of the design succeeds, who'll scrutinize that false simulation?

Isolated anomalies (due to roundoff ?) are unlikely to be diagnosed.

With tools currently available, **Time** and **Cost** impede diagnosis.

TIME:

Isolated anomalies were encountered during the development of SCALAPACK, intended to run on almost all platforms used for scientific and engineering computation. Many an anomaly would occur on only one of the many platforms on which SCALAPACK was run. A graduate student assigned to debug the anomaly would fail to diagnose it before it went away (or went somewhere else) when that platform's hardware or compiler was updated. Perhaps an over-optimizing compiler was at fault. Often we never found out.

COST:

Computers are cheap enough for use to compute results worth less than the time a PhD-bearing error-analyst would spend trying and probably failing to debug their anomalies.

What if a a tiny spot in a medical image indicates a cyst or tumor in the brain ?
A lucky patient's surgeon digs in and finds nothing.
A luckier patient's surgeon asks for more detailed images, and they show nothing amiss.
Who'll investigate this incident to see how often the software generates false positives?
How unlucky is a patient for whom the software generates a false negative?

By now I hope you have seen enough evidence to be persuaded that floating-point software is probably far more infested by lurking bugs than is generally believed.

Prophylaxis:

An ounce of prevention is worth a pound of cure.

Floating-point roundoff can do damage so bad, we should try to preclude it by *Default* use of extravagantly high precision .

`<www.cs.berkeley.edu/~wkahan/CS279/RRR.pdf>` explains how much extravagantly high precision is likely to work well if used by default.

The new IEEE 754 (2008) specifies 16-byte wide Quadruple Precision. It has been implemented by IBM, Sun and others. It should be extravagant enough, but serves our needs only if it is the default precision for real scratch-variables.

But that much precision may be impracticable. We still need tools to diagnose occasional anomalies *suspected* to be due to roundoff.

Currently, debuggers allow conditional break-points to be set into a program, and allow single-stepping through it. Grateful though I am for these capabilities, I have found them inadequate to cope with hundreds of lines of floating-point expressions each evaluated billions of times before something happens whose anomalous nature will not become evident until after billions more evaluations have occurred, all in a few seconds. And a program that must be recompiled to be debugged cannot be debugged if the bug is caused by over-optimization that recompilation changes.

The following case study describes one of the debugging tools I have found most helpful.

A Didactic Hypothetical Case Study: Bits Lost in Space

Imagine plans for unmanned astronomical observatories in orbits perpendicular to the ecliptic around the sun. They will (re)position themselves according to comparisons of an *Ephemeris* with telescopic observations of stars and planets. Extensive simulations exercise three different versions of the software that will manage these observatories. Each version is assembled from modules coming from diverse sources. Many modules come as object-modules precompiled and ready to be loaded from, say, DLL libraries.

Many modules come without source-code, or
with source-code nobody has the time to read.

Discrepancies appear during the simulations. Among *millions* of tests are a mere handful about which different software versions disagree significantly.

The disagreements are attributed to roundoff because they go away when data—
positions, attitudes, time, calibrations, …
— are changed slightly. Otherwise 4-byte `float` arithmetic would have been adequate.

How do we discover which software version (if any) is right? And what is wrong with the rarely inaccurate versions? These aren’t rhetorical questions.

The software is assembled from modules whose inputs are other modules' outputs. At some level the interfaces between modules are accessible to scrutiny and even alteration. So, what can I do that You Can't to identify possibly aberrant modules ?

I can *rerun* the software in question on *exactly* the same *PRECIOUS DATA* as generated the disagreements, but with selected modules altered

WITHOUT ALTERATION NOR ACCESS TO THEIR CODES

to round differently: all up, all down, or all towards zero. (I dare not change some non-default roundings in the Math. library.) Modules whose four results from four different rounding modes disagree too much become *suspected* (but not yet convicted) of numerical hypersensitivity to roundoff at the precious data in question.

What do I have that you haven't? My very old computer systems

from the late 1980s and early 1990s,

hardware, compilers, debuggers, … .

They let me inject control word changes that then over-ride default rounding modes to alter arithmetic only in chosen program modules, and with no other changes to them.

For details see §11 of `<www.cs.berkeley.edu/~wkahan/Mindless.pdf>` .

The modules that come under suspicion are supposed to compute the angles subtended at the observatory by stars or planets whose positions are read from a table (an *Ephemeris*).

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
$\theta,\ \Theta$	Azimuth	Right Ascension	Longitude
$\phi,\ \Phi$	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, actually $|\,\theta-\Theta\,|$ **mod** 2π. Three implementations of this function ψ have been compared; they were called **u**, **v** and **w**. From 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*	3.14082*050*	3.14055*681*
$\psi \approx$ **v** :	0.00159324*868*	0.0002441406*10*	0.0001726334*89*	0.00056223187*1*	3.140*61618*	3.140*61618*
$\psi \approx$ **w** :	0.00159324*868*	0.0002441406*10*	0.0001726334*89*	0.00056223187*1*	3.140*78044*	3.140*54847*

Which digits are *wrong* ? Which (if any) of subprograms **u**, **v** and **w** dare you trust ?

Which if any of subprograms **u**, **v** and **w** dare you trust? They have now been rerun on the suspect data in different rounding modes mandated by IEEE Standard 754. Fortunately, they were rerun on a system that permitted the directions of all default (to nearest) roundings to be changed without recompilation of the subprograms. Here are two of the six sets of results:

$\theta-\Theta$:	0.000244140625			2.58913445		
ϕ :	0.000244140625			1.57074428		
Φ :	0.000244140654			-1.56994033		
$\psi \approx \mathbf{u}$:	0.000598019920	NaN arccos(>1)	0.000598019920	3.14061594	3.14067936	3.14082050
$\psi \approx \mathbf{v}$:	0.000244140581	0.000244140683	0.000244140581	3.14039660	3.14159274	3.14039660
$\psi \approx \mathbf{w}$:	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

Only subprogram **w** seems practically indifferent to changes in rounding's direction. It uses an unobvious formula stable for all admissible `float` data. Subprogram **u** uses a naive 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. See §11 of `<www.cs.berkeley.edu/~wkahan/Mindless.pdf>` for the formulas. If not for roundoff all three subprograms would agree.

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

The ability to redirect rounding is mandated by IEEE Standard 754 (1985) for floating-point arithmetic. It is a valuable diagnostic aid albeit far from foolproof. We need it to help debug schemes contrived to exploit parallelism aggressively.

Some compilers have supported dynamically redirected rounding, but almost no programming languages and their debuggers support it. Except maybe C99 ?

Java outlaws redirected rounding.

See `<www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf>` .

The lack of use of this capability is leading to its atrophy. **Use it or lose it.**

For other desirable debugging tools we may wish were provided by software-development environments, tools that employ high-precision floating-point and interval arithmetic combined (they are not helpful enough by themselves), see §14 of my web posting `<www.cs.berkeley.edu/~wkahan/Mindless.pdf>` . One of the techniques discussed there runs two programs in lock-step. One is the program being debugged and the other is a version of it recompiled to use substantially higher precision.

This technique does not run them forward until their correspondingly named variables diverge too far; that would be futile. This technique is much easier to use than that.

And now for something entirely different:

Floating-Point Exceptions

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 Division-by-zero, End-of-file, or an attempt to Dereference a Null Pointer.

IEEE Standard 754 for Floating-Point Arithmetic uses "Exception" somewhat ambiguously for a class of events or one of them, like Division-by-zero, INVALID OPERATION SQRT(–5.0), or Overflow, that, by default (in the absence of a contrary request by the program), generates a value *presubstituted* for the exceptional operation and, as a side-effect, signals the event by raising a *flag* which the program can sense later, or (as happens most often) ignore.

An Exception is so called because a programmer might reasonably take exception (in the legal sense) to any policy, imposed in advance of the program's invocation, intended to cope with a specific class of arithmetic Exceptions. For instance, terminating execution upon an overflow and exhibiting its location plus a traceback of the subroutine return stack is a policy appropriate for debugging a program while it is under development, but may well be a perilous policy afterwards, as we shall see.

IEEE 754's Five Floating-Point Exception-Classes:

INVALID OPERATION	defaults to NaN Not-a-Number
OVERFLOW	defaults to ±∞
DIVIDE-BY-ZERO (∞ from finite operands)	defaults to ±∞
INEXACT RESULT	defaults to a rounded value

UNDERFLOW is GRADUAL and ultimately glides down to zero by default.

Each Exception raises one of five *flags* that will clear only upon the program's command. Each flag serves as a logical value and, ideally, as a subterranean pointer to the location of a *Milestone* in the program near where the flag was first raised after it was cleared.

Ideally, the path of a program's control would be memorialized by Hansel & Gretel's *Breadcrumbs*. More practical would be a circular file of entries of the IDs of *Milestones* passed by the program. The compiler should always drop a milestone at the start of every basic block and wherever else the programmer requests it, and every milestone should ideally have its own unique ID. The correlation between milestones in the source-code and those in the executable code will be imperfect because of aggressive optimization, but

> "… 'Nothing avails but perfection' may be spelt shorter: 'Paralysis'."
>
> Winston S. Churchill, 6 Dec. 1942

For better Exception-handling than is provided by current programming languages other than C99 and perhaps FORTRAN 2003, see suggestions in …

`<www.cs.berkeley.edu/~wkahan/Grail.pdf>` and `<…/ARITH_17U.pdf>`.

INVALID OPERATION	defaults to NaN Not-a-Number
OVERFLOW	defaults to ±∞
DIVIDE-BY-ZERO (∞ from finite operands)	defaults to ±∞
INEXACT RESULT	defaults to a rounded value

UNDERFLOW is GRADUAL and ultimately glides down to zero by default.

Floating-point Exception-handling is a crucially important issue because …

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
BASIC:	ON ERROR GOTO … ; ON ERROR GOSUB …
C :	setjmp/longjmp; ERRNO; abort
ADA:	Arithmetic Error *Falls Through* to a handler or the caller, or aborts
Java:	try/throw/catch/finally; abort showing error-message and traceback

These disruptions are not disallowed by IEEE Standard 754; but it requires a program to demand one of them. They must *not be the default* for any floating-point Exception-class.

Why ***not*** ?

Why does conformity to IEEE 754 disallow disruptions of control, unless demanded by the program, as the *handlers by default* for Floating-point Exceptions ?

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 (albeit not easy) ways to cope with Floating-point Exceptions, especially by programmers who incorporate other programmers’ subprograms into their own programs.

Error-Prone:

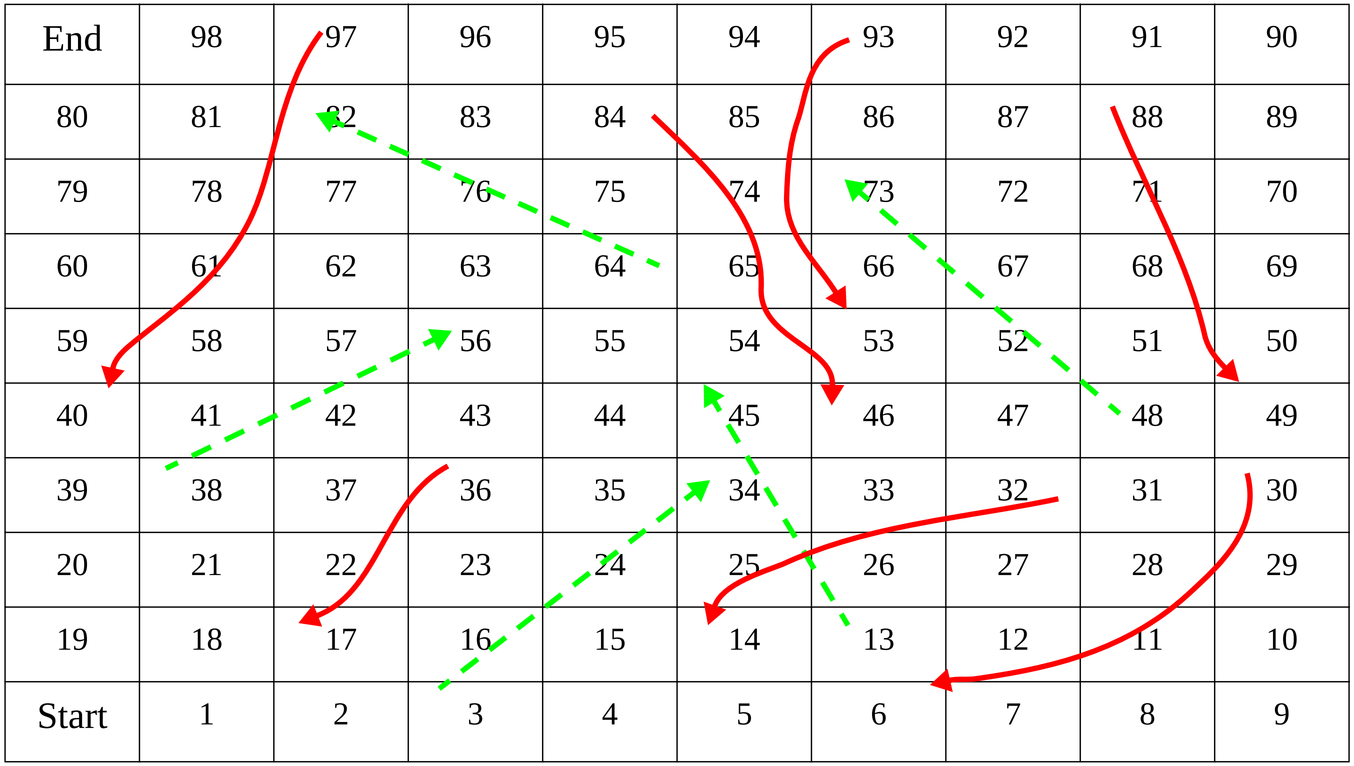
Prof. Westley Weimer’s PhD. thesis, composed at U.C. Berkeley, exposed hundreds of erroneous uses of try/throw/catch/finally in a few million lines of non-numerical code. Mistakes were likeliest in scopes where two or more kinds of exceptions may be thrown. See `<www.cs.virginia.edu/~weimer>`.

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

Floating-point programs are probably more prone to error than others because every operation is susceptible, unless proved otherwise, to more than one kind of Exception. And these programs have lots of operations; a handler could be entered from any one, quite possibly unanticipatedly.

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

Snakes-and-Ladders

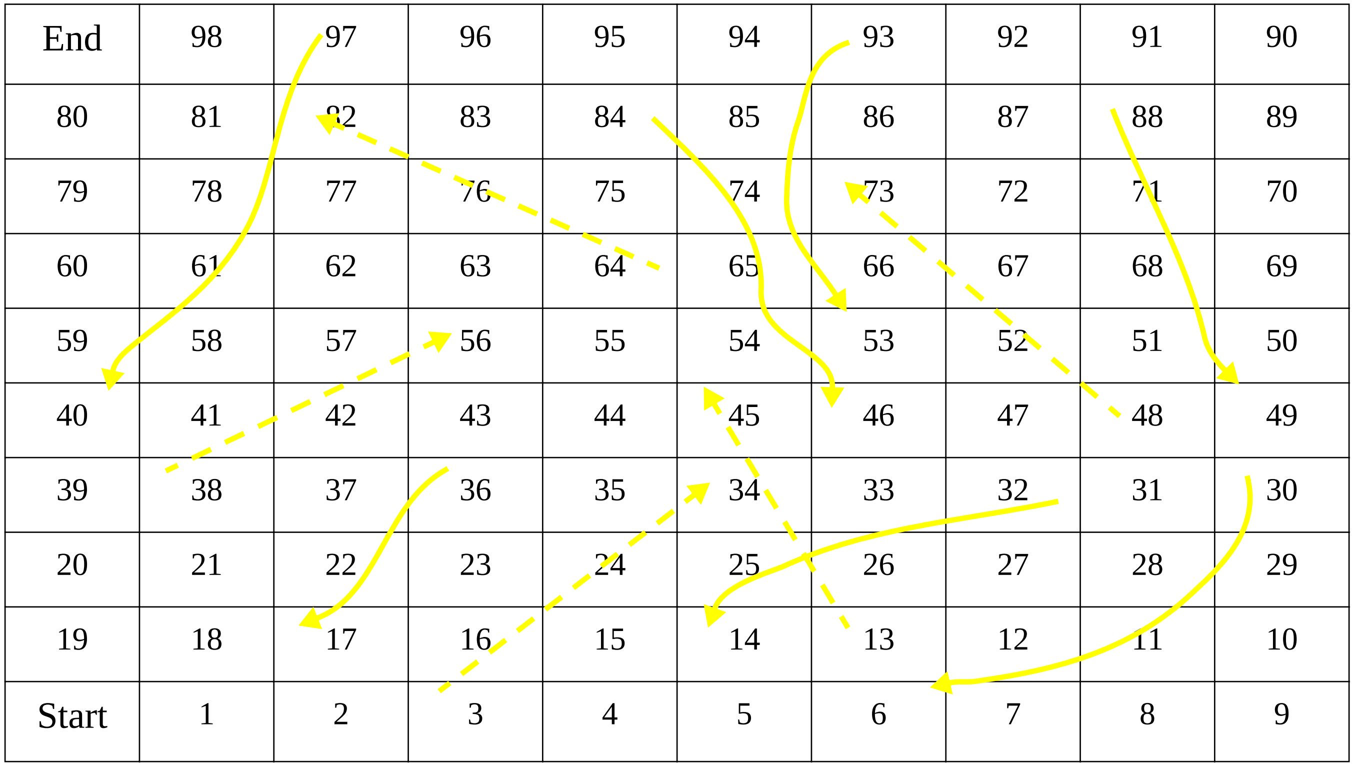


End	98	97	96	95	94	93	92	91	90
80	81	82	83	84	85	86	87	88	89
79	78	77	76	75	74	73	72	71	70
60	61	62	63	64	65	66	67	68	69
59	58	57	56	55	54	53	52	51	50
40	41	42	43	44	45	46	47	48	49
39	38	37	36	35	34	33	32	31	30
20	21	22	23	24	25	26	27	28	29
19	18	17	16	15	14	13	12	11	10
Start	1	2	3	4	5	6	7	8	9

… 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.

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

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

Now decommissioned, the USS Yorktown was among the first warhips extensively computerized to reduce crew (by 10% to 374) and costs (by $2.8 million per year).

On 21 Sept. 1997, the Yorktown was maneuvering off the coast of Cape Charles, VA, when a crewman accidentally ENTERed a blank field into a data base. The blank was treated as a zero and caused a Divide-by-Zero Exception which the data-base program could not handle. It aborted to the operating system, Microsoft Windows NT 4.0, which crashed, bringing down all the ship's LAN consoles and miniature remote terminals.

The Yorktown was paralyzed for $2\frac{3}{4}$ hours,
unable to control steering, engines or weapons,
until the operating system was re-booted.

Fortunately the Yorktown was not in combat nor in crowded shipping lanes.

See `<www.gcn.com/Articles/1998/07/13/Software-glitches-leave-Navy-Smart-Ship-dead-in-the-water.aspx>`

If IEEE 754's default had been in force, the division by zero would have insinuated ∞ and/or NaN into the data-base, which could have been debugged afterwards without a crash.

. . . .

The half-a-billion-dollars Ariane 5 disaster of 4 June 1996

The Ariane 5 is a French rocket that serves nowadays to lift satellites into orbit.

On its maiden flight it turned cartwheels shortly after launch and was blown up, scattering half a billion dollars worth of payload and the hopes of European scientists over a marsh in French Guiana. The disaster was traced to an Arithmetic Error,— Overflow,– in a software module monitoring acceleration (due to gravity and tidal forces) and used only while the rocket was on the launch-pad. This module's output was destined to be ignored after rocket ignition, so it was mistakenly left enabled; but it aborted upon overflow.

A commission of inquiry blamed the disaster upon software tested inadequately.
What software failure could not be blamed upon inadequate testing?

Since then the question "Who is to blame?" has spawned dozens of responses :
`<www.rvs.uni-bielefeld.de/publications/compendium/incidents_and_accidents/ariane5.html>`
…updated to 13 July 2005 by Prof. Peter B. Ladkin

Nobody else has blamed the *Fall-Through* policy of the programming language ADA.

If the overflow had not been trapped, but instead had raised a flag and generated an ∞ or any other value, both would have been ignored, and the Ariane 5 would not have crashed.

A trap too often catches creatures it was not set to catch.

. . . .

Modern commercial and military jet aircraft can achieve their efficiencies only because

they Fly by Wire :

The pilot’s control stick, wheel and pedals connect only to a computer. It commands the control surfaces (ailerons, elevators, rudder) to move in ways often counter-intuitive, sometimes limited for safety’s sake. And the computer can shake the pilot’s stick.

Suppose an aircraft in a banked turn suffers a lightning strike or severe turbulence that overwhelms a sensor that sends the computer an extraordinary signal that precipitates an *unanticipated* INVALID OPERATION that puts a message onto the pilot’s screen:

“ INVALID OPERATION at line 276 of CZRXPT in line …
[STOP**]** **[C**ONTINUE**]** ”

or, worse,

“ *PLEASE* RESTART.”

. . . .

We may never know what happened on 1 June 2009 to *Air France* #447 (Airbus 330) 35000 ft. over the Atlantic about 1000 mi. North-East of Rio de Janeiro.

. . . .

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 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.

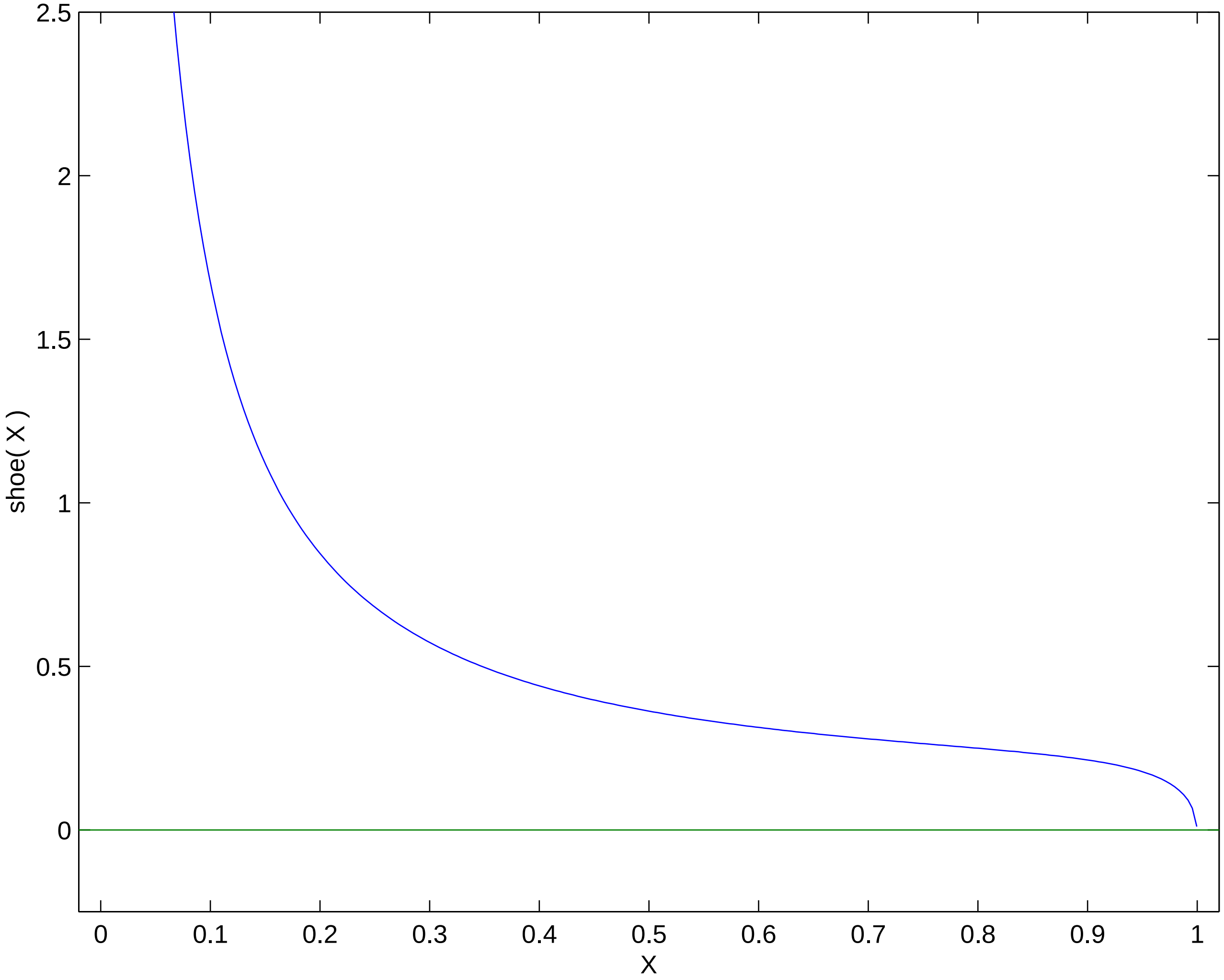
Does your rootfinder find Z ? Or does it persuade you that Z probably does not exist ?

Try, say, each of 19 initial guesses x = 0.05, 0.1, 0.15, 0.2, …, 0.5, …, 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.

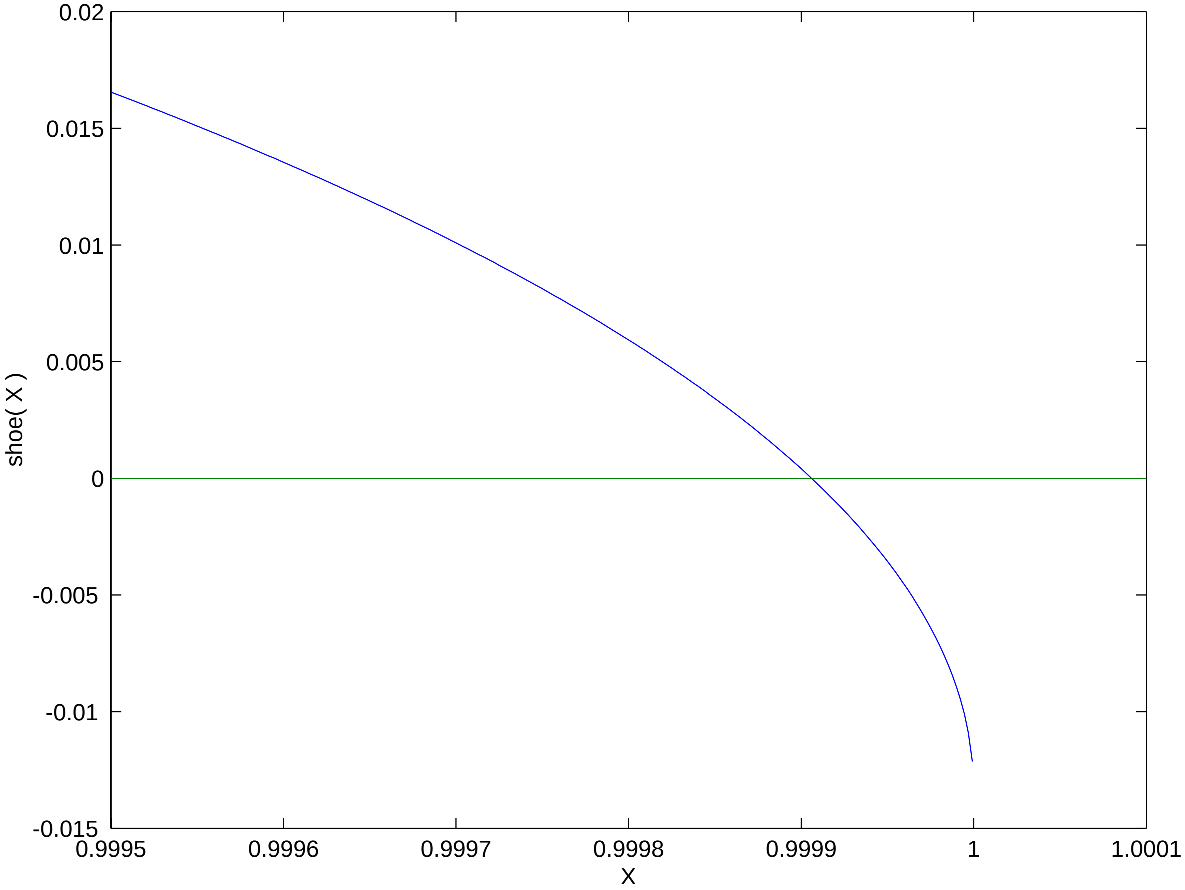
Why did [SOLV] on HP-18C, 19C and 28C handheld calculators find what they didn't ?

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



If no positive Z in shoe(x)'s domain satisfies shoe(Z) = 0 ,
then the SHOE leaks 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's "An Equation Solver for a Handheld Calculator" in the *Hewlett–Packard Journal* **38** #8 (Aug. 1987) pp. 30–34.

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 things, to be explained hereunder, are intended for Floating-Point computations.

Whether 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.

1• An Algebraically Completed Number System

… is one whose every operation upon members of the system produces a defined result, usually another member of the system, or else a LOGICAL or a character string or …

The system has no forbidden operation, none whose result is "undefined" or "Platform Dependent".

Consequently the control of computations in that system need never be trapped nor disrupted unless a program explicitly demands disruption for its selected class(es) of *Exceptions*,– occurrences of some operations with some operands or some results expected to occur only rarely. And each such occurrence generates a side-effect,— raises a *flag*.

IEEE 754's Floating-Point Numbers *approximate* the familiar Field of Real Numbers Algebraically Completed by the adjunction of ±∞ and NaN (to be described later). No roundoff nor over/underflow afflicts the Algebraically Completed Real Numbers, but they do admit Division-by-Zero. Their Exceptional arithmetic operations are, as expected,

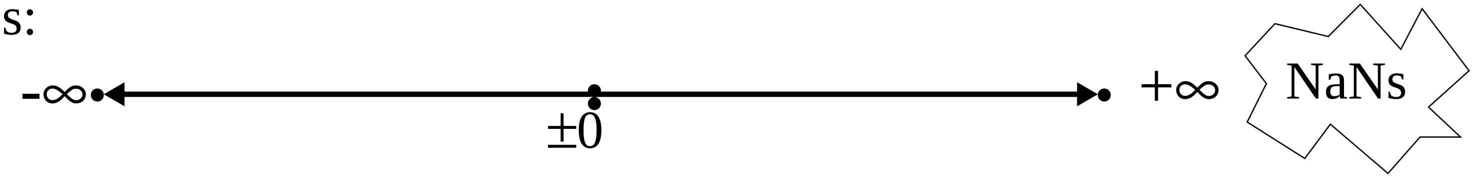
$$\text{finite}/0,\quad 0/0,\quad \infty/\infty,\quad \infty\cdot 0,\quad \infty-\infty,\quad \text{real}\sqrt{<0}\,,\quad \text{and order-comparison with NaN}\,.$$

They spawn potential violations of the Real Field's cancellation laws: $x-x = 0$, $x/x = 1$. Also of its *Trichotomy*: $x < y$ or $x = y$ or else $x > y$, unless x and/or y is NaN .

Algebraic Completion of the Real Numbers
is feasible in more than one way
no one of which always suits everybody.

Three Algebraic Completions of the Real Numbers

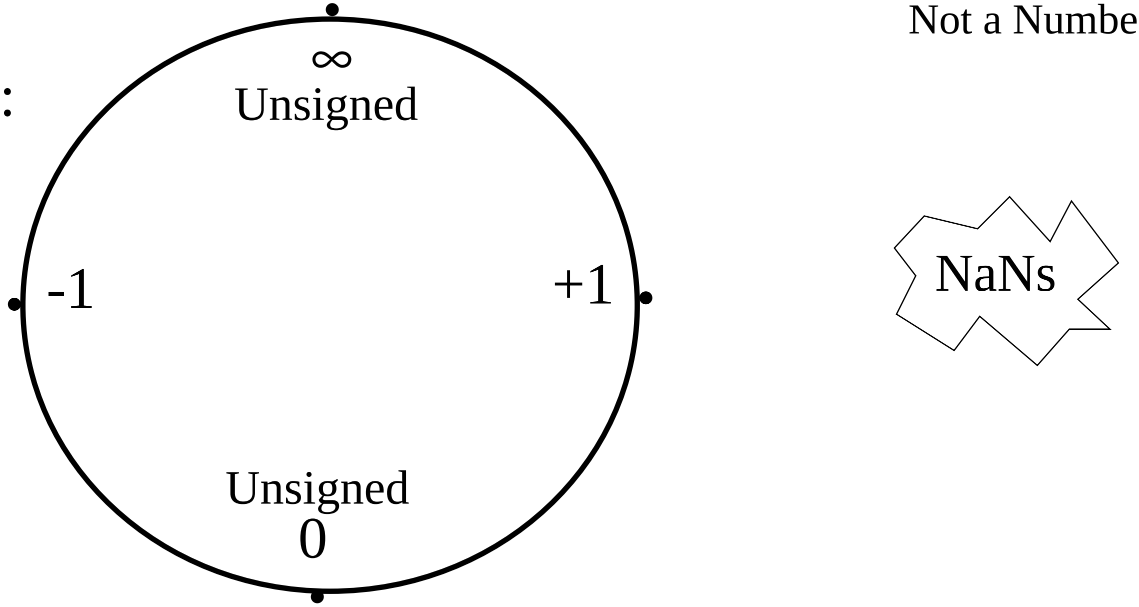
IEEE 754’s:

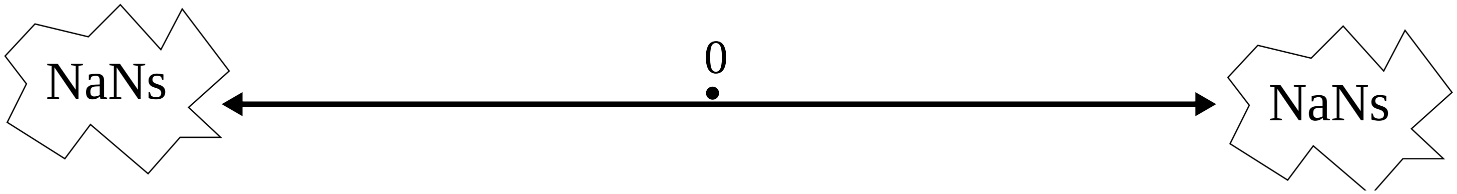


(A NaN is
Not a Number)

Projective Closure:

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





Digression about Algebraically Completed Real Numbers :

By violating cancellation laws, the adjunction of ∞ and NaN threatens the validity of the distributive law for Real expressions. Analysts invoke these laws and associative and commutative laws routinely to rearrange expressions in ways intended to speed up their evaluation in Floating-Point arithmetic and/or reduce their vulnerability to over/underflows and roundoff that do not afflict Real numbers. Here is an EXAMPLE :

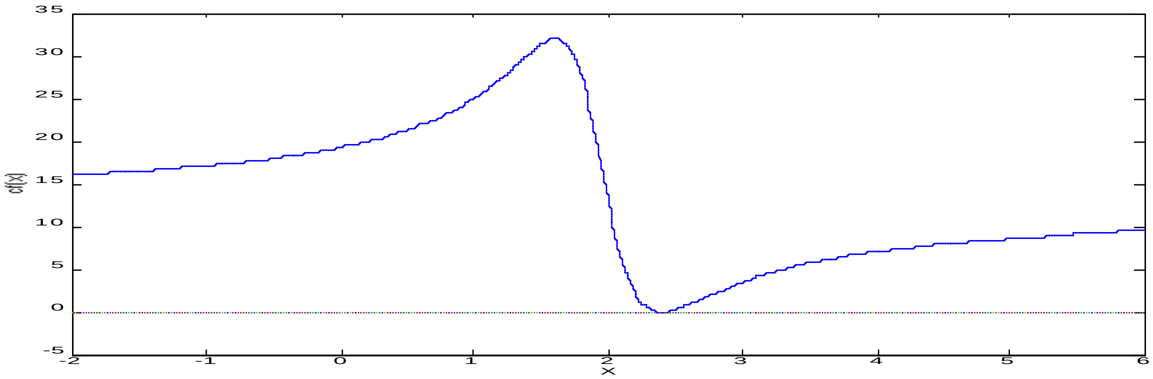
The *Continued Fraction* cf(x) …

$$\mathrm{cf}(x) = 13 - \cfrac{12}{x-2-\cfrac{1}{x-7+\cfrac{10}{x-2-\cfrac{2}{x-3}}}} = \frac{2152 - x(2551 - x(1080 - x(194 - 13x)))}{112 - x(151 - x(72 - x(14 - x)))} = 13 - \frac{12(x-2)((x-5)^2+4)}{x + (x-2)^2((x-5)^2+3)}$$

{4÷ 8±} {1÷ 7• 8±} {1÷ 5• 6±}

{Numbers in braces count operations.}

It has been rearranged in two ways requiring just one slow division each. The middle expression was obtained by ordinary algebraic simplification; the last expression was obtained by a rather unobvious method to need fewer multiplications. The graph of cf is smooth and altogether unexceptional …



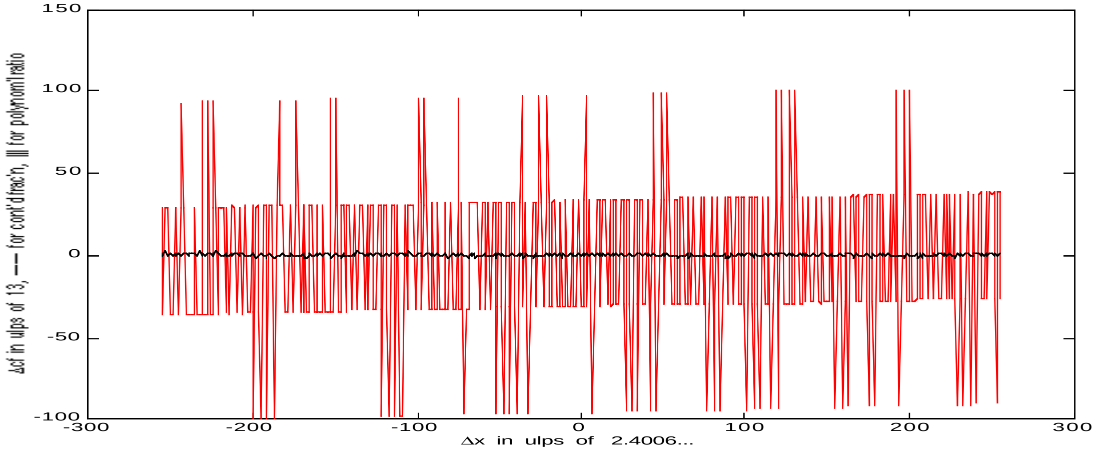
$$\mathrm{cf}(x) \;=\; 13 - \cfrac{12}{x-2-\cfrac{1}{x-7+\cfrac{10}{x-2-\cfrac{2}{x-3}}}} = \frac{2152-x(2551-x(1080-x(194-13x)))}{112-x(151-x(72-x(14-x)))} = 13 - \frac{12(x-2)((x-5)^2+4)}{x+(x-2)^2((x-5)^2+3)}$$

{4÷ 8±} {1÷ 7• 8±} {1÷ 5• 6±}

{Numbers in braces count operations.}

The evaluation of cf using the first expression is unexceptional too except for DIVISIONS-BY-ZERO at x = 1, 2, 3 and 4; they generate infinities that evaporate harmlessly. The other two expressions incur no divisions by zero but produce NaN instead of cf(∞) = 13. Consequently, when evaluated in Floating-Point at huge arguments x these two become infinite or NaN because of Overflows.

The middle expression loses about two more decimal digits than the others lose to roundoff. The plot below shows how the first and second expressions vary when evaluated at 513 consecutive floating-point arguments centered around x = 2.4006… where cf(x) achieves its minimum. The first expression fluctuates by at most a unit or two in the last digit of 13.000…000 as shown by the darker nearly horizontal line; the second expression generates the ragged oscillations and spikes.



END OF EXAMPLE

Digression about Algebraically Completed Real Numbers **resumes :**

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 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 both infinite then at least one is NaN .

The Completion chosen by IEEE Standard 754 does this.

So would some other less tolerant Completions; *e.g.* …

- Introduce NaN and only one unsigned ∞ , thus requiring just one unsigned 0 .
- Introduce NaN but no ∞ .

Yet other Completions, like *APL*’s 0/0 := 1 and MathCAD’s 0/0 := 0 , destroy Algebraic Integrity.

Floating-Point, unlike Real, evaluations usually conserve Algebraic Integrity at best approximately after the occurrence of roundoff and over/underflow.

For example, compare evaluations of the three algebraically equivalent expressions

$$2/(1 + 1/x)\ , \qquad 2\cdot x/(1 + x)\ , \qquad 2 + (2/x)/(-1 - 1/x)$$

at Real $x = 0$, Real $x = \infty$, Real $x = -1$ and Floating-Point $x \approx -1.000\ldots???$.

END of Digression about Algebraically Completed real Numbers

Presubstitution …

… provides for each Exception-class a short procedure that will supply a value for any Floating-Point Exception that occurs later, instead of aborting execution.

IEEE Standard 754 provides five presubstitutions by default for …

INVALID OPERATION	defaults to NaN Not-a-Number
OVERFLOW	defaults to ±∞
DIVIDE-BY-ZERO (∞ from finite operands)	defaults to ±∞
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 (finite/∞ = 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* .

- IEEE 754 (2008) allows a kind of *Flush-to Zero* (almost), but not as the default.

  … useful for very 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 ±∞ .

- Sometimes *Saturation* to ±(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.

Digression about NaNs :

“NaN” means “Not a Number”; it is *not* “UNDEFINED”.

… nor to be confused with a river in Siam, a nursemaid, grandmother, a girl’s name, nor Indian flatbread

An INVALID OPERATION, typically an attempt to evaluate a function outside its Real domain, defaults to a NaN whenever any other Real result would worsen confusion.

Examples: $0/0$, ∞/∞ , $\infty - \infty$, $0\cdot\infty$, Real $\sqrt{-5}$, $\arccos(2)$, $\log(-5.6)$, $(-7.8)^{0.3}$, …

Whenever a new NaN is created, the INVALID OPERATION FLAG must be raised too. Later this NaN propagates quietly through every arithmetic operation upon it *except* …

- Order Predicates “ $x < y$, $x \le y$, $x \ge y$, $x > y$ ” are all FALSE and will raise the INVALID flag (unless the program spurns it) when x and/or y is NaN . And then quietly (no flag) “ $x = y$ ” is FALSE , and “ $x \ne y$ ” is TRUE; $\text{NaN} \ne \text{NaN}$.
- If $f(x, y)$ is independent of x for some value of y , say $y = 0$, then $f(\text{NaN}, 0)$ takes the same value as every other $f(x, 0)$. For example, $\text{NaN}^0 = 1$.

IEEE 754 provides at least 4,000,000 NaNs distinguishable by non-numerical means, so each newly created NaN can point (indirectly) to the site of its creation in a program.

A program may use NaNs also for missing data, uninitialized variables, and/or for the result of an ambiguous or unsuccessful search perhaps for a nonexistent value.

END of Digression about NaNs

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.

References to *the* Flags are Floating-Point operations 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 ?

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 *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**.

Flags are Nuisances.
Why bother with them? … continued …

Why not use the DATA-FLOW idea instead ?

This idea attaches an Exception-History to each scalar Floating-Point variable. Its History records all Exceptions that may have affected the variable's value adversely.

Aside from the cost of Histories in memory and execution-time, no good way is known to combine Histories of combined variables that predicts which of a result's past experiences affect it adversely.

For instance, adding a variable affected by Underflow to another big enough renders the underflow inconsequential. Dividing an infinity into something small enough may render the infinity's History irrelevant. What is small *enough* ? Big *enough* ? How much History is enough?

Nobody can read Histories that are too long or too often irrelevant.

. . . .

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

- Apt Presubstitutions render most Exceptions and their Flags ignorable, not all.
- 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.

Why might a program export a raised Flag or pass it through?

- It's a consequence of an oversight, a programmer's mistake. Which programmer?
- A judgment has been deferred to a later stage of the computation. Recall shoe(x) .
- The program's result is Exceptional and deserves *its* Flag. *e.g.*, exp(exp(exp(999**.**))) .
- The programmer did not bother to clear a Flag intended correctly to be ignored.

Recall how IEEE 754's default presubstitutions were chosen.

Retrospective Diagnostics help a program's user sift through the possibilities.

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 Flags 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 logs 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”, 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 Needs Doing …

Yes, it will be controversial.

… by Programming Languages and Compilers:

- The *Default* for Fltg. Pt. scratch variables, constants and expression-evaluation should be the widest precision that the hardware does not run too slowly, in the style adopted by the original Kernighan-Ritchie ***C*** on the old PDP-11 , so that numerically naive programmers are less often betrayed by roundoff.
- Support at least one extravagantly higher precision despite that it runs too slowly.
- Support *Modes* for the choice of directed roundings and non-default presubstitutions
  Scoped in a convenient way, e.g., like *APL*’s System Variables; but …
  Insulate the Math. Library’s functions, especially when *Inlined* .
- Support IEEE 754’s NaNs, and Flags also Scoped in a convenient way.
- Exception-Handling control structures are O.K., but not as the default for Fltg. Pt.
- Disallow Fltg. Pt. optimizations that disregard parentheses unless *Associativity* is enabled explicitly by the program’s text (not command-line).
- Let a module’s local variables be initialized to NaNs that point to variables’ names

… by Operating Systems and Debuggers :

- Support Retrospective Diagnostics’ *Annunciator* and *Interrogator*.
- Support *Pause/Explore/Resume* at designated Fltg. Pt. Exceptions, to debug them.
- Let a Debugger override a selected module’s default roundings by directed roundings, as if the module’s text had invoked a directed rounding Mode, though that text is unavailable except for a symbol-table provided by the compiler.

Who shall bell the cat ?

Who has the knowledge, skills, incentive and time to implement debugging capabilities like those advocated in this document?

The necessary combinations of expertise and motivation might reasonably be thought to reside in Computer Science & Engineering Departments.

Maybe not. Maybe Computer Science has changed too much.

A compendium published in **1983** ,
Encyclopedia of Computer Science and Engineering 2nd ed.
ed. by A. Ralston & E.D. Reilly Jr., 1694 pp., Van Nostrand Reinhold,
explains at length Floating-Point Error Analysis (by J.H. Wilkinson) and control structures intended to handle all kinds of Exceptions (by J.L. Wagener).

A compendium published in **1997** ,
The Computer Science and Engineering Handbook
ed. by A.B. Tucker Jr., 2650 pp., CRC Press & ACM,
explains a few numerical methods but neither roundoff nor Floating-Point Exceptions.

In *Commun. ACM* v. 40 #4, **1997** ,
“The Debugging Scandal and What To Do About It”, pp. 26 - 78,
does not mention Floating-Point at all.

"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.