The Baleful Effect of Computer Benchmarks upon Applied Mathematics, Physics and Chemistry

by

Prof. W. Kahan
Mathematics Dept., and
Elect. Eng. & Computer Sci. Dept.
University of California
Berkeley CA 94720-1776

(510) 642-5638

Updated version of transparencies
first presented in Park City, Utah,
4 Aug. 1995

PostScript file:
`http://http.cs.berkeley.edu/~wkahan/ieee754status/baleful.ps`
For more details see also ···/`ieee754.ps`

The Baleful Effect of Computer Benchmarks upon Applied Mathematics, Physics and Chemistry

Abstract:

An unhealthy preoccupation with Speed, as if it were synonymous with Throughput, has distracted the computing industry and its marketplace from other important qualities that computer hardware and software should possess too ---

Correctness, Accuracy, Dependability, Ease of Use, Flexibility, ⋯

Worse, technical and political limitations peculiar to current benchmarking practices discourage innovations and accommodations of features necessary as well as desirable for robust and reliable technical computation. Particularly exasperating are computer languages that lack locutions to access advantageous features in hardware that we consequently cannot use though we have paid for them. That lack prevents benchmarks from demonstrating the features' advantages, thus affording language implementors scant incentive to accommodate those features in their compilers. It is a vicious circle that the scientific and engineering community must help the computing industry break.

What are Benchmarks ?

Suites of C and Fortran programs, in the custody of industry-acknowledged authorities, available (for a fee) to test the speeds of computers.

Whom are Benchmarks supposed to serve, and how ?

Two constituencies:

1. Designers and Vendors \ / of Computer Hardware and Software,
2. Purchasers and Users / \ especially of Compilers.

Two presumptions:

1. Benchmarks are representative samples of typical workloads.
2. Other things being equal (though they hardly ever are),
   computer systems are rated according to their speeds on benchmarks.

Designers of computer hardware and software " tune " their designs to maximize speed on benchmarks.

Purchasers compare speeds before they buy, presumably, the faster design.

What is wrong with current benchmarks ?

Their presumptions that ...

(Compare the RISC philosophy.)

1. " Higher Speed implies Higher Throughput."
2. " What does not appear in benchmarks does not matter much."

These are over-simplifications, not quite correct.
The belief that they are quite correct causes harm.

How do you choose which computer to buy?

Compare quantifiable features like ...

Price

Speed

Memory sizes: RAM
Disk(s)

Peripherals: Graphics Display
Multi-Media Capabilities
Instrumentation and Signal Processing
Comfortable Keyboard and Mouse

Available Software: ... (a long list) ...

Hardware Reliability and Maintenance

Do no other features matter ? What about details like ...

Accuracy and Range of Floating-Point Arithmetic ... Hardware
... Library

Correct Handling of Floating-Point Exceptions and Special Cases

(Adaptability and Extensibility : Has the future been considered?)

(Ease of Use : How much arbitrariness must be memorized?)

?

(Only Floating-Point features lie within my competency, so I shall not explore the others here.)

Practically all commercially significant North American computer hardware largely conforms to

IEEE Standard 754 for Binary Floating-Point Arithmetic.

The principal exceptions, --
Cray X-MP, Y-MP, *C90*, *J90*, IBM /370, 3090, DEC VAX, --
are mostly passing rapidly from the scene though still commercially significant.

Among conforming computers are these:

... all well-known.

IBM PC's and clones based upon
Intel 386 & 387, 486, Pentium or P6 processors
or clones thereof by Cyrix, IBM, AMD, TI

Apple Macintosh based upon Motorola 68020 + 68881/2 or 68040

…

IBM RS/6000 family, and Power PC - based descendants.
Apple Power Macintosh, based upon Power PC chips too.

Sun Microsystems, formerly based upon M 68020+68881/2,
currently based upon SPARC chips.

Silicon Graphics, now based upon MIPS chips.

DEC Alpha, based upon DEC 21064 and 21164 chips.
Cray T3D, based upon DEC 21064 too.

Hewlett-Packard, based upon PA-RISC chips.

But floating-point arithmetics differ despite IEEE 754.

Which computers have better arithmetics ?

Among IBM, Intel, Apple, Motorola, Sun, SGI, DEC, Cray, H-P, ...

Floating-Point Hardware is intrinsically and substantially
more accurate on some of those computers than on others.

The faster software libraries of
Elementary Transcendental Functions
(exp, log, cos, sin, tan, arctan, ...)
are substantially
more accurate on some of those computers than on others.

For example, while this slide is being prepared,

Transcendental Functions on Intel Pentium and P6, Cyrix ‘87, and on Motorola 68040 are generally 3 dec. more accurate than on the rest.

Next come IBM RS/6000 and the Power PCs, and the Sun SPARCs, and the public-domain library distributed with 4.3 BSD UNIX.

The library that comes with H-P workstations is substantially least accurate.
(According to tests by Vinod K. Stokes in 1993-4.)

Where can you obtain this kind of information ?

Not from Published Benchmarks.

Published Benchmarks
tend to be preoccupied with

speed

to the near exclusion of everything else.

Consequently, the Computer analog of Gresham's Law goes ⋯

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

Wrong ?

Some controversial mathematical conventions are embedded in computers, in hardware and/or in programming languages, and persist only because little commercial incentive exists to expend the considerable effort required to resolve controversy and attend to details that could not affect the speed of current benchmarks.

Example: Why do systems disagree about 35035.0D0 / 15.0 - 7007.0 / 3.0 ?

Example: Why do systems disagree about whether $0.0^{0.0} = 1.0$ or ERROR ?

Nit-Picky Example: What should be done with the sign of ± 0.0 ?

(This example was chosen because a smaller error than the difference between +0 and -0 is hard to imagine; and yet the computing industry appears unable to correct such mistakes, and bigger mistakes too, after they become entrenched. Thus are the sins of the fathers visited upon succeeding generations, all in the name of " Compatibility.")

Where does the sign of ± 0.0 matter ?

Complex Arithmetic

Example: **Define** complex analytic functions

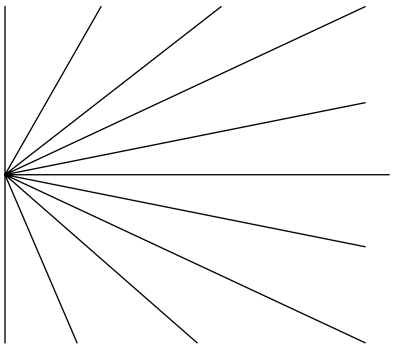
$$g(z) = z^2 + z \cdot \sqrt{z^2 + 1} \quad , \quad \text{and}$$

$$F(z) = 1 + g(z) + \log(g(z)) \quad .$$

Plot the values taken by $F(z)$ as z runs along eleven rays

$$z = r \cdot \mathrm{i} \; , \quad z = r \cdot e^{4i \cdot \pi/10}, \quad z = r \cdot e^{3i \cdot \pi/10}, \quad z = r \cdot e^{2i \cdot \pi/10}, \quad z = r \cdot e^{i \cdot \pi/10}, \quad z = r$$

and their Complex Conjugates, taking positive r from near 0 to near $+\infty$.



The expected picture, called " Borda's Mouthpiece," shows eleven streamlines of an ideal fluid flowing into a channel under such high pressure that the fluid's surface tears free from the inside of the channel.

But a streamline goes astray when the complex functions SQRT(⋯) and LOG(⋯) are implemented, as is customary in Fortran and in libraries currently distributed with C++ compilers, in a way that disregards the sign of ± 0.0 and consequently **violates identities** like

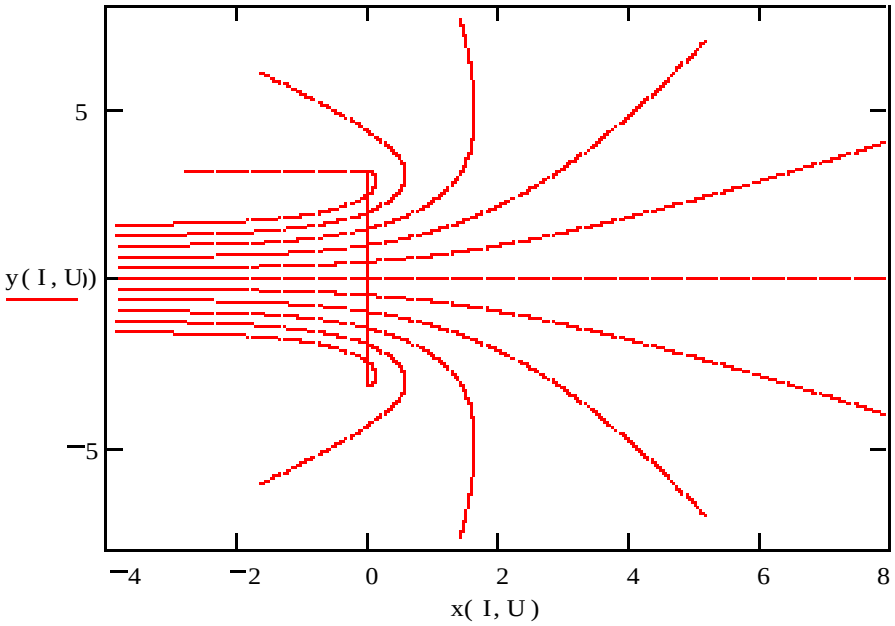
$$\mathrm{SQRT}(\,\mathrm{CONJ}(\,Z\,)\,) \;=\; \mathrm{CONJ}(\,\mathrm{SQRT}(\,Z\,)\,) \qquad \text{and}$$

$$\mathrm{LOG}(\,\mathrm{CONJ}(\,Z\,)\,) \;=\; \mathrm{CONJ}(\,\mathrm{LOG}(\,Z\,)\,)$$

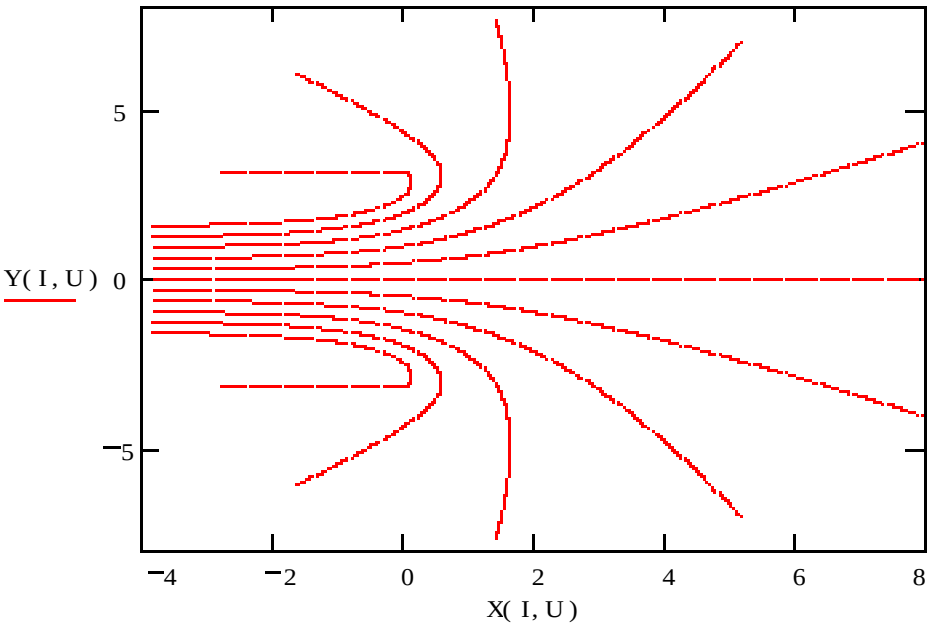
whenever the COMPLEX variable Z takes negative real values.

Pictures of Borda's Mouthpiece come next.

Borda's Mouthpiece, plotted without -0



Borda's Mouthpiece, plotted correctly



This plot shows the streamlines of a flow of an Ideal Fluid under high pressure escaping to the left through a channel with straight horizontal sides. Inside the channel, the flow's boundary is free, not touching the channel walls. Without -0 , the flow along the outside of the lower channel wall is misplotted across the inner mouth of the channel and, though it does not show above, also as a short segment in the upper wall at its inside end. W. Kahan

Why such plots malfunction, and a very simple way to correct them, were explained long ago in my paper

> " Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit," ch. 7 in *The State of the Art in Numerical Analysis* (1987) ed. by M. Powell and A. Iserles for Oxford University Press.

A controversial proposal to incorporate that correction, among other things, in a Complex Arithmetic Extension to the programming language C has been put before ANSI X3J11 , custodian of the C language standard, by Jim Thomas of Taligent and myself. It is controversial because it purports to help programmers cope with physically important discontinuities by suspending a logical proposition,

" x = y " implies " f(x) = f(y) " ,

at certain kinds of discontinuities. However, regardless of that proposal's merits, it is barely worth discussing because ...

> Little incentive exists to incur the costs of corrections (even if principally to documentation) that will not be rewarded by improved performance in current benchmarks and a consequent commercial advantage.

If benchmarks did include the graph-plotting example above, they could not enforce its correctness anyway.

Why not ?

Benchmarks have to be capable of running successfully on *all* commercially significant computers. But older computers, which do not conform to IEEE Standard 754, lack hardware support for - 0.0 , and are therefore intrinsically incapable of plotting Borda's Mouthpiece correctly from the simplest program that would suffice on conforming computers. On nonconforming computers,

"successful" could not mean "correct."

Every Benchmark passes through a sequence of steps:

Benchmark program, written in a standard language like Fortran or C , ...

is submitted to a computer's **Compiler**, ...

which translates that program into the **machine language program** that ...

runs on the **hardware** under test, producing ...

results that are usually disregarded except for the **time** taken to produce them.

The " Computer " that a benchmark tests consists of hardware running some versions of hardware-specific software, namely its
Operating System (e.g. Windows 95, or UNIX) and a
Compiler (e.g. Microsoft C v. 7.0, or GNU-Fortran),
any of which may spoil or obscure the hardware's capabilities.

Advantageous features built into the hardware but inaccessible through the compiler might as well be left out of the hardware.

Example: Inaccessible Floating-Point Accuracy and Range
that you may have paid for but cannot enjoy.
……
This needs some explanation · · ·

Names of Floating-Point Formats:

Single-Precision	`float`	REAL*4
Double-Precision	`double`	REAL*8
Double-**Extended**	`long double`	REAL*10+
(Doubled-Double	`long double`	REAL*16)
(Quadruple-Precision	`long double`	REAL*16)

(Except for the IBM 3090, no current computer supports either of the last two formats fully in its hardware; at best they are simulated in software too slowly to run routinely, so we ignore them.)

Spans and Precisions of Floating-Point Formats :

Format	Min. Normal	Max. Finite	Rel. Prec'n	Sig. Dec.
IEEE Single:	1.2 E-38	3.4 E38	5.96 E-8	6 - 9
IEEE Double:	2.2 E-308	1.8 E308	1.11 E-16	15 - 17
IEEE Extended:	3.4 E-4932	1.2 E4932	5.42 E-20	18 - 21
(Doubled-Double:	2.2 E-308	1.8 E308	≈ 1.0 E-32	≈ 32)
(Quadruple:	3.4 E-4932	1.2 E4932	9.63 E-35	33 - 36)
(IBM hex. REAL*4:	5.4 E-79	7.2 E75	9.5 E-7	≈ 6)
(IBM hex. REAL*8:	5.4 E-79	7.2 E75	2.2 E-16	≈ 15)
(CRAY X-MP etc. REAL*8:	≈ 1 E-2466	≈ 1 E2466	≈ 7 E-15	≈ 14)

Except for Cray X-MP etc., all computers mentioned so far fully support **Single**- and **Double**-Precision floating-point arithmetic in **hardware**.

The following computer chips also support Double-**Extended** in **hardware**:

Intel's 80x86+87, 486, Pentium, P6,
and their clones by IBM, Cyrix, AMD and TI

Intel's 80960KB (found mainly in Embedded Systems like printers)

Motorola's 68020+68881/2, 68040 ... fading.

Motorola's 88110 (very rare)

These chips are designed to evaluate *every* floating-point expression in Double-**Extended** regardless of arithmetic operands' formats.

If you purchased a Macintosh, or NeXT, or Sun III (all 680x0-based), or an Intel-based PC or Cyrix/IBM/AMD/TI-based clone, you paid for the extra precision and range of the hardware's Double-**Extended** format.

Did you actually benefit from it?

This third Double-**Extended** format resembles the unmentionable outcasts of
India (formerly *Untouchables*, now called "Harijan") and of
Japan (formerly called "Etta," now called "Buraku-Min") ;

it is preordained for dirty work.

Its 11 extra bits of precision and 4 extra bits of exponent range are intended
rarely to be seen by most computer users,
but instead to help typical applications programmers look better by rendering their ordinary `double` or REAL*8 results more reliable than might be expected from
usually numerically naive programmers.

This **Extended** format is designed to be used, with negligible loss of speed, for all but the simplest arithmetic with `float` and `double` operands. For example, it should be used for scratch variables in loops that implement recurrences like

polynomial evaluation, scalar products, partial and continued fractions.

It often averts premature Over/Underflow or severe local cancellation that can spoil simple algorithms.

Without an **Extended** format, ...

- some ostensibly straightforward `double` computations are prone to malfunction unless carried out in devious ways known only to experts;

- matrix computations upon vast arrays of `double` data degrade too rapidly as increasing dimensions engender worsened roundoff.

The idea of an **Extended** format has been amply vindicated by its use in Hewlett-Packard's financial calculators, which all perform all arithmetic and financial functions to three more sig. decimals than they store or display. Doing so has helped to earn the HP-12C a deserved reputation for dependability that has kept it prominent for over 11 years in a market where other electronic products enjoy a lifetime shorter than a Mayfly's.

Among 680x0-based Macintosh, NeXT and Sun III, and Intel-based PCs , all of which contain Double-**Extended** floating-point hardware,

ONLY the Macintosh's compilers routinely supported Double-**Extended** via the **S.A.N.E.** (**S**tandard **A**pple **N**umerical **E**nvironment) ;
see Apple *Numerics Manual, Second Edition* (1988) Addison-Wesley, Mass.

Owners of other Double-**Extended** hardware were denied their just deserts by ...

Crippled Compilers:

No compilers for the old Sun III family, based upon Motorola's 68020+68881/2, ever supported its Double-**Extended** format, so that has atrophied. Current Sun SPARC hardware supports only Single and Double.

No commercially significant **Fortran** compiler for Intel-based PCs supports their Double-**Extended** format; and only Borland's and Microsoft's C / C++ compilers support it, the latter only grudgingly. Other C / C++ compilers **ignore** " `long double` " or else treat it as if it were merely " `double`."

No benchmark programs exercise Double-**Extended** since it is unavailable on many workstations, and since " `long double` " has no well-defined meaning.

Therefore, little incentive exists to incur the costs of supporting Double-**Extended** fully since that effort will not be rewarded by improved performance in current benchmarks and a consequent commercial advantage.

And yet, despite a general lack of support, Double-**Extended** confers a detectable advantage upon computers that have it in their hardware. This advantage would be obvious to everybody if the computing community ran

Benchmarks to Test Range, and

Benchmarks to Test Accuracy.

What kinds of calculations tax Range ?

1. Three-Term Recurrences

$$P_{n+1}(x) := a_n(x)\cdot P_n(x) - b_n(x)\cdot P_{n-1}(x)$$

are used to compute Orthogonal Polynomials, Bessel Functions, Spherical Harmonics, and many others of the transcendental functions of Mathematical Physics. Their values usually transgress the ranges of DEC VAX and IBM hex. arithmetics ($10^{\pm 38}$ and $10^{\pm 79}$), often transgress the range $10^{\pm 308}$ of IEEE 754 Double, almost never transgress the range $10^{\pm 4930}$ of Extended. Programs that would work well with Extended would sometimes crash with Double, and often crash with DEC VAX or IBM hex. Double.

A program that crashes commercially significant machines would not be acceptable to their custodians as a benchmark.

Crashes can be precluded by *Scaling* the recurrences, at the cost of defensive tests and branches. Defensive code wastes time since it must wait for every test though it rarely branches. Benchmarks that obliged a machine with narrower range to preclude crashes that way would be even more objectionable to its custodian if competing machines with wider range were allowed to omit defensive code and therefore run faster.

2. Every computer's range is taxed by **Quotients of Prolonged Products** like

$$Q = \frac{(a_1 + b_1)\cdot(a_2 + b_2)\cdot(a_3 + b_3)\cdot(\ldots)\cdot(a_N + b_N)}{(c_1 + d_1)\cdot(c_2 + d_2)\cdot(c_3 + d_3)\cdot(\ldots)\cdot(c_M + d_M)}$$

when N and M are huge and when the numerator and/or denominator are likely to encounter premature OVER/UNDERFLOW even though the final value of Q would be unexceptional if it could be computed. This situation arises in certain otherwise attractive algorithms for calculating eigensystems, or Hypergeometric series, for example.

The hardware of IBM hex. and of machines that conform to IEEE 754 can easily compute Q accurately and quickly, and so can other machines with some fiddling; but compiled programming languages lack the necessary locutions. (See Ch. 2 of *Floating-Point Computation* P.H. Sterbenz (1974) Prentice-Hall, N.J., for a brief description of how it was done in the 1960s on an IBM 7094.) Therefore computations like Q cannot figure in a benchmark for range.

Qualifications for Benchmarks to Test Accuracy.

Some formidable political and technical obstacles must be overcome if accuracy is to figure in benchmarks besides speed :

1. Compilers generally must support a reasonable consensus about the meanings of different precision specifications if these are to figure in benchmarks.

No such consensus is in sight yet, so let us try to get along without it for a while. In other words, by not mentioning " Double-Extended " nor " REAL*10 " nor " `long double`," at least not for the time being, an accuracy benchmark can be eligible to run on every commercially significant computer of interest to us.

2. A benchmark must be realistic enough to deserve serious attention.

It must perform a task typical of tasks somebody may plausibly need performed repeatedly; and accuracy should be an important aspect of the task.

3. Benchmarks must avoid the computational counterpart of the ...

Stopped Clock Paradox: Why is a mechanical clock more accurate stopped than running? A running clock is almost never exactly right, whereas a stopped clock is exactly right twice a day.
(*But WHEN is it right? Alas, that was not the question.*)

To avoid this, we must avoid results that an inferior computer might get exactly right although superior computers get merely excellent approximations. For instance, if the perfect result were 0.5 , it might be obtained *exactly* by accident using only low-precision floating-point while higher precision got something " infinitely worse " like 0.499999999999999 .

4. Input data should be composed from simple integers and fractions that will not be mishandled by the compiler's Decimal-Binary conversion, which might otherwise alter the data before it reached the floating-point hardware under test.

Such alteration could cause the benchmark to disparage hardware that got the right answer for the wrong question. Worse, tiny changes to data critically contrived to expose a weakness might thwart that intent. For instance, a critical datum 94906267.0 treated as a REAL*4 or `float` constant would be changed to an uninformative 94906264.0 if not rewritten as 94906267.0 D0 or, better, expressed as 6847*DBLE(13861) using only small integers we expect to use safely.

Solving the Quadratic Equation

$$p \cdot x^2 - 2 \cdot q \cdot x + r = 0 \ .$$

This illustrates the hazards that beset an accuracy benchmark.

The roots x_1 and x_2 will be computed using a " stable " numerical procedure:

```
Qdrtc( p , q , r , x1 , x2 ) :
    s := SQRT( q·q - p·r ) ;
    If  q > 0  then  t := q + s
               else  t := q - s ;
    x1 := r/t  ;    x2 :=  t/p .
```

Data will NOT be chosen at random. In practice, coefficients p, q, r are often correlated; and that is the kind of data that will be supplied *exactly* here:

For each chosen datum $r >> 1$, set $q := r - 1$ and $p := q - 1$.

Consequently the roots are known to be $x_1 = 1$ and $x_2 = 1 + 2/p$ *exactly.* These can be compared with the roots computed in floating-point by the procedure `Qdrtc` above, and the worst errors detected will shed light upon the intrinsic accuracy of the computer's floating-point arithmetic.

A benchmark program `Qtest`, combining `Qdrtc` with a battery of fifteen values r chosen maliciously to reveal the worst errors possible on various computers, was prepared for them. Results are tabulated below.

(The battery of trial values r and the details of the program `Qtest` can be found in my *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*, accessible by electronic mail from my home page:
`http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps` .)

Results from Qtest(Qdrtc) on 8-byte Floating-Point

Computer Hardware	Software System	Precision sig. bits	Accuracy sig. bits	How far < 1 sig. bit
ix86/87- & Pentium-based PCs	Fortran, C, Turbo-Basic, Turbo-Pascal	53	32	33.3
680x0-based Sun III, Macintosh	Fortran, C			
DEC VAX D	Fortran, C	56	28	29.3
ix86/87 & Macintosh	MATLAB, MathCAD	53	26.5	27.8
SGI MIPS, SPARC, HP, DEC VAX G DEC Alpha	Fortran, C, MATLAB			
IBM /370 etc.	Fortran, C	56	26.4	26.4
CRAY Y-MP	Fortran, C	48	24	25.3
PowerPC/Mac, IBM RS/6000	Fortran, C	53	NaN from √(< 0)	NaN from √(< 0)

Precision = how many sig. bits are stored in the named system's 8-byte format.
(Different systems trade off precision and range differently.)

Accuracy = fewest sig. bits delivered by `Qdrtc` over the whole test battery.
(Evidently as many as half the sig. bits stored in computed roots can be wrong.)

The smaller computed root can fall short of 1.0 in the sig. bit whose position is tabulated last. (In the absence of roundoff, no root would fall below 1.0 .)

These findings cry out for Explanations: How can so simple a program get worse accuracy on some computer systems than on others that store the same number of significant bits or fewer?

Explanations:

Best accuracy, 32 sig. bits, is achieved on inexpensive ix86/87-based PCs and 680x0-based Macintoshes by software that evaluates each subexpression to 64 sig. bits by default in their Extended registers though it be rounded to 53 sig. bits in Double when stored.

(These computer systems also accept, without premature Over/Underflows, a far wider range of input data {μ·p, μ·q, μ·r} than do the others, though this robustness cannot be explored by `Qtest` without crashing some other systems upon Over/Underflow .)

Why do MATLAB and MathCAD achieve no better accuracy on ix86/87 and 680x0 platforms with Extended registers than on the other machines without?

These programs are written mostly in C in a purportedly portable fashion with no mention of `long double`, so they store almost every subexpression into `double` scratch variables, thereby wasting time as well as the Extended registers' superior accuracy and range.

Why do IBM's /370 and 3090 etc. do worse than the DEC VAX D format, though both store the same number 56 of sig. bits?

IBM's notorious old Hexadecimal floating-point format is intrinsically as much as three sig. bits less accurate than a Binary format of the same width.

Explanations, continued:

Whence comes **NaN** (**N**ot **a N**umber) on RS/6000s and PowerPC/Macs ?.

It arises from the square root of a negative number $q \cdot q - p \cdot r$.

However, tests performed upon input data would find that $QQ := q \cdot q$ and $PR := p \cdot r$ do satisfy $QQ \geq PR$ whenever `Qtest`'s $q \cdot q - p \cdot r < 0$.

This paradox arises out of the *Fused Multiply-Accumulate* instruction possessed by these machines. They can compute expressions like

$$\pm x \pm y \cdot z$$

in one operation with just one rounding error instead of two. This is faster and usually more accurate than separately rounded multiply and add operations, but sometimes less accurate, so it should not be used indiscriminately.

Is $\pm x = q \cdot q$ rounded and $\pm y \cdot z = p \cdot r$, or is $\pm x = -p \cdot r$ rounded and $\pm y \cdot z = q \cdot q$?

The paradox can be avoided by inhibiting Multiply-Accumulate at compile time.

Alas, doing so generally would slow these machines; therefore, their compiler was designed to render that inhibition inconvenient and unusual, thereby achieving better speeds on benchmarks that lack locutions to enable or disable a Multiply-Accumulate.

Accuracy benchmark `Qtest` could run successfully on these machines, getting the same mediocre results as do MIPS, SPARC, HP, DEC VAX G and Alpha, if run in their unusual and slower Multiply-Accumulate-inhibited mode.

Would that be considered a *fair test* ?

Fairness raises troublesome issues for a benchmark.

What if custodians of a computer family allege *Unfairness* ? Letting them tweak a benchmark slightly to render it " *fair* " lets them overcompensate in devious ways very difficult to expose. For example, replace `Qdrtc` by an ostensibly algebraically equivalent procedure …

```
PPCQdrtc( p ,  q ,  r ,   x1 ,  x2 ) :
    ß := p·r ;  ø := p·r - ß ;                    |
    s := SQRT( (q·q - ß) - ø ) ;                  |
    If  q > 0  then  t := q + s
               else  t := q - s ;
    x1 := r/t  ;    x2 :=  t/p .
```

For comparison, here is the original ...

```
Qdrtc( p ,  q ,  r ,   x1 ,  x2 ) :
    s := SQRT( q·q - p·r ) ;                      |
    If  q > 0  then  t := q + s
               else  t := q - s ;
    x1 := r/t  ;    x2 :=  t/p .
```

Aside from running slightly longer to compute ø , which just vanishes for most computer arithmetics, `Qtest(PPCQdrtc)` differs from `Qtest( Qdrtc )` only by awarding the prize for accuracy to PowerPC and RS/6000, which get 53 correct sig. bits instead of NaN from `PPCQdrtc` .

> Which of `Qtest( PPCQdrtc )` and `Qtest( Qdrtc )`
> do you deem the fairer assessment of computers' accuracies ?

Of course, `Qdrtc` could be replaced by a different yet ostensibly algebraically equivalent procedure `EQdtrc` devised to deliver 53 correct sig. bits only on machines with Extended registers (but without mentioning " Extended ") and to match `Qdrtc` on all other machines.

Dilemma: To insist that a benchmark exist in just one version, and that it run successfully (no NaNs `!`) on *every* computer, may cripple speed or accuracy or robustness on computers with advantageous features others lack. But to permit local variations may permit skulduggery that invalidates comparison.

As it is now, `Qtest( Qdrtc )` tells us something I think worth knowing regardless of whether it is admitted to the ranks of industry-approved benchmarks.

Solving quadratic equations is not generally regarded as so IMPORTANT a computation that anyone would pay big bucks for a better way. As a benchmark it would not likely be taken seriously.

What computations are both important and technically challenging enough that they could earn real money if accomplished significantly better?

1. Solving big systems of linear equations: $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$.

2. Computing eigenvalues/vectors: $\mathbf{X}^{-1}\cdot\mathbf{A}\cdot\mathbf{X}$ = diagonal.

Despite phenomenal improvements in numerical methods over the past three or four decades, we still lack software that will always solve these problems as accurately as their data deserve.

For instance, solving $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$ can still run afoul of certain *pathologies*:

Gargantuan dimension.
Unfortunate column ordering —> poor pivot choice.
Disparate scaling of rows —> poor pivot choice.
Systematically severe ill-condition (near singularity).

One way to ameliorate such pathologies is to follow Gaussian elimination by Iterative Refinement, which is believed to cope with them. But that is not the whole story: ....

Roundoff Degrades an Idealized Cantilever

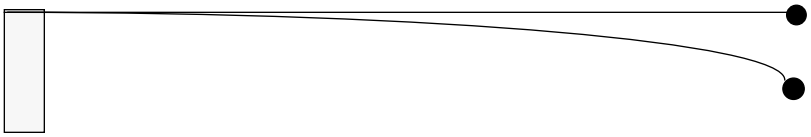
Prof. W. Kahan and Ms. Melody Y. Ivory
Elect. Eng. & Computer Science Dept. #1776
University of California
Berkeley CA 94720-1776

Abstract:
By far the majority of computers in use to-day are Intel-based PCs, and a big fraction of the rest are old 680x0-based Apple Macintoshes. Owners of these machines are mostly unaware that their floating-point arithmetic hardware is capable of delivering routinely better results than can be expected from the more prestigious and more expensive workstations preferred by much of the academic Computer Science community. This work attempts to awaken an awareness of the difference in arithmetics by comparing results for an idealized problem not entirely unrepresentative of industrial strength computation. The problem is to compute the deflection under load of a discretized approximation to a horizontally cantilevered steel spar. Discretization generates N simultaneous linear equations that can be solved in time proportional to N as it grows big, as it must to ensure physical verisimilitude of the solution. The solution is programmed in MATLAB which, like most computer languages nowadays, lacks any way to mention those features that distinguish better arithmetics from others. None the less this program yields results on PCs and old Macs correct to at least 52 sig. bits for all values N tried, up to N = 18827 on a Pentium. However the other workstations yield roughly 52.3 - 4.67 log N correct sig. bits from the same program despite that it tries two styles of Iterative Refinement; at N = 18827 only a half dozen bits are left. This kind of experience raises troublesome questions about the coverage of popular computer benchmarks, and about the prospects for a would-be universal language like JAVA to deliver identical numerical results on all computers from one library of numerical software.

The MATLAB program used to get the aforementioned results is available by electronic mail from the authors: ivory@cs.berkeley.edu and wkahan@cs... .

Roundoff Degrades an Idealized Cantilever

A uniform steel spar is clamped horizontal at one end and loaded with a mass at the other. How far does the spar bend under load?



The calculation is discretized: For some integer N large enough (typically in the thousands) we compute approximate deflections

$$x_0 = 0\ ,\quad x_1,\ x_2,\ x_3,\ \ldots,\ x_{N-1},\quad x_N \approx \text{deflection at tip}$$

at uniformly spaced stations along the spar. Discretization errors, the differences between these approximations and true deflections, tend to 0 like $1/N^2$. These x_j 's are the components of a column vector $\mathbf{x}$ that satisfies a system $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$ of linear equations in which column vector $\mathbf{b}$ represents the load (the mass at the end plus the spar's own weight) and the matrix $\mathbf{A}$ looks like this for $N = 10$:

$$A = \begin{bmatrix} 9 & -4 & 1 & o & o & o & o & o & o & o \\ -4 & 6 & -4 & 1 & o & o & o & o & o & o \\ 1 & -4 & 6 & -4 & 1 & o & o & o & o & o \\ o & 1 & -4 & 6 & -4 & 1 & o & o & o & o \\ o & o & 1 & -4 & 6 & -4 & 1 & o & o & o \\ o & o & o & 1 & -4 & 6 & -4 & 1 & o & o \\ o & o & o & o & 1 & -4 & 6 & -4 & 1 & o \\ o & o & o & o & o & 1 & -4 & 6 & -4 & 1 \\ o & o & o & o & o & o & 1 & -4 & 5 & -2 \\ o & o & o & o & o & o & o & 1 & -2 & 1 \end{bmatrix}$$

The usual way to solve such a system of equations is by Gaussian elimination, which is tantamount to first factoring $\mathbf{A} = \mathbf{L}\cdot\mathbf{U}$ into a lower-triangular $\mathbf{L}$ times an upper-triangular $\mathbf{U}$, and then solving $\mathbf{L}\cdot(\mathbf{U}\cdot\mathbf{x}) = \mathbf{b}$ by one pass of forward substitution and one pass of backward substitution. Since $\mathbf{L}$ and $\mathbf{U}$ each has only three nonzero diagonals, the work goes fast; fewer than $30\cdot N$ arithmetic operations suffice. But this solution $\mathbf{x}$ is very sensitive to rounding errors; they can get amplified by the *condition number* of $\mathbf{A}$, which is of the order of N^4 .

To assess the effect of roundoff we compare this computed solution $\mathbf{x}$ with another obtained very accurately and very fast with the aid of a trick: There is another triangular factorization $\mathbf{A} = \mathbf{R}\cdot\mathbf{R}^T$ in which $\mathbf{R}$ is an upper-triangle with three nonzero diagonals containing only small integers 1 and ± 2 . Consequently the desired solution can be computed with about $4\cdot N$ additions and a multiplication. Such a simple trick is unavailable for realistic problems.

The loss of accuracy to roundoff during Gaussian elimination poses a Dilemma:

Discretization error —> 0 like $1/N^2$, so for realistic results we want N big.

Roundoff is amplified by N^4 , so for accurate results we want N small.

For realistic problems (aircraft wings, crash-testing car bodies, ...), typically $N > 10000$. With REAL*8 arithmetic carrying the usual 53 sig. bits, about 16 sig. dec., we must expect to lose almost all accuracy to roundoff occasionally.

Iterative Refinement mollifies the dilemma:
Compute a *residual* $\mathbf{r} := \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ for $\mathbf{x}$. Solve $\mathbf{A}\cdot\Delta\mathbf{x} = \mathbf{r}$ for a correction $\Delta\mathbf{x}$ using the same program (and triangular factors $\mathbf{L}$ and $\mathbf{U}$) as "solved" $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$ for an $\mathbf{x}$ contaminated by roundoff. Update $\mathbf{x} := \mathbf{x} - \Delta\mathbf{x}$ to refine its accuracy.

Actually, this Iterative Refinement as performed on the prestigious work-stations (IBM RS/6000, DEC Alpha, Convex, H-P, Sun SPARC, SGI-MIPS, ...) does not necessarily refine the accuracy of $\mathbf{x}$ much though its residual $\mathbf{r}$ may get much smaller, making $\mathbf{x}$ look much better to someone who does not know better.

Only on Intel-based PCs and 680x0-based Macintoshes (not Power-Macs) can Iterative Refinement *always* improve the accuracy of $\mathbf{x}$ substantially *provided* the program is not prevented by a feckless compiler from using the floating-point hardware as it was designed to be used:

Accumulate residual $\mathbf{r} := \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ in the computer's REAL*10 registers. They carry 11 more bits of precision than REAL*8's 53 sig. bits. Using them improves accuracy by at least about 11 sig bits whenever more than that were lost.

To get comparable or better results on the prestigious workstations, somebody would have to program simulated (SLOW) extra-precise computation of the residual, or invent other tricks.

e.g.: Accuracies from a MATLAB program (WITH ***NO MENTION*** of REAL*10)

N = 18827	PCs & 680x0 Macs	Others	Condition no. > 2^{57}
Unrefined Residual	156 ulps.	≈156 ulps.	Why N = 18827 ? Because for bigger N MATLAB's Stack Overflowed on a Pentium with 64 MB RAM .
Refined Residual	0·41 ulps.	≈0·7 ulps.	
Unrefined Accuracy	6 sig. bits	≈6 sig. bits	
Refined Accuracy	53 sig. bits	≈5 sig. bits	

The foregoing tabulated results are misleading because they compare results from the *same* MATLAB program run on *different* computers, which is exactly how current benchmarks are expected to assess different computers' comparative merits. But this refinement program would probably not exist if the only computers on which it had to run were prestigious workstations that lack fast extended-precision; on those computers, iterative refinement is best performed in a different way. The difference is subtle and yet important, if only because it raises questions about a popular notion, promulgated especially by *JAVA* enthusiasts, that software ought to work identically on every computer.

Every iterative refinement program repeats the three steps

{ **r** := **A**·**x** - **b** ; *solve* **A**·Δ**x** = **r** for Δ**x** ; *update* **x** := **x** - Δ**x** ; }

until something stops it The programs most in use nowadays, like _GERFS in LAPACK, employ an **r**-based stopping criterion:

> Stop when the residual **r** no longer attenuates, or when
> it becomes acceptably small, whichever occurs first.

Usually the first **x** , if produced by a good Gaussian elimination program, has an acceptably small residual **r** , often smaller than if **x** had been obtained from $\mathbf{A}^{-1}\mathbf{b}$ calculated exactly and then rounded off to full REAL*8 precision! Therefore, that criterion usually inhibits the *solve* and *update* operations entirely.

What if **r** is initially unacceptably big? This can occur, no matter whether **A** is intrinsically ill conditioned, because of some other rare pathology like gargantuan dimension N or disparate scaling. Such cases hardly ever require more than one iteration to satisfy the foregoing criterion. That iteration always attenuates **x** 's inaccuracy too, but only on PCs and Macs that accumulate **r** extra–precisely.

On workstations that do not accumulate **r** extra-precisely, updating **x** often degrades it a little and almost never improves it much unless inaccuracy in **x** is caused initially by one of those rare pathologies other than intrinsic ill-condition.

Thus, the **r**-based stopping criterion serves these workstations well by stopping them as soon as they have achieved a goal appropriate for them, namely ...

> Locate an approximate solution **x** whose *computed*
> residual **r** := **A**·**x** - **b** will not much exceed the roundoff
> that may accrue while it is being computed.

Such an approximate **x** may still be very inaccurate; this happens just when **A** is intrinsically "ill-conditioned," which is what we say to blame inaccuracy upon the data instead of our own numerical (in)expertise.

Reconsider now the results tabulated earlier for the cantilever with $N = 18827$. A smaller N would do as well; for all of them, $\mathbf{x}$ is accurate to 52 or 53 sig. bits after refinement on a PC or 680x0-based Mac. How do these machines achieve accuracy to the last bit or two in the face of condition numbers so huge that the survival of any sig. bits at all surprises us? Not by employing an $\mathbf{r}$-based stopping criterion; it would too often terminate iteration prematurely.

The stopping criterion employed to get those results is $\mathbf{x}$-based:

> Stop when decrement $\Delta\mathbf{x}$ no longer attenuates, or when it becomes acceptably small, whichever occurs first.

To get those results, "acceptably small" here was set to zero, which seems rather tiny but shows what the program can do. At $N = 18827$ the cost of those 53 sig. bits was 10 iterations of Iterative Refinement; at lesser dimensions N the cost was roughly $1/(1 - 0.091 \log N)$ iterations, which suggests that dimensions N beyond 55000 (with condition numbers $> 2^{63}$) lie beyond the program's reach.

> This $\mathbf{x}$-based stopping criterion that so enhances the accuracy of results from PCs and 680x0-based Macs must not be employed on other workstations lest it degrade the accuracy of their results and, worse, waste time.

Different strokes for different folks.

. . . . . .

How relevant is this *idealized* cantilever problem to more general elastostatic problems whose coefficient matrices $\mathbf{A}$ generally do not have entries consisting entirely of small integers? Small integers make for better accuracy from a simpler program, but they are not essential. What is essential is that we preserve important *correlations* among the many coefficient entries, which are determined from relatively few physically meaningful parameters, despite roundoff incurred during the generation of those entries. Such a correlation is evident in the example explored here; all but the first two row-sums of $\mathbf{A}$ vanish, as do row–sums for a non–uniform cantilever whose matrix $\mathbf{A}$ has varying rows of non-integer coefficients. We must force the same constraint, among others, upon the rounding errors in $\mathbf{A}$, and then they will do us little harm.

But the rounding errors incurred later during Gaussian elimination cannot be so constrained. Though tiny, they become dangerous when amplified by big condition numbers. Thus we are compelled either to attenuate them by employing inverse iteration with extra-precise residuals, or to devise other tricks that do not incur such dangerous errors.

If you do not know how much Accuracy you have, what good is it?
Like an expected inheritance that has yet to "mature," you can't bank on it.

Iterative refinement programs like _GERFS that employ the **r**-based stopping criterion can also provide, at modest extra cost, an almost–always–over–estimate of the error in **x** . They do so by first computing a majorizer **R** that dominates **r** := **A**·**x** - **b** plus its contamination by roundoff. Then they estimate $||\mathbf{A}^{-1}\mathbf{R}||_\infty$ without ever computing $\mathbf{A}^{-1}$ to obtain the desired bound upon error in **x** . This estimate costs little more than a few steps of Iterative Refinement.

Unfortunately, it is not infallible, though serious failures (gross under–estimates of the error in **x**) must be very rare since the only known instances are deftly contrived examples with innocent–looking but singular matrices **A** . Worse, this error bound tends to be grossly pessimistic when **A** is very ill-conditioned and/or its dimension N is extremely big. The pessimism often amounts to several orders of magnitude for reasons not yet fully understood.

In short, versions of Iterative Refinement working on prestigious workstations can provide error bounds but they are too often far too pessimistic, and they can fail.

Iterative Refinement programs that employ the **x**-based stopping criterion can also provide, at no extra cost, an almost–always–over–estimate of the error in **x** . They do so by keeping track of $||\Delta\mathbf{x}||$ which, if convergence is not too slow, gives a fair (rarely much too pessimistic) indication of the error in **x** . This is not an infallible indication; it fails utterly whenever the computed residual

r := (**A**·**x** - **b** plus roundoff)

happens to vanish. Iterative Refinement produces residuals that vanish surprisingly often, sometimes because **x** is exactly right.

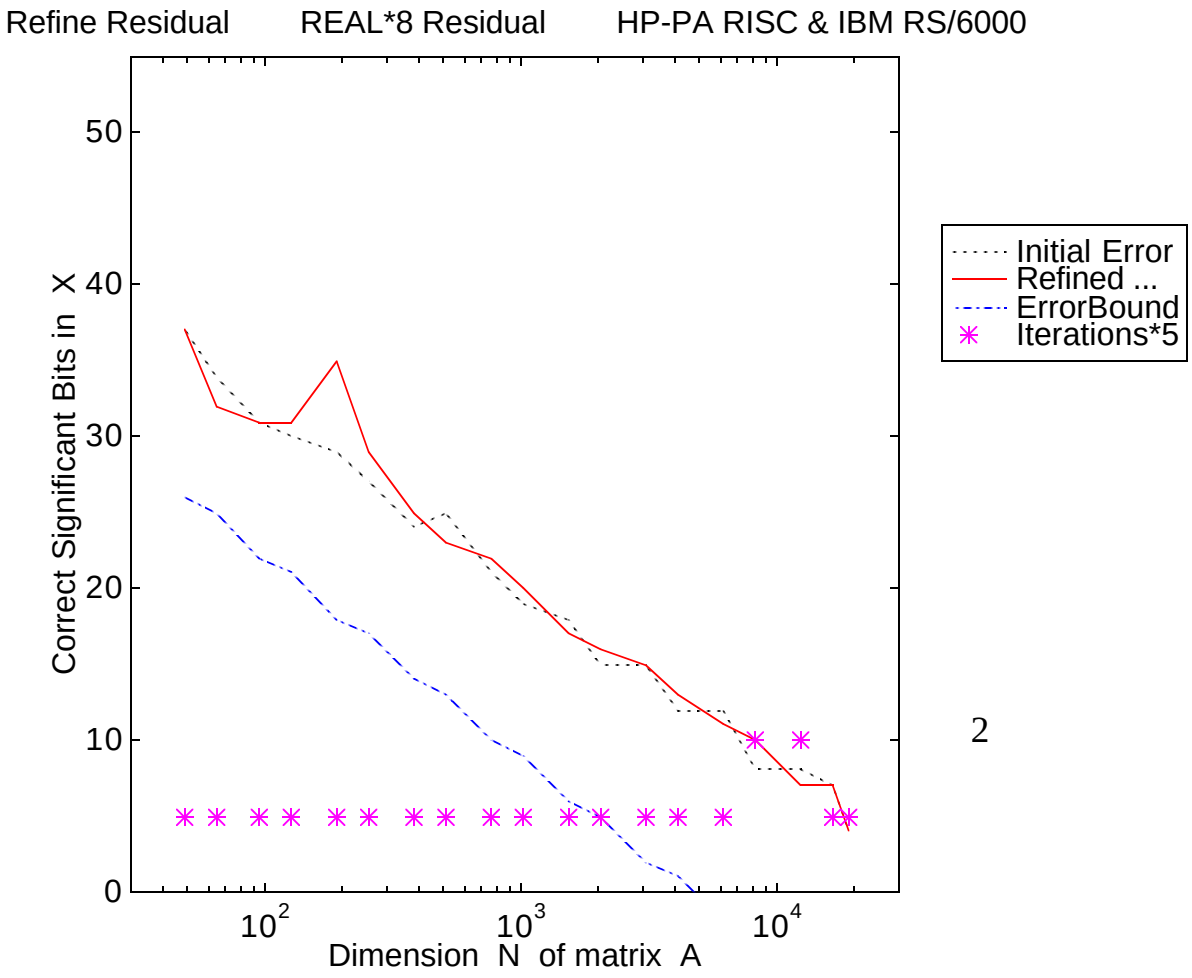
(The INEXACT flag mandated by IEEE Standard 754 for Binary Floating–Point Arithmetic would, if MATLAB granted us access to that flag, help us discriminate between solutions **x** that are exactly right and those, perhaps arbitrarily wrong, whose residuals vanish by accident.)

In short, Iterative Refinement appropriate for PCs and 680x0-based Macs comes with a cost-free indication, usable if hardly infallible, of its superior accuracy.

The other workstations have nothing like it.

The following figures exhibit some evidence to support the foregoing claims.

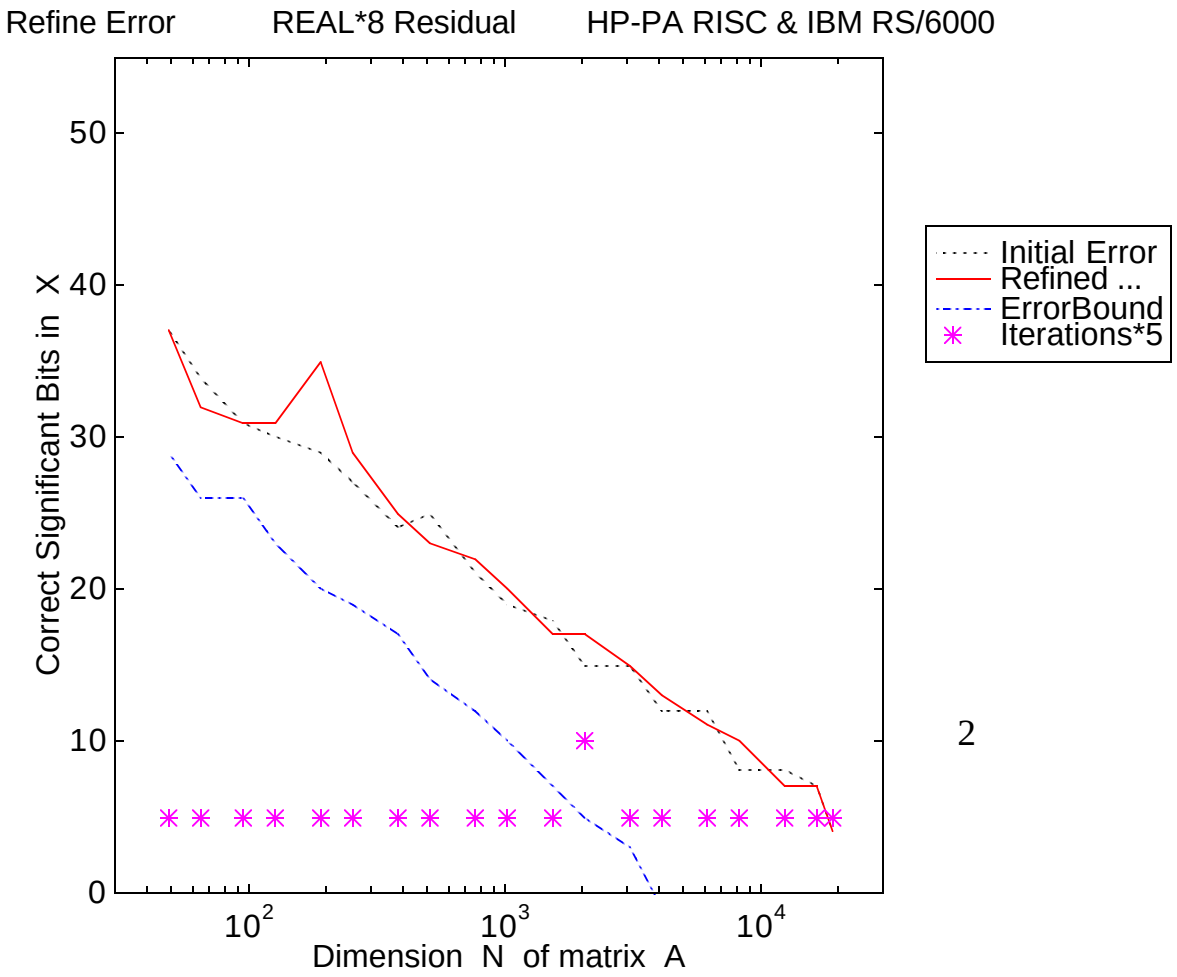
ACCURACY
of a Cantilever's Deflection after Iterative Refinement by a MATLAB 4.2 program run on workstations



2

Iterative Refinement of residuals **r** (employing the **r**-based stopping criterion), as does LAPACK program _GERFS, always reduces the residual **r** below an ulp or two, but rarely improves the accuracy of the solution **x** much, and often degrades it a little, on workstations that do not accumulate residuals to extra precision. And the error-bound on **x** inferred from **r** is too pessimistic. But on those workstations it is difficult to do better.

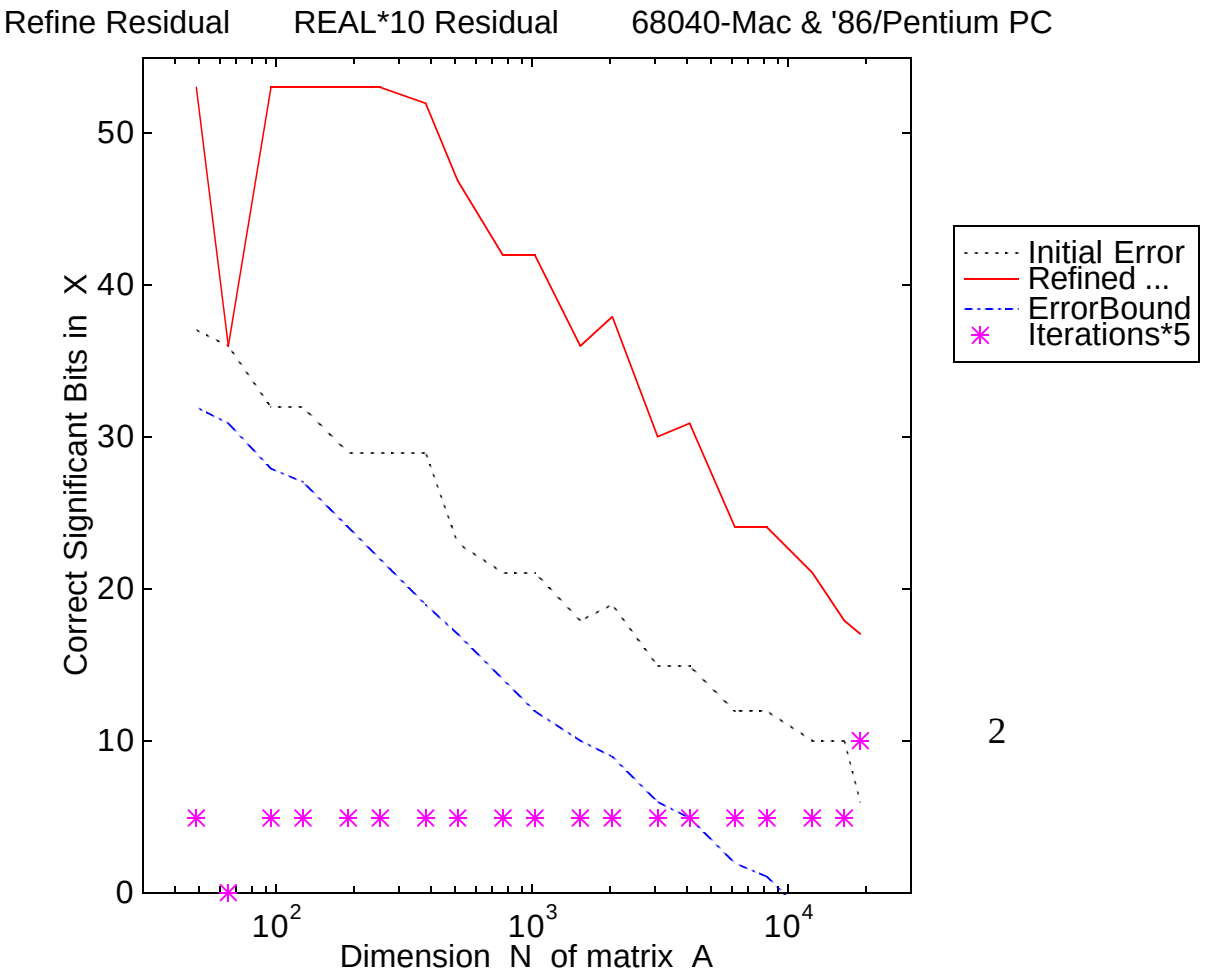
ACCURACY
of a Cantilever's Deflection after Iterative Refinement by a MATLAB 4.2 program run on workstations



2

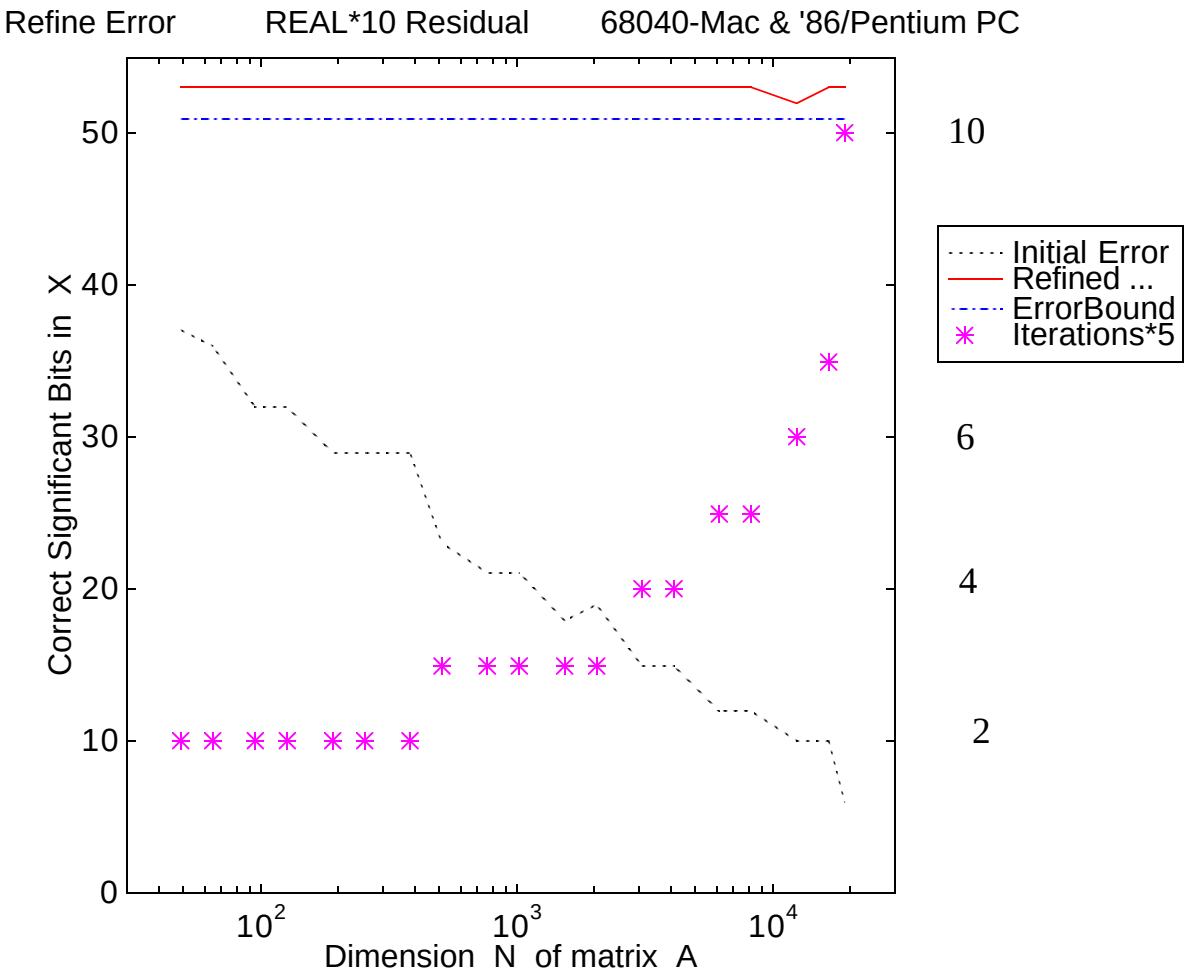
Iterative Refinement of solutions **x** (employing the **x**-based stopping criterion) is no more accurate than refinement of **r** for the Cantilever problem (and rarely more accurate for other problems) on workstations that do not accumulate residuals to extra precision. And the error-bound on **x** inferred from $\Delta\mathbf{x}$ is still too pessimistic for this problem (and too optimistic for others). Worse, refining **x** usually takes more iterations than refining **r** , though not for cases shown here. Therefore this kind of Iterative Refinement does not suit those workstations.

ACCURACY
of a Cantilever's Deflection after Iterative Refinement by a MATLAB 4.2 program run on PCs and old Macs



Iterative Refinement of residuals **r** (employing the **r**-based stopping criterion), as does LAPACK program _GERFS, always reduces the residual **r** below an ulp or two, and also improves the accuracy of the solution **x** if not stopped too soon (as occurred above at N = 64 because the initial **r** was below 1 ulp) on PCs and Macs that accumulate residuals to extra precision. But the error-bound on **x** inferred from **r** is still too pessimistic. On these computers we can do better.

ACCURACY
of a Cantilever's Deflection after Iterative Refinement by a MATLAB 4.2 program run on PCs and old Macs



Iterative Refinement of solutions **x** (employing the **x**-based stopping criterion) far surpasses the accuracy of refinement of **r** for ill–conditioned Cantilever problems (and also for other problems) on PCs and Macs that accumulate residuals to extra precision. And the error-bound on **x** inferred from $\Delta\mathbf{x}$ is satisfactory for this problem (and almost always for others). Of course, the required number of iterations rises sharply as **A** approaches singularity. Still, this kind of Iterative Refinement is the right kind for those popular computers.

Would the Cantilever problem make a good benchmark?

Perhaps not. Since different families of computers are best served by different versions of Iterative Refinement with different capabilities, like rather different kinds of error over–estimates, comparisons would become confounded.

A good bench mark has to be a single program that does something worth–while on every computer even if it does better on some of them.

I have devised such a program: **RefinEig**.

RefinEig -- towards a better benchmark for accuracy:

For any square matrix B the *MATLAB* statement

```
[Q, V] = eig( B )
```

computes an eigenvector matrix Q and a diagonal matrix V of eigenvalues.

Ideally, $V = Q^{-1} \cdot B \cdot Q$ is diagonal.

Numerical accuracy deteriorates as B approaches a set of measure zero, the algebraic variety of *Defective* matrices B , on which V cannot be diagonal.

No *single* algorithm can compute Q and V as accurately as deserved by *every* datum B , if a theorem proved recently by Ming Gu at Berkeley can be taken at face value.

Therefore **`eig(...)`** must be imperfect;
and it is, as examples will demonstrate.

Examples: Werner Frank's NxN matrices, exemplified here for N = 5 :

$$F = \begin{bmatrix} 5 & 4 & 3 & 2 & 1 \\ 4 & 4 & 3 & 2 & 1 \\ o & 3 & 3 & 2 & 1 \\ o & o & 2 & 2 & 1 \\ o & o & o & 1 & 1 \end{bmatrix} \qquad F' = \begin{bmatrix} 5 & 4 & o & o & o \\ 4 & 4 & 3 & o & o \\ 3 & 3 & 3 & 2 & o \\ 2 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 1 & o & o & o \\ 1 & 2 & 2 & o & o \\ 1 & 2 & 3 & 3 & o \\ 1 & 2 & 3 & 4 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

F' is obtained by transposing, and *P* by reversing rows and columns of *F* .

F , *F'*, *P* and *P'* have the same eigenvalues, all positive in reciprocal pairs.
If f is an eigenvalue, so is $1/f$, and then $\sqrt{f} - 1/\sqrt{f}$ is a zero of the N^{th} Hermite polynomial.

> The smaller eigenvalues are the more ***ill-conditioned*** (i.e. sensitive to perturbation), exponentially more so for bigger N , the same for all four of *F* , *F'*, *P* and *P'*. Consequently `eig(...)` computes none of their " significant " bits correctly when N > 17 .

However, for 7 < N < 17 , `eig(...)` computes those smaller eigenvalues several sig. bits more accurately for *F'* than for the other matrices, thus demonstrating that

`eig(...)`'s accuracy depends in part upon mathematically irrelevant accidents.

Remedy:

```
[Q, V] = RefinEig(Q, V, B)
```

is my *MATLAB*-language program designed to try to improve the accuracy of

```
[Q, V] = eig( B )
```

in cases when it has been degraded by some accident.

Sometimes the improvement is spectacular.

How to invoke **RefinEig** from within *MATLAB* :

```
[Q, V] = eig( B ) ;          % ... Initial approximations Q, V .
[Q, V] = RefinEig(Q,V,B) ; % ... Iterate until "convergence."
```

Convergence is *cubic* (extremely fast) if it occurs at all,
so one iteration usually does about as well as can be done.

How **RefinEig** works:

Ideally $Q^{-1}\cdot B\cdot Q = V$ would be diagonal, but because of roundoff we find

$$\Delta C = Q^{-1}\cdot B\cdot Q - V ,$$

when computed, to be nonzero and nondiagonal. We shall replace the approximate eigensystem [Q, V] by the exact eigensystem $[\, Q + Q\cdot\Delta Z,\ V+\Delta V\,]$ wherein, ideally, ΔV is a diagonal correction and ΔZ is an eigenvector corrector, normalized by $\mathrm{diag}(\Delta Z) = O$. They have to satisfy the equation

$$\Delta V = \Delta C + V\cdot\Delta Z - \Delta Z\cdot V - \Delta Z\cdot\Delta V + \Delta C\cdot\Delta Z .$$

To solve it for ΔV and ΔZ , we first rewrite it in a form that suggests an iteration :

$\Delta V = \mathrm{diag}(\Delta C + \Delta C\cdot\Delta Z) = \mathrm{diag}(\Delta C) + O(\Delta...)^2$;
U := the NxN matrix full of 1 s ;
E := $U\cdot V - V\cdot U + U\Delta V - \Delta V + I$; ... presumably entirely nonzero
$\Delta Z = (\Delta C - \Delta V + \Delta C\cdot\Delta Z)./E = (\Delta C - \Delta V)./E + O(\Delta...)^2$.
(The division (...)./E is to be performed elementwise.)

Initializing ΔZ to O and running through these equations in turn would yield first-order approximations to ΔV and ΔZ with a fatal defect; they degenerate in case some eigenvalues of B are too closely paired though otherwise well separated, which is the most common situation for which `eig(...)` is inaccurate.

`RefinEig`'s innovation is a better initialization of ΔZ inspired by a half-century old formula of Jahn and Magnier discussed in Bodewig's *Matrix Calculus* (1959). This ΔZ would be exactly right if ΔC were a permuted diagonal sum of 1x1 and 2x2 matrices, and is correct to first order otherwise.

`RefinEig` computes *residuals* like $R = B\cdot Q - Q\cdot V$, needed for $\Delta C = Q^{-1}\cdot R$, in a slightly peculiar way. Instead of the *MATLAB* expression `B*Q - Q*V`, `R = [B, Q]*[Q; -V]` is computed in one matrix multiplication for a reason that will become evident momentarily.

RefinEig as an Accuracy Benchmark

MATLAB scripts were prepared to assess the accuracies to which first `eig(...)` and then `RefinEig(...)` can compute the smallest few eigenvalues of W. Frank's matrices *F*, *F'*, *P* and *P'* for dimensions N from 8 to 24.
Since the accuracy of `eig(...)` is sometimes affected by *equilibration* or *balancing*, it was run both with and without; see *MATLAB*'s documentation for `eig(..., 'nobalance')`.

Since *MATLAB* runs on practically every commercially significant computer that conforms to IEEE Standard 754, and *MATLAB* provides no way to mention its floating-point format (all its variables use the 8-byte `double` format), and ...

since `RefinEig` performs a valuable function (as the following results will confirm) regardless of the floating-point formats available, and ...

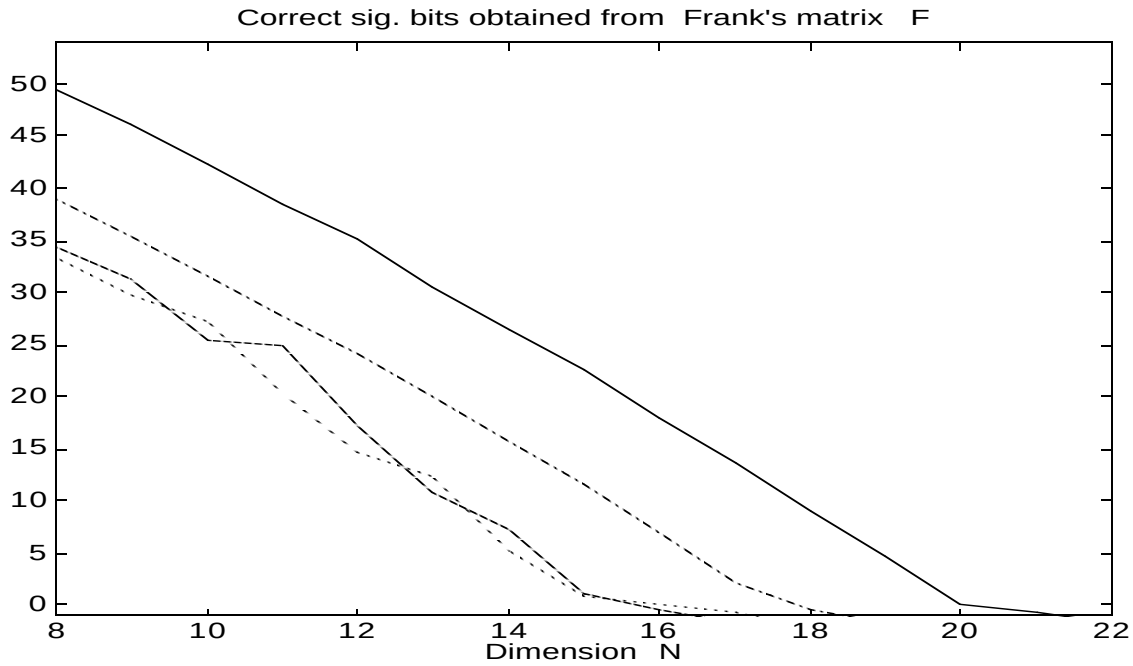
since the input data consists of arrays of easily converted small integers, and ...

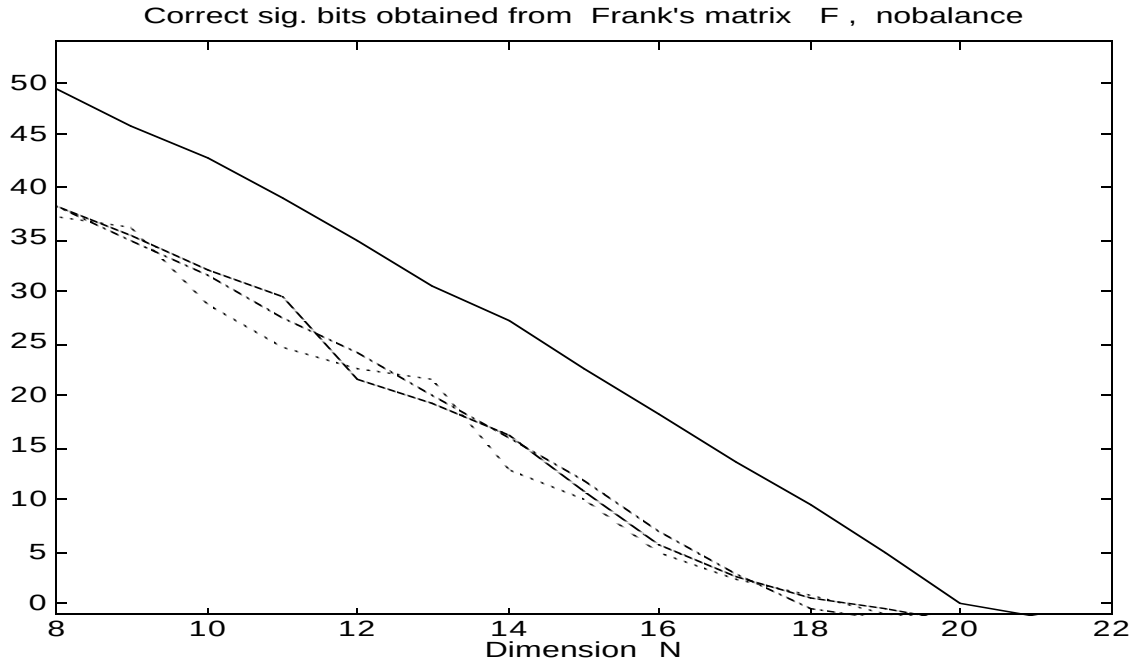
since the desired eigenvalues are not vulnerable to the Stopped Clock Paradox,

RefinEig possesses the four qualifications enunciated above for an acceptable Accuracy Benchmark.

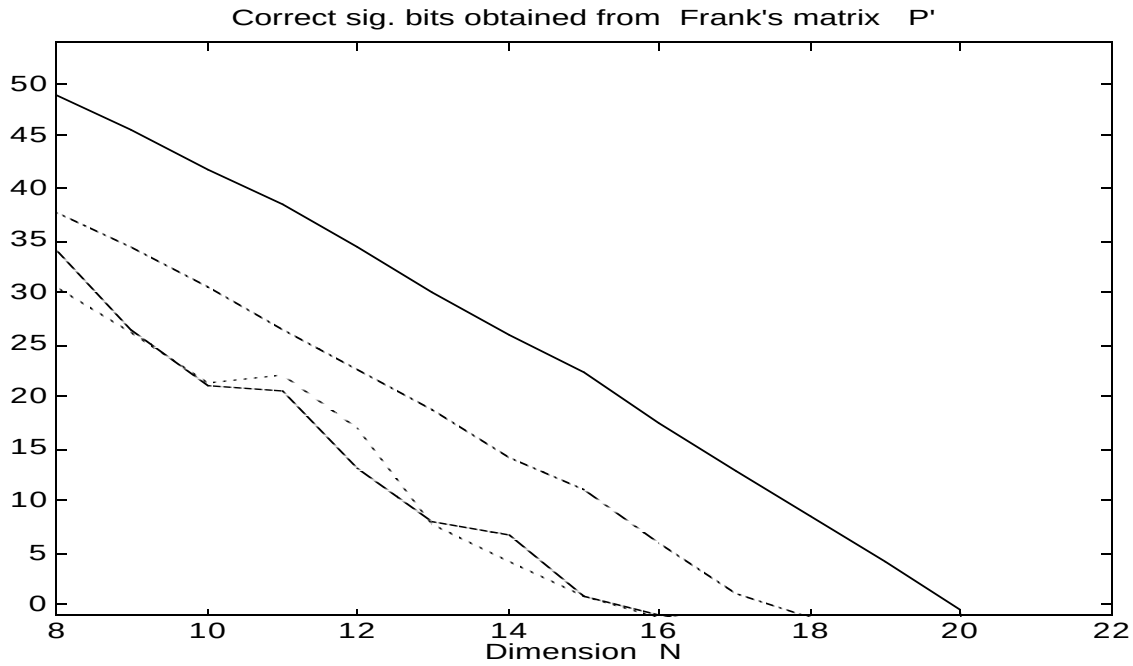
Let us view its results:

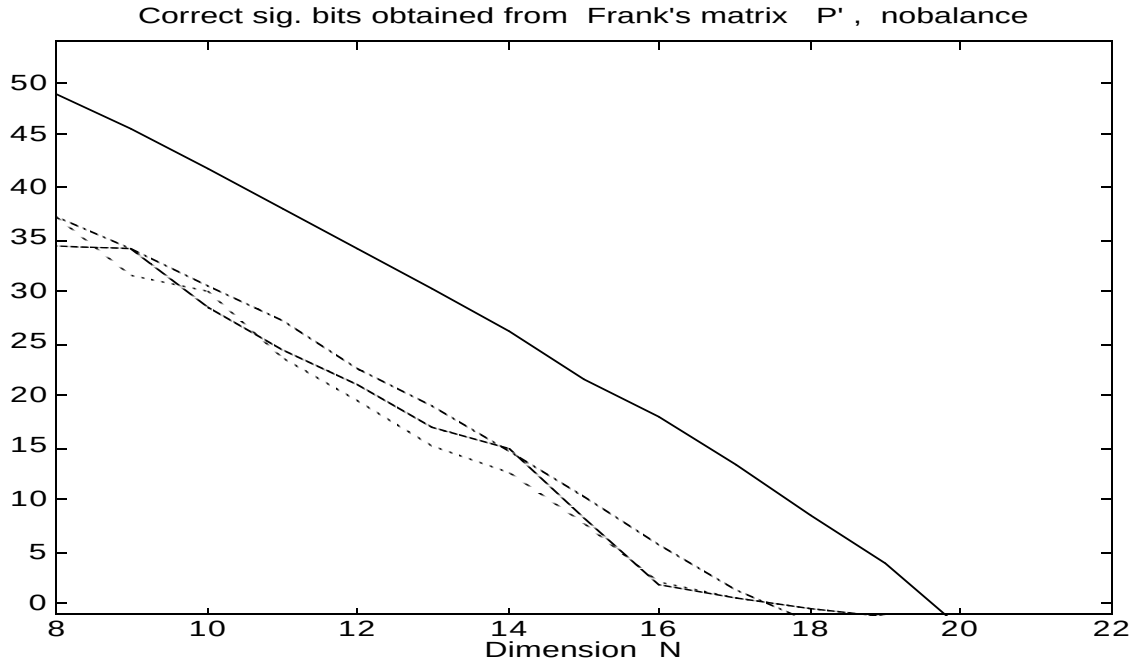
These results were obtained off my 68040-based Macintosh Quadra 950, and differ negligibly from results off my 386/387-based PC. The results for other computers, such as the MIPS, SPARC, H-P PA, PowerPC/Mac and DEC Alpha, were simulated by setting the Mac's and PC's *Precision Control* to emulate the other computers' arithmetics. The emulation is imperfect, but close enough.



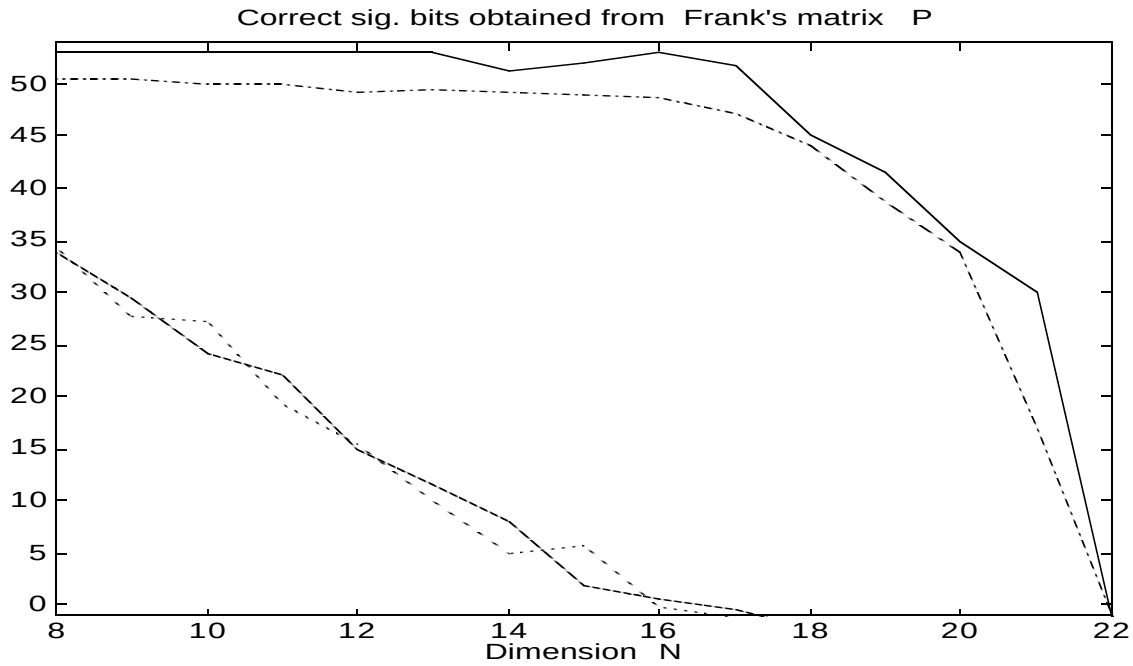


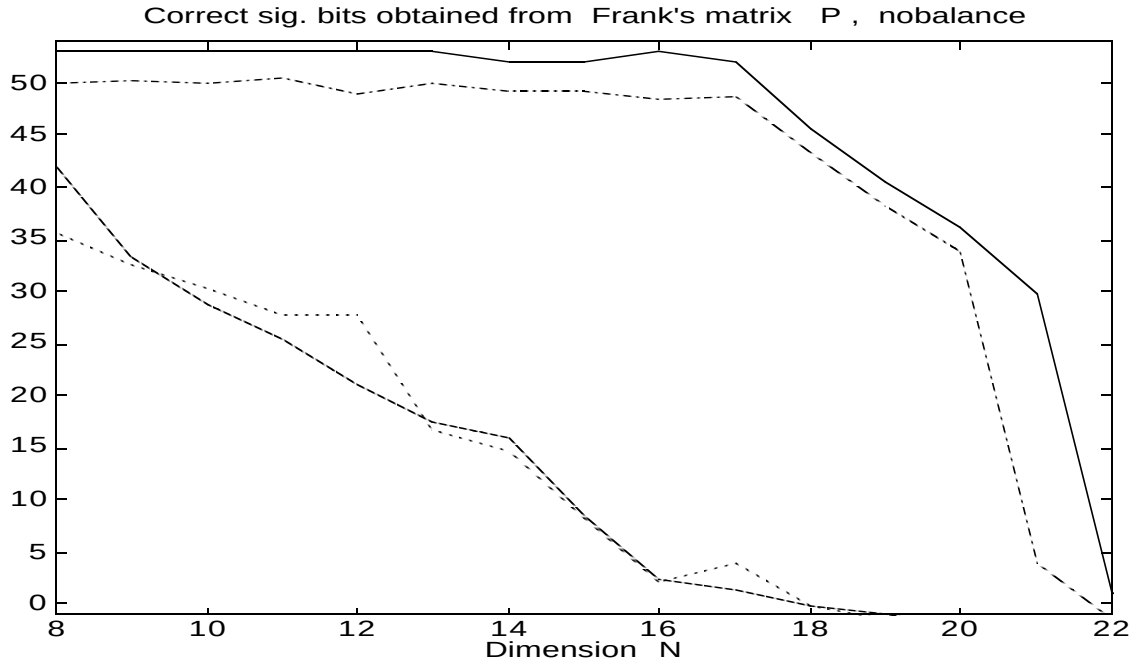
Legend:
- `-----` `eig` on 680x0-Mac or Intel-PC
- `_____` `RefinEig` on 680x0-Mac or Intel-PC
- `......` `eig` on others
- `·-·-·-·` `RefinEig` on others.





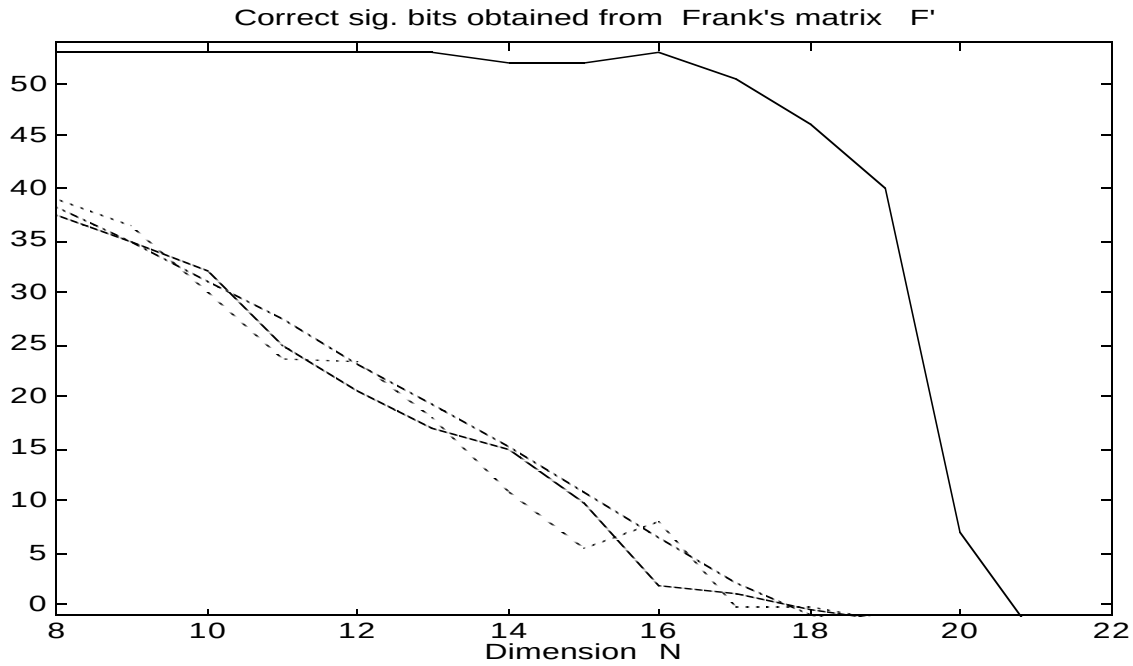
Legend:
- `-----` `eig` on 680x0-Mac or Intel-PC
- `_____` `RefinEig` on 680x0-Mac or Intel-PC
- `......` `eig` on others
- `·-·-·-·` `RefinEig` on others.

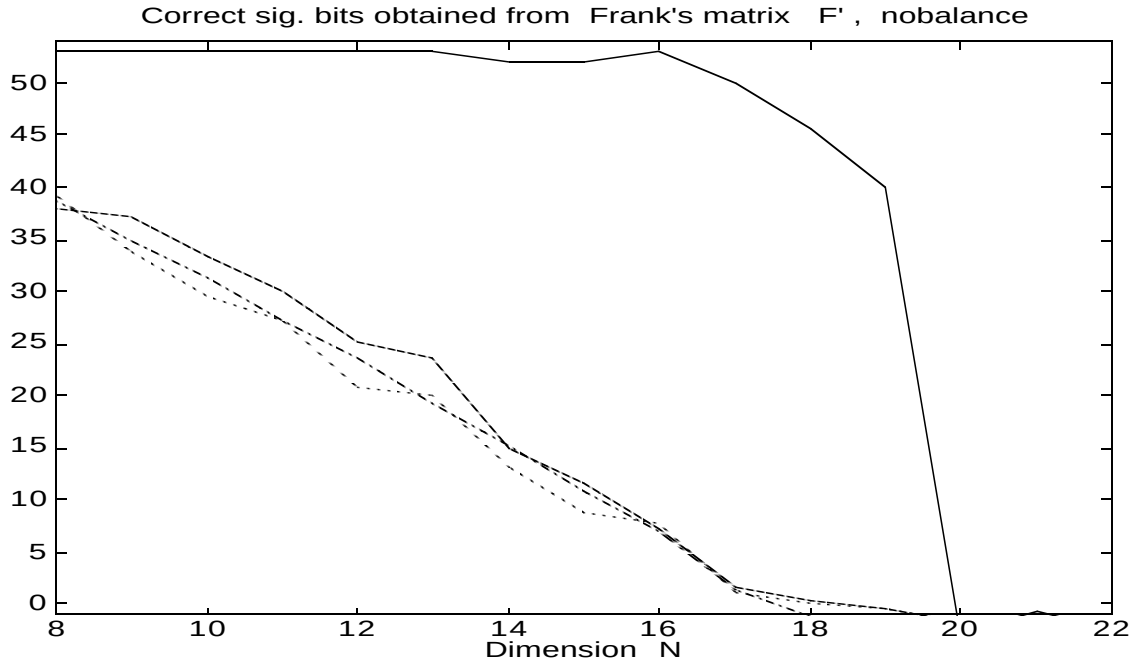




Legend:

-----	`eig`	on 680x0-Mac or Intel-PC
_____	`RefinEig`	on 680x0-Mac or Intel-PC
.....	`eig`	on others
·-·-·-·	`RefinEig`	on others.





Legend:
- `-----` `eig` on 680x0-Mac or Intel-PC
- `_____` `RefinEig` on 680x0-Mac or Intel-PC
- `......` `eig` on others
- `·-·-·-·` `RefinEig` on others

How does *MATLAB* benefit from an Extended format which `Qdrtc` showed us *MATLAB* eschews ?

MATLAB 's matrix multiplication operation is programmed carefully, differently for every different computer, in order to reach

the highest possible speed.

Every element of a matrix product is a

$$\textbf{Scalar Product} \;=\; a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 + \ldots + a_N \cdot b_N \quad .$$

By keeping products $a_j \cdot b_j$ and their sums in fast registers to maximize speed, *MATLAB* computes them to the precision of the registers; on computers with Extended precision, that is 64 sig. bits even though the operands a_j and b_j carry only 53 sig. bits.

`RefinEig` computes its residuals like $R = B \cdot Q - Q \cdot V$ as matrix products

$$R = \begin{bmatrix} B & Q \end{bmatrix} \cdot \begin{bmatrix} Q \\ -V \end{bmatrix}$$

which, after massive cancellation, come out almost as accurate as if evaluated in 64 sig. bit arithmetic though they are stored to only 53 sig. bits.

Thus, on computers that have it,

Extended precision can enhance `RefinEig`'s accuracy,
typically by 11 sig. bits,
without ever being mentioned.

Summary of Observations so far:

1. The Non-Symmetric Eigenproblem has
no fast foolproof solution.

Occasionally trial-and-error is inescapable.

2. **`RefinEig`** usually improves accuracy
regardless of the underlying arithmetic.

`RefinEig` never hurts much, even if it cannot help much. (*F'*)
Sometimes it recovers accuracy lost to (im)balancing. (*F* , *P'*)
Sometimes it improves accuracy spectacularly. (*P*)

3. Computers with Double-Extended registers
always gain accuracy through **`RefinEig`**.

If more than 11 sig. bits would be lost, those registers recover at least 10 .
They sometimes recover spectacularly more. (*F'*)

4. This accuracy benchmark reveals something interesting,
about the different accuracies inherent in different computers,
impossible to glean from benchmarks dedicated solely to
speed.

Incidentally, the accuracy achieved by `RefinEig` on my Mac Quadra and on my PC is achieved in less than half the time *Mathematica* and *Maple V* consume to achieve the same accuracy from their multi-precision arithmetic software.

The Threat: Atrophy and Stagnation

For lack of benchmarks that assess accuracy or other desirable attributes other than speed,
Apple's S.A.N.E never received the accolade it deserved from the marketplace.

Consequently, Apple's management cut its losses, dispersed much of Apple's numerical expertise, and abandoned the Double-Extended format when they chose to move from the 680x0 to the faster Power-PC-based " Power Mac " (which goes faster for reasons other than its omission of an Extended format).

For lack of benchmarks that would reward their diligence, compiler writers have not supported novel capabilities of IEEE 754, so atrophy threatens them:

Fast flexible handling of exceptions like Division-by-Zero and Gradual Underflow.

Directed roundings, necessary for good Interval Arithmetic.

Extended precision, capable of evolving into arbitrarily high precision.
Extended range.

More generally, for lack of ways to accommodate innovations, current benchmarks tend to stifle innovations regardless of their merits.

Computer Languages and Compilers hold center stage.

Mediaeval thinkers held to a superstition that

Thought is impossible without Language.

That is why "dumb" changed in meaning from "speechless" to "stupid."

With the advent of computers, "Thought" and "Language" have changed their meanings, and now there is some truth to the old superstition:

In so far as programming languages constrain utterance,
they also constrain what a programmer may contemplate productively.

Few compiler writers address challenges to mathematical, scientific and engineering computation, and these few are preoccupied with keeping their handiwork abreast of rapidly changing hardware in a bitterly competitive marketplace where no architecture enjoys more than a few months of ascendancy.

They have to run as fast as they can just to stay in the same place.

Consequently, computer languages have not been evolving towards scientifically desirable goals, swayed as they are by over-reliance upon standards committees' aesthetic fads, on the one hand, and industrial demands for compatibility with past practice on the other. For instance, a case could be made for ...

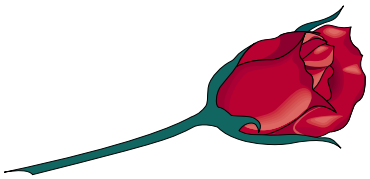
The Baleful Effect of
C++
upon
Applied Mathematics,
Physics and Chemistry.

The challenges facing the Scientific Community:

Although Computer Science ought to be a branch of Applied Mathematics distinguished solely by its preoccupation with the cost of computation, we cannot rely upon the mathematical probity of computer professionals among whom few harbor hospitality towards mathematical thought. We have educated them badly:

Some think Mathematics is a Religion
whose rules they have been taught not to break for fear of moral condemnation.
e.g., Division by Zero, Discontinuity .
Although violating some rules is perilous, others are intended to be broken;
the trick is to tell which are which.

Some think Mathematics has at most Aesthetic value.
If you believe Beauty is *the* criterion by which Mathematics should be judged,
please recall that Beauty lies in the Eye of the Beholder ;
in the eyes of a bug, a rose is mere fodder.



Mathematics is a miraculous reward for penetrating thought.

To render that kind of thought ever more economical is the computer's most worthwhile promise. We had best not entrust it entirely to people antipathetic to mathematical thought or motivated too much by mere pecuniary rewards.

The Scientific Community has to help promulgate Appropriate Benchmarks

and other schemes that will reward diligence and encourage useful innovation while discouraging unnecessary and anarchic diversity that fragments the marketplace.

This problem is difficult technically and politically.