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

W. Kahan, Prof. Emeritus
Mathematics Dept., and
Elect. Eng. & Computer Science Dept.
University of California @ Berkeley

Prepared for the 2nd Heidelberg Laureates Forum, 22 - 26 Sept. 2014

This document is posted at `www.eecs.berkeley.edu/~wkahan/Hdlbrg2.pdf`.
For many details omitted here see `www.eecs.berkeley.edu/~wkahan/Boulder.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 is posted at `www.eecs.berkeley.edu/~wkahan/Hdlbrg2.pdf`. More details have been posted at `…/Boulder.pdf`, `…/NeeDebug.pdf` and `…/Mindless.pdf`.

"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 obviously wrong ?

Nobody knows. Nobody keeps score.

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

Two Kinds of Evidence:

- Persistence in Software and in Textbooks of numerically flawed formulas that have *withstood* rather than *passed* the *Test of Time* . For example, … Numerically naive formulas used in Finance.

- Occasional Revelations of gross inaccuracies in widely used and respected packages like MATLAB and LAPACK, caused by bugs lying unseen for years. *E.g.*, … Over 40 years of occasional ***under***estimates, some severe, of matrices' ranks.

The evidence shows up when programs malfunction rarely (*how rarely***?**) on otherwise innocuous inputs.

An instance of Persistence: a classical financial formula used for savings and mortgages:

A Misleading Classical Formula

Margaret sells marijuana. It's legal in her State, though illegal under USA Federal law. She finds a local bank willing to handle her money. Every day she and a big man with a bulge under his jacket go to the bank to deposit $1000 in a Savings account and the rest in a Checking account. The Savings account pays interest, nominally at 0.9 % per annum but compounded daily at (0.9/365)% per diem. After 365 deposits, Margaret evaluates a Classical Formula to find out how much the Savings account should hold:

-PMT := $1000 daily for n = 365 days at fractional interest rate i := 9/365000

accumulates to $fv := PMT\cdot((1+i)^n - 1)/i$ = $366,649.84 computed carrying 10 sig.dec.

But she finds only FV = $366,642.90 actually in her account.

"You thought I'd not notice you're *skimming* $6.94 from my Savings**!**" she accuses the banker, and her companion slips his hand under his jacket. "No**!**" protests the banker; "You have been given an inaccurate formula to compute fv. A better formula is …

$fv := PMT\cdot((1+i)^n - 1)/((1+i) - 1)$ = $366,642.92 " computed carrying 10 sig.dec.

Seeing fv algebraically equivalent to fv, the big man drops his hand from his jacket.

Why is fv more accurate than fv ? Is fv 's formula always accurate within a few cents?

Whence comes FV ? From my 33-years old reliable HP-12C Financial calculator.
It adheres to the SIA Handbook, has been adopted as a Standard, and is still made and sold for $70.

The Details of FV, *f*v and fv Explained:

A discretization of the ***Derivative*** $h'(x) := dh(x)/dx$ is the ***Divided Difference***

$$h^\dagger(\{x, y\}) := \big(h(x) - h(y) \big)/(x - y) \quad \text{when } x \neq y ;$$

$$h^\dagger(\{x, x\}) := h'(x) \quad \text{otherwise.}$$

Wherever h is differentiable, $h^\dagger$ is continuous.

Let $f(x) := 1000 \cdot x^n$. Ideally (absent roundoff) $FV = fv = fv = f^\dagger(\{1+i, 1\})$. But …
$i = 0.0000\,2465\,7\,53425$ is so tiny that $1+i$ rounds to $1+\imath$ for $\imath = 0.0000\,2465\,8$.
Then $fv = f^\dagger(\{1+\imath, 1\}) \approx f^\dagger(\{1+i, 1\}) = FV$ closer than it is approximated by fv .

To compute FV more accurately than fv , we used to use and could still use a …

Theorem: If h is an *Algebraic* function, $h^\dagger(\{x, y\})$ can be computed at roughly the cost of two computations of h without ever dividing a tiny $x - y$ into mostly cancelling $h(x) - h(y)$.
But f(x) is too costly when n is huge — about a dozen multiplications at $n = 365$.

How does the HP-12C approximate FV more accurately and quickly? Its Math. library includes

$$expm1(x) := x \cdot exp^\dagger(\{x, 0\}) \quad \text{and} \quad log1p(x) := x \cdot log^\dagger(\{1+x, 1\})$$

implemented accurately enough that it can compute $FV = 1000 \cdot expm1(n \cdot log1p(i))/i$ extra-precisely no matter how big n nor how small $i \neq 0$ may be. In 1982, 4.3 BSD Berkeley UNIX and, in 1984, the earliest Apple Macintoshes' Math. libraries included exp1m and log1p for the same reason.

For more about Divided Differences see my www.eecs.berkeley.edu/~wkahan/Math185/Derivative.pdf .

Other Occasionally Misleading Classical Formulas and their Cures:

Edges, Angles and Areas of Triangles: www.eecs.berkeley.edu/~wkahan/Triangle.pdf

Angles Between Sightlines to Stars …/Math128/angle.pdf

Angles Between Subspaces: …/Math128/NearestQ.pdf

Applications of Cross-Products in 3D: …/MathH110/Cross.pdf

Mean and Variance: …/Math128/MeanVar.pdf

Real Quadratic and Cubic Equations: …/Qdrtcs.pdf and …/Math128/Cubics.pdf
Avoid §5.6 of *Numerical Recipes* … by Press et al.

Differential Equations: …/Math128/Fl0Trik.pdf

Linear Matrix Equations $A \cdot x = b$: …/Math128/FailMode.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, Persistence and Revelation:

- Persistence of numerically naive and thus vulnerable formulas in the source-code of some programs, and in some published papers and textbooks.

- Revelation, after long use, that a widely trusted program has produced, for otherwise innocuous input data, results significantly more inaccurate than previously believed.

Hereunder come several of those Revelations:

• •

Accumulated over a few days, rounding errors caused a PATRIOT Anti-Missile Missile to miss an Iraqi SCUD that hit a barracks and killed many soldiers during the Gulf War.
Would a hit have prevented those casualties? Hitting the fuel tank might not have deflected the warhead.

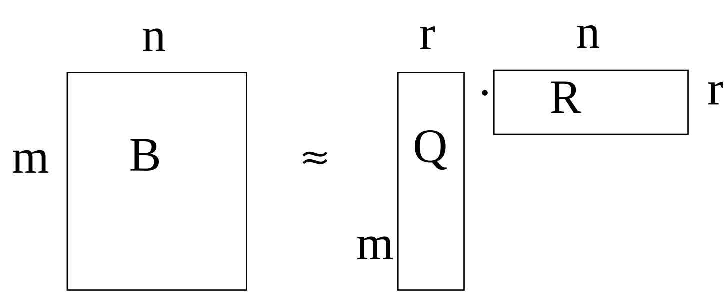
Long-running accumulation of roundoff aren't uncommon; *cf.* my …/Math128/Fl0Trik.pdf and …

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 that much over a weekend. Roundoff had accumulated over years. "Experts" called in took two weeks to find a commonplace bug, and then "fixed" it.

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



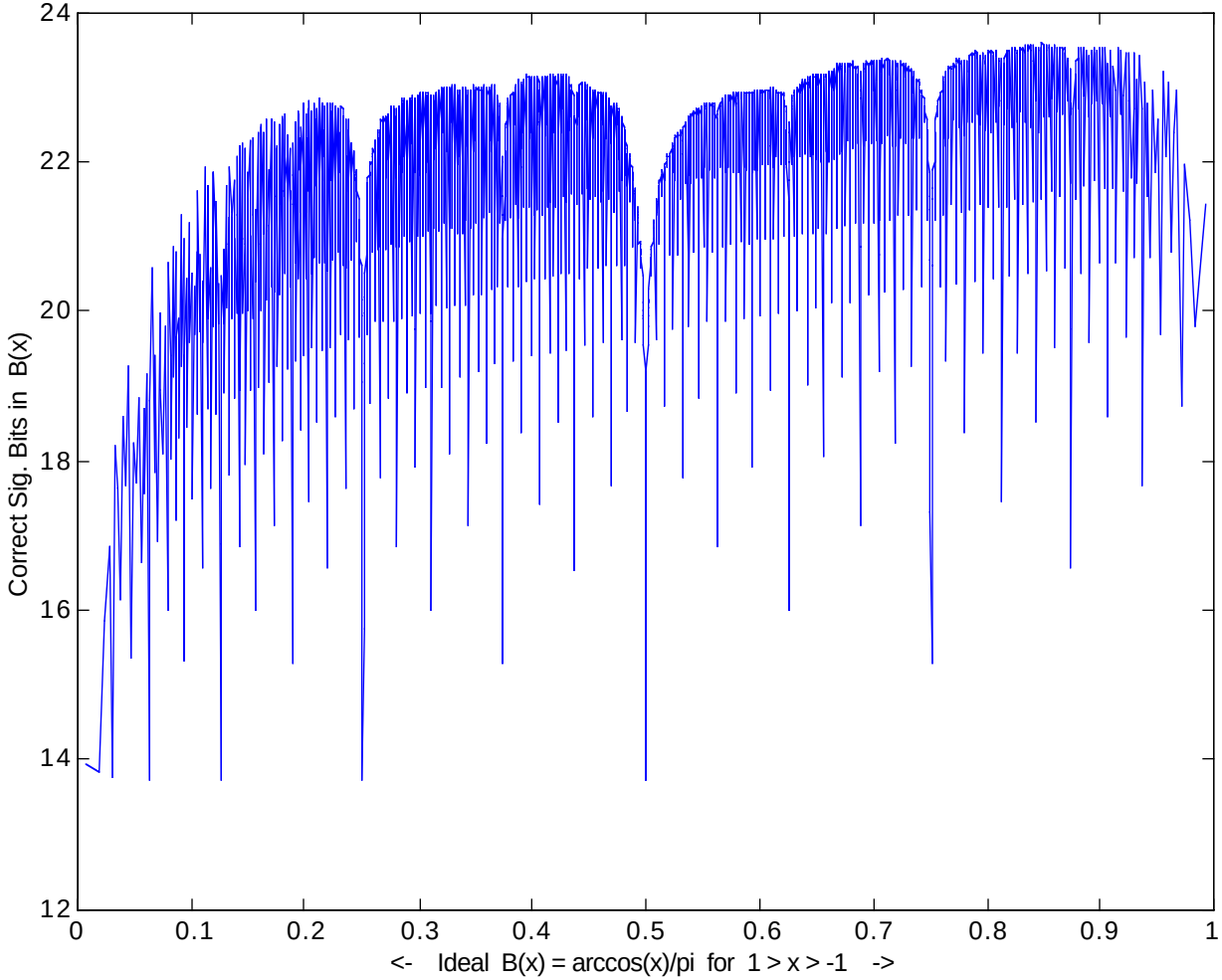
within $\pm\tau$.

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 so severely 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) small odd integer multiples of powers of 1/2. The smaller the integer, the wider and deeper the spike, down to near half the sig. bits lost. Intense testing missed such arguments x, though frequent in practice.

Roundoff-Induced Anomalies Evade Expert Searchers for Too Long:

- 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 `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.

Anomalies due to Over/Underflow can evade expert searchers 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 Causes Bugs in Numerical Software?

- Numerically naive algorithms may be chosen out of ignorance of better ones.

  #(Programmers Graduating) >> #(Takers of apt Numerical Analysis courses) , so
  Few programmers can cope with the vagaries of approximate arithmetic.

- Adequate tests can be too difficult to construct; later we'll see why. And, …
  If you don't know the right result, how can you tell whether a result is wrong?
  So, producers of numerical software need help to debug it;
  they need lots of assistance from a few of many users.

  But how much debugging of numerical software is included in,
  say, a chemist's job-description?

- Debugging is deterred when it costs more than the result is worth.

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

Our choices: Either …

- Change the programming ambience to render unexposed bugs in numerical software so rare as to be practically ignorable.

Or …

- 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 rare but innocuous data are highly likely to be inaccuracies, at other less rare data, barely bad enough to arouse suspicions.

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?

- The producers of software development systems don't know that such tools could be produced, and don't think there is a demand for them.

- The scientists and engineers who would benefit most from such tools are hardly aware of them, and consequently do not request them.

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. Almost all their schemes disregard the peculiarities of Floating-Point. What is it about Floating-Point that repels Computer Scientists?

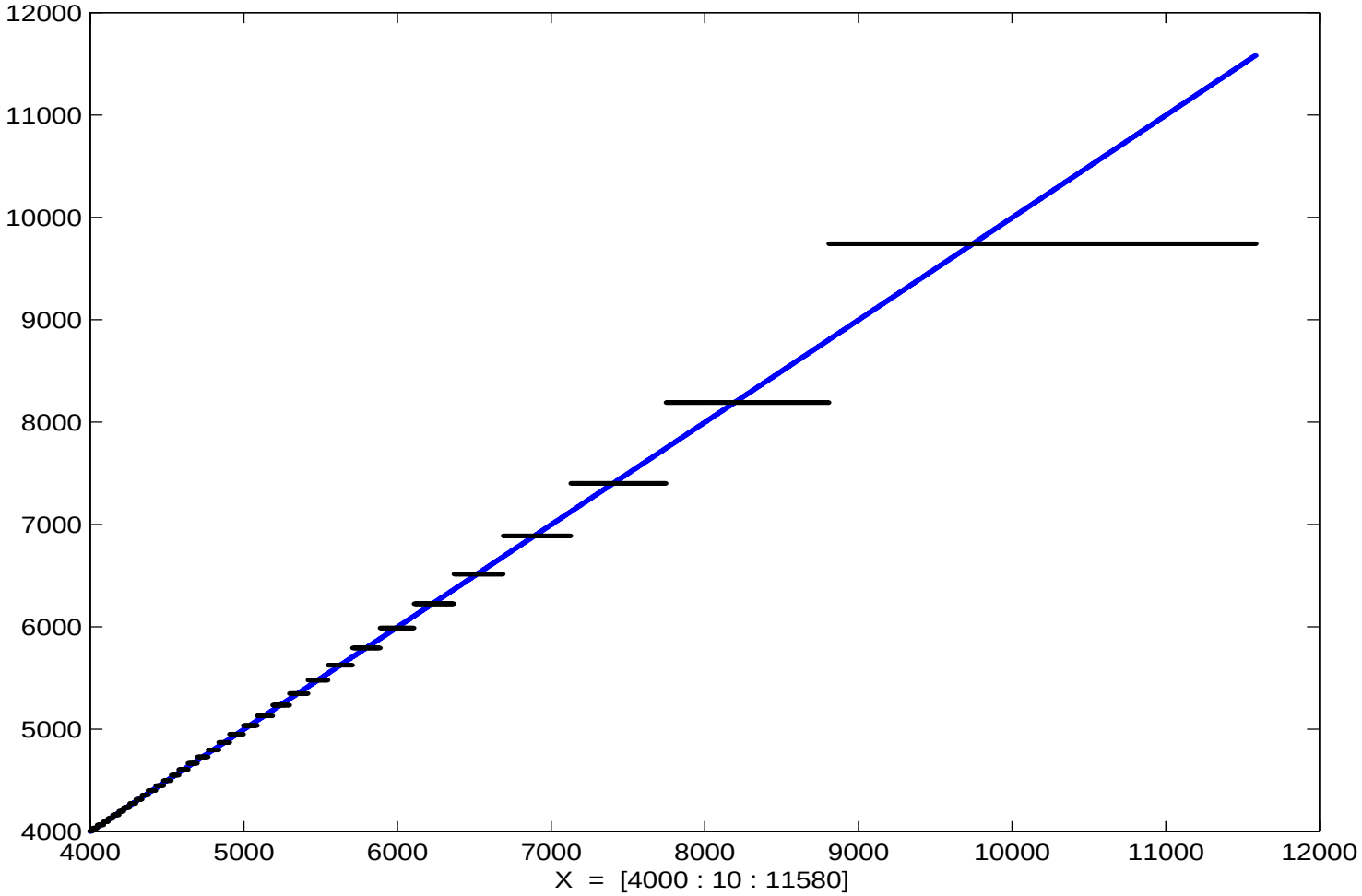
Floating-Point arithmetic usually approximates *Real* arithmetic closely, but not always.

- 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)) can approximate function $f(x) := g(h(x))$
  Utterly Wrongly over a large part of its domain.
  It happens rarely. Here is a simple didactic example, albeit contrived:

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

$f(x) = x$ *vs.* `G(H((x)) = (-log(exp(-x^-4)))^-1/4` rounded to 53 sig.bits



Only one of several rounding errors matters. Which? See pp. 24 - 25 of my posting …/MxMulEps.pdf.

What exposes a misjudgment due to rounding errors ?

Unlikely events:

- 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 ever.

- 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/Fl0Trik.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 do that ?

Arithmetic precision of `double`: 53 sig. bits ~ 16 sig.dec. $\varepsilon \approx 2^{-52}$
of `float`: 24 sig. bits ~ 7 sig.dec. $\varepsilon \approx 2^{-23}$

7 correct sig. dec. is more than adequate accuracy
for almost all computed results desired by scientists and engineers.
But the final results you see are not ***always*** accurate in all digits displayed.

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.

Summary so far:

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

Some numerical software, though programmed by experts, is too complicated for the programmer to debug thoroughly with the tools available. Inevitably, users will be the first to encounter some bugs, perhaps without noticing them.

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 and inadequate testing, or at least enhance the likelihood that those programs' numerical vulnerabilities will be exposed, preferably before too late?

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."

(Suspicions cause more trouble than anything else.)

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.

This 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

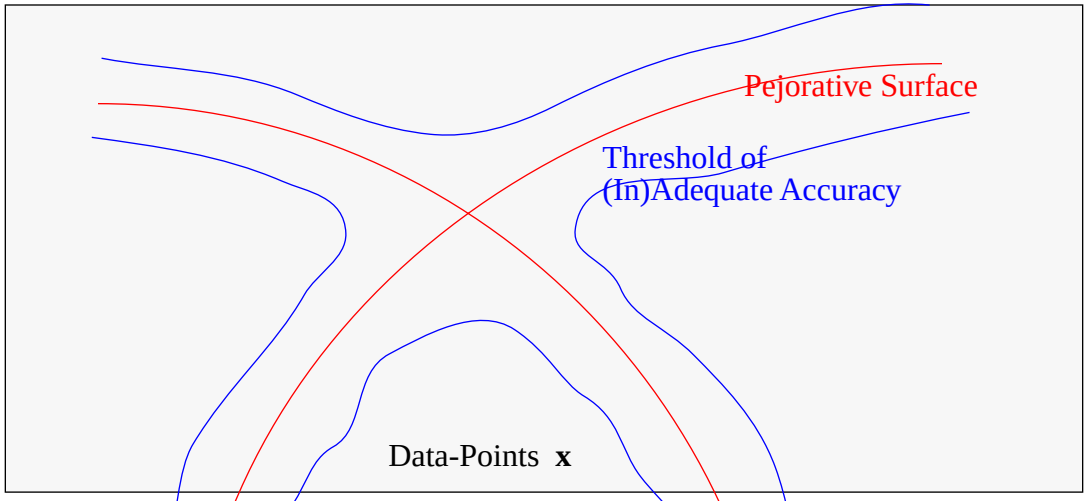
$$f` = \infty .$$

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 $\|f`(\mathbf{x})\| \to \infty$ as $\mathbf{x} \to$ (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(X) is intended to compute $f(\mathbf{x})$ but actually F(X) = f(X, **r**) in which column **r** represents the rounding errors in F and $f(\mathbf{x}, \mathbf{o}) = f(\mathbf{x})$. 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 X we have equals the data **x** we wish we had.

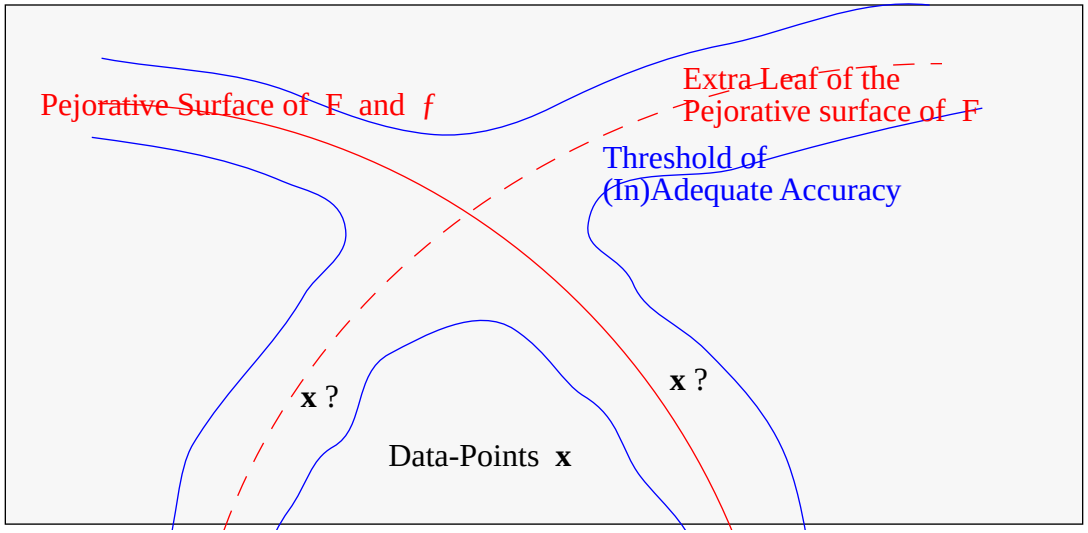
Let $f_r(\mathbf{x}) := \partial f(\mathbf{x}, \mathbf{r})/\partial \mathbf{r}\big|_{\mathbf{r}=\mathbf{o}}$. Because ρ is so tiny, program F(**x**) actually computes
$f(\mathbf{x}, \mathbf{r}) \approx f(\mathbf{x}, \mathbf{o}) + f_r(\mathbf{x})\cdot\mathbf{r} = f(\mathbf{x}) + f_r(\mathbf{x})\cdot\mathbf{r}$, 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 **r** is tiny only if derivative f_r is gargantuan, which can happen only if **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 **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 **x** too near that Extra Leaf of Pejorative Surface the program F(**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(**x**) is accurate enough only at data **x** far enough from all pejorative surfaces.

An opportunity to discover whether the program's pejorative surface has an Extra Leaf arises when F(**x**) is inaccurate enough to arouse suspicion. Does F(**x**) deserve its inaccuracy because **x** is "Ill-Conditioned" — too close to the Pejorative Surface of *f* ? Or is the inaccuracy undeserved because innocuous data **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?

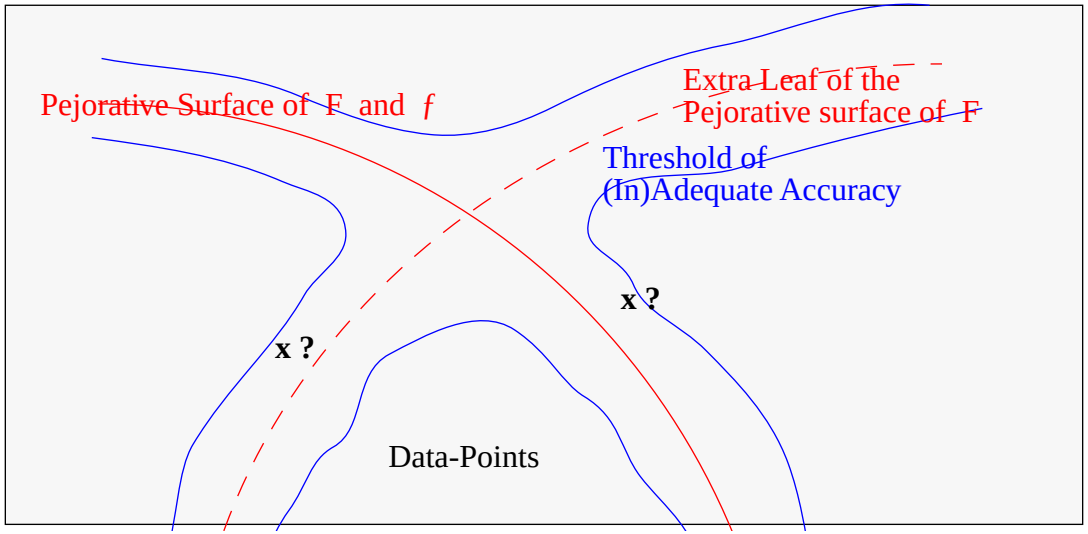
We must find out lest later we accept unwittingly an utterly inaccurate result at some other innocuous data much closer to the program's Extra Leaf of its Pejorative Surface, of whose existence we had chosen to remain unaware.

Two choices present themselves:

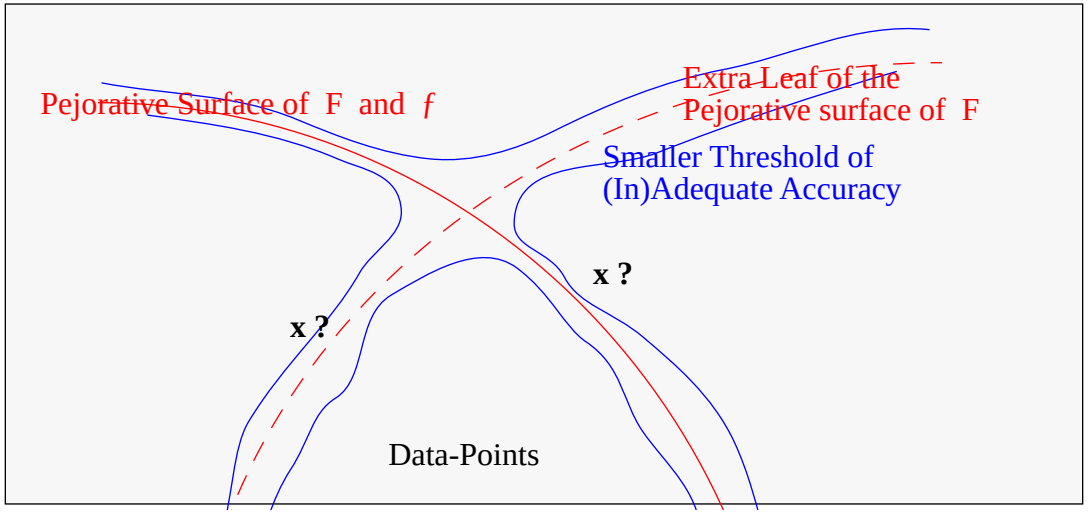
- Enhance the likelihood of these difficult questions' resolution by supplying tools to reduce by orders of magnitude the cost in talent and time to resolve them. OR …
- Reduce by orders of magnitude the likelihood that these questions will arise or matter.

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* suffices, as does COBOL's *Comp* format, both with at least 33 sig.dec. of precision.

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



Higher Precision $\Rightarrow$ Smaller ρ $\Rightarrow$ 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 **x** comes to a Pejorative Surface.

Some Examples:

- Nearly redundant Least-Squares problems.
- Nearly double zeros of polynomials, like the quadratic equation.
- 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 $FV(n, i) := -PMT\cdot((1 + i)^n - 1)/i$ 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 := $-PMT\cdot((1 + i)^n - 1)/((1 + i) - 1)$. **Honor Parentheses!**

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 .

Does that "Ample experience" justify small amendments to programming languages?

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 forseeable 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 **x** at which F(**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(**x**) . Data **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(**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 computations mentioned in the lengthy list on p. 22 of …/NeeDebug.pdf or in this document. For example, Margaret's *fv* and fv :

n = 365 ; i = 9/(1000·n) ; -PMT = 1000

Formulas:	Classical *fv*	Better fv
Rounded to …	$1000 \cdot \frac{(1+i)^n - 1}{i}$	$1000 \cdot \frac{(1+i)^n - 1}{(1+i) - 1}$
… Nearest	366,921.88	366,642.50
… Zero	365,142.75	366,635.91
… +∞	366,926.72	366,647.34
… –∞	365,142.75	366,635.91

Without having to know the correct value 366,642.90 , we can infer roughly the uncertainty due to roundoff from the spread among the four (re)computed values.

Provided only the last rounding error in each of the Math. library's functions is altered by Redirected Rounding, it affects only the last sig.dec. of FV = 1000·expm1(n·log1p(i))/i .

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.

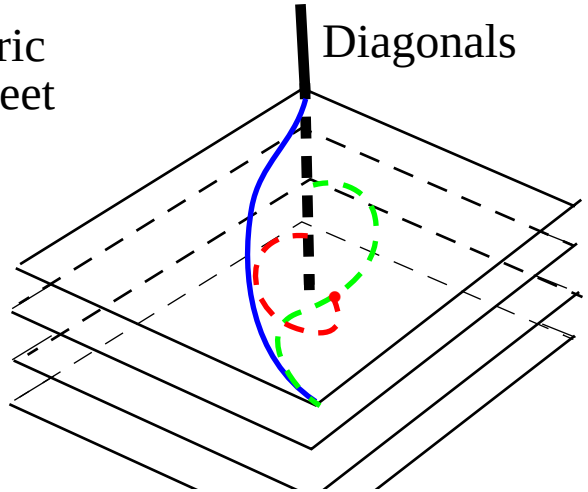
A Tool for (Slower) Recomputation with Higher Precision

This tool would ease the task of running two programs F(**x**) and $\mathbb{F}$(**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 **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.



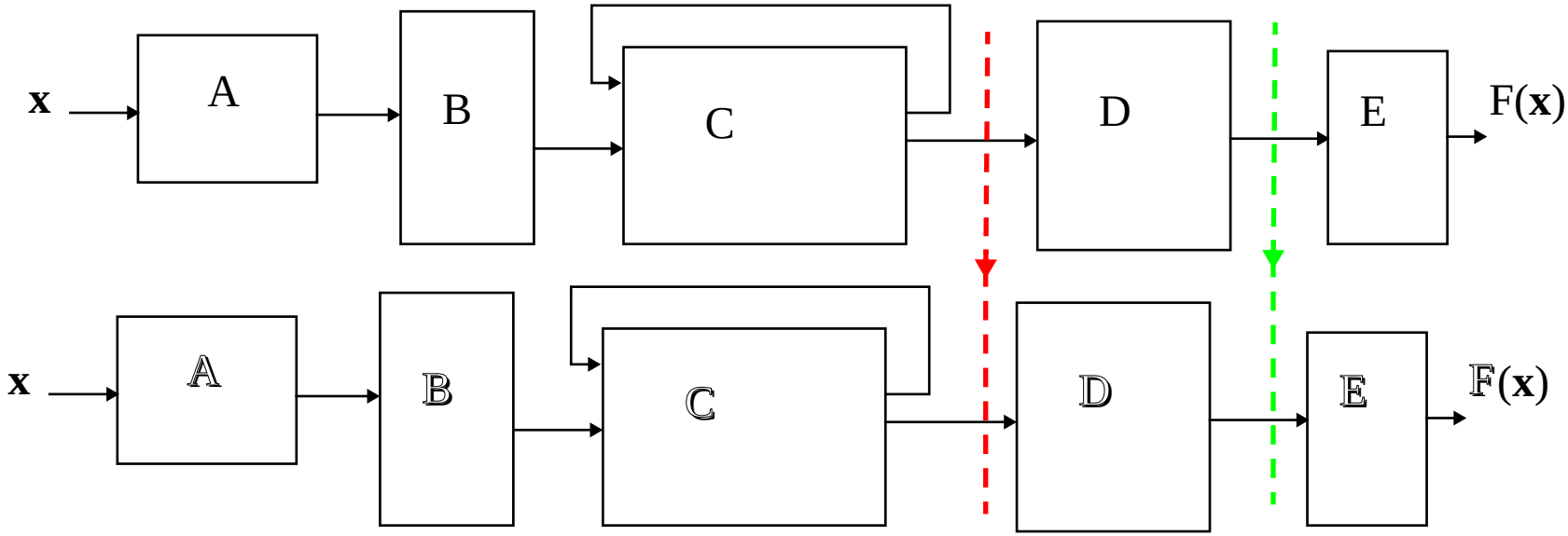
Paths followed during a program's computation of eigenvalues with …

… no rounding errors

… the usual rounding errors

… and altered rounding errors

Instead of running F and 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(**x**) up to a chosen stage and then copy the values of all the variables alive at that stage exactly to their counterparts in 𝔽 ; then run 𝔽 to its end to see how much its result disagrees with F(**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(x) and 𝔽(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 some programs. 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, and its *Affine* variant
- Significance Arithmetic (used by *Mathematica* among others)
- Repeated runs with Random Rounding (cf. Vignes' *CESTAC, CADNA*)

cf. my …/Improber.pdf .

- Searches for Singularities by Theorem Provers & Computerized Algebra
- … ?

So far, all those schemes lack at least two of these four 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 false alarms 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.

<4>: If supported adequately by a compiler and debugger, can be used easily even if source-code is unavailable for some of the program's object modules.

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 for 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 occurence 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.

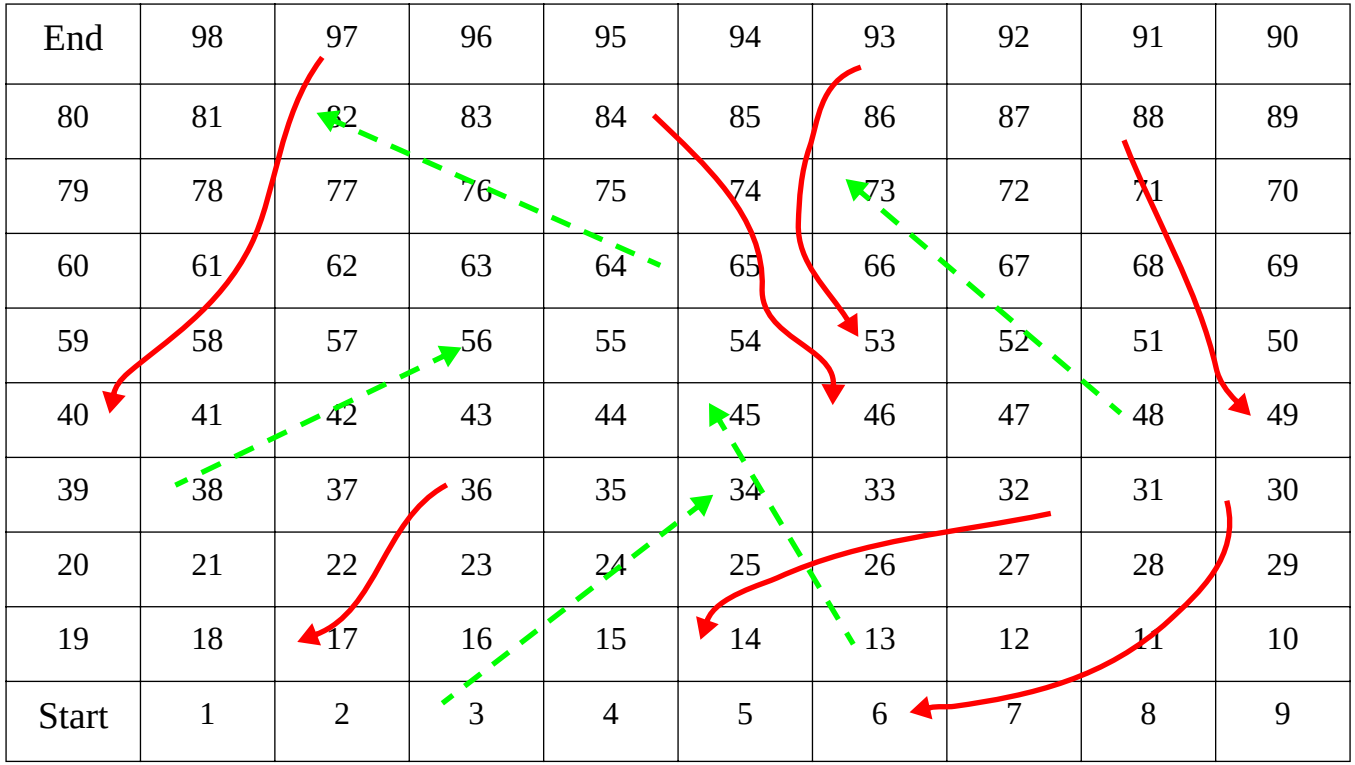
See `<www.cs.virginia.edu/~weimer>`.

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



… 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 3 hrs. in 1997
- The Ariane 5 satellite-carrying rocket blown up in 1996
- Air France #447 (AirBus 330) crashed in 2009

Let's look into the last example …

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

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***.

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?

<…/Boulder.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 deemed Errors ?

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 *f*lags 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.

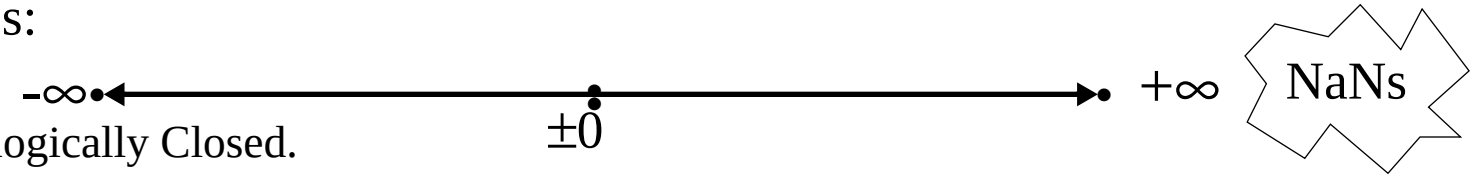
These, explained in <…/Boulder.pdf>, are intended for Floating-Point computations.

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:

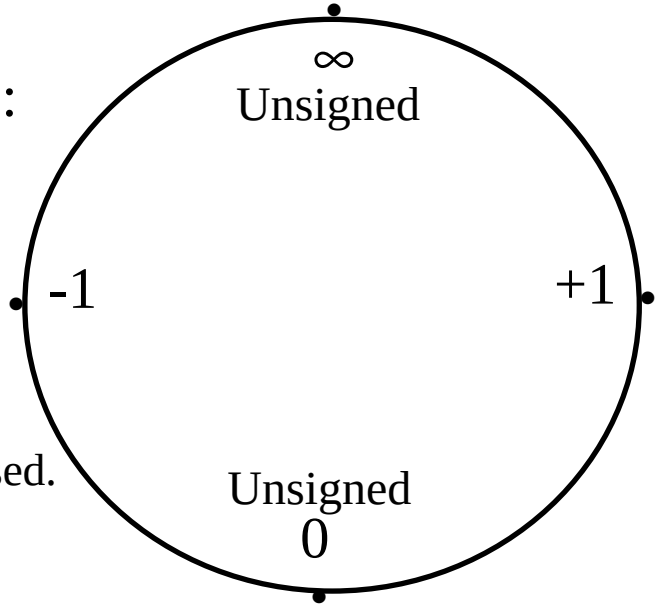


… is Topologically Closed.

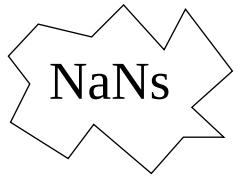
(A **NaN** is
Not **a** **N**umber)

Projective Closure:

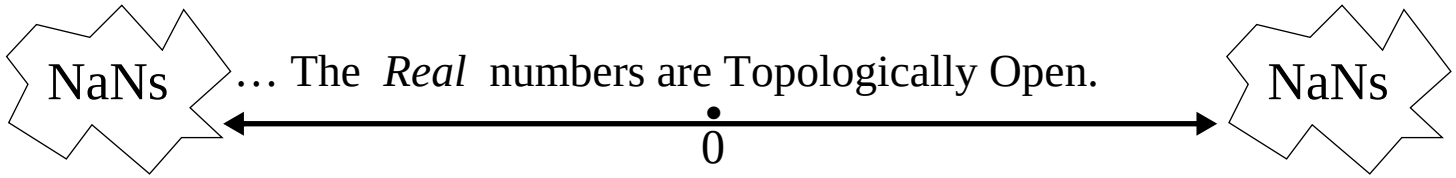
(Stereographic Projection, like the Riemann Sphere of the Complex Plane)



… is Topologically Closed.



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



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·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 …/NeeDebug .

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 ±∞
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*, 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) *flag*s 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 *flag*s 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 *flag*s' 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 !**

But programming numerical software without *flags* is worse.

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 *flag*s.

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 *flag*s' 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.

The essential advantage of human civilization is that we can benefit from someone's experience without having to relive it.

"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.