Back to the Future of Undebuggable Floating-Point Computation in Science and Engineering

by Prof. W. Kahan
Math. Dept., and Computer Science Dept.,
University of California at Berkeley

For presentation to BASCD08
30 March 2008

Abstract: When I began to program an electronic computer in 1953, von Neumann was still disparaging floating-point computation, which was generally deemed impervious to error-analysis. Occasional anomalous results were expected. Often they were attributed wrongly to "Ill-Condition". Putting one's data through several numerical methods some of whose results might agree was a prudent policy. Those days are back. Their challenges will be illustrated by a program like some used by structural engineers for forty years. To cope, we need debugging aids like those in §14 of my web page's <…/Mindless.pdf>. Help can come only from the designers of hardware, compilers and software development systems after they are persuaded that the demand for such aids is commercially significant.

This document will be posted at <www.eecs.berkeley.edu/~wkahan/BASCD08K.pdf>

Reminiscences from 1953

- I programmed an electronic computer in a language cruder than Assembly Language.

- John von Neumann was still alive and still disparaged floating-point arithmetic.
  His fixed-point error-analysis of linear equation solving published in 1948 was still regarded as authoritative despite a gaffe that made it grossly pessimistic.

- Alan M. Turing was still alive. His more modern fixed-point error-analysis of linear equation solving published in 1949 had implications not yet appreciated.

- Alston H. Householder's book *Principles of Numerical Analysis* was published:
  An authoritative but terse text. No mention of "Floating-Point Arithmetic" in it.
  It included a few fixed-point error-analyses and cited others.

- Floating-point arithmetic was generally deemed unreliable because it was believed to be not susceptible to error-analysis. Also, implemented in software, it was slow.

- Engineers and scientists were using it anyway.

Reminiscences from 1957

- In early summer I met Gene Golub at the U. of Illinois in Champaign-Urbana.
  In Sept. at Wayne State U. in Detroit I presented some results from my thesis on
  Successive Over-Relaxation in 15 min. Met R.S. Varga, D.M. Young, …
  Young: "Have you actually *proved* all that?" Varga: "Show me!".

- I wrote floating-point programs to help G.E. (Canada) design transformers and motors.

- *Backward Error-Analysis* was found applicable to some floating-point computations …
  … by J.H. Wilkinson in Teddington, England, inspired by Turing's experience
  … by myself in Toronto, Canada, inspired by a comment in Turing's 1949 paper
  … by F.L. Bauer in Munich (I have a copy of his notes in German)
  all independently. Wilkinson's publications in 1959 were the first and most lucid.

- Mathematicians generally still deemed floating-point arithmetic unreliable, but now
  it was in the hardware of the IBM 650 and 704, among many other machines.

Reminiscences from 1958 - 1960

- Enjoyed a post-doc. as a "graduate student" at Cambridge.
  Taught numerical analysis including floating-point error-analysis. (… Mike Powell)
  Floating-point error-analyses with David Wheeler (… μ-coded EDSAC 2)
  Visited Wilkinson: "Yes, I have that." (Blows off dust) "Do you have this?"

- "Supervised" by J.C.P. Miller, who taught me innumerable computing tricks.

- 1959: With Sheila & Gene Golub, went to a conference in Paris. Met Fritz Bauer, …

- Numerical software using floating-point was still generally deemed unreliable,
  and deservedly so all too often.
  For every numerical problem there were numerous algorithms published, and
  innumerable programs promulgated.

  e.g.: for the matrix eigenproblem there were methods by Hyman, Wielandt,
  Danilewsky, Frame-Faddeev-Souriau-Leverrier, Lanczos, Givens, …
  none of which worked reliably until John Francis' QR iteration.

Reminiscences from the Early 1960s

- I returned to the Univ. of Toronto as a professor teaching N.A., C.S., Math. etc.
  In order to solve differential equations my way I first had to …
  Improve IBM's wretched library of elementary functions for the 7090/7094,
  Build up a library of reliable quadratures, solvers of (non)linear equations, …
  Put humane exception-handling, gradual underflow, retrospective diagnostics,
  and even graph-oriented data-structures into Fortran II and IBSYS.

- I became an active contributor to IBM's SHARE and
  joined Bruno Witte's committee to standardize floating-point software (Hopeless!)

Meanwhile …

To cope with unreliable numerical software, John Rice at Purdue U. recommended

Polyalgorithms :

If the first numerical program you choose turns your data into a poor result (how would you know?) or none, try another program, maybe another, until a consensus emerges.

Starting in the mid-1960s, comparatively reliable numerical software began to proliferate, guided in most cases by a growing appreciation of error analysis which supplied families of proven stable sub-procedures, like orthogonal matrix multiplications.

e,g,: In 1964 Gene Golub and I wrote up the first fast Singular-Value Decomposition that was proven at least about as accurate as the data deserved.

The situation now is different.

We are being thrown back to the state of floating-point software as it was five decades ago.

Our energies are so consumed by attempts to exploit concurrency as fully as possible that

we are obliged to devise and use programs whose numerical stability has not been established as thoroughly as it should be, and

many of them can’t be debugged with the tools now available.

Why can't we debug?

Too much is happening, and too much of it is invisible.

Too much is happening:

- Too many gigaflops to single-step through them, even aided by break-points.
- Too many gigabytes to survey, even aided by data-mining technology.
- Computation is too cheap. Most computations are not worth the cost of assessing their validity if doing so will require a rare and expensive person with a Ph.D. to spend an unpredictable number of weeks to reach quite possibly no useful conclusion.
- Other tasks running on the same aggregation of hardware as my program may cause reconfigurations that change what my program does when rerun.

"You can't step twice into the same river." (Heraclitus, near 513 B.C.E.)

Why can't we debug?

Too much is happening, and too much of it is invisible.

Too much is invisible:

- What else is the computer doing to my variables while executing the code I'm reading?
- If an exception occurred, whence came the program's control to where I'm reading?
  Set_jump/Long_jump ? Try/Catch/Finally? ON ERROR GOTO … ?
- Rounding errors have no names; otherwise they would overwhelm a program's text.
  So roundoff is invisible to a programmer except in the mind's eye.
- Well-intentioned but overly aggressive compiler optimization may change the execution of my program to differ subtly but significantly from the text that I am reading.

Now let's concentrate upon how to debug roundoff.

- Rounding errors have no names; otherwise they would overwhelm a program's text. So roundoff is invisible to a programmer except in the mind's eye.

My web-page posting *How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?* at `<www.cs.berkeley.edu/~wkahan/Mindless.pdf>` explains why no foolproof way exists to assess nor to remedy the effects of roundoff.

"If you have to resort to roundoff you put yourself in a state of sin." (Derrick Lehmer)

Even so, that posting advocates affordable ways to …

- Reduce the incidence of roundoff-induced embarrassment in sofware written by people who are clever at what they have studied but naive about roundoff.
- Assuage or strengthen suspicions about computed results with unknown accuracy.
- Localize short stretches of a program that may be hypersensitive to roundoff.
- Prioritize the scrutiny of potentially maleficent software modules.

But that document's examples, chosen for didactic effect, may seem unrealistic to the people who have to be persuaded to implement and/or adopt the schemes I advocate.

A Case Study:

Frequencies and Modes of Vibration of Undamped Elastic Structures

These give rise to Generalized Symmetric Definite Eigenproblems: $A \cdot x = \lambda \cdot H \cdot x$

Given symmetric matrices A and H, with positive definite H, compute all modes (eigenvectors) x and their eigenvalues λ that determine the vibrational frequencies.

Circumstances arise when this problem must be solved iteratively:

A convergent sequence of approximations E_1, E_2, E_3, … to the matrix of eigenvectors makes $E_k^T \cdot H \cdot E_k \to I$ and $E_k^T \cdot A \cdot E_k \to \Lambda$, a diagonal matrix of eigenvalues.

A program that performs the iteration and an explanation of it are posted on my web page at `<www.cs.berkeley.edu/~wkahan/Math128/GnSymEig.pdf>` .

The algorithm is a sequence of Jacobi-like congruences that simultaneously annihilate pairs of off-diagonal elements in A and in H.

Structural engineers have used similar programs for about four decades. As usual, …

- They did not know whether it would always work. I have proved that it does.

- Assessing its accuracy is, in general, a difficult task they could not afford.
  I have found that an obvious implementation can produce very wrong results for otherwise innocuous data. I have also corrected this flaw in my program.

How were wrong results first recognized?
They weren't. They were just suspected.

How were wrong results confirmed and their cause located?
I used one of the techniques discussed in Mindless.pdf, namely four recomputations with redirected roundings using just the data that led to suspicious results.

Why could I do that but you probably can't?
I have an old version of MATLAB 3.5 running on an old Intel 386/486 whose system and compiler I have tweaked for the purpose. You should be jealous,