Why can I Debug some Numerical Programs that You Can't ?

Why should we care? What should we do?

Presented in 23 min. on 30 March 2007 to the
"Stanford 50" celebration of Stanford University's
50th Anniversary of George Forsythe's founding
of Stanford's Computer Science Dept., and also
in anticipation of Prof. Gene H. Golub's 75th birthday
(which, alas, he did not quite reach):
"The State and Future Directions of Computational Mathematics and Numerical Computing"

This is posted at `<www.cs.berkeley.edu/~wkahan/`**`Stnfrd50.pdf`**`>`

Abstract: The future promises teraflops and terabytes in your laptop, petaflops and petabytes in your supercomputer, and the inability to debug numerical programs on either. Why can't they be debugged? What should we do instead?

Though this presentation does include some mathematical symbols and Greek letters, it contains no Mathematics.

Reminiscences

I owe a lot to Gene Golub and to George Forsythe.

To Gene:

1957	Hospitality at University of Illinois @ Champaign-Urbana
1959	Shared an office at Cambridge University
1964	Joint paper on good Singular Value Decomposition
1966	Arranged for me to visit Stanford, where …

To George:

1966	Encouraged my hobby: Proofs about Floating-Point Arithmetic

I would rather have spent my time solving differential equations.

Part 1 of this presentation is to honor George. **Part 2** is for Gene.

Part 1:

"Whoever forgets the past is doomed to repeat its mistakes."

(Not what George Santayana said.)

Do You Remember …?

Do You Remember?

When I started programming computers in 1953, and for over a decade after, the consensus among almost all Numerical practitioners was that …

“ Floating-Point Error-Analysis is Hopelessly Intractable. ”

Do you remember John Rice’s *Polyalgorithms*? Like the *Alka-Seltzer* advts.:

“*Try it! You’ll like it !*” And if you don’t, try something else.

e.g., For non-symmetric matrix eigenvalues, try the Power Method, or the Leverrier-Souriau-Frame-Faddeev Method, or Danilewski’s, or … .
(And if one or two seemed to work they often gave excessively inaccurate results.)

The first signs that floating-point error-analysis might be feasible for some huge calculations, leading to what came to be called “Backward Error-Analysis”:

1949 A. Turing, Teddington.
1954 W. Givens, Oak Ridge, later at Argonne Nat’l Labs.
1957 F. Bauer, Munich; J. Wilkinson, Teddington; W.K., Toronto

1958-60 W.K. visits J.H.W often to share new (?) results; J.H.W. asks “And have you this?

We are headed back to the past:

Thesis: Too much of the Numerical Software developed for
Scientific and Engineering Computation is
now Impossible to Debug.

Our community needs better support for the
diagnosis of numerical embarrassment,
especially if due to roundoff and
thus unnoticed until too late
if ever.

Hardware conforming to IEEE Standard 754 (1985) for Binary Floating-Point supports better diagnostic tools than you get now from programming languages (except perhaps from a few implementations of C99) and program-development environments. That hardware support is atrophying for lack of exercise.

Use it or lose it.

Realistic examples supporting the **Thesis** are too complex for C.S. students.
E.g., see `http://www.cs.berkeley.edu/~wkahan/Math128/GnSymEig.pdf`.
Instead I must resort to artificial examples created for didactic purposes, like …

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 desires 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 be adequate.

How do we discover which software version (if any) is right?
And what is wrong with the others? 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 to identify possibly aberrant modules that You Can't ?

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.) 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, …,

which let me inject control word changes that then over-ride default rounding modes with no changes to the program modules whose arithmetic is so altered.

For details see §11 of `.../Mindless.pdf` on my web page.

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
θ, Θ	Azimuth	Right Ascension	Longitude
ϕ, Φ	Elevation	Declination	Latitude

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

Two stars whose coordinates are (θ, ϕ) and (Θ, Φ) subtend an angle ψ at the observer's eye. This ψ is a function $\psi(\theta-\Theta, \phi, \Phi)$ that depends upon θ and Θ only through their difference $|\theta-\Theta| \bmod 2\pi$. Three implementations of this function ψ will be compared; they are called u, v and w. Of millions of tests, here are the few that aroused suspicion:

$\theta-\Theta$:	0.00123456784	0.000244140625	0.000244140625	1.92608738	2.58913445	3.14160085
ϕ :	0.300587952	0.000244140625	0.785398185	-1.57023454	1.57074428	1.10034931
Φ :	0.299516767	0.000244140654	0.785398245	-1.57079506	-1.56994033	-1.09930503
$\psi \approx$ u :	0.0015*8221229*	*0.0*	0.000*345266977*	0.0005*98019978*	3.1408*2050*	3.14055*681*
$\psi \approx$ v :	0.001593248*68*	0.00024414061*0*	0.000172633489	0.00056223187*1*	3.140*61618*	3.140*61618*
$\psi \approx$ w :	0.001593248*68*	0.00024414061*0*	0.000172633489	0.00056223187*1*	3.140*78044*	3.1405*4847*

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 were 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 roundings (to nearest) to be changed without recompilation of the subprograms. Here are some results:

$\theta - \Theta$:	0.000244140625			2.58913445		
ϕ :	0.000244140625			1.57074428		
Φ :	0.000244140654			-1.56994033		
$\psi \approx$ u :	0.000598019920	NaN arccos(>1)	0.000598019920	3.14061594	3.14067936	3.14082050
$\psi \approx$ v :	0.000244140581	0.000244140683	0.000244140581	3.14039660	3.14159274	3.14039660
$\psi \approx$ 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 `.../Mindless.pdf` for the formulas. If not for roundoff all three 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 agressively.

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 `http://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 program-development environments, tools that employ high-precision floating-point and interval arithmetic combined (they are not helpful enough by themselves), see §14 of my `http://www.cs.berkeley.edu/~wkahan/Mindless.pdf` .

For better exception-handling than provided by current programming languages other than C99 and perhaps Fortran 2003, see my …`/Grail.pdf` and …`/ARITH_17U.pdf` . Floating-point exception-handling is a crucially important story for another day.

What’s Holding Up Progress towards Numerical Reliability?

Compared with 50 years ago, today’s computers run millions of times faster, and hold millions of times more memory. More important, now floating-point computation is so much *cheaper* than it was then as to be almost free. So it is used now mostly for games and entertainment. (*E.g.*: the IBM-Sony-Toshiba *Cell* computer.)

The Tragedy of the Commons: Every free good is destined for abuse.

E.g.: Spammers and phishers abuse e-mail because it is free.

Now only a tiny fraction of floating-point computations
are worth the cost of ascertaining their validity,
much less the cost of correcting them if found wrong.

Unintended numerical anomalies in computer games become *Features* celebrated in *BLOGS*.

Gresham’s Law:

“*Bad* money (debased or counterfeit) drives out the *Good*” (from circulation).

Sir Thomas Gresham (1519-1579)

Gresham’s Law for Computing:

The *Fast* drives out the *Slow* even if the *Fast* is Wrong.

PART 2: A Bad Example for a Bad Policy:

MATLAB 7 now “supports” floating-point arithmetic with 4-byte-wide `single`-precision variables as well as the previously supported 8-byte-wide `double`-precision variables. But MATLAB evaluates any expression that mixes `single`- with `double`-precision variables entirely in `single`-precision arithmetic because it goes faster this way on the most popular architectures. Actually, …

This policy can slow down large-scale computations.

For instance, consider the discretization of an elliptic boundary-value problem

$$\mathrm{Div}\big(p(\mathbf{x})\bullet\mathbf{Grad}\ U(\mathbf{x})\big) + q(\mathbf{x}, U(\mathbf{x}))\cdot U(\mathbf{x}) = b(\mathbf{x}, U(\mathbf{x}))\ .$$

When discretized this boundary-value problem turns into a system of linearized equations

$$(\mathrm{A} + \mathrm{Diag}(\mathbf{q}))\cdot\mathbf{u} = \mathbf{b}\ .$$

Here matrix A represents the discretization of $\mathrm{Div}(p(\mathbf{x})\bullet\mathbf{Grad}\ \ldots)$. For many reasons not necessarily spawned by roundoff, the solution **u** has to be computed by an iteration, and that *always* entails the computation of a residual

$$\mathbf{r} := \mathbf{b} - (\mathrm{A} + \mathrm{Diag}(\mathbf{q}))\cdot\mathbf{u}$$

The final accuracy of the computed **u** is limited by the accuracy with which the residual **r** can be computed. The accuracy of **u** , which is typically a potential, has to be sufficient to support differencing to estimate the *Gradient* **Grad** $U(\mathbf{x})$, a field strength, without too much loss of accuracy to cancellation.

Residual $\mathbf{r} := \mathbf{b} - (A + \mathrm{Diag}(\mathbf{q}))\cdot\mathbf{u}$

If all data, variables and arithmetic have the same precision, and if **r** is computed from literally the foregoing formula, then **u** cannot be computed more accurately than to about half as many sig. digits as the arithmetic carries. If everything is `double`-precision (≈16 sig. dec), that is accurately enough.

But the speed of computation is limited mostly by the speed at which data and variables travel through the memory system. Arithmetic (other than division) is almost instantaneous by comparison.

`single`-precision moves through the memory system twice as fast as `double`.

But losing half of `single`-precision's digits (≈7 sig. dec.) leaves too few for **Grad**.

If all *array* data and variables, except possibly the diagonal of A , are declared `single`, but all arithmetic is performed in `double` before being rounded off to be assigned to `single`-precision elements of an array, **u** can be computed to almost 6 sig. dec. **Kernighan-Ritchie C used to do this by default**. Not Java …

When arithmetic more precise than `single` data is unavailable or too slow the programmer must resort to *trickery* to achieve 6 sig. dec. accuracy.
See `http://www.cs.berkeley.edu/~wkahan/Math128/FloTrik.pdf` .

How likely is the programmer to know about this trickery? Should he have to?
I think it best that programmers NOT have to know about numerical trickery.

“In every army large enough there is always someone who does not get the message, or gets it wrong, or forgets it.”

Whatever students learn gets forgotten if not exercised soon enough afterwards.

Applied Math. students at Berkeley have to learn some Numerical Analysis, though probably not the aforementioned trickery. CS grads from Berkeley, Stanford and most other places don’t have to know more about floating-point than an hour’s worth in a programming language course. Apparently …

Numerical Analysis has become a sliver
under the fingernail of Computer Science.

No Numerical Analysis appears in *Educational Testing Service*’s COMPUTER SCIENCE Major Field Test (4CMF) of students’ mastery of a CS curriculum.

Therefore we must (re)design computer architectures, languages and program-development environments to diminish rather than enlarge the capture cross-section for numerical misadventure of programs written by clever but numerically naive programmers.

See my innumerable web postings on the subject:
www.cs.berkeley.edu/~wkahan/MxMulEps.pdf, …/Mindless.pdf,
…/JAVAhurt.pdf, …/MktgMath.pdf, …/MathH110/Cross.pdf, …/…

Epilogue for the Younger Reader

The foregoing *Jeremiad* exhorts you, the reader, to do something:– to read, research, reflect, and react. Can you see what needs doing? Why? Who should do it? Who'll pay?

If you see what needs doing, will you do your part? Will you exert your influence?

What needs doing will take several years. I am willing to help, but I cannot lead the charge.

According to life-insurance premiums, actuaries seem to have estimated that the death rate for non-smokers of my age is roughly 1/2 % per month and increasing rapidly with age. How likely am I to be still active when what needs doing has been done? Not very.

You'll have to do it.

Prof. W. Kahan