The Baleful Influence of **SPEC** Benchmarks
upon
Floating-Point Arithmetic

Prepared for **SPEC**
17 May 2005

by
Prof. W. Kahan

Math. Dept., and
Elect. Eng. & Computer Science Dept.
University of California @ Berkeley

and the
Committee to Revise IEEE Standard 754

Three Challenges:

- How can **SPEC** benchmarks take Correctness and Robustness into account as well as Speed?
- How can **SPEC** benchmarks inhibit petty “optimizations” that turn into pejorations, which degrade the correctness and mathematical integrity of numerical software generally?
- How can **SPEC** benchmarks reward improved arithmetic designs instead of eschewing them, thus penalizing their designers?

“Correctness” is usually construed as “Accuracy within Acceptable Limits”.

Accuracy is NOT the goal of applications software used directly by scientists and engineers for their own numerical computations.

They perform those computations only in order to Predict.

Prediction entails Extrapolation.

Extrapolation practically ignores some errors while amplifying others.

Approximation, without which computation would take longer than we can wait, can be justified only if we know its errors will not be amplified intolerably later.

In general, no way exists to know that.

Approximations acceptable in one context can be intolerable in another, and only a possibly difficult error-analysis can be expected to tell which is which. See “… Mindless Assessments of Roundoff …?” `http://www.cs.berkeley.edu/~wkahan/Mindless.pdf` .

This is why we require support software -- the Math. library and compilers -- to maintain mathematical integrity and accuracy as well as economically possible.

Petty compiler “optimizations” that undermine mathematical integrity are actually pejorations that benchmarks should disallow or at least discourage.

Petty Compiler "Optimizations"
that actually pejorate floating-point computations:

- Compile-time algebraic rearrangements that override "redundant" parentheses to apply distributivity, presumably to exploit common subexpressions.

- Compile-time algebraic rearrangements that override "redundant" parentheses to apply associativity without the programmer's explicit licence.
  Most programmers will licence it to speed up almost all matrix multiplications.

- Compile-time replacement of divisions, sqrt, exp. log, trig functions, etc. by faster but less accurate versions *with the same names*. If a programmer needs a fast-but-dirty sqrt, say, he should either call his own `mysqrt` or call a `dirtysqrt` from the Math. library.

- Register-spill to and from anonymous variables narrower than the registers.

- Replacement of *Gradual Underflow* by *Flush-to-Zero* bundled with other optimizations, some of which may be good ones.

Benchmark programs that allow or encourage these pejorations impose them unwittingly upon innocent programmers who opt for speed when they cannot appreciate the consequent degradation of mathematical integrity.

Example: 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 := $\sum_{k \geq 1}$ 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 . …

Compensated Summation:

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

But now the computed Sum can be wrong in the worst way: Occasionally its error will be too small to be obvious but not small enough to be inconsequential.

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

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” floating-point?

What Computational Style(s) should benchmarks promote?

In the absence of a competent error-analysis, programmers will almost never be embarrassed by roundoff if they opt for old-fashioned Kernighan-Ritchie ***C*** semantics, which by default evaluated every expression and constant in `double` even if all operands were `floats`. This policy accords with an ancient rule-of-thumb inherited from the days of slide-rules and electromechanical calculators:

In the absence of a competent error-analysis, perform all intermediate arithmetic in a little more than twice the precision to which data and final results are stored.

An updated rule-of-thumb would replace "a little more than twice ..." by "the widest precision available that does not run too slow."

Except perhaps for C99 , today's programming languages and compilers are stuck with a mind-set adopted as a disagreeable but necessary expedient in the late 1950s when compilers had to fit entirely into 128KB and pass just once over the program being compiled. Benchmarks should allow excursions beyond that mindset.

Here is a possibility that current benchmarking policies would disallow:

A Candidate Worth Considering as a Benchmark:

Iterative Refinement of Computed Eigenvectors and Eigenvalues

Eigenvectors and Eigenvalues characterize the “Natural” modes and frequencies of vibration of elastic structures of aircraft, bridges and buildings, among many other things. Stimulation of some natural modes can cause failures.
Examples: “Galloping Gertie, the Tacoma Narrows bridge. Marching army “Breaks Step” when crossing a bridge.

Computed eigensystems may lose accuracy to roundoff in several ways:

- Losses worsen as dimensions (degrees of freedom) increase.
- Eigenvectors lose accuracy as their eigenvalues approach coincidence.
- Severe losses can occur if data’s structural symmetries are lost to roundoff.
- Severe losses … if software mishandles systematically wide-ranging data.

Example: A flea atop a dog atop an elephant atop the Eiffel tower.
The flea’s vibrational frequencies so dominate the tower’s that the tower’s can be lost to roundoff unless appropriate special methods are used.

Iterative Refinement is a scheme that usually attenuates those losses without requiring that their cause(s) be identified. The scheme starts by computing a *Residual* that measures how badly the solution computed so far dissatisfies its defining equations. Then the residual guides refinement of that solution.

First Illustrative Example: n-by-n *Pascal* matrices’ elements range ever wilder as dimension n increases. We seek at least 10 correct sig. bits.

The n-by-n *Pascal* matrix is an n-by-n corner of an infinite matrix constructed from Pascal's Triangle. Here is how it looks when n = 6 :

$$\begin{matrix}
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 3 & 6 & 10 & 15 & 21 \\
1 & 4 & 10 & 20 & 35 & 56 \\
1 & 5 & 15 & 35 & 70 & 126 \\
1 & 6 & 21 & 56 & 126 & 252
\end{matrix}$$

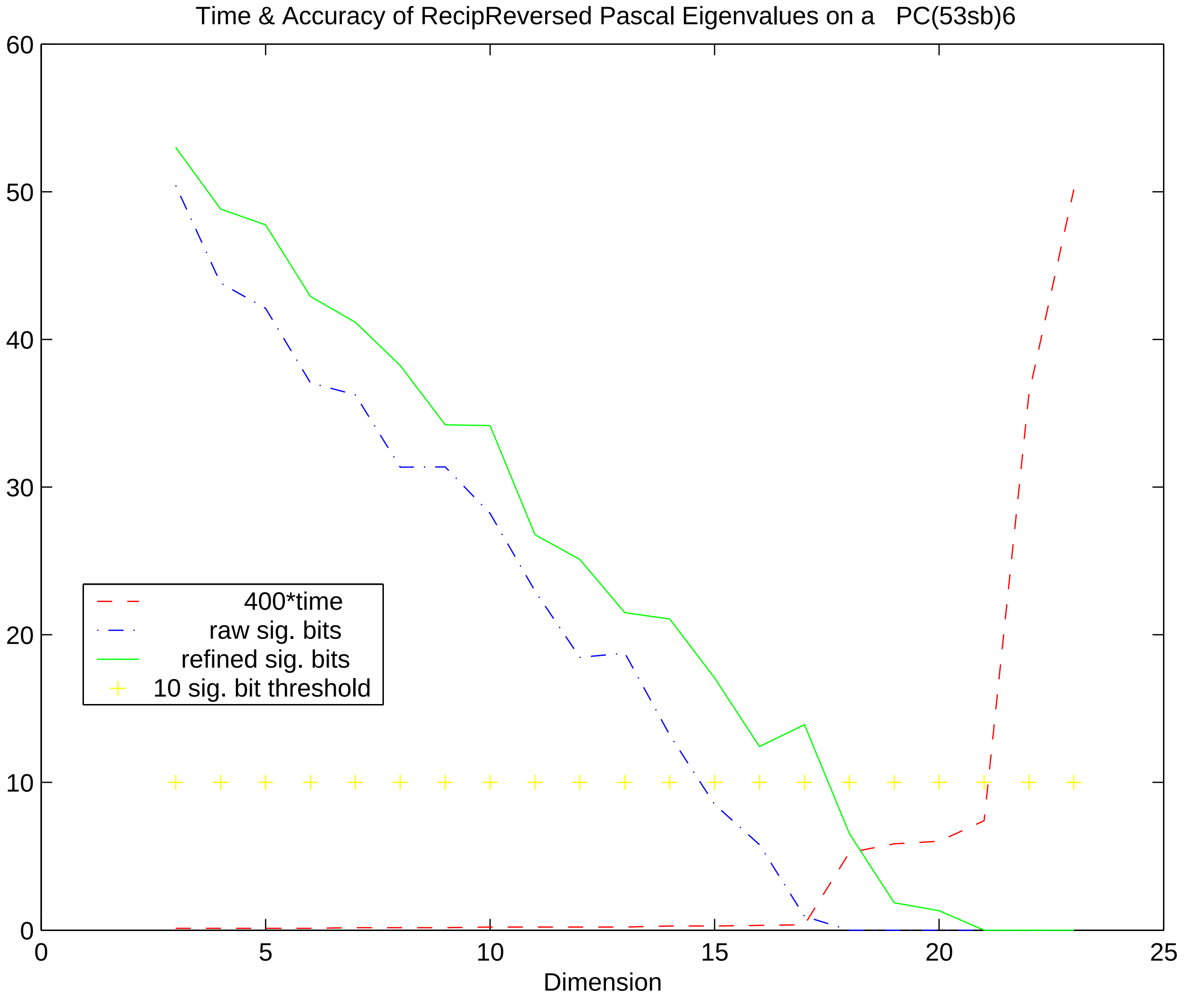
Though no simple formulas for its n eigenvalues are known, they are known to be positive and come in reciprocal pairs: If λ is an eigenvalue, so is $1/\lambda$.

We shall gauge the accuracy of computed eigenvalues
by how close products of appropriate pairs come to 1 .

Because the ratio (biggest eigenvalue)/(smallest) grows like $2^{4n}/(n\pi)$, we expect smaller computed eigenvalues to lose sig. bits at a rate proportional to the dimension n . The loss rate depends upon details of the computation; most algorithms used today lose accuracy faster if rows and columns are reversed thus:

$$\begin{matrix}
252 & 126 & 56 & 21 & 6 & 1 \\
126 & 70 & 35 & 15 & 5 & 1 \\
56 & 35 & 20 & 10 & 4 & 1 \\
21 & 15 & 10 & 6 & 3 & 1 \\
6 & 5 & 4 & 3 & 2 & 1 \\
1 & 1 & 1 & 1 & 1 & 1
\end{matrix}$$

Then most programs lose almost 4n of the sig. bits carried by their arithmetic.



MATLAB v. 6.5 on a Wintel PC accumulating matrix products to 53 sig. bits:
Refinement boosts successful dimensions n from n ≤ 14 to n ≤ 17 in a tolerable time.

Similar results are obtained on Sun SPARCs, SGS MIPS, HP PA-RISC, IBM Power PCs and Apple Power Macs:

Iterative Refinement increases from n = 14 to n = 17 the
largest dimension for which at least 10 sig. bits are achieved.

For larger dimensions computation time rises steeply mainly to issue warnings of possibly severe loss of accuracy.

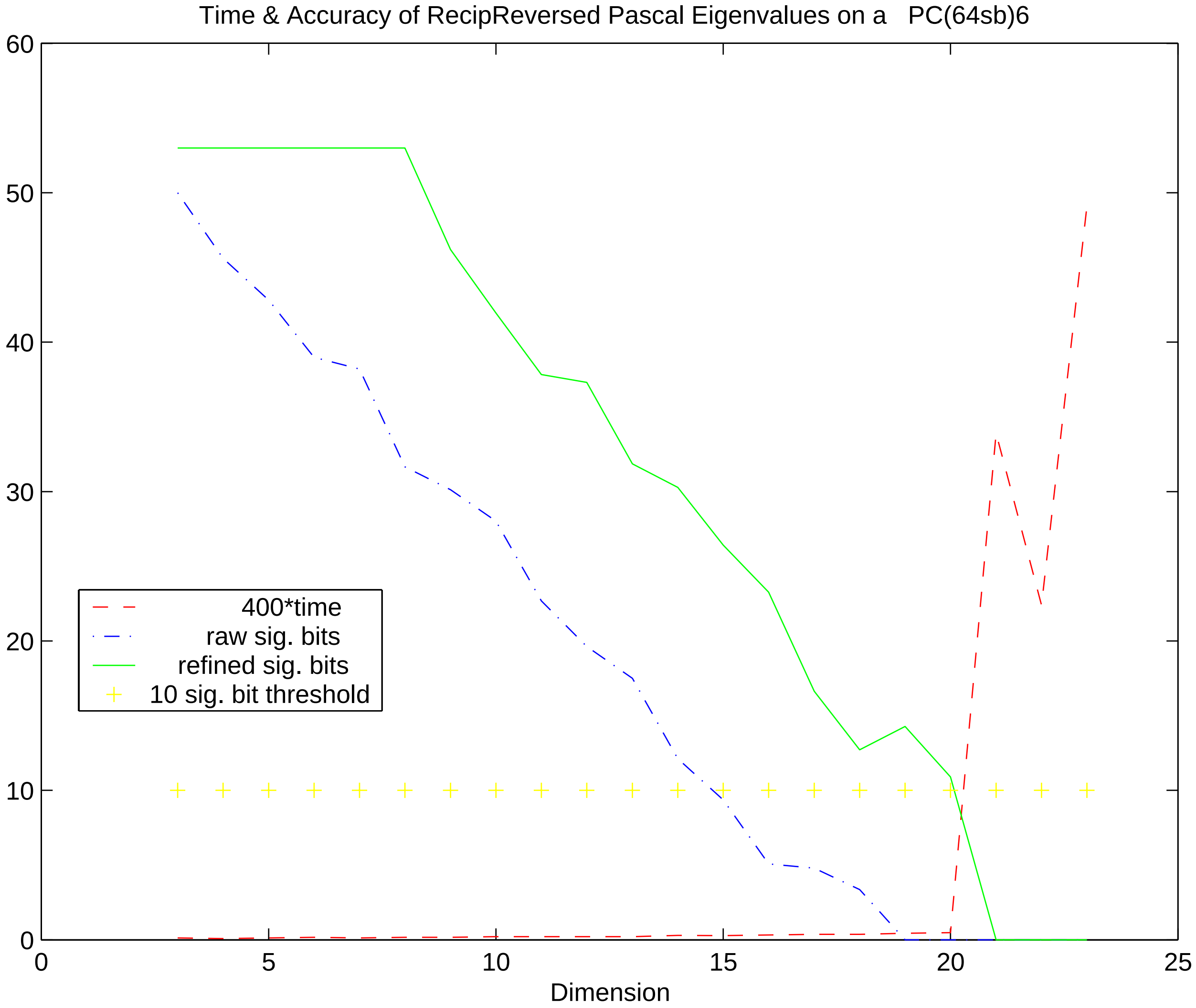
However, the foregoing are **UNFAIR** as **BENCHMARKS** for Wintel PCs.

These machines can get better results in the same time running exactly the same MATLAB programs on the same version 6.5 of MATLAB after invoking the prefatory command

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

(or on version 4.2 without that command) to accumulate matrix products to 64 sig. bits before storing them back to 53. This is how Intel's floating-point was originally (back in 1978) designed to be used.

With that extra-precise accumulation, Iterative Refinement increases from n = 14 to n = 20 the largest dimension for which 10 sig. bits are achieved,
and with no significant increase in running time.



MATLAB v. 6.5 on a Wintel PC accumulating matrix products to 64 sig. bits:
Refinement boosts successful dimensions n from $n \le 14$ to $n \le 20$ in a tolerable time.

Second Illustrative Example:

Wallace Givens' n-by-n 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 when the dimension n is huge. Iterative refinement can undo some of that degradation.

Alas, something goes awry when dimension n gets huge.

The following results for n = 1000 were obtained from a Wintel PC.

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 (20 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 matrix multiplications. These can run enormously faster than v. 4.2's.

But v. 6.5 uses a matrix multiplication subroutine (BLAS 3), programmed by Intel, that spills individual block products, each accumulated to 64 sig. bits, into memory holding only 53. This squanders almost all the advantage of extra-precise accumulation, spoiling residuals while adding negligibly to speed. The consequent loss of 10 sig. bits of ultimate accuracy would have been overlooked if we could compare only computed residuals instead of correct eigenvectors.

Thus does petty optimization for speed become serious pejoration for accuracy.

Stories for Another Day: Ants at a Picnic

- How slow handling of "Rare" Infinities and NaNs messes up parallelism.
- How slow handling of Underflows and Subnormal operands induces unwise flush-to-zero handling of underflows, like the Little Boy Who Was Ignored Wrongly When He Cried "WOLF" Again.

See `http://www.cs.berkeley.edu/~wkahan/Grail.pdf` and `…/ARITH_17U.pdf` .