

# The Top of a Wish-List for the Integration of Hardware Floating-Point Computation into Computerized Algebra Systems



Prof. W. Kahan  
Math. Dept., and Computer Science Dept.,  
University of California  
Berkeley CA 94720-1776






For presentation to ISSAC 2007  
30 July 2007  
University of Waterloo, Canada

**Abstract:** Numerical software for Scientific and Engineering Computation using fast floating-point hardware is becoming impossible to debug when afflicted by data-dependent hypersensitivity to roundoff. At the same time, the experience that would help programmers avoid that hypersensitivity is disappearing, replaced all too often by mere superstition. Neither Interval Arithmetic nor arbitrarily high precision but slow arithmetic by themselves can help enough. What is needed is known but not by enough programmers to create a market worth entering by the vendors of program development systems.

This document will be posted at [www.eecs.berkeley.edu/~wkahan/ISSAC\\_07](http://www.eecs.berkeley.edu/~wkahan/ISSAC_07)

## My Wish List:



- Diagnostic Tools to help debug roundoff-induced anomalies in computations:  
Which subprograms are above suspicion, which not?  
Which statement's rounding error spawned the fatal departure from accuracy? 
- Better control of variable precision, especially if arbitrarily high. 
- Truly linear algebra and geometry instead of lists and arrays. (cf. LINALG system) 
- Integration of *Interval Arithmetic* with geometry and with arbitrarily high precision.
- Help to locate singularities of sufficiently simple kinds. 
- Appeal to the *Monodromy Theorem* to help **correctly** simplify complex arithmetic expression involving principal branches of multi-valued algebraic functions. 
- ...

But there is time now only for the top of the list.

## **We urgently need ...**

- Diagnostic Tools to help debug roundoff-induced anomalies in computations:
  - Which subprograms are above suspicion, which not?
  - Which statement's rounding error spawned the fatal departure from accuracy?

## **... and ...**

- Better ways to lower such anomalies' incidence in programs inadequately debugged
  - ( Has any large program ever been fully debugged ? )

## **How Harmful are Anomalies Induced by Roundoff ? We cannot know.**

Such anomalies are almost always misdiagnosed if not shrugged off as too costly to find.

e.g.: PATRIOT anti-missile missile fails to stop SCUD from falling upon barracks.

The failure to hit the SCUD was ultimately traced to roundoff-induced drift.

(Even if a hit had been scored, it would not have stopped the warhead.)

So there are exceptions.



## Why are anomalies due to roundoff almost never diagnosed correctly?

The biggest change in computers over the past few decades has **not** been ...

... millions-fold increase in speed, nor ...

... millions-fold increase in memory capacity, ...

... but their decrease in price. Now almost everybody can afford one or more, and most computers are cheap enough to be left idle most of the time.

Floating-point computation has become so cheap that most of it is used for entertainment.

**Most computed results are not worth the cost of hiring a Ph.D. to find out whether they are wrong, much less why.**



For lack of a steady demand, the kind of expertise needed to perform competent floating-point error-analysis gets harder to find. There are places to learn about error-analysis:

- Web pages like mine and Jim Demmel's and a few others ...
- Texts like Nick Higham's (700 pages) and Pete Stewart's and a few others ...

But it is so difficult that hardly any students choose to learn about it.

CS grads don't have to know more about floating-point than about an hour's worth in a programming language course. Apparently ...

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

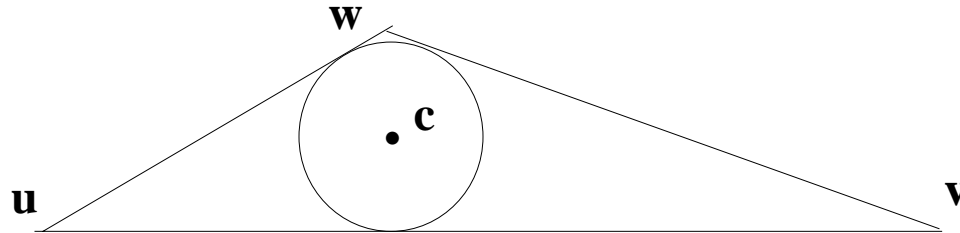
No Numerical Analysis appears in *Educational Testing Service's* COMPUTER SCIENCE Major Field Test (4CMF) of students' mastery of a CS curriculum.

Applied Math. students have to learn some Numerical Analysis, but the readable texts treat N.A. as algebra applied to the transformation of standard problems into algorithms. And those algorithms are simplified for didactic purposes, not the algorithms packaged software like MATLAB uses to solve those problems. Hardly any students learn about the algorithms' failure modes:

- Roundoff can bloat intolerably only if a problem's data is too close to a singularity.
- Some singularities are intrinsic to the problem --- "Ill Conditioned data"
- Some are spurious artifacts of the program's algorithm --- "Numerical Instability"
- Recognizing a singularity can be difficult; deciding whether it is spurious more so.

Let's look at an example ...

**Example:** Given the vertices  $\mathbf{u}$ ,  $\mathbf{v}$  and  $\mathbf{w}$  of a triangle in Euclidean 3-space, we seek its *incenter*  $\mathbf{c}$ , the center of its inscribed circle.



$\mathbf{c}$  is characterized by these requirements:

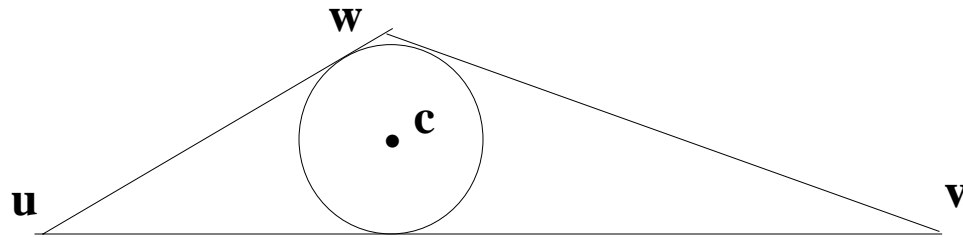
- $\mathbf{c}$  lies in the plane containing the vertices.
- The line segment joining  $\mathbf{c}$  to any vertex makes equal angles with the triangle's two edges at that vertex.

The requirements must be translated by vector algebra into equations whose solution is  $\mathbf{c}$ .

This example is one of many in my classroom notes


“Computing Cross-Products and Rotations in 2- and 3-Dimensional Euclidean Spaces”  
posted at [www.eecs.berkeley.edu/~wkahan/MathH110/Cross.pdf](http://www.eecs.berkeley.edu/~wkahan/MathH110/Cross.pdf) .

These notes have a better cross-product notation and details omitted from what follows.




**c** satisfies a system  $M \cdot \mathbf{c} = \mathbf{m}$  of linear equations whose 3-by-3 matrix  $M := [\mathbf{p}, \mathbf{b}, \mathbf{q}]^T$  and column  $\mathbf{m} := [\pi, \beta, \theta]^T$  are computed from these formulas:

$$\begin{aligned}
 \mathbf{p} &:= (\mathbf{v}-\mathbf{u}) \times (\mathbf{w}-\mathbf{u}), & \pi &:= \mathbf{p} \cdot \mathbf{u}, & \dots & \text{in the triangle's plane} \\
 \mathbf{b} &:= \|\mathbf{u}-\mathbf{v}\| \cdot (\mathbf{w}-\mathbf{v}) - \|\mathbf{w}-\mathbf{v}\| \cdot (\mathbf{u}-\mathbf{v}), & \beta &:= \mathbf{b} \cdot \mathbf{v}, & \dots & \text{bisect angle at } \mathbf{v} \\
 \mathbf{q} &:= \|\mathbf{v}-\mathbf{w}\| \cdot (\mathbf{u}-\mathbf{w}) - \|\mathbf{u}-\mathbf{w}\| \cdot (\mathbf{v}-\mathbf{w}), \quad \text{and} & \theta &:= \mathbf{q} \cdot \mathbf{w}. & \dots & \text{bisect angle at } \mathbf{w}
 \end{aligned}$$

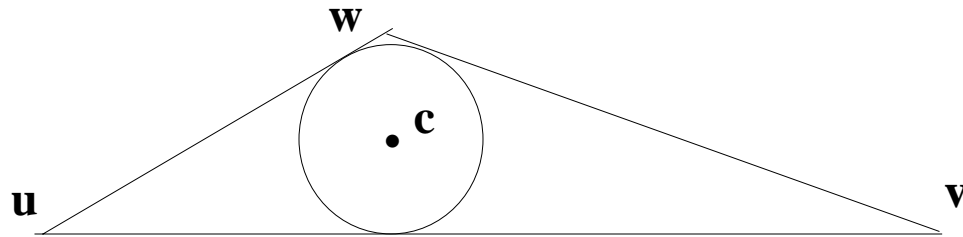
Now  $\mathbf{c} = M^{-1} \cdot \mathbf{m}$ . 

Where are its singularities? This linear system's determinant is

$$\det(M) = \mathbf{p} \cdot \mathbf{b} \times \mathbf{q} = \|\mathbf{v}-\mathbf{w}\| \cdot (\|\mathbf{u}-\mathbf{v}\| - \|\mathbf{v}-\mathbf{w}\| + \|\mathbf{w}-\mathbf{u}\|) \cdot \|(\mathbf{v}-\mathbf{u}) \times (\mathbf{w}-\mathbf{u})\|^2.$$

which vanishes just when the given triangle is *degenerate* (has area zero). This suggests that  $\mathbf{c} = M^{-1} \cdot \mathbf{m}$  should lose accuracy when applied to excessively narrow triangles. 

Let's apply the formula  $\mathbf{c} = M^{-1} \cdot \mathbf{m}$  to some numerical data:



$$\mathbf{u} := [ 255429.53125, -139725.125, 140508.53125 ]^T,$$

$$\mathbf{v} := [ 10487005., 8066347., -11042884. ]^T,$$

$$\mathbf{w} := [ -5243521., -4033150., 5521499. ]^T .$$

These are vertices of a narrow triangle 80 times as long as wide. Its accurate incenter is

$$\mathbf{c} = [ 128615.61552\dots, -69127.510715\dots, 69282.163604\dots ]^T$$

$$\approx [ 128615.6171785, -69127.5078125, 69282.1640625 ]^T$$

when rounded to 4-byte wide 24 sig. bit floats. The data all fit into floats too.

When  $\mathbf{c} = \mathbf{M}^{-1} \cdot \mathbf{m}$  is computed carrying just 24 sig. bits throughout, its every element is in error by about  $\pm 400.00$ , leaving fewer than half as many correct digits as were carried.




Would this much error disturb you? If so, what would you blame for it? The condition number  $\kappa(\mathbf{M}) \approx 100$ ; should a loss of 2 sig. dec. be expected? If so the error should not be worse than  $\pm 1.00$ , far smaller than what is observed. Is this puzzle worth solving?



The foregoing is a taste of what passes too often for error-analysis. It is utterly wrong, but in ways that almost all practitioners find extremely difficult to unravel.


First, the condition number  $\kappa(M) \approx 100$  that is generally interpreted as an amplification factor for roundoff tells only a small part of the story. Actually, roundoff did most of its damage when  $M := [\mathbf{p}, \mathbf{b}, \mathbf{q}]^T$  and  $\mathbf{m} := [\pi, \beta, \theta]^T$  were computed from the formulas

$$\begin{aligned} \mathbf{p} &:= (\mathbf{v}-\mathbf{u}) \times (\mathbf{w}-\mathbf{u}), & \pi &:= \mathbf{p} \cdot \mathbf{u}, & \dots & \text{in the triangle's plane} \\ \mathbf{b} &:= \|\mathbf{u}-\mathbf{v}\| \cdot (\mathbf{w}-\mathbf{v}) - \|\mathbf{w}-\mathbf{v}\| \cdot (\mathbf{u}-\mathbf{v}), & \beta &:= \mathbf{b} \cdot \mathbf{v}, & \dots & \text{bisect angle at } \mathbf{v} \\ \mathbf{q} &:= \|\mathbf{v}-\mathbf{w}\| \cdot (\mathbf{u}-\mathbf{w}) - \|\mathbf{u}-\mathbf{w}\| \cdot (\mathbf{v}-\mathbf{w}), & \text{and } \theta &:= \mathbf{q} \cdot \mathbf{w}. & \dots & \text{bisect angle at } \mathbf{w} \end{aligned}$$

Were the triangle degenerate, at least one row of  $[M, \mathbf{m}]$  and perhaps three would vanish; most digits of  $[M, \mathbf{m}]$  get lost before  $\mathbf{c} = M^{-1} \cdot \mathbf{m}$  is computed if the triangle is narrow. 

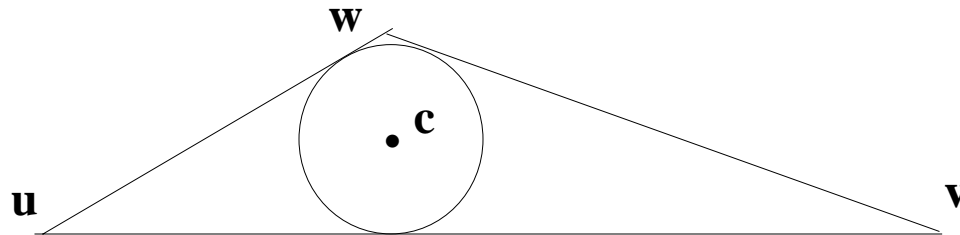
**This loss is utterly unnecessary.**

If a computerized algebra system could *Simplify* the foregoing formula for  $\mathbf{c}$  (but I know none that can do so) the result would be a numerically far superior formula

$$\mathbf{c} = ( \mathbf{u} \cdot \|\mathbf{v}-\mathbf{w}\| + \mathbf{v} \cdot \|\mathbf{w}-\mathbf{u}\| + \mathbf{w} \cdot \|\mathbf{u}-\mathbf{v}\| ) / ( \|\mathbf{v}-\mathbf{w}\| + \|\mathbf{w}-\mathbf{u}\| + \|\mathbf{u}-\mathbf{v}\| ). \quad \text{$$

When this formula is used carrying, as before, just 24 sig. bits, the error is not  $\pm 400$ . but about  $\pm 0.3$ , and is that big only because  $\|\mathbf{w}\|/\|\mathbf{c}\| > 53$  and  $\|\mathbf{v}\|/\|\mathbf{c}\| > 106$ .

How easily could you explain all that?



$$\mathbf{c} = ( \mathbf{u} \cdot \|\mathbf{v}-\mathbf{w}\| + \mathbf{v} \cdot \|\mathbf{w}-\mathbf{u}\| + \mathbf{w} \cdot \|\mathbf{u}-\mathbf{v}\| ) / ( \|\mathbf{v}-\mathbf{w}\| + \|\mathbf{w}-\mathbf{u}\| + \|\mathbf{u}-\mathbf{v}\| ) . \quad \text{📄}$$



How many people know this formula nowadays? How many of the rest could find it if they had to compute  $\mathbf{c}$  somehow? Can *Google* find it before I post it on my web page?

## It is unreasonable to expect that ...

- ... so benign a solution exists for every numerical problem.
- ... everyone who needs such a solution can find it when it exists.
- ... every numerical program, benign or not, is error-analyzed by its programmer.

How can we defend ourselves and our loved ones from numerical computations programmed by clever and knowledgeable but numerically naïve programmers?

How can we defend ourselves and our loved ones from numerical computations programmed by clever and knowledgeable but numerically naïve programmers?

 We need a defence against programs that have been tested for accuracy with all due diligence but suffer from spurious singularities that amplify rounding errors intolerably for very rare and otherwise innocuous data that the program seems to dislike. Though very rare, such data can be distributed either almost everywhere (as was the case for the Pentium FDIV bug) or tightly close to some smooth surface in data space. A frightful example of the almost-everywhere kind afflicted one of the earliest computers for two  years; see “Marketing vs. Mathematics” [<.../MktgMath.pdf>](#) on my web page. Other perplexing examples appear in “Do MATLAB’s `lu(...)`, `inv(...)`, `/` and `\` have *Failure Modes* ?” [<.../Math128/FailMode.pdf>](#). Such anomalous behavior often escapes notice during diligent testing only to be discovered by one of the program’s users later. Then the program’s vendor and programmer need help to find and cure the anomaly.

Don’t think “very rare” will probably protect you. If your data lies in data-space on a surface that intersects the spurious singularity’s surface tangentially, then “rare” will become “often” for you.

How can we defend ourselves and our loved ones from numerical computations programmed by clever and knowledgeable but numerically naïve programmers?

I propose three steps:



**Step 1:** Programming languages that support floating-point hardware must by default (unless the program text demands otherwise) declare all local variables and perform all floating-point arithmetic with the widest precision available at hardware's speed, even if this precision exceeds extravagantly the precision of the input data and the accuracy desired for the result put out.


Usually every extra decimal digit carried by these extra-precise intermediate variables and arithmetic reduces the incidence of unanticipated embarrassment due to roundoff by a factor near  $1/10$ . Carrying somewhat more precision than twice the data's and the result's almost always eliminates that embarrassment, as experience has shown.

This step was taken serendipitously in the mid 1970s by Kernighan and Ritchie when they designed the language *C* for their DEC PDP 11s. This serendipity was undone by ANSI X3J11 in the mid 1980s at the behest of CDC and CRAY. Where are they now?



How can we defend ourselves and our loved ones from numerical computations programmed by clever and knowledgeable but numerically naïve programmers?

**Step 2:** Program development environments (compilers and their debuggers) that support floating-point computation, regardless of the precision(s) supported, must permit users to rerun subprograms in all three directed rounding modes (to  $-\infty$ , to 0, to  $+\infty$ ) in lieu of the default (to nearest), allowing the user to specify the modes' scope, so that the math. library of elementary functions and decimal-binary conversions will not be spoiled by altered rounding modes. For more details see “How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?” [<.../Mindless.pdf>](#) on my web page.

No matter which formula is chosen to compute the incenter  $\mathbf{c}$ , its four values computed in all four rounding modes differed among themselves by at least roughly as much as the computed  $\mathbf{c}$  was in error. The same happens to every geometrical computation described in [<.../Cross.pdf>](#), and to all the examples cited in [<.../Mindless.pdf>](#) except a few examples constructed to show why there is no foolproof way to locate modules that deserve scrutiny for possible numerical instability. Rerunning with redirected roundoff almost always works. (But randomized rounding has been proved unreliable.) 

Steps 2 and 3 are needed when step 1 fails or is unavailable.

How can we defend ourselves and our loved ones from numerical computations programmed by clever and knowledgeable but numerically naïve programmers?

**Step 3:** Program development environments (compilers and their debuggers) that support floating-point computation with very high precision(s) must permit programs to be debugged by running two versions, one with higher precision from a designated point to its end. This is a subtle technique described in more detail near the end of “How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?” posted at [<.../Mindless.pdf>](#). The subtlety is necessary because two programs, identical but for one’s precision rather higher than the other’s, often diverge in the middle but come together at the end. Think of iterative equation-solvers, or Gaussian elimination when two candidates for pivot differ inly by roundoff.

Step 3 is needed mostly when step 2 fails to locate a small suspicious module inside a large program suspected or convicted of hypersensitivity to roundoff for some otherwise innocuous data.

## **What can you do now?**

Are you a consumer of compilers and program development environments?

Now you know what to demand from your software vendors.

Are you a designer or vendor of compilers or program development environments?

Now you know what the world of scientific and engineering computation needs to debug software that affects buildings, bridges, tunnels, vehicles, medical imaging, etc., upon which you or your loved ones may depend some day.

Think not of your *Duty to Truth in Computation.*

Think about *Self-Defence.*