How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?

Why should we care? What should we do?

Presented on 23 May 2005 to the Householder Symposium XVI in Seven Springs PA

FOR DETAILS SEE `<www.eecs.berkeley.edu/~wkahan/`**`Mind1ess.pdf`**

The purpose of this presentation is to tempt you to read that very long document.

The purpose of that document is to persuade you to demand better support for the diagnosis of numerical embarrassment, much of it due to roundoff.
Better support? See `<www.eecs.berkeley.edu/~wkahan/`**`Boulder.pdf`**`>`.

Hardware conforming to IEEE Standard 754 for Binary Floating-Point does support better diagnostic tools than you are getting 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.

Rather than try to persuade you that Rounding-Error Analysis should be as important to you as it is to me, and therefore deserves your generous support, I shall assume that you would rather have nothing to do with it.

Almost all students of Mathematics and Computer Science incline this way.

I believe that,
should a (presumably rare) numerical anomaly embarrass you,
you would prefer to determine as quickly and quietly as possible
(in case it's your own fault) whom to blame.

Rather than present a general assessment of ways to diagnose and sometimes cure numerical embarrassments …

(that can be found in my lengthy …/Mindless.pdf and …/Boulder.pdf),

I shall titillate you with some examples drawn from …/Mindless.pdf .

Our first examples are **Errors Designed Not To Be Found**.

Parentheses in Microsoft's *Excel 2000* spreadsheet can have uncanny powers:

Values *Excel 2000* Displays for Several Expressions

Expression	`1.23456789012345000E+00`	<– Entered to help count digits
`V = 4/3  displays ...`	`1.33333333333333000E+00`	Does *Excel* carry 15 sig. dec.?
`W = V - 1`	`3.33333333333333000E-01`	Whence comes the 15th 3 ?
`X = W*3`	`1.00000000000000000E+00`	Where went all 15 of the 9s ?
`Y = X - 1`	`0.00000000000000000E+00`	They all went away !
`Z = Y*2^52`	`0.00000000000000000E+00`	Really all gone.
`(4/3 - 1)*3 - 1`	`0.00000000000000000E+00`	Yes, gone.
`((4/3 - 1)*3 - 1)`	`-2.22044604925031000E-16`	(But not *ENTIRELY* gone !)
`((4/3 - 1)*3 - 1)*2^52`	`-1.00000000000000000E+00`	Excel's arithmetic is weird.

Besides generating an extra digit "3" and rounding away 15 "9"s, *Excel* changed the value of an expression placed between parentheses from zero to something else. Why?

Apparently *Excel* rounds *Cosmetically* in a futile attempt to make Binary floating-point appear to be Decimal. This is why *Excel* confers supernatural powers upon some (not all) parentheses and induces other inconsistencies.

11 floating-point numbers X between $1 - 5/2^{53}$ and $1 - 13/2^{53}$ all look the same displayed:

11 Consecutive DistinctValues X Displayed as " 0.99999999999999000…"

#	(X–1)	SIGN(X–1)	FLOOR(X)	(X < 1)	(X = 1)	ACOS(X)	X–1
8	… < 0	–1	0	TRUE	FALSE	… > 0	… < 0
3	… < 0	–1	0	TRUE	FALSE	… > 0	0

27 distinct floating-point numbers X between $1 - 4/2^{53}$ and $1 + 22/2^{52}$ all look the same displayed.

27 Consecutive Distinct Values X Displayed as " 1.00000000000000000… "

#	CEIL(X)	FLOOR(X)	(X < 1)	(X = 1)	X–1	(X–1)	SIGN(X–1)	ACOS(X)
4	1	1	FALSE	TRUE	0	… < 0	–1	… > 0
1	1	1	FALSE	TRUE	0	0	0	0
7	1	1	FALSE	TRUE	0	… > 0	+1	#NUM!
15	1	1	FALSE	TRUE	… > 0	… > 0	+1	#NUM!

43 Consecutive Distinct Values Y Displayed as " 1024.5000000000… "

#	Displayed Y	ROUND(Y)	ROUND(Y–25)	ROUND(Y–925)
19	1024.500000000…	1025	999	99
2	1024.500000000…	1025	1000	99
22	1024.500000000…	1025	1000	100

How can a user of *Excel* debug his work without knowing which operations depend not upon the values of their arguments but upon how they display?

How can Microsoft cure those of *Excel*'s anomalies exhibited here?

• Switch *Excel*'s floating-point to honest decimal floating-point conforming to IEEE Standard 754 (2008) . This would be the best remedy.

Maybe after IBM's *Lotus 123* does that, *Excel* will imitate it.

In promotional advertisements for a certain software company,
the word "Innovate" often appears where "Imitate" would be more truthful.

Decimal's great advantage over binary is that, if enough digits are displayed,
What You See is What You Get.

Meanwhile, until then, . . .

• Allow *Excel*'s users to display up to 17 sig. dec. instead of at most 15;
and eschew Cosmetic Rounding;
and put some advice about the differences between binary
and decimal into Excel's HELP files.

What's so special about *15* sig. dec. and *17* sig. dec. ?

Displaying at most 15 sig. dec., as *Excel* does, ensures that a number entered with at most 15 sig. dec., converted to binary floating-point rounded correctly to 53 sig. bits (which is what *Excel*'s arithmetic carries), and then displayed converted back to decimal floating-point rounded correctly to at least as many sig. dec. as were entered but *no more* than 15, will *always* display

exactly the same number as was entered.

A decision to make *Excel*'s arithmetic seem to be Decimal instead of Binary restricted *Excel*'s display to at most 15 sig. dec., thus hiding the deception well enough to reduce greatly the number of calls upon *Excel*'s technical help-desk.

When symptoms of the deception are perceived they are routinely misdiagnosed; *e.g.*, see David Einstein's column on p. E2 of the *San Francisco Chronicle* for 16 and 30 May 2005.

If distinct 53 sig. bit binary floating-point numbers are converted to decimal and displayed correctly rounded to 17 sig. dec., they will *always* display differently. And if the displayed numbers are converted back to binary and rounded correctly to 53 sig. bits, they will reproduce the original binary floating-point numbers.

But displaying 17 sig. dec. exposes the non-decimal nature of the underlying arithmetic as soon as a number entered as, for example, " 8.04 " displays as

" 8.0399 9999 9999 9991 " .

And now for something entirely different …

Over-Zealous Compiler “Optimization”

. . . exploits the associative laws of arithmetic disregarding parentheses.

Consider slowly converging sums for infinite series, for updating averages, for amortization schedules, for quadrature (numerical integration), and for trajectories (differential equations), among other things.

Ideal infinite sum := $\sum_{k\ge1}$ term(k) is approximated by

Computed Sum := $\sum_1^N$ Term(k) + Tail(N)

in which Tail(N) approximates $\sum_{k>N}$ term(k) ever better as N increases.

But we shall not know N in advance. It may mount into billions.

Billions of rounding errors can degrade severely a sum computed naively :

```
[xxxxxx... Old Sum …xxxxxx]
+                   [xxxxxx… New Term …xxxxxx]
-------------------------------------
[xxxxxx… New Sum …xxxxx] […lost digits…]
```

The lost digits affect the Computed Sum about as much as if those digits had first been discarded from each New Term. The effect is severe if N is gargantuan.

The following program compensates for those lost digits;. For simplicity, it has been written assuming every Term(k) > Term(k+1) > Term(k+2) > … > 0 . …

```
Sum := 0.0 ;  Oldsum := –1.0 ;  comp := 0.0 ;  k := 0 ;
While  Sum > Oldsum  do …
    k := 1+k ;  Oldsum := Sum ;  comp := comp + Term(k) ;
    Sum := comp + Oldsum ;
    comp := (Oldsum – Sum) + comp ;
  End While Loop;
Sum := Sum + ( Tail(k) + comp ) .
```

However, an over-zealously "optimizing" compiler deduces that the statement

comp := (Oldsum – Sum) + comp ;

is merely an elaborate way to recompute comp := 0.0 , and therefore scrubs out all references to comp, thus simplifying and slightly speeding up the Loop:

```
Sum := 0.0 ;  Oldsum := –1.0 ;  k := 0 ;
While  Sum > Oldsum  do …
    k := 1+k ;  Oldsum := Sum ;
    Sum := Term(k) + Oldsum ;
  End While Loop;
Sum := Sum + Tail(k) .
```

Example of Pejoration by Over-Zealous “Optimization”:

Our task is to compute Sum := $\sum_1^N$ Term(k) + Tail(N) given that

$$\mathrm{Term}(k) := 3465/(k^2 - 1/16) + 3465/((k + 1/2)^2 - 1/16) \ ,$$
$$\mathrm{Tail}(k) := 3465/(k + 1/2) + 3465/(k + 1) \ ,$$

using each of the foregoing programs, one compensated, the other “optimized”.

Of course, a little mathematical analysis might render the programs unnecessary, but programming a computer is easier and running it is cheaper than analysis.

Here are the results from a Fortran program run on an IBM T21 Laptop:

Table 1: Final Computed Sum

Program:	Compensated	“Optimized”
Final Sum :	9240.000000000000	9240.000001147523
Time :	13.7 sec.	17.8 sec.
Loop-count K :	61,728,404	87,290,410
Time per Loop :	2.22E–7 sec.	2.04E–7 sec.

Even though the “Optimized” program’s Loop runs almost 10% faster, the program run as written got a significantly better result about 25% sooner.

Do you see why?
If someone doesn’t, would you like him to “optimize” your floating-point?

How can a programmer unaware of the "optimization" debug it?

There is a way: Rerun both programs in different rounding modes afforded by IEEE Standard 754 on fully conforming systems. Currently the only fully conforming standard programming language is C99, and only on a very few machines, but let's not dwell on that now. On my machines each program can be rerun first rounding every arithmetic operation Down (towards –∞) and again rounding Up (towards +∞) *without recompilation*. Here are the results:

Final Sums from Two Programs Rounded Differently

Program:	Compensated	"Optimized"
Rounded to Nearest :	9240.000000000000	9240.000001147523
Rounded Down :	9239.999999999998	9239.999994834162
Rounded Up :	9240.000000000002	Ran almost forever

Evidently "optimization" has actually worsened the program's accuracy and its speed, and also its sensitivity to roundoff, which exposes the "optimization".

Do you see why the "optimized" program Rounded Up ran almost forever?
If someone doesn't, would you like him to "optimize" your floating-point?

“Optimized” Register Spill

Sometimes compilers “spill” the contents of a wide register temporarily to a narrow cell in memory and later reload it having lost, presumably inadvertently, some of the bits generated in the wide register. We would like to think nobody concerned with the integrity of floating-point arithmetic would do such a thing.

Alas, MATLAB 6.5 does it on Wintel machines. Let’s see the evidence:

Wallace Givens’ n-by-n test matrix looks like this when n = 6 :

```
22    18    14    10     6     2
18    18    14    10     6     2
14    14    14    10     6     2
10    10    10    10     6     2
 6     6     6     6     6     2
 2     2     2     2     2     2
```

It can be derived from a discretization of an integral equation. Its eigenvalues and eigenvectors can be computed accurately from simple formulas that shall be used only to check the accuracy of MATLAB’s and my eigensystem software.

The smallest eigenvalues cluster just above 1 ; the biggest reach over $(4n/\pi)^2$. The eigenvectors have a special structure: Every eigenvector’s elements can be obtained from any other’s by permuting its elements and reversing some signs.

The accuracy of computed eigenvectors belonging to small clustered eigenvalues can be degraded by roundoff to an extent that grows about as fast as n^4 .

I wrote a MATLAB program `refiheig` designed to *Iteratively Refine* eigensystems computed for Hermitian matrices by MATLAB's `eig`. It works for versions 4.2 to 6.5 of MATLAB on PCs, versions 4.2 to 5.2 on old 68040-based Macs. It is needed when dimensions are so big that roundoff accumulates to obscure the results of `eig` excessively, or when roundoff prevents structural symmetries in the data from propagating into the computed eigensystem.

Iterative refinement by `refiheig` often undoes these and other kinds of accuracy loss without obliging its user to know what caused the loss.

The accuracy of `refiheig` is limited by the accuracy with which it can compute residuals by matrix multiplication. By default on Wintel machines, MATLAB 6.5 accumulates these to 53 sig. bits. However, after the prefatory command

```
system_dependent('setprecision', 64)
```

(or in version 4.2 without that command), matrix products are accumulated to 64 sig. bits before being stored back to 53. This is how Intel's (in 1978) and Motorola's (in 1980) floating-points were originally designed to be used.

The results tabulated hereunder were obtained for n = 1000 on an IBM T21 laptop running Windows 2000 from MATLAB v. 6.5 accumulating matrix products with 53 and then 64 sig. bits, and from v. 4.2 only with 64 sig. bits. The tabulated residuals are compared with the "minimal" residual for the true eigensystem computed almost correctly rounded from trigonometric formulas.

Alas, something goes awry.

Table 2: Execution Times

MATLAB v:	v. 6.5	v. 6.5	v. 4.2
MxM sig. bits	53 s.b.	64 s.b.	64 s.b.
eig	52.5 sec.	52.9 sec.	122 sec.
refiheig	67.1 sec.	66.7 sec.	1171 sec.

Table 3: Residuals vs. minimal 2.3E-11

MATLAB v:	v. 6.5	v. 6.5	v. 4.2
MxM sig. bits	53 s.b.	64 s.b.	64 s.b.
eig	2.1E-9	1.2E-10	3.1E-9
refiheig	1.2E-10	2.9E-11	7.4E-12

Table 4: Eigenvector Accuracies in Sig. Bits

MATLAB v:	v. 6.5	v. 6.5	v. 4.2
MxM sig. bits	53 s.b.	64 s.b.	64 s.b.
eig	18.4 s.b.	23.4 s.b.	18.6 s.b.
refiheig	25.9 s,b.	30.2 s.b.	40.7 s.b.

Why is MATLAB version 6.5 so much (18 x) faster than version 4.2 ?

Why is v. 6.5's refinement so much (3 sig. dec.) less accurate than v. 4.2's ?

V. 6.5 splits big matrices into small blocks to incur fewer cache misses during its blocked-matrix multiplications. These can run enormously faster than v. 4.2's ordinary unblocked matrix multiplications which incur many cache misses.

But v. 6.5 spills sums of block products, each accumulated to 64 sig. bits, into memory holding only 53. This squanders almost all advantages of extra-precise accumulation, obscuring residuals while adding negligibly to speed. The consequent loss of 10 sig. bits of ultimate accuracy could not have been detected if we had compared only computed residuals instead of comparing computed with known correct eigenvectors. Has anybody else noticed this spill anomaly?

The anomaly should not be blamed entirely upon MATLAB. It uses matrix-multiply subprograms "optimized" by Intel for its *Pentium* architecture taking account of cache-line sizes and management. If the subprograms stored sums of block products retaining all 64 sig. bits accumulated, the extra time and memory required would be practically inconsequential.

Thus does petty optimization for speed induce a subtle but severe pejoration of accuracy difficult to debug, even if noticed, for lack of access to source-code.

Sometimes inaccessible source-code cannot prevent diagnosis of formulas that are algebraically correct but numerically dubious, as in the next example …

A Hypothetical Case Study: Bits Lost in Space

Imagine plans for unmanned astronomical observatories in outer space. For details see §11 of `Mindless.pdf` .

Directions to planets and distant stars are specified by `float` 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 $|\theta-\Theta|$ mod 2π . We'll compare 3 `float` implementations of this function ψ ; 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*	3.14082*050*	3.14055*681*
$\psi \approx$ v :	0.001593248*68*	0.00024414061*0*	0.00017263348*9*	0.00056223187*1*	3.140*61618*	3.140*61618*
$\psi \approx$ w :	0.001593248*68*	0.00024414061*0*	0.00017263348*9*	0.00056223187*1*	3.140780*44*	3.140548*47*

Which digits are *correct* ? 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. In fact, it uses an unobvious formula stable for all admissible data.

Subprogram u uses a 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 `Mindless.pdf` for formulas.

Without access to source code, nor to another subprogram known to be reliable, how else might you have decided which program(s) to distrust 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.

Some compilers have supported dynamically redirected rounding, but almost no programming languages support it. The exceptions are a few C99 compilers.

Java outlaws redirected rounding. See

`www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf` .

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

For other desirable debugging tools we may wish were provided by programming development systems, tools that employ high-precision floating-point and interval arithmetic combined (they are not helpful enough by themselves), see §14 of `www.cs.berkeley.edu/~wkahan/Mindless.pdf`, and …`/Boulder.pdf`.

For better exception-handling than provided by current programming languages with the exceptions of C99 and perhaps Fortran 2003, see …`/Grail.pdf` and …`/ARITH_17U.pdf` , and pp. 46 - 73 of …`/Boulder.pdf` .