The

John von Neumann Lecture

on

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

presented by

Prof. W. Kahan
Mathematics Dept., and
Elect. Eng. & Computer Sci. Dept.
University of California
Berkeley CA 94720-1776
(510) 642-5638

at the

45th Annual Meeting of S.I.A.M.
Stanford University
15 July 1997

PostScript file:
`http://http.cs.berkeley.edu/~wkahan/SIAMjvnl.ps`
For more details see also …`/Triangle.ps` and …`/Cantilever.ps`

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

Abstract:

An unhealthy preoccupation with floating–point arithmetic's Speed, as if it were the same as Throughput, has distracted the computing industry and its marketplace from other important qualities that computers' arithmetic hardware and software should possess too, qualities like

Accuracy, Reliability, Ease of Use, Adaptability, ···

These qualities, especially the ones that arise out of Intellectual Economy, tend to be given little weight partly because they are so hard to quantify and to compare meaningfully. Worse, technical and political limitations built into current benchmarking practices discourage innovations and accommodations of features necessary as well as desirable for robust and reliable technical computation. Most exasperating are computer languages like Java® 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' benefits, thus denying language implementors any incentive to accommodate those features in their compilers. It is a vicious circle that the communities concerned with scientific and engineering computation must help the computing industry break. But our communities are still entangled in misconceptions that becloud floating–point issues, so we send the industry mixed and murky signals.

" The trouble with people is not that they don't know
but that they know so much that ain't so."
Josh Billings' Encyclopedia of Wit and Wisdom (1874)

Floating–Point Arithmetic has always been the Sliver under a Fingernail of Computer Science.

For the Princeton IAS machine, von Neumann rejected floating–point hardware thus:

[Without it,] "… human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time so consumed is a very small percentage of the total time we will spend ..." [programming in machine language.]
. . .
" We wish to incorporate into the machine … only such logical concepts as are either necessary … or highly convenient …" J.v.N. *Collected Works* vol.5 p. 43; §5.3 of Burks, Goldstine & von Neumann (1946/7). [Anticipating RISC ?]

" While a floating binary point is undoubtedly very convenient in coding problems, building it into the computer adds greatly to its complexity …" *ibid*. §6.6.7 p. 73.

" We formulated there [§6.6.7] various criticisms which caused us to exclude this facility, at any rate, from the first model of our machine. … Besides, the floating binary point represents an effort to render a thorough mathematical understanding of at least part of the problem unnecessary, and we feel that this is a step in a doubtful direction." *ibid*. §8.6 p. 113.

What could have motivated such specious rationalization?
I have to speculate …

Immediately upon the end of World War II in Europe, von Neumann inspected British and German computing facilities. He must have learned of the sad (for the Nazis) fate of Konrad Zuse's proposal to build an electronic computer with carefully rounded floating–point arithmetic that would have included $\pm\infty$ and things like *NaN*s, anticipating features of IEEE Standard floating–point by four decades. But in 1939 the German Air Ministry had refused to fund the proposal because, they said, it was too complicated to build before 1942, by which time they expected to have won the war and rendered the computer unnecessary.

Perhaps von Neumann wished to avert such a fate for the IAS machine.

Von Neumann's writings did more than disparage floating–point.

Omission of a guard digit could be justified by quoting this:

" Numbers which decrease in these cases [leading zeros created by cancellation] are really (mathematically) losing precision. Hence it is perfectly proper to omit [normalization] in this event. (Indeed, such a true loss of precision cannot be obviated by any formal procedure, but, if at all, only by a different mathematical formulation of the problem.)" *ibid*. §6.6.7, p. 74. This has been taken to mean

1.0xxxx???	the values of digits beyond the rightmost
-**0.1xxxx**x???	digit stored cannot be known, so the last
0.0000x????	digit x of the subtrahend doesn't matter.

It's almost true, and therefore a lie.

This misconception degraded the arithmetics of the IBM /360 until 1967, all Seymour Cray's CDC and CRAY designs, TI calculators, ...

Skepticism of floating–point error–analysis could be justified …

Papers in 1947/8 by Bargman, Goldstein, Montgomery and von Neumann seemed to imply that 40-bit arithmetic would hardly ever deliver usable accuracy for the solution of so few as 100 linear equations in 100 unknowns; but by 1954 engineers were solving bigger systems routinely and getting satisfactory accuracy from arithmetics with no more than 40 bits.

In 1957, soon after von Neumann died, the flaw in his reasoning was exposed (by A.S. Householder if my memory is right). To solve it more easily without floating–point von Neumann had transformed equation $Bx = c$ to $B^TBx = B^Tc$, thus unnecessarily doubling the number of sig. bits lost to ill-condition. Now that transformation deserves to be used only to solve least–squares problems with a rectangular matrix B , and only if the arithmetic carries somewhat more than twice as many sig. bits as are trusted in the data (B, c) or desired in the solution x .

. . . . . .

Indignor quandoque bonus dormitat Homerus. Horace.

(" Even the great Homer nods occasionally, and it annoys me.")

Considering how prodigious von Neumann's output was, his few mistakes are amply forgivable. Compared with his, my output is infinitesimal and my mistakes innumerable. To list a dozen of them here would seem boastful, so I won't.

Thesis:

Floating–Point Arithmetic
becomes an extremely dull subject
(of interest only to anal-compulsives)
when it is done correctly.

It is not being done correctly.

Numerical results obtained by the
overwhelming majority of computer owners are
less reliable, less robust, and
less accurate, sometimes much less,
than their hardware was designed to provide.

Computer owners derive scant benefit
from advantageous hardware features they have paid for
since these are practically inaccessible through
commercially dominant compilers and computer languages.

Why?

Why do so few of us benefit from improved floating–point capabilites we paid for in our hardware?

1. Most of us do not know they're there. We don't even have standard names by which to ask for them.

2. Current benchmarking practices test only those attributes of floating–point arithmetic common to all computers of the 1970s. Improved capabilities are not exercised by any benchmark, so no incentive exists to support them in commercial compilers for standard programming languages, so we cannot write suitable programs in a fashion sufficiently portable to be acceptable as benchmarks. It's a vicious circle.

3. The improved capabilities are features of IEEE Standard 754 for Binary Floating–Point Arithmetic mistakenly deemed arcane, as if intended for only a few specialists.

Bunk !

This is just one of a host of persistent misunderstandings that impede attempts to deal with floating–point issues.

14 Prevalent Misconceptions about Floating–Point Arithmetic

1• Floating–point numbers are all at least slightly uncertain.

2• In floating–point arithmetic, every number is a " Stand–In " for all the numbers that differ from it in digits beyond the last digit stored.

3• Arithmetic much more precise than the data it operates upon is needless.

4• In floating–point arithmetic nothing is ever exactly 0 ; but if it is, no purpose is served by distinguishing +0 from -0 .

5• Subtractive cancellation always causes numerical inaccuracy, or is the only cause of it.

6• A singularity always degrades accuracy when data approach it.

7• Classical formulas taught in schools and found in handbooks and software must have passed the Test of Time, not merely withstood it.

8• When better formulas are found, they supplant the worse.

9• Modern " Backward Error–Analysis " explains all error, or excuses it.

10• Algorithms known to be " Numerically Unstable " should never be used.

11• " Ill–Conditioned " data or problems deserve inaccurate results.

12• Bad results are the fault of bad data or bad programmers, not programming languages.

11• Most features of IEEE Floating-Point Standard 754 are too arcane.

14• ' Beauty is truth, truth beauty.' — that is all ye know on earth, and all ye need to know. ... Keats' *Ode on a Grecian Urn*

~~~~~~~~~~~~~~~~~~~~~~~~~~

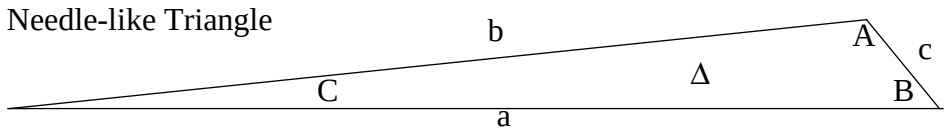
Misconceptions like these prevent programming languages and their compilers from conveying to owners of over 90% of computers now on desk–tops the benefits of floating–point semantics built into their hardware and superior to what is now considered acceptable.

And we continue to disseminate these misconceptions.
~~~~~~~~~~~~~~~~~~~~~~~~~~

Case Study: Heron's Formula for the Area Δ of a Triangle

Given a triangle's side–lengths a, b, c , its area is computed from a classical formula:

$$\Delta := \sqrt{(s\ (s-a)\ (s-b)\ (s-c))}\ , \qquad \text{where} \quad s := (a+b+c)/2\ .$$



Example: a := 65536 - 1.5/128 ; b := a + 3.5/128 ; c := 64 + 1/256.0 .

These data are representable exactly as Single Precision (`float` in C , or REAL*4 in Fortran) numbers. What we get for Δ depends upon the precision to which arithmetic operations are rounded:

Single (Fortran and newer C compilers): Δ = 2,097,**023.5**
Double (older C compilers) Δ = 2,097,279.621
Why so much difference?

Recompute Δ for 27 sets of Single data differing from the given data by at most one ulp (*U*nit in the *L*ast *P*lace stored, the 24th sig. bit).

Single: Δ varies from 2,096,**895.25** to 2,097,**407.5** .

Recompute Δ in three Rounding Modes, *Up*, to *Nearest*, and *Down*.

Single: Δ varies from 2,097,**023.25** to 2,09**8,048.25** .

Diagnosis: Δ is an **ill–conditioned** function of this data. Roundoff contributes little more uncertainty to Δ than it inherits from last–digit uncertainty in this data, so Heron's formula does as well as can be done.

This diagnosis is WRONG.

Heron's Classical Formula is Numerically Unstable for Needle-like Triangles.

Here is a better formula: First sort the data so that $a \geq b \geq c$. Then

$$\Delta = \sqrt{(\ (a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))\ )}/4\ .$$

DO NOT REMOVE PARENTHESES.

Cancellation will occur but will cause no harm on any commercially significant North American computer except Crays X-MP, …, J90.

Single (Fortran and newer C compilers): Δ = 2,097,279**.5**
Double (older C compilers) Δ = 2,097,279**.**621
Different only in bits beyond the last bit stored.

Recompute Δ for 27 sets of Single data differing from the given data by at most one ulp (*U*nit in the *L*ast *P*lace stored, the 24th sig. bit).

Single: Δ varies from 2,097,279**.** to 2,097,280**.** .

Recompute Δ in three Rounding Modes, *Up*, to *Nearest*, and *Down*.

Single: Δ varies from 2,097,279**.**25 to 2,097,280**.** .

Diagnosis: Δ is a **well–conditioned** function of this data. Roundoff contributes uncertainty to only the last bit or two of Δ . (In fact, this better formula computes Δ fully accurately for ill-conditioned data too.).

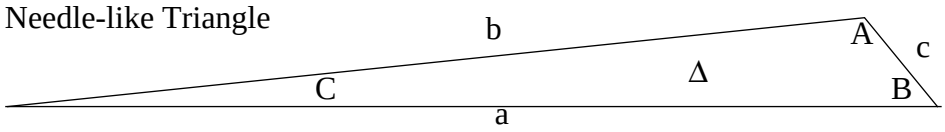
P.S.: The foregoing results were obtained from MATLAB on an ancient Intel PC upgraded with Cyrix chips, and also on another Pentium PC. Without changes to .m files that compute Δ from each of two formulas,

- how do I make MATLAB round to Single (24 sig. bits) ?
- how do I make MATLAB round *Up* and *Down*, not *to Nearest* ?

I can, but you can't.

I ran an old version, MATLAB 3**.**5, plus my DOS program that diddles the floating–point control word on Intel-based PCs.

Heron's formula is not the only schoolbook trigonometric formula that dislikes triangles of certain shapes.



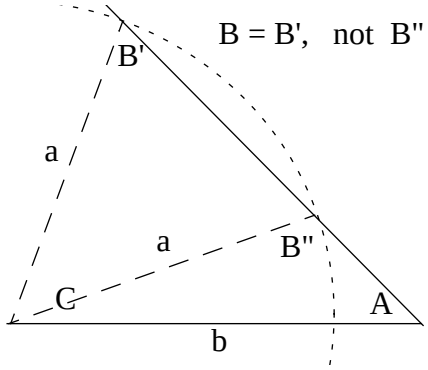
Unnecessarily inaccurate results can be obtained also from …

$$C := \arccos((a^2 + b^2 - c^2)/(2ab)) = 2 \arctan(\sqrt{((s-a)\,(s-b)/(s\,(s-c)))})$$

$$c := \sqrt{(a^2 + b^2 - 2\,a\,b \cos C)}$$

$$B := \arcsin((b/a) \sin A)$$

This formula for B dislikes triangles with B too near 90˚.



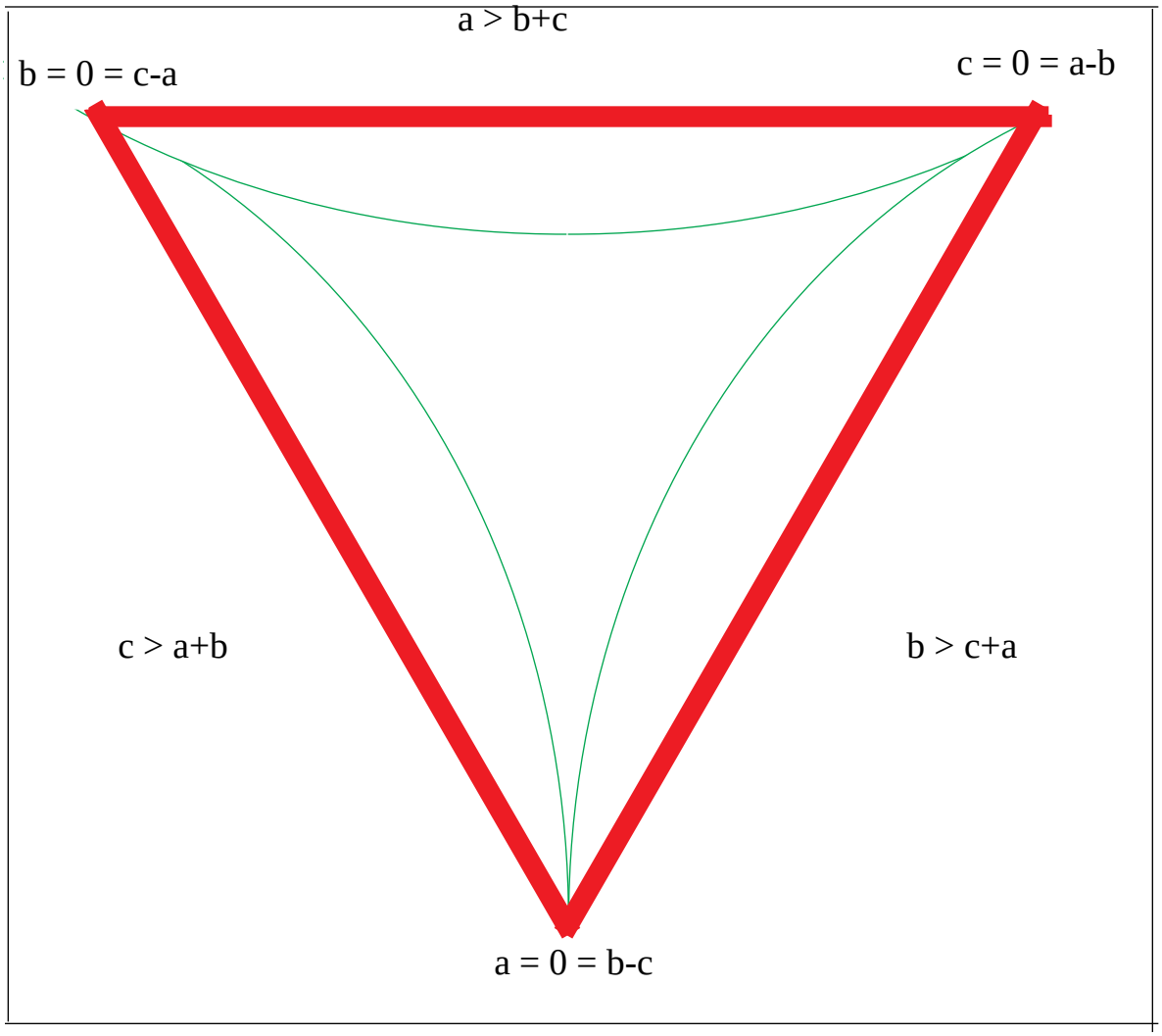
These classical formulas have *withstood* the Test of Time, not *passed* it.

For better formulas see http://http.cs.berkeley.edu/~wkahan/Triangle.ps .

Where does Heron's classical formula for Area Δ go wrong?

Almost nowhere.

Map Triangles to Points in the Plane by taking a Triangle's side–lengths (a, b, c) as Barycentric Coordinates:



Every point in the **Bold Triangle** above represents a family of Similar triangles.
All triangles Similar to a given generic triangle map to six of those points.
Points near the boundary represent Needle–like Triangles.

Points inside the *curvilinear triangle* represent triangles with all angles acute and utterly well–conditioned areas $\Delta(a,b,c)$. (The *curves* are hyperbolic arcs.)

Triangles for which Heron's formula miscalculates $\Delta(a,b,c)$ are at points inside the **thick edges**, with thickness proportional to the roundoff threshold.
Every extra decimal digit of arithmetic precision shrinks their thickness by 1/10.

The revised formula $\Delta(a,b,c)$ with sorted a, b, c is accurate at all triangles.

Conclusion: Everybody should use the better formula for Δ .

But they won't.

The better formula has been published at least four times, but not where most programmers who might need it are likely to look it up. Heron's formula is what they will almost surely find instead.

In general, the programmers who use a little (or a lot of) floating–point arithmetic may be very clever at things they care about, but not at error–analysis of floating–point. (Not even von Neumann got it quite right.) And all of us shall occasionally run their programs unwittingly, and be thus exposed to risks of which they were unaware.

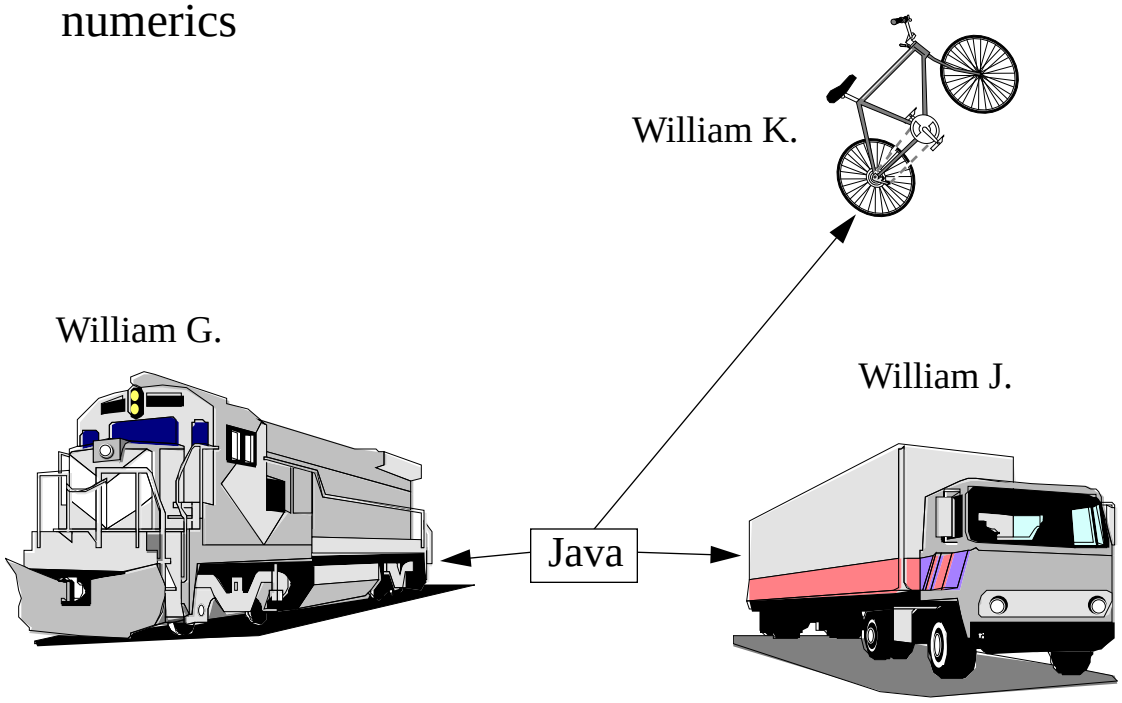
The floating–point arithmetics on Motorola and AMD/Cyrix/Intel chips in Macs, PCs and Power-PCs, were designed to attenuate the risks you face and to help you diagnose them.

They were designed to evaluate every subexpression to 8-byte Double–Precision (Power-PC/Macs) like old–fashioned C, or to 10-byte Extended Precision (PCs, older Macs), to attenuate the incidence of dangerously inaccurate results. But Java doesn't do this.

They were designed to let you rerun program modules, whose source-code you can't or won't change, unchanged but in different rounding modes upon data that produce suspicious results. If different roundings change a module's results too much, you may wish to question its provenance. But Java does not allow you to do this.

Attenuating risks does not eliminate them; neither is the foregoing diagnostic technique anywhere near foolproof. Still, those hardware designs do improve your chances. But not with Java programs.

Three Williams contend for Java's numerics



It seems bizarre that a programming language being promoted as the way for Everyman to program Everything to run Everywhere should have floating–point semantics chosen to be so disadvantageous to the overwhelming majority of programmers and users of the overwhelming majority of computers on desktops. I have been assured that it was not done in an attempt to secure commercial advantage for a few at the expense of the many, so I must presume that it was done out of ignorance.

An attempt is under way to supplant Java's treatment of floating–point by something better. It is called Borneo, devised by some students at U.C. Berkeley, and being implemented at least in part by one of them, Joe Darcy < darcy@cs.berkeley.edu > with support from some folks at Sun.

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

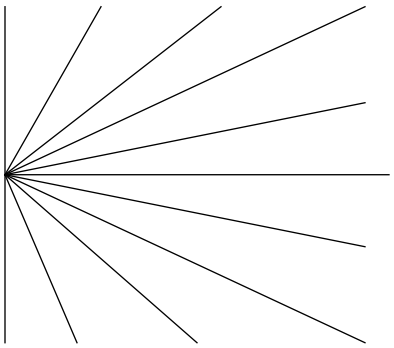
$$g(z) = z^2 + z \cdot \sqrt{z^2 + 1} \quad , \quad \text{and}$$

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

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

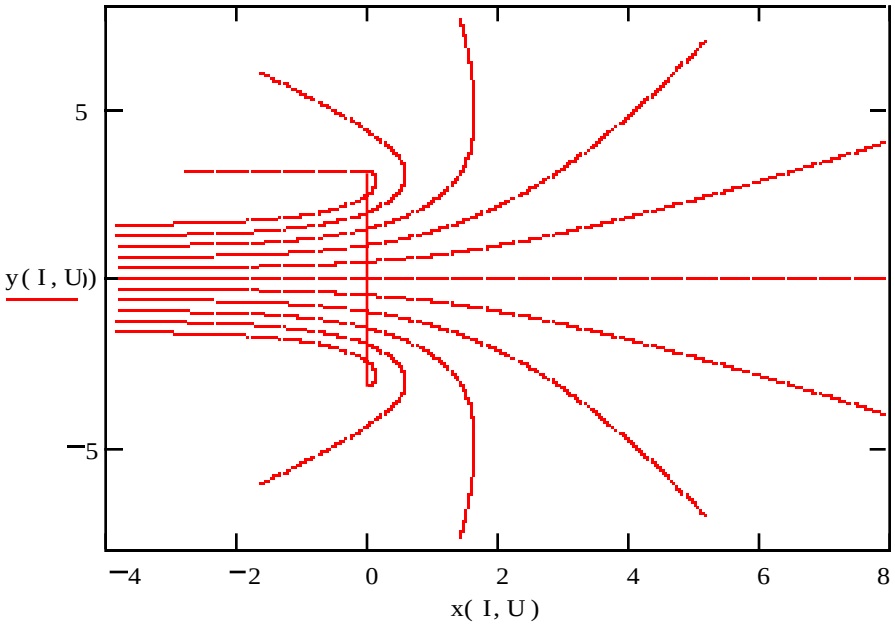
$$\mathrm{SQRT}(\,\mathrm{CONJ}(\,Z\,)\,) \;=\; \mathrm{CONJ}(\,\mathrm{SQRT}(\,Z\,)\,) \qquad \text{and}$$

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

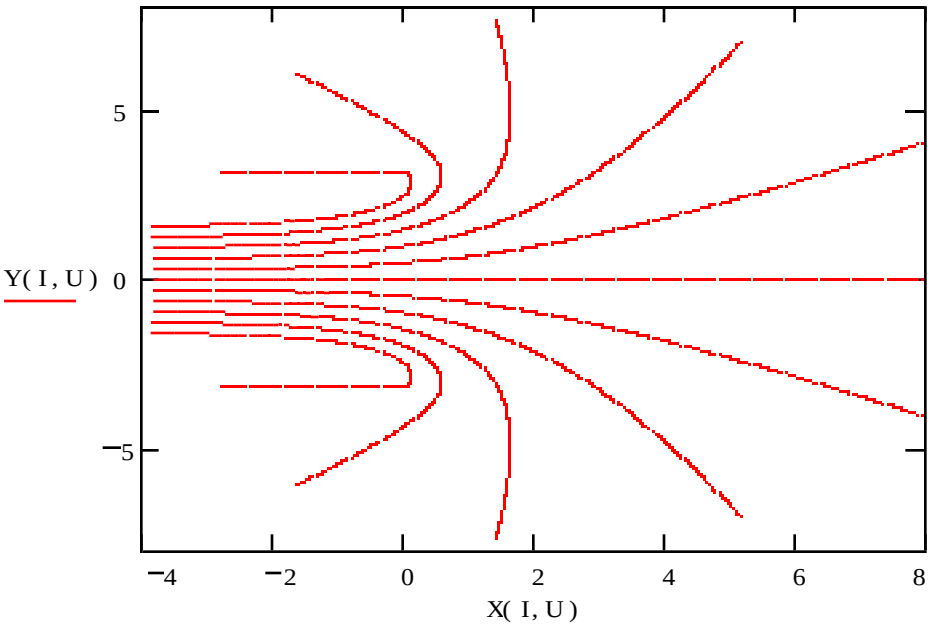
whenever the COMPLEX variable Z takes negative real values.

Pictures of Borda's Mouthpiece come next.

Borda's Mouthpiece, plotted without -0



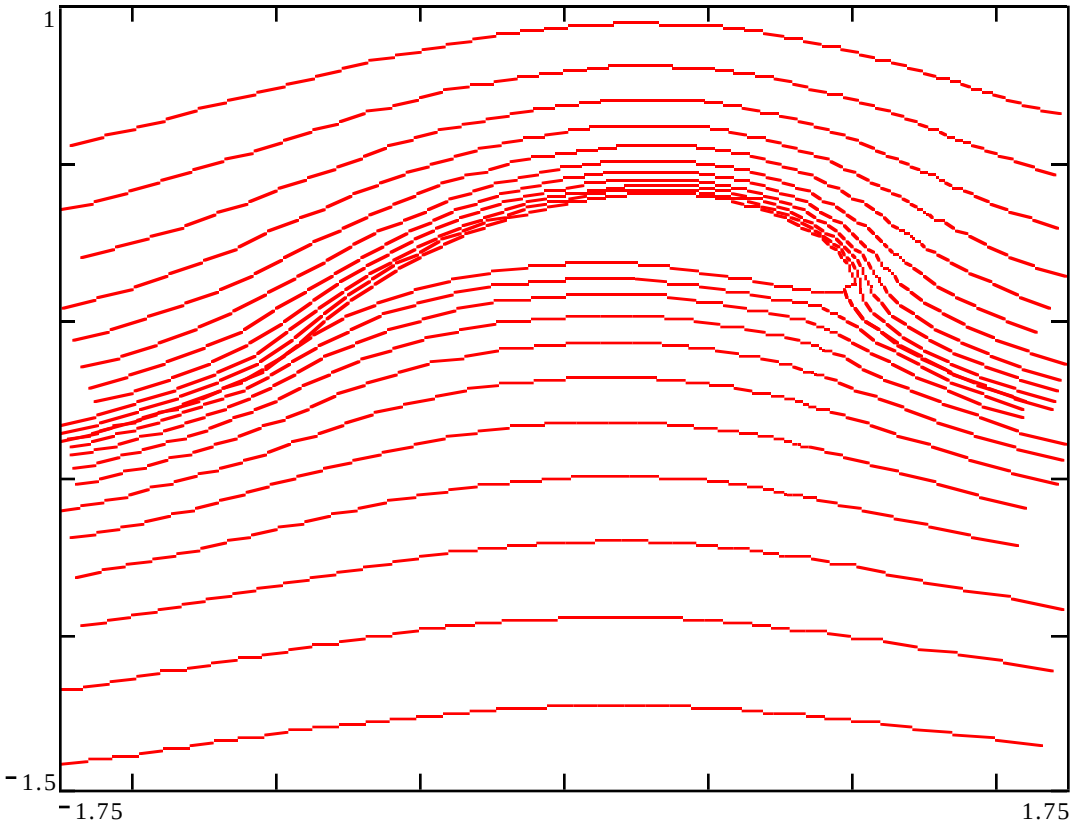
Borda's Mouthpiece, plotted correctly



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

CIRCULATING FLOW PAST JOUKOWSKI 'S AIRFOIL

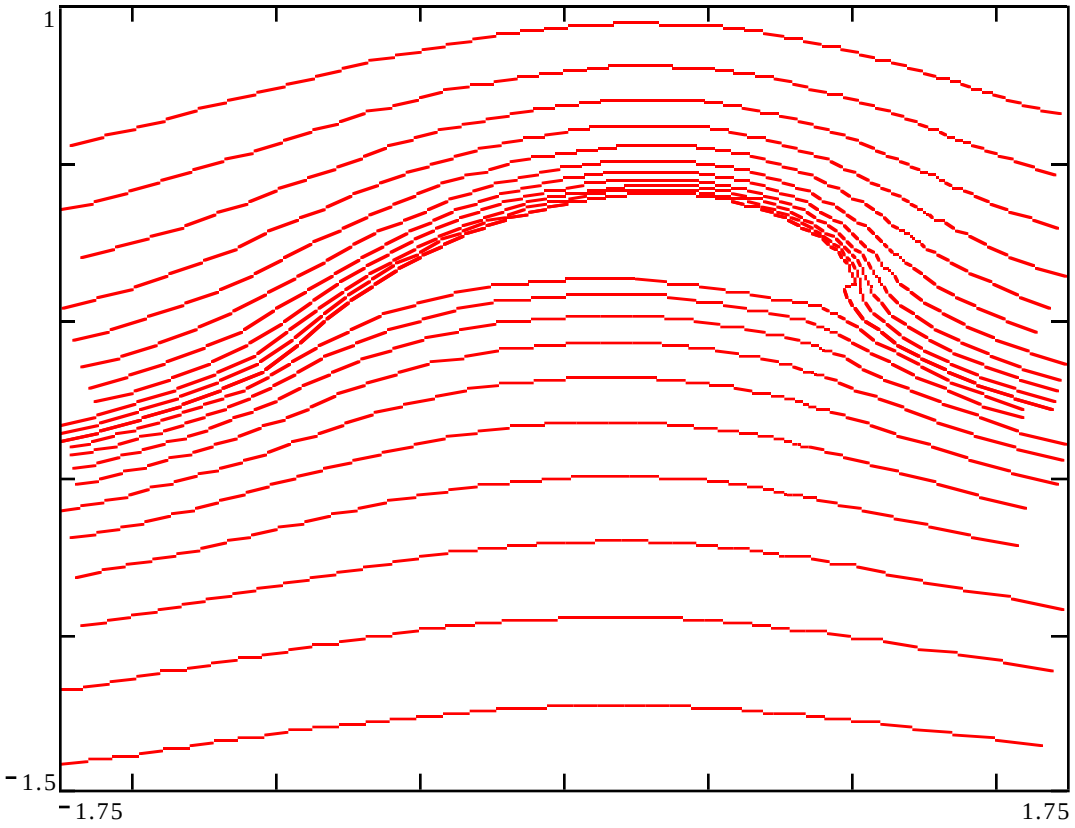
Shape: $\sigma = 0.3 + 0.1i$ Angle of Attack: $\alpha = 0.125$ Circulation: $\gamma = 0.381346$



This shows the stream-lines of a two-dimensional flow of an Ideal Fluid around a cross-section called " Joukowski's Airfoil." Its Shape and its Angle of Attack determine the intensity of Circulation by means of what is called the " Kutta Condition," which says that the flow should not be Singular (turn sharply) at the airfoil's sharp trailing edge. Circulation around a wing engenders lift. It is evident from the plot because the stream-lines flow closer together, indicating higher speed and lower pressure, on one side of the airfoil than on the other. Only at small Angles of Attack is this picture realistic; then a real wing roughly satisfies the Kutta condition for circulation by creating Vortices off the wing-tips. These vortices are sometimes visible as " Contrails " of condensed moisture following an aircraft. At large Angles of Attack the Kutta Condition cannot be maintained; the flow separates from the airfoil, and the wing stalls.

CIRCULATING FLOW PAST JOUKOWSKI 'S AIRFOIL

Shape: $\sigma = 0.3 + 0.1i$ Angle of Attack: $\alpha = 0.125$ Circulation: $\gamma = 0.381346$



What happened to the aerofoil's lower surface? It got lost in a computer that does not support -0 , or else in a programming language that supports the declarations REAL and COMPLEX but not IMAGINARY.

(There are ways to program around a bug like this, but they can be tricky. Besides, from accumulations of little bugs like these does software die the Death of a Thousand Cuts.)

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 standard, by Jim Thomas <jthomas@best.com> 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."

What computations are both important and technically challenging enough that they could earn real money if accomplished significantly better?
These are promising candidates for service in benchmarks.

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*:

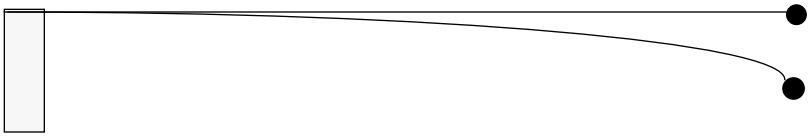
Gargantuan dimension.
Unfortunate column ordering —> poor pivot choice.
Disparate scaling of rows —> poor pivot choice.
Systematically severe ill-condition (near singularity).

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: ....

Case study: **Roundoff Degrades an Idealized Cantilever**

(Work done jointly with Ms. Melody Y. Ivory)

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.. These x_j's are the components of a column vector **x** that satisfies a system $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$ of linear equations in which column vector **b** represents the load (the mass at the end plus the spar's own weight) and the matrix **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 **L** times an upper-triangular **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 **L** and **U** each has only three nonzero diagonals, the work goes fast; fewer than $30\cdot N$ arithmetic operations suffice. But this solution **x** is very sensitive to rounding errors; they can get amplified by the *condition number* of **A** , which is of the order of N^4 .

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.

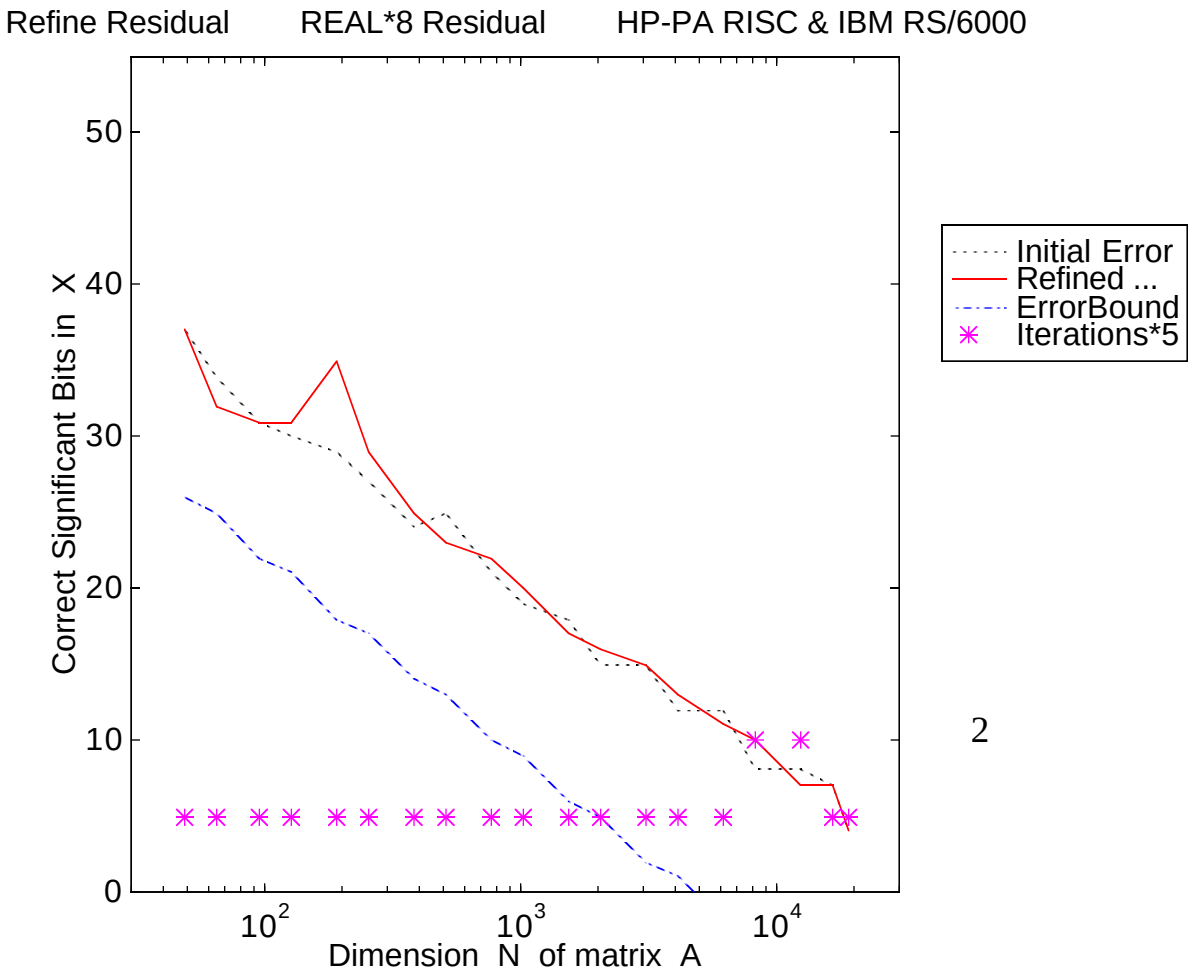
Iterative Refinement mollifies the dilemma:
Compute a *residual* $\mathbf{r} := \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ for $\mathbf{x}$. Solve $\mathbf{A}\cdot\Delta\mathbf{x} = \mathbf{r}$ for a correction $\Delta\mathbf{x}$ using the same program (and triangular factors $\mathbf{L}$ and $\mathbf{U}$) 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:

The following figures exhibit some evidence to support the foregoing claims. For more details see http://http.cs.berkeley.edu/~wkahan/Cantilever.

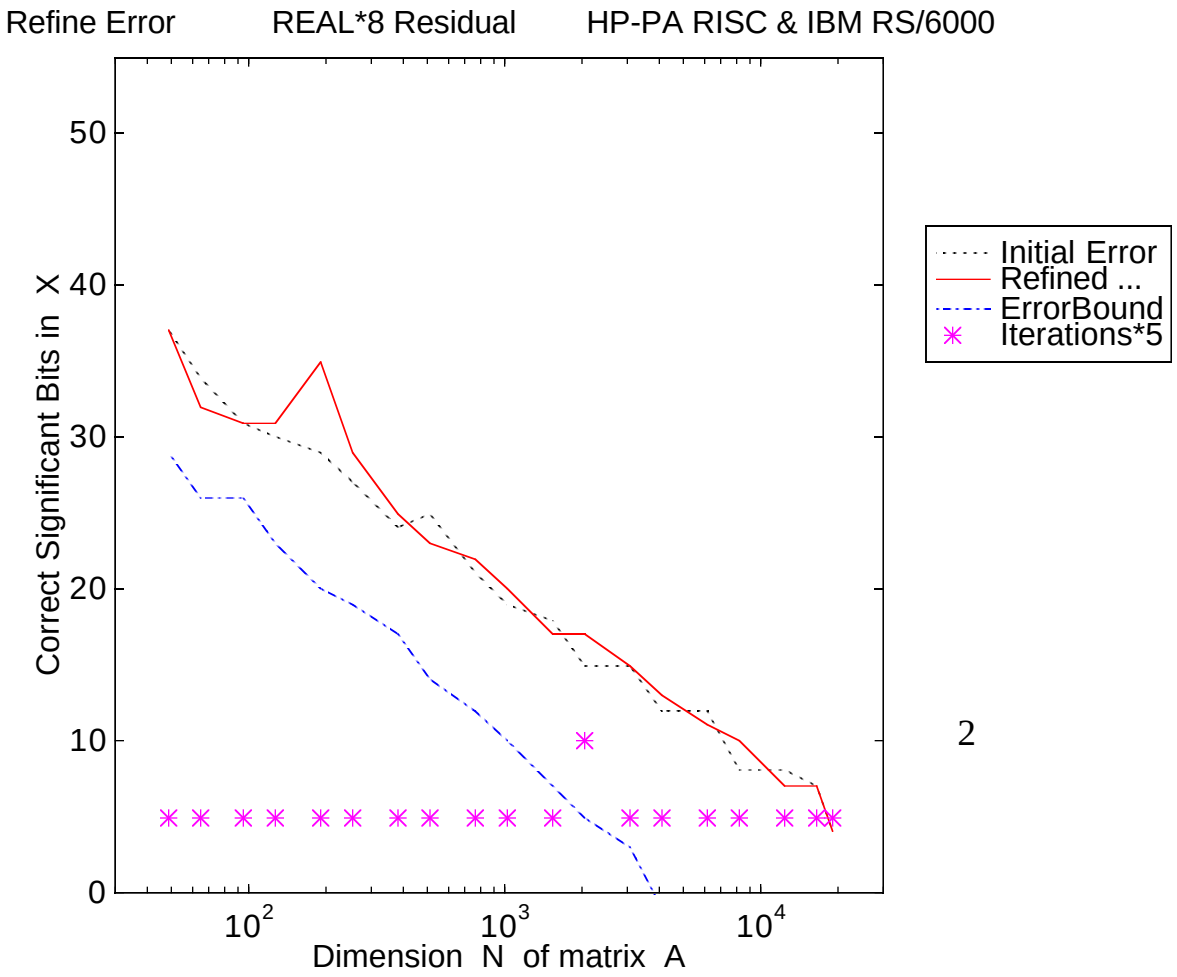
ACCURACY
of a Cantilever's Deflection after Iterative Refinement by a MATLAB 4.2 program run on workstations



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.

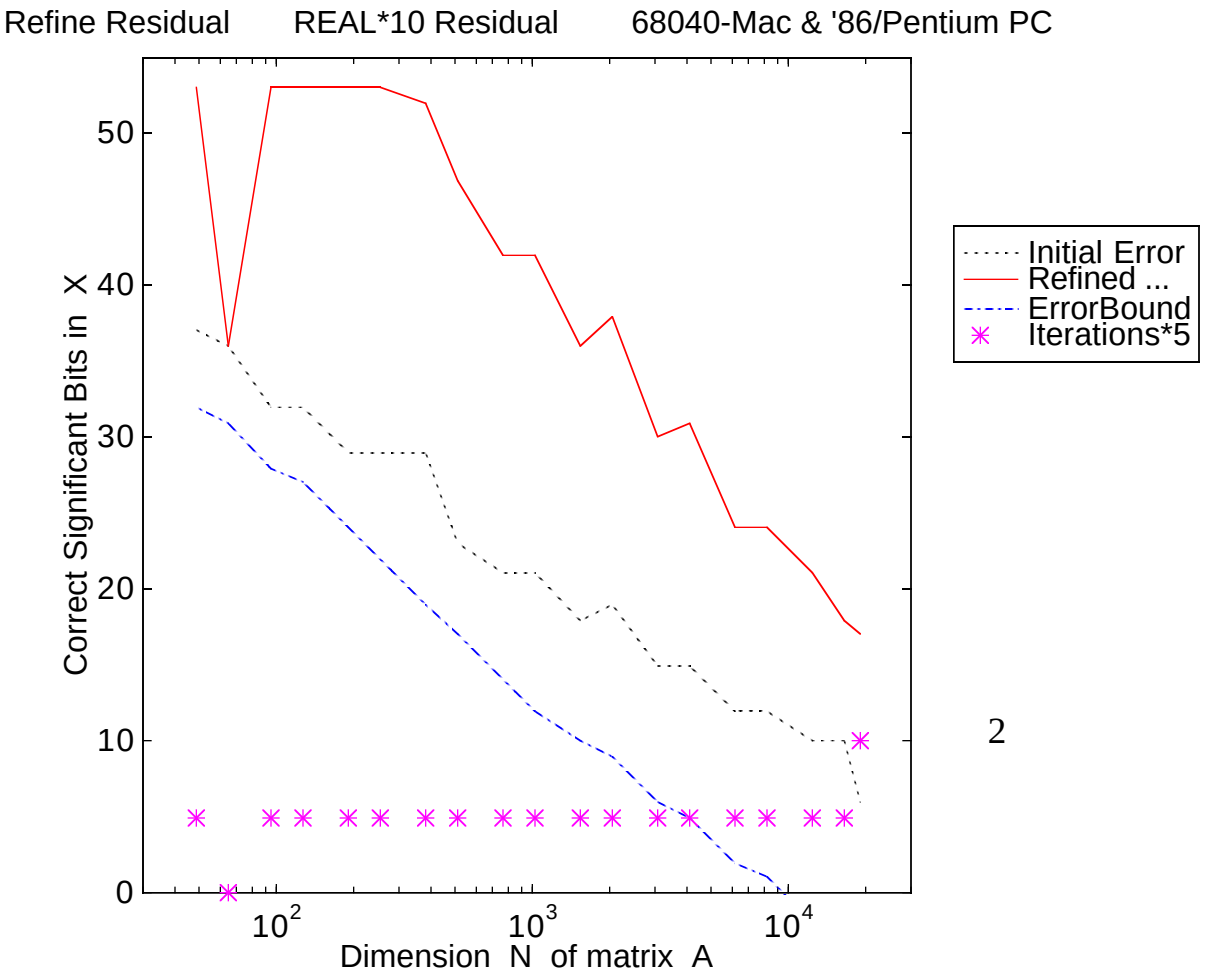
ACCURACY
of a Cantilever's Deflection after Iterative Refinement by a MATLAB 4.2 program run on workstations



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.

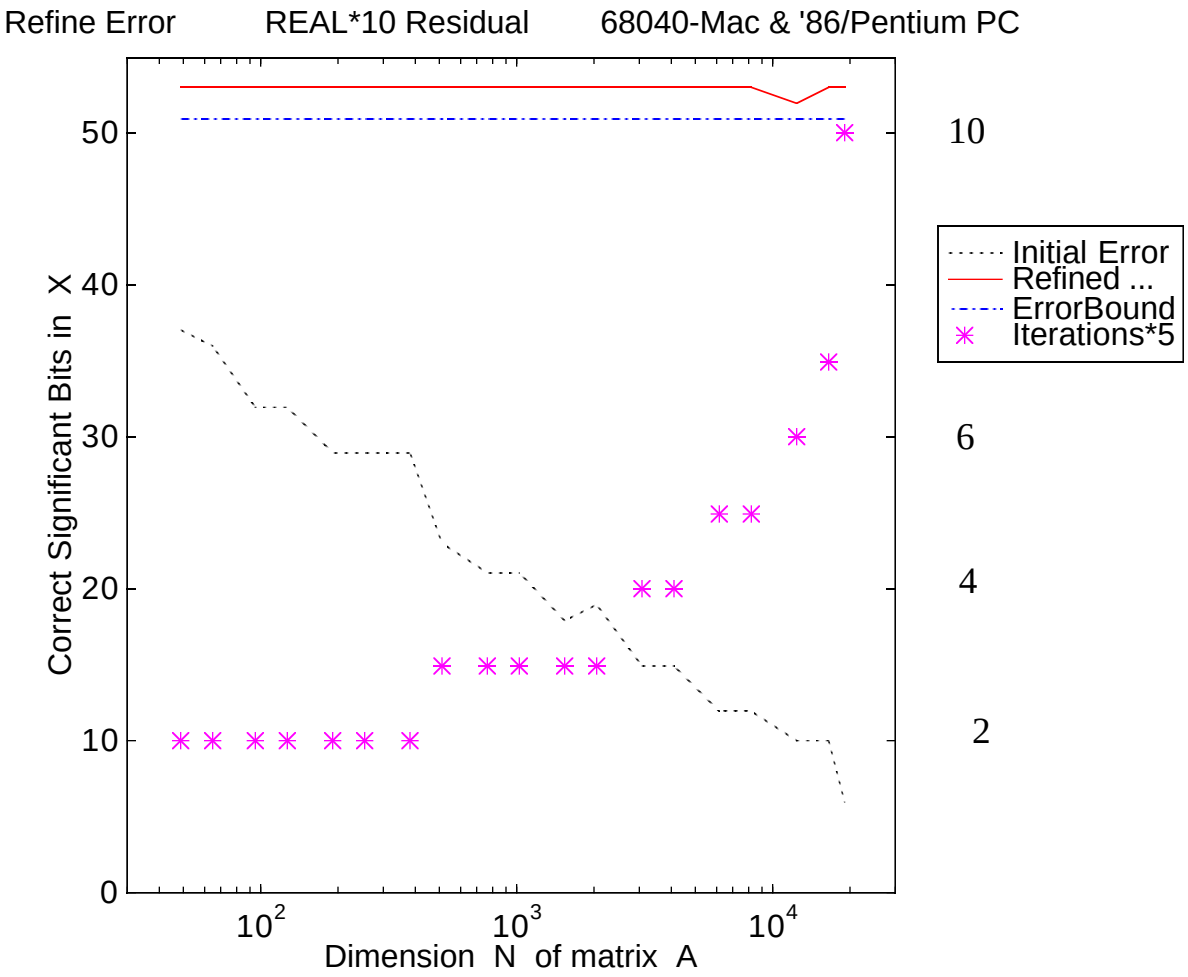
ACCURACY
of a Cantilever's Deflection after Iterative Refinement by a MATLAB 4.2 program run on PCs and old Macs



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, 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, $V = Q^{-1} \cdot B \cdot Q$ should be 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 (he is at UCLA now) can be taken at face value; see his "Finding Well-Conditioned Similarities to Block-Diagonalize Nonsymmetric Matrices is NP-Hard" *Journal of Complexity* **11** (1995), pp. 377-391.

Therefore **`eig(...)`** must be flawed;
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:

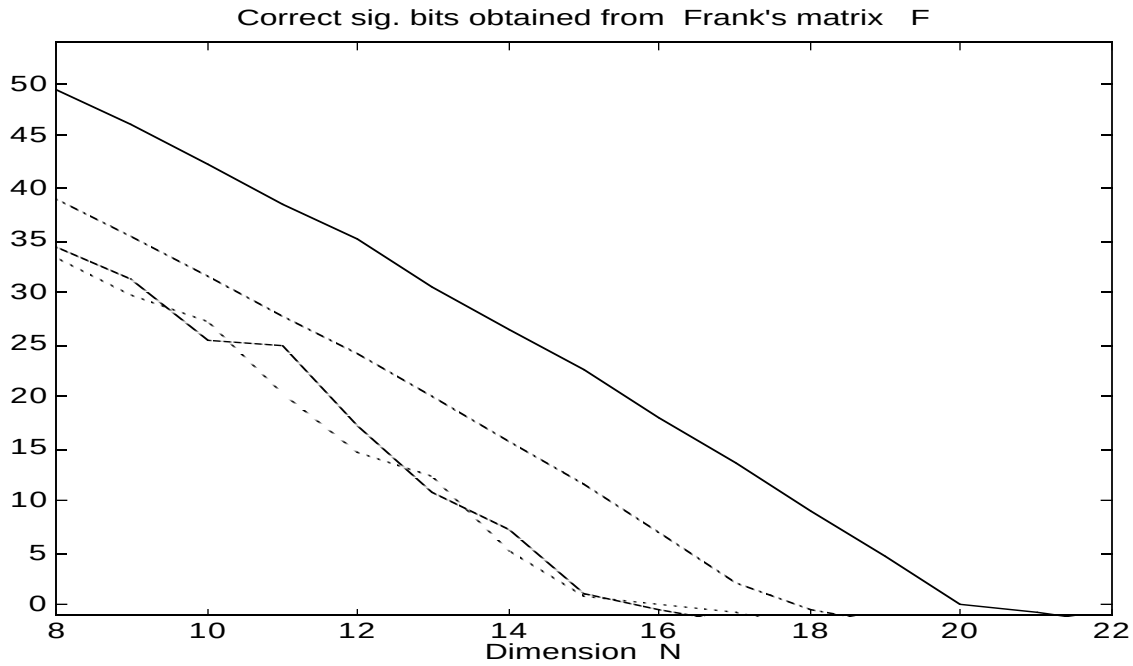
`[Q, V] = RefinEig(Q, V, B)`

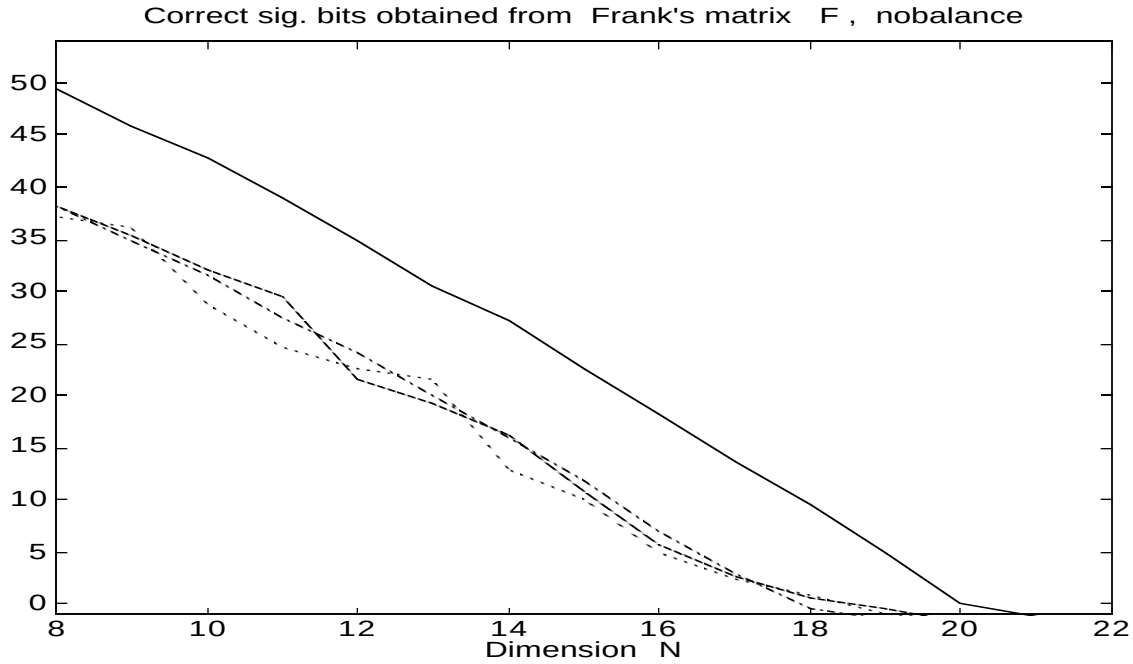
is my *MATLAB*-language program designed to try to improve the accuracy of

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

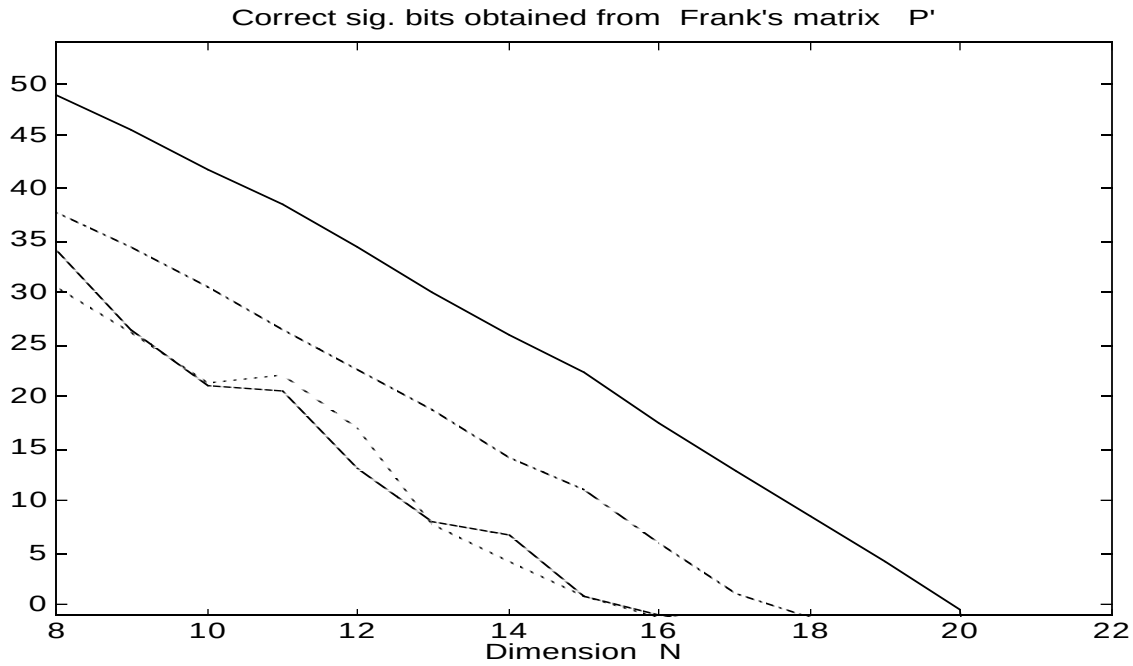
in cases when it has been degraded by some accident.

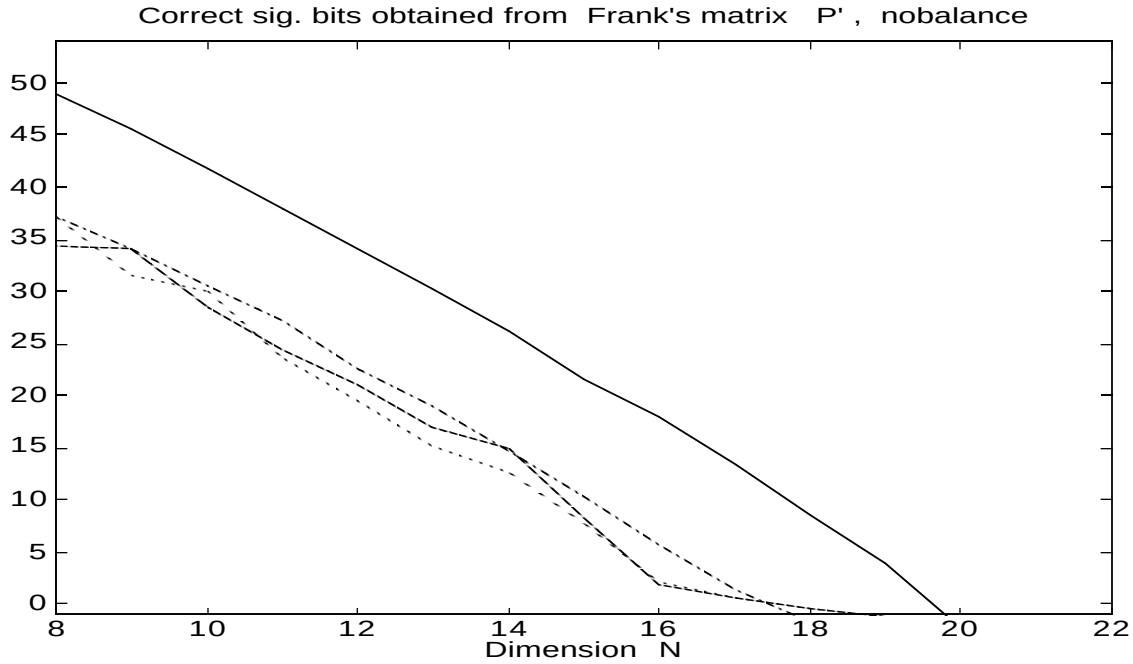
Sometimes the improvement is spectacular.



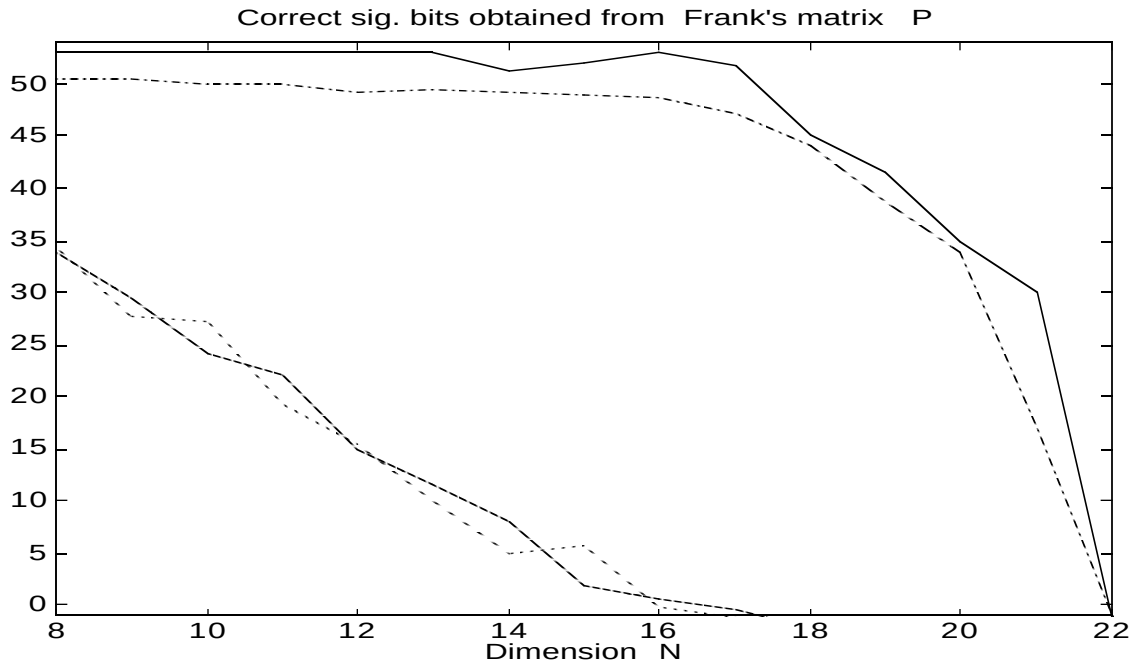


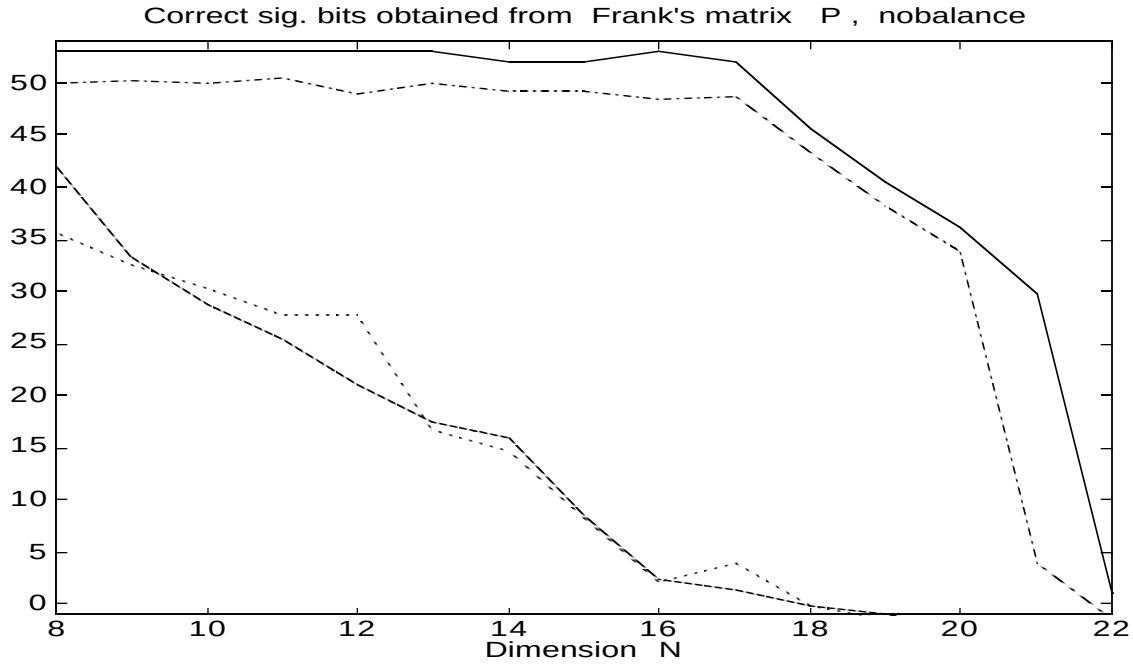
Legend:
- `-----` `eig` on 680x0-Mac or Intel-PC
- `_____` `RefinEig` on 680x0-Mac or Intel-PC
- `......` `eig` on others
- `·-·-·-·` `RefinEig` on others.





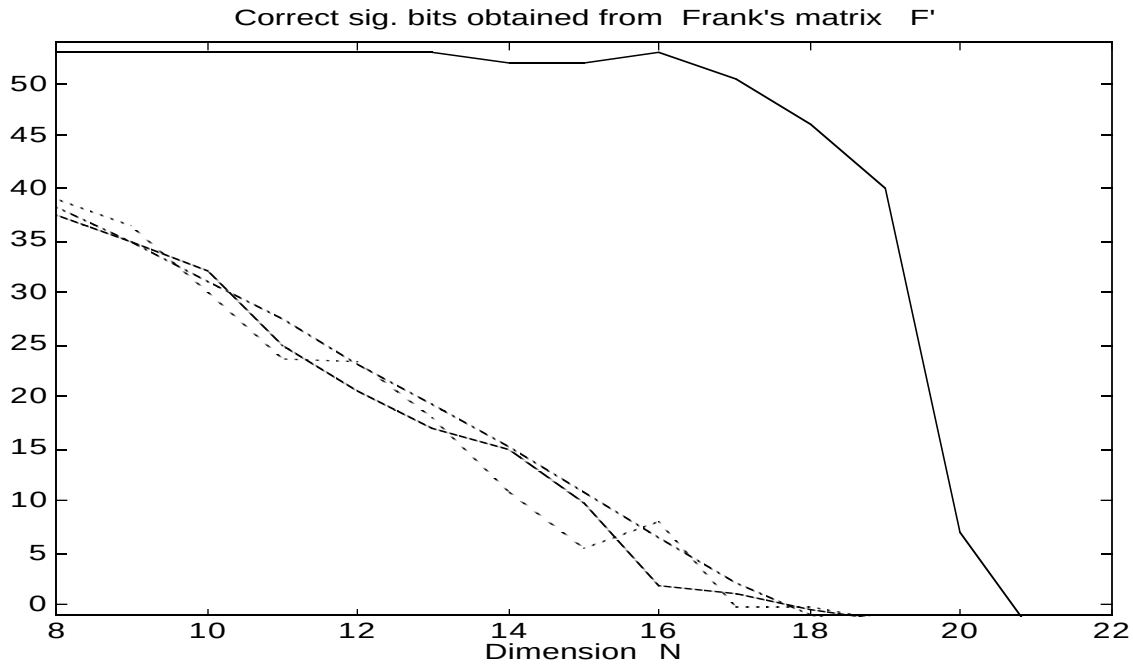
Legend:
- `-----` `eig` on 680x0-Mac or Intel-PC
- `_____` `RefinEig` on 680x0-Mac or Intel-PC
- `......` `eig` on others
- `·-·-·-·` `RefinEig` on others.

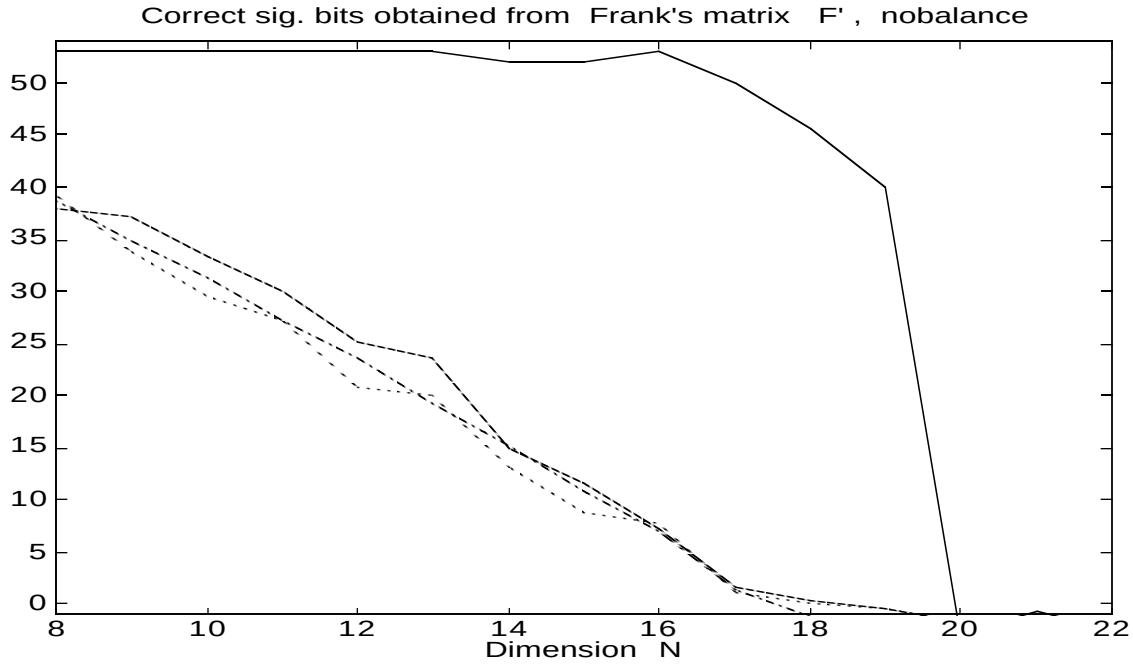




Legend:

-----	`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 without ever mentioning it?

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

Scalar Product = $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 + ... + a_N \cdot b_N$.

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.

The Threat: Atrophy and Stagnation

For lack of benchmarks that assess accuracy or other desirable attributes
other than speed,
Apple's Standard Apple Numerical Environment (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 and helpful for diagnosing numerical instability.

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.

Mediæval 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 new product 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.

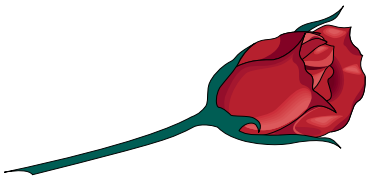
… and now Java.

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 a bit;
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.