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

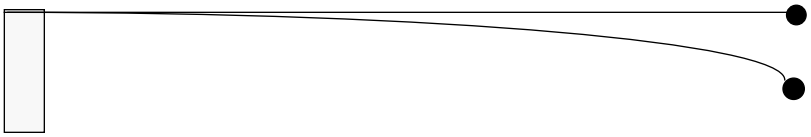
Abstract:
By far the majority of computers in use to-day are AMD/Cyrix/Intel-based PCs, and a big fraction of the rest are old 680x0-based Apple Macintoshes. Owners of these machines are mostly unaware that their floating-point arithmetic hardware is capable of delivering routinely better results than 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. Their solution is programmed in MATLAB 4.2 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* that purports to deliver identical numerical results on all computers from one library of numerical software.

The MATLAB 4.2 program used to get the aforementioned results is available by electronic mail from the authors: ivory@cs.berkeley.edu and wkahan@cs... . For related work see also http://http.cs.berkeley.edu/~wkahan/triangle.ps .

This document was created with FrameMaker 4.0.4

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 each of forward and 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, the usual way to compute $\mathbf{x}$, is extremely 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 much better.

Only on AMD/Cyrix/Intel-based PCs and 680x0-based Macs (not Power–Macs) can Iterative Refinement *always* improve the accuracy of $\mathbf{x}$ substantially *provided* the program is not prevented by a feckless language or 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 4.2 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 a bug in MATLAB 4.2 made its stack overflow on a Pentium with 64 MB RAM . 80 MB on a Mac went no better.
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 4.2 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 these 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 all software ought to work identically on every computer.

Every iterative refinement program repeats the three steps

{ $\mathbf{r} := \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$; *solve* $\mathbf{A}\cdot\Delta\mathbf{x} = \mathbf{r}$ for $\Delta\mathbf{x}$; *update* $\mathbf{x} := \mathbf{x} - \Delta\mathbf{x}$; }

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 $\mathbf{r} := \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ 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. It carries 64 sig. bits to compute residuals. Coincidence?

> 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 less 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 $\mathbf{r} := \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ 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

$\mathbf{r} := (\mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ plus roundoff)

happens to vanish. Iterative Refinement produces residuals that vanish surprisingly often and not necessarily because **x** is exactly right.

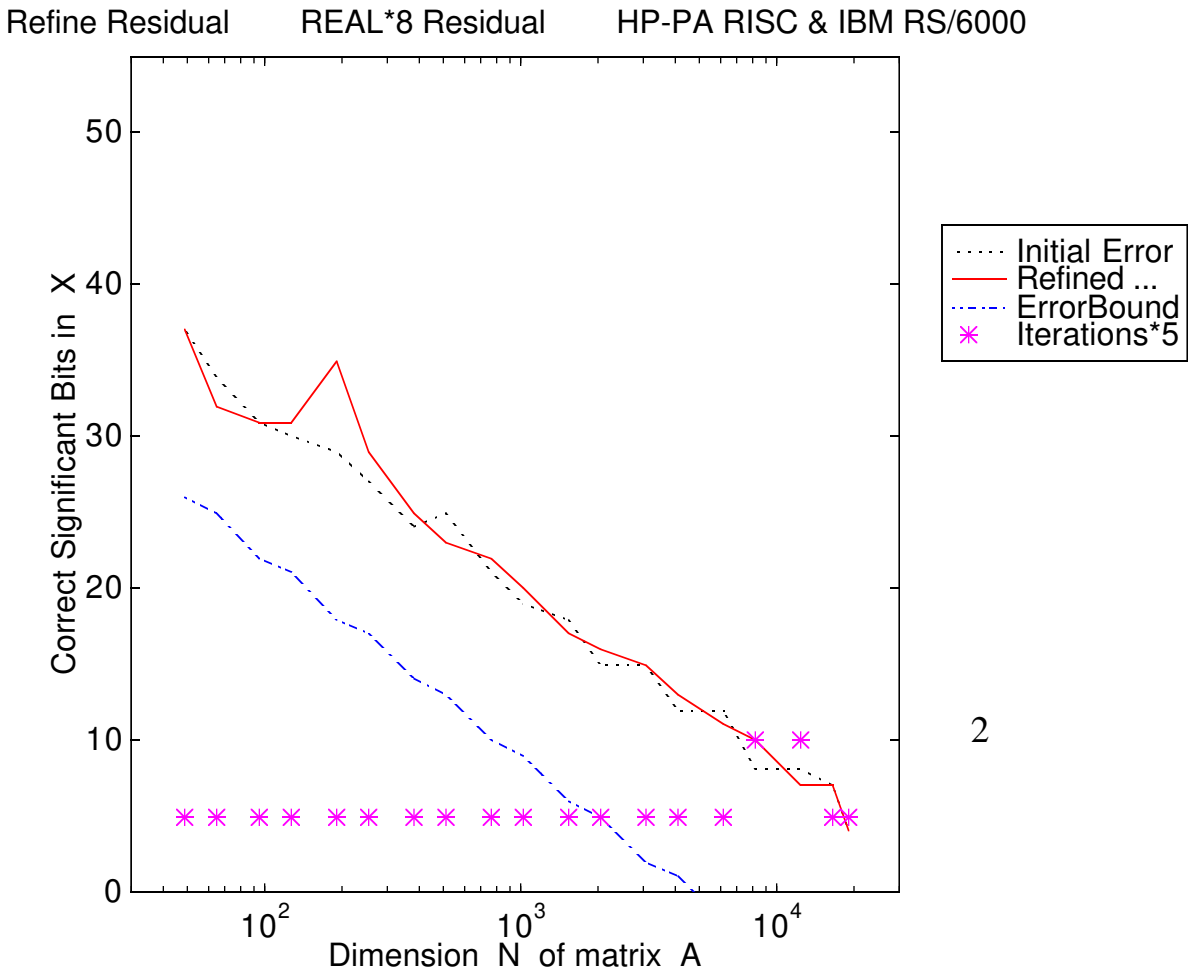
(The INEXACT flag mandated by IEEE Standard 754 for Binary Floating–Point Arithmetic would, if MATLAB granted us access to that flag, help us discriminate between solutions **x** that are exactly right and those, perhaps arbitrarily wrong, whose residuals vanish by accident.)

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.

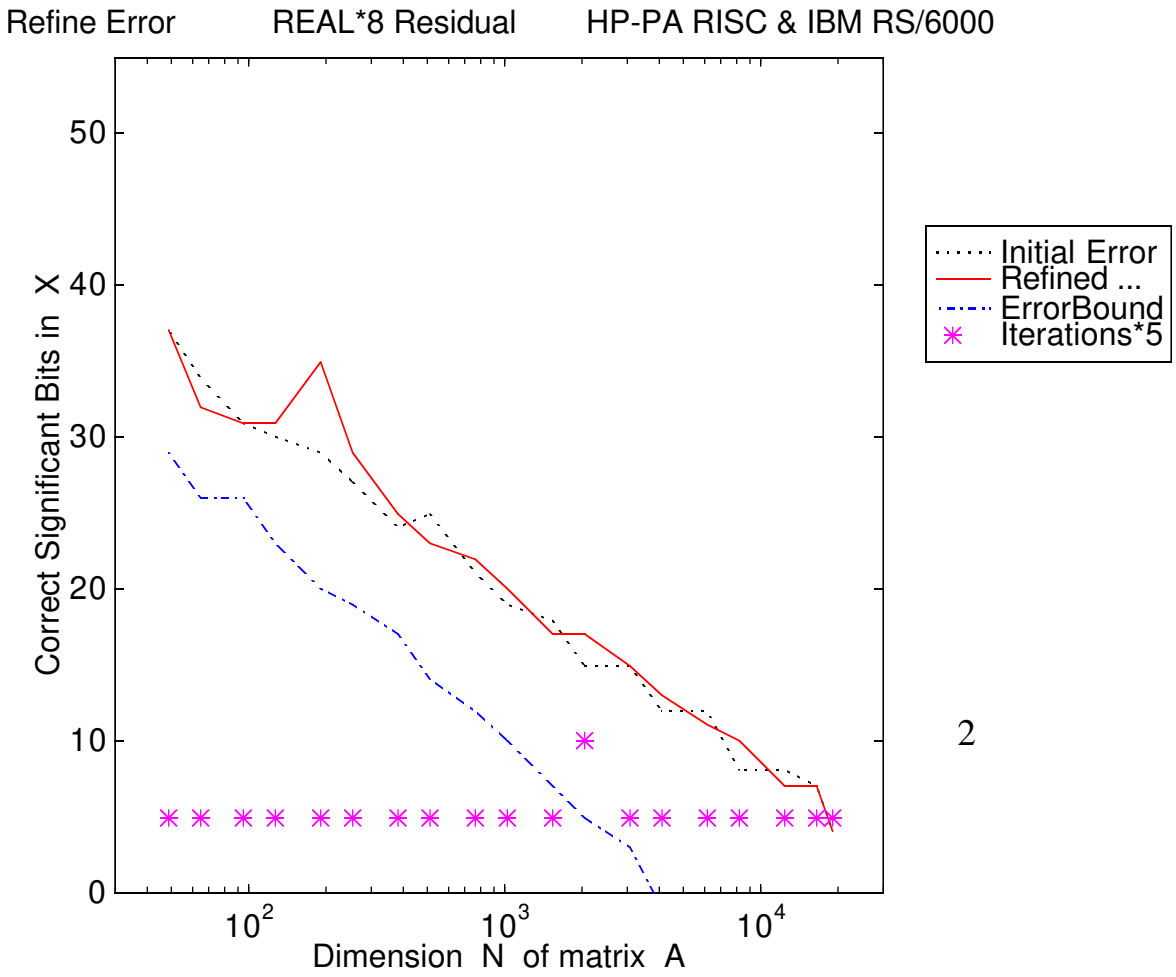
ACCURACY
of a Cantilever's Deflection after Iterative Refinement
by a MATLAB 4.2 program run on workstations



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 sometimes degrades it a little, on workstations that do not accumulate residuals to extra precision. And the error-bound on **x** inferred from **r** is often too pessimistic. But to do better on those workstations is too difficult in MATLAB to be worthwhile.

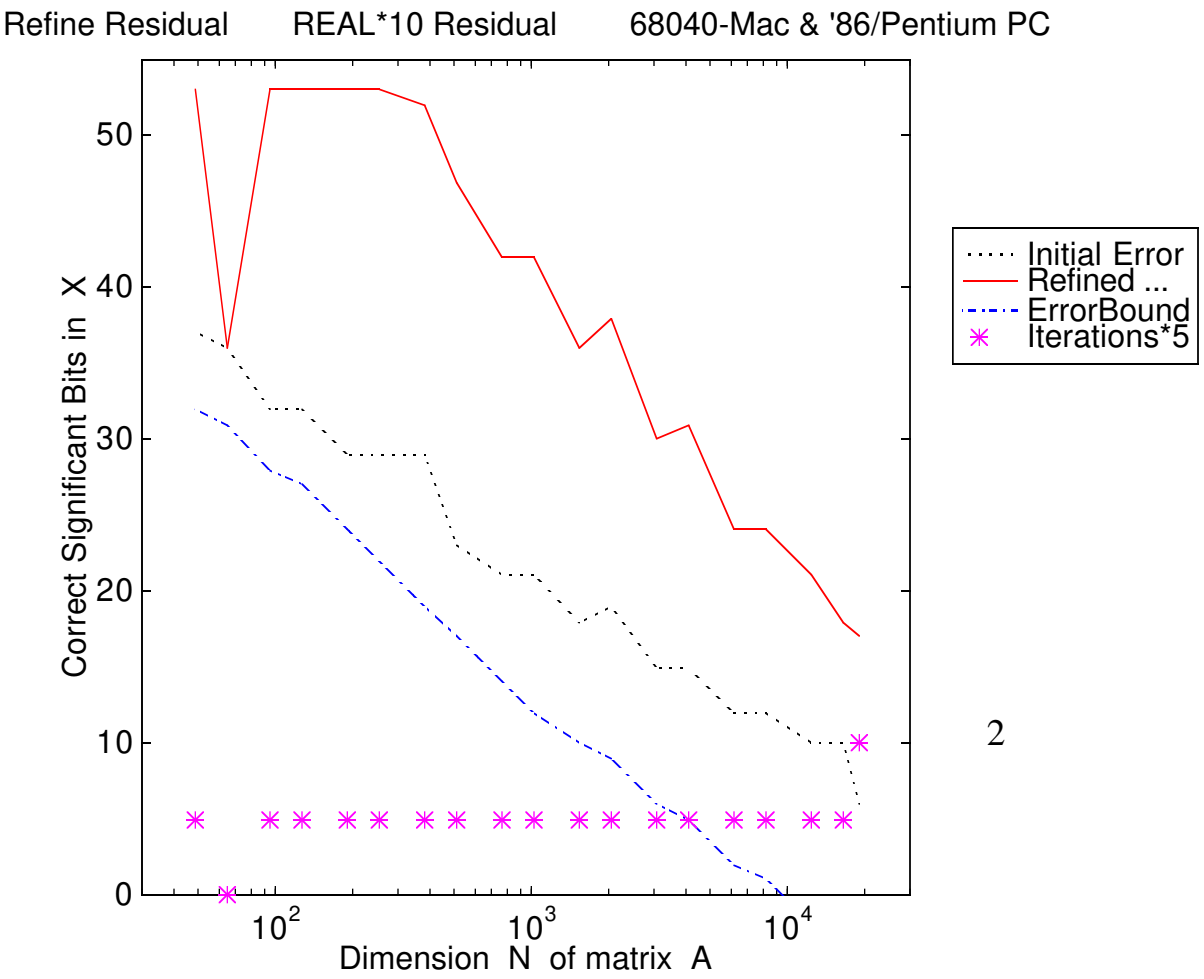
ACCURACY
of a Cantilever's Deflection after Iterative Refinement
by a MATLAB 4.2 program run on workstations



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.

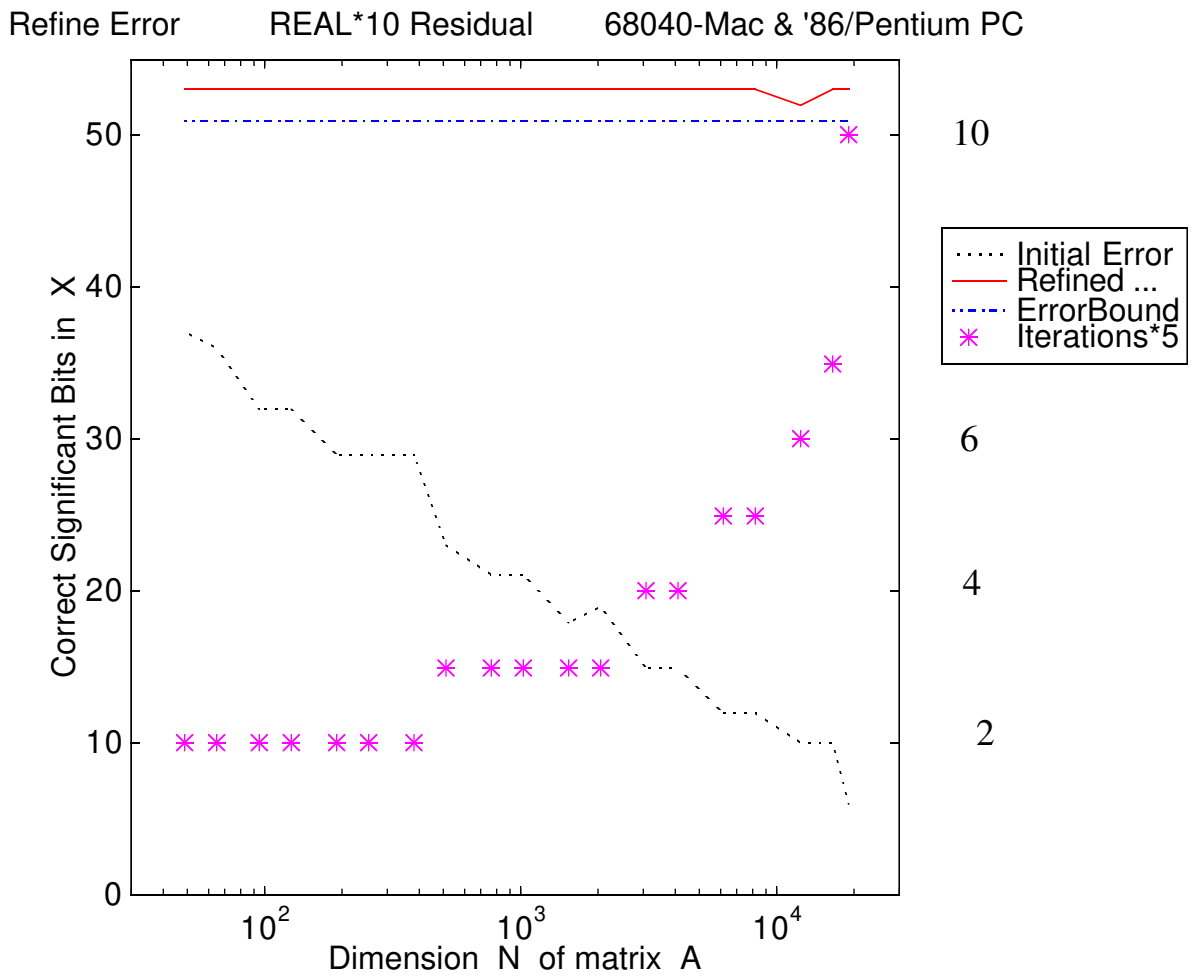
ACCURACY
of a Cantilever's Deflection after Iterative Refinement by a MATLAB 4.2 program run on PCs and old Macs



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.

How does our program get MATLAB 4.2 to compute residuals with 11 extra bits of precision, on machines that have it, without mentioning it and without incurring a noticeable speed penalty? The program uses two tricks, both justifiable by their contribution to speed regardless of their contribution to accuracy.

The first trick is built into MATLAB 4.2 and earlier versions. To compute non–sparse matrix products $\mathbf{P} = \mathbf{C}\cdot\mathbf{Q}$ as quickly as it can, MATLAB accumulates the scalar products $p_{ij} = \sum_k c_{ik}\cdot q_{kj}$ in the computer's internal registers rather than store each partial sum in memory only to reload it immediately afterwards. These internal registers carry 64 sig. bits on AMD/Cyrix/Intel–based IBM–compatible PCs and 680x0–based Apple Macintoshes, more than the 53 sig. bits carried in the other machines' registers and stored by MATLAB into 8-byte words in all machines' memories. Therefore MATLAB's matrix products $\mathbf{P} = \mathbf{C}\cdot\mathbf{Q}$ tend to be more accurate, sometimes much more, on PCs and old Macs than elsewhere.

With the possible exception of exponentiation z^y , all operations other than non–sparse matrix multiplication appear to be carried out by MATLAB without taking advantage of extra–precise registers. This would contaminate residual $\mathbf{r} = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ by rounding $\mathbf{A}\cdot\mathbf{x}$ from 64 to 53 sig. bits before it has a chance to mostly cancel with $\mathbf{b}$. To avoid this contamination we should compute $\mathbf{r}$ in one matrix–vector multiplication $\mathbf{r} = [\mathbf{A}, \mathbf{b}]\cdot[\mathbf{x}; -1]$, and we would do so were $\mathbf{A}$ not sparse.

The second trick exploits the repetitiveness of $\mathbf{A}$'s rows; all but the first two and last two are [1, -4, 6, -4, 1] amidst long strings of zeros. We copy $\mathbf{x}$ and $\mathbf{b}$ into

$\mathbf{Z}$ = [[0;0;0;0;**x**], [0;0;0;**x**;0], [0;0;**x**;0;0], [0;**x**;0;0;0], [**x**;0;0;0;0], [0;0;**b**;0;0]]

and compute $\mathbf{s} = \mathbf{Z}\cdot[1; -4; 6; -4; 1; -1]$. Discarding its first three elements yields all but the first one and last two elements of $\mathbf{r}$, which are computed separately. The copying, on machines with fast memory–to–memory block transfers, goes at least about as fast the extraction of elements from MATLAB's sparse matrices, so no machine suffers a noticeable performance penalty to execute a trick that enhances accuracy only on PCs and old Macs.

Evidently *Wider is Better* where arithmetic reliability is concerned, but part of the computing industry has a different agenda. *JAVA* forbids the use of extra precision, and Microsoft's *Windows* turns it off. It brings a line from *Othello* to mind:

" Perplex'd in the extreme ..., threw a pearl away
Richer than all his tribe."