Matlab’s Loss is Nobody’s Gain

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

Abstract:

Matlab has become the software package most widely used by engineers and scientists for their numerical computations. Part of its appeal arose from its early availability on Intel x86-based IBM PCs and their clones, the hardware most widely used by engineers and scientists for their numerical computations nowadays. Almost all these users are unaware of enhanced accuracy Matlab used to enjoy on PCs but lost in its latest releases, versions 5.x , under Windows. Offered here is a function `mxmuleps` that, when run on various versions of Matlab on various computers, tells which ones have enhanced accuracy, whose benefits are illustrated by examples. Also addressed are pervasive misunderstandings about how much that lost accuracy was worth.

Contents:

Matlab's Loss is Nobody's Gain

Introduction:

A Matlab function `mxmuleps` has been programmed to reveal Matlab's roundoff threshold for matrix multiplication. This is smaller than `eps`, the roundoff threshold for almost all other expressions evaluated by Matlab, on certain machines whose *extra-precise internal registers* carry more sig. bits than Matlab stores in the `double` (8-byte) format it uses for its floating-point variables. Earlier versions of Matlab used these extra bits during matrix multiplication whenever they were available; Matlab 5.x continues to do so only on 680x0-based Apple Macintoshes, and later versions of Matlab may not support these machines. On the ubiquitous Intel Pentium-based PCs and their clones running under Microsoft Windows™, Matlab now eschews use of extra precision that would enhance the accuracy of its results on these machines. Why Matlab 5.x abandoned the extra accuracy is a question not answered in this note, which first explains how `mxmuleps` reveals whether Matlab accumulates matrix multiplication to extra precision and then illustrates its benefits by examples, some of them surprising.

Example 1 shows how difficult is the task of designing a benchmark to compare the accuracies of different computer systems. Two programs designed to compute the same integer-valued function of an integer argument, but one program rather more accurately than the second, are run on versions of Matlab some of which compute exponentials `x^n` rather more accurately than others. Can you guess which programs and which versions of Matlab will produce the best and which the worst results? Don't bet on it.

Example 2 is a program that normally benefits from arithmetic with greater precision but, when run on computers with a *Fused Multiply-Accumulate* (FMAC) instruction that commits fewer rounding errors than other computers do, gets disconcertingly worse results. However, revising the program to exploit an FMAC, when it is available, then produces impeccable results.

Example 3 applies *Iterative Refinement* to offset the effects of certain numerical pathologies upon computed solutions of linear systems $\mathbf{A}\cdot\mathbf{z} = \mathbf{b}$. The process is reliable only if residuals $\mathbf{A}\cdot\mathbf{x}-\mathbf{b}$ are accumulated extra-precisely; but this interpretation of experimental and theoretical results is obscured by misunderstandings of *Backward Error-Analysis* spawned by treacherous mathematics. A fair appraisal of accuracy's worth is subtle; it reduces risks though they are not random. This is illustrated again by Example 4, a simple geometrical computation.

Modest modifications are proposed to restore Matlab's superior accuracy when it runs on Intel-based PCs and clones, even if results from other platforms remain no better than they are now, without detracting from Matlab's speed. This proposal runs counter to a notion promulgated with Java™ to the effect that all computers should deliver identical results; but that notion is inimical to both speed and accuracy so long as computer architectures remain diverse.

An appendix presents a Matlab program `divulge.m` that divulges small divergencies among the arithmetic properties of different versions of Matlab on diverse platforms.

What `eps` means:

Matlab's `eps` is the difference between 1.0 and the next larger floating-point number stored in the `double` (8-byte) format Matlab uses for all its floating-point variables. Currently `eps` $= 1/2^{52} \approx 2.2/10^{16}$ for all computers on which Matlab runs; earlier versions of Matlab ran on computers with diverse arithmetics and consequently diverse values of `eps` . Currently all arithmetics on which Matlab runs conform to IEEE Standard 754 for Binary Floating-Point Arithmetic and round every rational (+, –, ·, /) and square root (√) operation upon `double` operands by default to 53 sig. bits or more. Most conforming hardware carries more, at least 64 sig. bits, in an internal register file that provides extra accuracy for subexpression evaluation; however the extra sig. bits can be turned off if a programmer doesn't want them perhaps because he prefers to get the same results as he would get from computers that lack the 11 extra bits.

In the past, Matlab did not turn the 11 extra bits off, but discarded them after evaluating all but a few kinds of (sub)expressions. The difference between " discard " and " turn off " is subtle but can be detected easily. In Matlab, compute

```
e = eps ,   z = 1 + (1 + e)*e/2 ,   d = z - 1
```

to see the following results displayed …

if extra bits are turned off: `e = 2.2204e-16 , z = 1.0000 , d = 2.2204e-16` .
if extra bits are discarded: `e = 2.2204e-16 ,  z = 1 ,      d = 0` .

(To distinguish extra bits turned off on an old 680x0-based Mac from nonexistent extra bits on other computers is impossible, and on a PC requires an expression that barely underflows. Such an expression can be found in the program `divulge.m` in an appendix at the end of this note.)

Here is the explanation for the results above. If the extra bits are turned off or nonexistent, the right-hand side expression computed for `z` exceeds `1 + eps/2` before it is rounded off to 53 sig. bits, and consequently rounds to `z = 1 + eps` , the `double` number next larger than 1 . If the 11 extra bits are active in the computer's registers, the right-hand side expression is first rounded to `1 + eps/2` in 64 sig. bits, and then rounded again to the `double` number `z = 1` in 53 sig. bits when its last 11 bits are discarded as it is stored.

During non-sparse matrix multiplication, and during exponentiation $(\text{real})^{\text{integer}}$, Matlab used to keep the extra sig. bits, thus saving both time and higher accuracy on Pentiums and old Macs. Now Matlab turns off the Pentium's extra 11 sig. bits. This will seems perverse to vastly many Pentium owners thus denied some of the accuracy they paid for when they bought their hardware. Those 11 extra bits were not discarded to ensure Matlab will get identical results on all hardware since this is impractical for several reasons of which one comes next.

The Fused Multiply-Accumulate (FMAC) Instruction:

Hardware conforming to IEEE 754 is often augmented by features that can accelerate matrix multiplication if programmers exploit them. One such feature is pipelined arithmetic; it allows arithmetic operations to be initiated at the rate of one per computer clock cycle though each operation may take a few cycles to complete. At any moment a few arithmetic operations can be simultaneously in progress through the pipeline. A programmer can exploit this capability by computing at least two elements p_{ij} of a matrix product $P = X \cdot Y$ simultaneously, interleaving the products and sums in $p_{ij} = x_{i1} \cdot y_{1j} + x_{i2} \cdot y_{2j} + x_{i3} \cdot y_{3j} + \ldots$ with those of a neighbor, thereby also reusing elements of X and Y fetched from slow memory into a cache or faster registers.

Expensive hardware with more than one floating-point pipeline can keep them all full only by issuing more than one instruction per cycle, thus moving at least four operands per cycle from the register file to the pipelines and at least two results per cycle back. Less expensive hardware can multiply matrices at the same speed by using a Multiply-Accumulate (MAC) instruction that moves three operands per cycle into the pipeline and one back after performing a multiplication and addition; this instruction evaluates one expression of the form $\pm s \pm x \cdot y$ per cycle. A *Fused* Multiply-Accumulate (FMAC) instruction evaluates this expression with one rounding error instead of two. Few hardware architectures have a FMAC; a few more have a MAC, and most have neither. How do results computed with a FMAC differ from those computed without?

The difference is subtle enough to require some notation to explain it. Let $\{x + y\}$ stand for the rounded value of $x + y$, and $\{x \cdot y\}$ for the rounded value of $x \cdot y$, thus distinguishing values rounded to 53 sig. bits from exact values uncontaminated by roundoff. The **C** programmer who writes " T = S + R*Q " intending to compute $s + r \cdot q$ actually gets instead $t = \{s + \{r \cdot q\}\}$ with two rounding errors, except on a machine whose FMAC produces $t = \{s + r \cdot q\}$ with only one rounding error. This FMAC has advantages and disadvantages.

The obvious advantage, speed, is shared with an unfused MAC that delivers a result with two rounding errors the same as slower separate multiplication and addition operations deliver. A less obvious advantage of a FMAC is slightly more accurate matrix products, which tend to an extra sig. bit of accuracy because they accumulate about half as many rounding errors. An unobvious advantage of a FMAC, and the prime reason for its existence, is a trick that computes correctly rounded quotients at adequate speed without division hardware built specifically for that purpose. The trick can also be used to compute a product $r \cdot q = s + t$ exactly as a sum of two floating-point numbers with two FMACs thus: Given r and q first obtain $s = \{r \cdot q + 0\}$ and then $t = \{r \cdot q - s\}$; it turns out that $t = r \cdot q - s$ exactly. This trick has many applications but here is not the place to explain them.

The FMAC has subtle disadvantages that argue against its indiscriminate use and hence deserve explanation. The trouble is that FMACs render statements like " T = U*V – R*Q " ambiguous in languages like **C** or Fortran. Which of $t = \{\{u \cdot v\} - r \cdot q\}$ and $t = \{u \cdot v - \{r \cdot q\}\}$ does the programmer prefer? Even if he can indicate his choice by inserting algebraically redundant parentheses, say " T = (U*V) – R*Q " for the first outcome or " T = U*V – (R*Q) " for the second, or by some other linguistic convention, the trouble does not go away; how shall he decide which to choose? The choice almost never matters much, but now and then it matters noticeably, as we shall see.

For instance, the product of a complex number $z = x + \imath y$ and its complex conjugate $x - \imath y$ should be the real number $x^2 + y^2 + \imath(y\cdot x - x\cdot y) = x^2 + y^2$, and is real in rounded arithmetic without a FMAC; but a FMAC can deliver a tiny nonzero value $\{y\cdot x\} - x\cdot y$ or $y\cdot x - \{x\cdot y\}$ for the imaginary part. This anomaly could probably be ignored if it did not have to be explained.

In Matlab the FMAC’s anomaly shows up when a complex column `c` turns out to have a squared norm `c'*c` that is a slightly complex number instead of the expected nonnegative real number. The anomaly shows up again when, for most complex matrices `C`, the hermitian products `A = C'*C` produce slightly nonzero differences `A–A'` instead of exactly vanishing differences obtained on machines with no FMAC. Resetting `A = 0.5*(A+A')` cures that.

A simpler way to avoid its anomalies is to eschew the FMAC; but similar and equally avoidable anomalies would persist in Matlab even without FMACs. For instance, although every complex hermitian matrix `A = A'` has only real eigenvalues, Matlab’s `eig(A)` still delivers, on all computers, eigenvalues with tiny nonzero imaginary parts and occasionally with

utterly non-orthogonal eigenvectors ; *this last defect is a blunder.*

And if `B` has the same dimensions as `C` above then the hermitian character of `H = B*A*B'` is slightly spoiled by roundoff no matter whether all data is real or no FMAC is used; the best cure is to reset `H = 0.5*(H+H')` .

Matlab’s non-sparse matrix multiplication exploits a FMAC or MAC on any hardware that has one, rather than forego a factor of 2 in speed; and `mxmuleps` determines whether a FMAC is in use that way. When evaluating most other expressions Matlab eschews the FMAC.

How `mxmuleps` works:

The constant `eps`, a constant function in Matlab 5.x , used to be a global variable in earlier versions. In case `eps` has been changed `mxmuleps` compares it with the computed value of

```
u  =  abs( (4.0/3 - 1)*3 – 1 )
```

which produces the same result as $u = 1.000\ldots0001 - 1.000\ldots0000$ from binary floating-point arithmetic correctly rounded to the same precision as stored variables; this difference u is one ULP (Unit in the Last Place stored) of `double` numbers barely bigger than 1 . If comparison reveals that `u` $\neq$ `eps` then `mxmuleps` issues a warning message asking whether `eps` has been changed. Such a change could invalidate `mxmuleps`’s computation, as would non-binary or incorrectly rounded floating-point arithmetic.

How does the computation of u work? In binary, $4/3 = 1.010101\ldots$. The rounded quotient $z = \{4/3\}$ differs from $4/3$ by $\pm1/3$ ULP . Then $(z - 1)$ suffers no rounding error; its last two sig. bits are zeros. Consequently $(z - 1)\cdot 3$ is computed exactly and differs from 1 by just ±1 ULP .

The computation of u foreshadows the way `mxmuleps` works. A Matlab scalar product

$$s = [\, x_1, x_2, x_3 \,]*[\, y_1; y_2; y_3 \,] = x_1\cdot y_1 + x_2\cdot y_2 + x_3\cdot y_3$$

that would vanish in the absence of roundoff is designed to expose whatever roundoff occurs.

For this purpose $x_1 \cdot y_1 = \{x_1 \cdot y_1\}$ and $x_3 \cdot y_3 = \{x_3 \cdot y_3\}$ have been chosen to be computable exactly; and $x_2 \cdot y_2$ has been chosen as close to 4/3 as possible so as to differ from its rounded value « $x_2 \cdot y_2$ » by very nearly ±1/3 ULP of the precision Matlab carries during matrix multiplication, provided this precision is less than twice the precision of `double`. Otherwise, or if Matlab uses a FMAC, the product $x_2 \cdot y_2$ is carried forward exactly to the subsequent addition, either $x_1 \cdot y_1 + x_2 \cdot y_2$ or $x_2 \cdot y_2 + x_3 \cdot y_3$ depending upon the order in which scalar product s is evaluated.

Since s = 0 in the absence of roundoff, both $x_1 \cdot y_1 + x_2 \cdot y_2 = -x_3 \cdot y_3$ and $x_2 \cdot y_2 + x_3 \cdot y_3 = -x_1 \cdot y_1$ are computable exactly; thus the computed s = 0 whenever Matlab uses a FMAC or at least twice the precision of `double` during matrix multiplication. Currently no hardware supports quadruple-precision floating-point fully, so it runs too slowly for Matlab to use it routinely for matrix multiplication; for the time being s = 0 implies the use of a FMAC. In this case `mxmuleps` returns `NaN` rather than suggest wrongly that matrix multiplication is exact.

A nonzero value computed for s is intended to expose the rounding error committed when $x_2 \cdot y_2$ was rounded to a slightly different value « $x_2 \cdot y_2$ » perhaps more accurate than $\{x_2 \cdot y_2\}$. Now the order in which the scalar product is evaluated matters; the computed value of s is either

$$\{\text{« « } x_1 \cdot y_1 + \text{« } x_2 \cdot y_2 \text{ » » } + x_3 \cdot y_3 \text{ »}\} \quad \text{or} \quad \{\text{« } x_1 \cdot y_1 + \text{« « } x_2 \cdot y_2 \text{ » } + x_3 \cdot y_3 \text{ » »}\}\ .$$

Matlab opts for the first on some machines, the second on others, so `mxmuleps` computes both. The second happens to vanish. In the first the choice $x_1 \cdot y_1 \approx -\{x_2 \cdot y_2\}$ implies that $x_1 \cdot y_1 +$ « $x_2 \cdot y_2$ » mostly cancels and is therefore computed exactly. Then, since $x_3 \cdot y_3 = -x_1 \cdot y_1 - x_2 \cdot y_2$, the computation of s yields

$$\{\text{« } (x_1 \cdot y_1 + \text{« } x_2 \cdot y_2 \text{ »}) + x_3 \cdot y_3 \text{ »}\} = \{\text{« « } x_2 \cdot y_2 \text{ » } - x_2 \cdot y_2 \text{ »}\} = \text{« } x_2 \cdot y_2 \text{ » } - x_2 \cdot y_2$$

exactly, whence 3·s must be very nearly ±1 ULP of the precision Matlab carries during matrix multiplication.

Since |3·s| is very nearly a power of 1/2 no bigger than `eps`, rounding `eps`/|3·s| to the nearest integer must produce a power of 2 exactly, and dividing `eps` by this power of 2 delivers the result `mxmuleps` promised:

`mxmuleps` = 1 ULP of 1.xxx… at the precision Matlab carries during matrix multiplication unless it uses a FMAC, in which case `mxmuleps` returns NaN .

If `mxmuleps` is the answer, what is the question? It concerns the difference between the intent behind a Matlab assignment `P = X*Y` and what it accomplishes. The computed product **P** usually differs from the exact matrix product **X**·**Y** slightly; how much? Elementwise,

$$|\mathbf{P} - \mathbf{X}\cdot\mathbf{Y}| \le (\texttt{eps}/2)\cdot|\mathbf{P}| + k\cdot ç\cdot|\mathbf{X}|\cdot|\mathbf{Y}|$$

where k is the number of columns in **X** and rows in **Y**, and ç = `mxmuleps`/2 unless it is NaN , in which case ç = `eps`/2 . This inequality applies to real and complex non-sparse matrices. (For sparse matrices k = `max(max( (X ~= 0)*(Y ~= 0) ))` and ç = `eps`/2 .) However, this inequality tends to overestimate the difference |**P** – **X**·**Y**| grossly; rounding errors rarely conspire to reinforce each other as much as this inequality would allow. A more realistic estimate would replace k by something smaller than √k .

The details of the inequality matter to finicky error analysts more than to the rest of Matlab's users, for whom the following summary should suffice:

If `mxmuleps` = `eps` then the error in the computed `P = X*Y` is rarely much worse than if each element of the data **X** and **Y** had been perturbed by an ULP or two, or more if the dimension k is huge.

If `mxmuleps` is NaN then the error in the computed `P = X*Y` is typically about 70% of what it would be were `mxmuleps` = `eps` since FMACs commit fewer rounding errors.

If `mxmuleps` = `eps`/2048 then the computed `P = X*Y` rarely differs noticeably from what would be obtained by rounding off each element of the exact matrix product **X**·**Y** once.

The Program `mxmuleps.m` :

```
function  y = mxmuleps
%  mxmuleps  =  the roundoff threshold for  MATLAB's  matrix multiplication
%            =  eps     if no  Extra-Precise Accumulation occurs    ... (1)
%            =  eps/2048   if  Extra-Precise Accumulation occurs    ... (2)
%            is  NaN   if a Fused Multiply-Accumulate is enabled    ... (3)
%  on three important classes of computers conforming to  IEEE Standard 754
%  for  Binary Floating-Point Arithmetic.  These three classes include ...
%  1)  Sun SPARC;  H-P PA RISC-1;  SGI MIPS;  DEC Alpha;  and some others;
%           and  Matlab 5.x  running on  Intel x86-based  PCs  and clones.
%  2)  Old Apple Macintoshes based upon the  Motorola 680x0;  and
%           Matlab 3.5 and 4.x  running on  Intel x86-based PCs and clones.
%  3)  IBM RS/6000 & Apple Power Mac;  HAL SPARC;  SGI R8000;  H-P PA RISC-2.
%  Matlab 3.5  does not run on computers in class  3)  whereon later versions
%    of  Matlab  use  Fused Multiply-Accumulation  only during non-sparse
%    matrix multiplication so far as I have been able to determine.
%  On computers in class  2)  MATLAB  accumulates  Extra-Precisely  only
%    non-sparse matrix multiplication and perhaps exponentiation  ( y^n )
%    so far as I have been able to determine.    (C)  W. Kahan,  2 Aug. 1998.

e = eps ;  %  eps = 1/2^52  on all machines listed above.
z = 4/3 ;  u = (1-z)*3 + 1 ; % ...  |u| = 1 ULP of  1.xxxx ,  z = 4/3 - u/3
if  abs(u) ~= e,  % ...  Question whether arithmetic is anomalous:
      disp(' Has precision been altered?  Why do the following differ? ...')
      AnULPofOne = abs(u) ,  Eps = e
      disp(' Now  mxmuleps  cannot be trusted.'),  end % ...  if |u| ~= e .
z = (z-1)*4 ;  zzz = [z; z; z] ; % ...  z = (1 - u)*4/3
y = max(abs([-u, 1+u, -1 ; -1, 1+u, -u]*zzz))*3 ; % = 0 for Fused Mult-Acc.
if y == 0 ,  y = NaN ;  else  y = u/round(u/y) ;  end  %... End mxmuleps
```

`mxmuleps` has delivered these results:

Hardware all with eps = $1/2^{52}$	Matlab 3.5	Matlab 4.2	Matlab 5.2
DEC Alpha, H-P PA-RISC-1, SGI MIPS not R8000, Sun SPARC	—	eps	eps
Intel x86/Pentium-based PC & clones	$eps/2^{11}$	$eps/2^{11}$	eps
68040-Macintosh (Quadra 950)	$eps/2^{11}$	$eps/2^{11}$	$eps/2^{11}$
Power Macintosh, IBM PowerPC, SGI MIPS R8000	—	NaN	NaN

How well do these values of `mxmuleps` correlate with the accuracies achievable using Matlab ? This question is explored in the following examples. The first explains why "… `and perhaps exponentiation ( y^n )` …" appears in the comments above; the accuracies of Matlab's exponentiation `y^n` , its matrix multiplication, and the hardware's arithmetic do not correlate.

Example 1 . The Fibonacci Numbers:
The Fibonacci numbers F_n can be generated by the following recurrence:

$F_0 = 0$, $F_1 = 1$, and $F_{n+1} = F_n + F_{n-1}$ for $n = 1, 2, 3, \dots$ in turn.

And this recurrence is a good way to compute them if all of $F_1, F_2, F_3, \dots, F_n$ are needed, up to a point. Two questions require attention:

How accurately can these numbers be computed by Matlab's floating-point arithmetic?

How fast can F_n be computed for each n in a given set of nonconsecutive positive integers?

At first sight these questions may seem to have obvious answers since

$F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, $F_9 = 34$, …

are small integers computable exactly and quickly; F_n costs $n–1$ additions. But what about

$F_{1471} = 1178511447879147184988 0\dots15229$?

It has 308 decimal digits, too many to fit into an 8-byte word, so F_{1471} must be rounded off. Must 1470 operations be performed to compute an approximate value? We'll find a faster way.

First the accuracy question. $F_{78} = 8944394323791464$ is the last Fibonacci number that fits into Matlab's `double` format. $F_{79} = 14472334024676221 > 2^{53} = 9007199254740992$ requires more than that format's 53 sig. bits, so it gets rounded off. So does F_n for every $n > 78$. Still, Matlab's floating-point can compute F_n exactly for larger values of n , up to around 150 , as an unevaluated difference $F_n = f_n – e_n$ between two floating-point numbers. Here's how it's done:

Approximate $F_1, F_2, F_3, \dots, F_n$ respectively by floating-point numbers $f_1, f_2, f_3, \dots, f_n$ obtained in any way that is not excessively inaccurate. Starting with $e_1 = f_1–1$ and $e_2 = f_2–1$, compute $e_k = ((f_k – f_{k-1}) – f_{k-2}) + e_{k-1} + e_{k-2}$ for $k = 3, 4, \dots, n$ in turn. Don't disregard parentheses! They ensure that the first two subtractions' cancellations occur exactly; then the additions occur exactly until $|e_k|$ gets too big, which first occurs typically for some $k > 140$. Until then, $e_k = f_k – F_k$ exactly, as can be confirmed by induction. And e_k can be shown to approximate the error $f_k – F_k$ pretty well for larger values of k too because that error grows about as rapidly as F_k does.

In short, given approximations f_k to the first n consecutive Fibonacci numbers F_k , and provided they are not too inaccurate, we can estimate their errors $e_k = f_k – F_k$ adequately also.

Now we turn to two ways to compute an isolated F_n faster than by performing $n–1$ operations. A simple way faster for very large integers n , taking fewer than $16\cdot\log_2 n$ floating-point multiplications and additions, is based upon an inductively verifiable formula

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} .$$

Here the computation of Y^{n-1} for a matrix Y and positive integer n is accomplished by repeated matrix squarings and multiplications; for example, $Y^{259} = Y\cdot Y^2\cdot(((((((Y^2)^2)^2)^2)^2)^2)^2)^2$ in 10 matrix multiplications instead of 258 .

A way that goes several times faster than matrix multiplication uses approximate (not exact) floating-point arithmetic in an algorithm derived from another inductively verifiable formula

$F_n = (\mu^n – (–1/\mu)^n)/(\mu + 1/\mu)$ wherein $\mu = (1 + \sqrt{5})/2 = 1 + 1/\mu = 1.61803398875\dots$.

Since $\mu + 1/\mu = \sqrt{5} > 2$, and $1/\mu^n \le 1$, the foregoing formula simplifies to

$$F_n = (\text{ the integer nearest } \mu^n/\sqrt{5}\ ) \quad \text{for each nonnegative integer } n ,$$

and this is the formula upon which our algorithm is based. Almost. We need one more trick:

Instead of μ we obtain a rounded value u whose error $v = \mu - u$ is tinier than `eps` but gets amplified significantly when we compute u^n instead of $\mu^n = u^n + u^n \cdot n \cdot (v/u) + u^n \cdot n \cdot (n-1) \cdot (v/u)^2/2 + \dots$ for large n . To offset that error we estimate v and then compute the leading two terms in this series for μ^n , ignoring the rest because they contribute less than $(n\cdot\texttt{eps})^2 \cdot u^n$ which is negligible for $n < 1475$. (For larger values n , overflow beyond Matlab's floating-point range renders F_n inaccessible.) We need only the first several sig. bits of v , and they come from a formula

$$v = (\ 207/128 - u\ ) + 31/(\ 18304 + 8192\cdot\sqrt{5}\ )$$

whose validation is left to the diligent reader. (Don't disregard parentheses!) Matlab gets

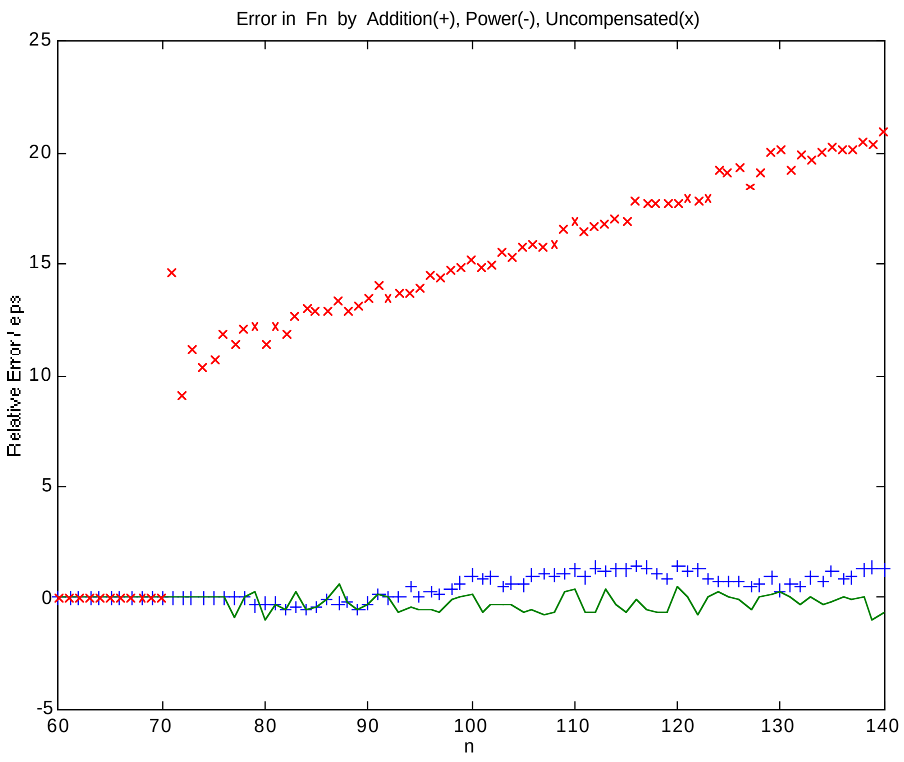
```
      s = sqrt(5)                                      … ≈ 2.23606797749979
      u = (1 + s)*0.5                                  … ≈ 1.61803398874989
      v = ( 207/128 – u ) + 31/( 8192*s + 18304 )      … ≈ -5.43185288415238e-17
```

Actually, Matlab's $u = 910872158600853/2^{49}$, and Matlab's v matches u 's error $\mu - u = -5.4321152\dots$**e**–17 beyond 3 sig. dec., which is more than good enough.

Here is Matlab function `fibon(N)` intended to deliver an array of Fibonacci numbers:

```
function  F = fibon(N)
%   F = fibon(N)  is the array of  Fibonacci  numbers corresponding to array
%   N of nonnegative integers.  Fibonacci  numbers are defined recursively:
%     fibon(0) = 0 ,  fibon(1) = 1 ,  and  fibon(n+1) = fibon(n) + fibon(n-1)
%   for  n = 1, 2, 3, 4, ...  in turn.  Fibon(N)  uses this recurrence if  N
%   is a long row or column of consecutive integers;  otherwise a faster but
%   sometimes less accurate direct formula is used.  The largest integer  n
%   for which  fibon(n)  can be computed exactly is  n = 78 ,  beyond which
%   roundoff cannot be avoided.  Some computers’ roundoff degrades  fibon(n)
%   for  n  as small as  71 .                        (C) W. Kahan 7 July 1998

if  any(any( (N ~= round(N)) | (N < 0) )),
      error(' fibon(n)  is defined only for nonnegative integers  n .'),  end
u = min(size(N)) ;  L = max(size(N)) ;
if ((u==1)&(L>5)),             %...  Cope with a long row or column  N ...
    if  (N(L)-N(1))==(L-1) ,  %...  of consecutive integers.
        F = N ;               %...  Allocate storage.
        F(1:2) = fibon(N(1:2)) ;  %...  Recursive call (!) to initialize  F .
        for k = 3:L ,  F(k) = F(k-1) + F(k-2) ;  end  %... Recurrence.
        return, end, end
%...  Otherwise  N  is not a long row nor column of consecutive integers:
s = sqrt(5) ;  u = (1+s)*0.5 ;             %...  u = rounded (1 + sqrt(5))/2 .
r = ((207/128 - u) + 31/(8192*s + 18304))/u ; %...  r = rel. error in  u .
F = u.^N ;  F = round(((r*N).*F + F)/s) ;     %...  F = round((u+r*u).^N/s) .
%  End fibon
```

How much does `fibon(N)` benefit from the trick compensating for the rounding error $\mu - u$? It varies among different computers and different versions of Matlab. Let's plot the relative error (`fibon`(n) – F_n)/`fibon`(n) both with (–) and without (x) that compensation as computed by Matlab 4.2 and 5.x on a 68040-based Macintosh Quadra 950 for $60 \le n \le 140$. (For $n \le 60$ that error is 0 .) Shown too (+) is the error in `fibon`(60:140) computed by additive recurrence.



Results from
Matlab 4.2 & 5
on a 68040-Mac

Relative error
in `fibon`(n) is
divided by `eps`.

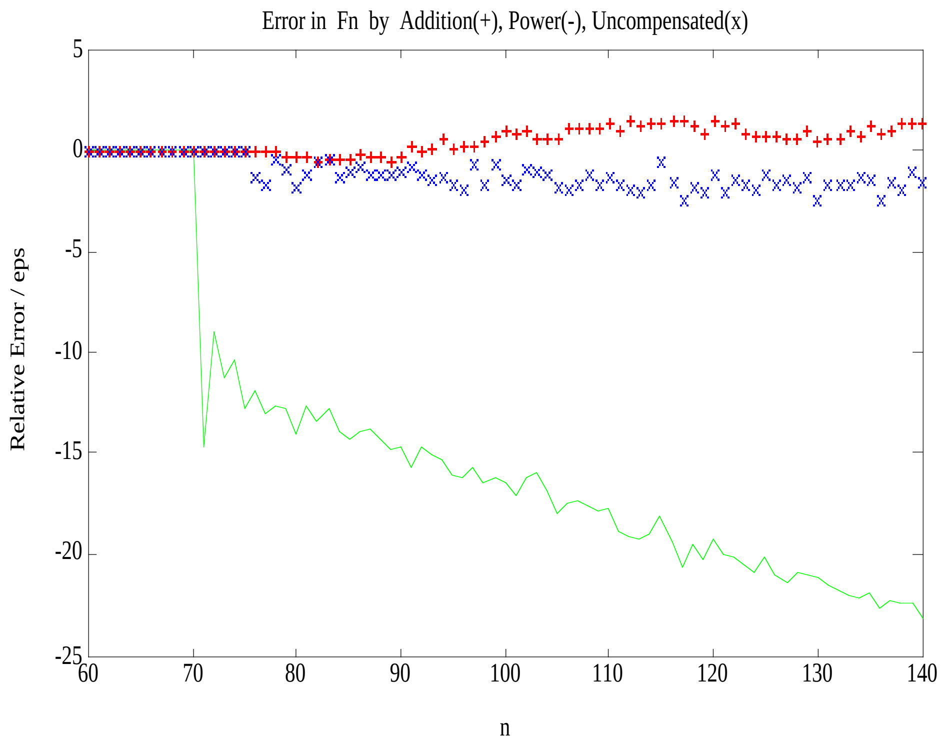
Legend:

Uncompensated
u^n
xxxxxxxxxxx

Additive
Recurrence
+++++++++

Compensated
μ^n
—

A graph like the above was obtained on Power Macs and from Matlab 3.5 on x86-based PCs. Something bizarre happens to `fibon`(n) with other versions of Matlab or on other computers:



Results from
Matlab 3.5
on a 68040-Mac

Relative error
in `fibon`(n) is
divided by `eps`.

Legend:

Additive
Recurrence
+++++++++

Uncompensated
u^n
xxxxxxxxxxx

Compensated
μ^n
—

The first graph above makes a convincing case for the error compensation provided by variable
`r = ((207/128 - u) + 31/(8192*s + 18304))/u ; %...  r = rel. error in  u .`
in the program `fibon(N)` . Without it (or with `r = 0` instead), the error in `fibon`(n) would grow roughly linearly with n . Compensated, that error stays smaller than 2.5 ULPs, usually smaller than the error accumulated by the additive recurrence (which behaves somewhat like a random walk). But this exemplary accuracy is achieved by only certain versions of Matlab on certain computer families, namely Matlab 4.2 and 5.x (not 3.5) on Apple Macintoshes, and Matlab 3.5 (not 4.x nor 5.x) on Intel x86/Pentium-based PCs and clones.

The second graph above shows what happens with other versions of Matlab or other computers, though it was obtained from 680x0-based Mac Matlab 3.5. They reverse the compensated and uncompensated errors; their compensated error in `fibon`(n) grows linearly with n far faster than the uncompensated error, which hardly grows at all. This perverse phenomenon cries out for an explanation. It has been figured out as follows:

Experiments suggest that, for real scalars y and positive integers n , Matlab computes y^n by repeated squarings and multiplications, as illustrated above for y^{259} , because larger integers n tend to take a little longer. Matlab 4.2 and 5.x must accumulate those squarings and multiplications extra-precisely on a 680x0-based Mac because its y^n is accurate to about an ULP; for instance, `t3(y,n) = y^(3*n) - (y*y*y)^n` cancels except perhaps for an ULP whenever y is so chosen (say $y = 1 \pm m/2^k$ for small positive integers m and k) that `(y*y*y)` gets computed exactly. Matlab 3.5 must *not* accumulate y^n extra-precisely because `t3(y,n)` grows with n when n is huge and `(y*y*y)` is still exact. though near 1 (to avoid over/underflow). Our inferences about extra-precise multiplication are corroborated by the behavior of `t2(y,n) = y^(2*n) - (y*y)^n` for arbitrary scalars y and positive integers n : it cancels to zero under Matlab 3.5 but not under Matlab 4.2 nor 5.x on a 680x0-Mac. This occurs because `(y*y)` gets rounded to $\{y^2\}$, to 53 sig. bits, before the second exponentiation.

Now recall $\mu = (1 + \sqrt{5})/2$ and $u = \{\mu\}$ and $v \approx \mu - u$. Under Matlab 3.5, the first few squarings during attempted computations of μ^2, μ^4, μ^8, … actually produce $u_2 = \{u \cdot u\}$, $u_4 = \{u_2 \cdot u_2\}$, $u_8 = \{u_4 \cdot u_4\}$, … rounded to 53 sig. bits (in the absence of extra-precise accumulation) instead of u^2, u^4, u^8, … respectively. The errors committed by these first few roundings {…} are the crucial errors because they get amplified most by subsequent operations, and these first few errors do something surprising. Tabulated below in ULPs of u_k are differences between computed squares u_k and the true squares μ^k and u^k . Apparently the first five $u_k = \{\mu^k\}$, not $\{u^k\}$.

k :	1	2	4	8	16	32	64
$u^k - u_k$ in ULPs :	0	0.27	0.85	1.86	2.95	6.17	14.05
$\mu^k - u_k$ in ULPs :	–0.24	–0.12	–0.18	0.09	0.35	0.56	1.00

In short, when Matlab exponentiates without extra-precise squarings and multiplications, an accident of roundoff in the computed large powers of u makes them approximate corresponding large powers of μ far more closely than they approximate true powers of u . It's magical, as if the computer divined from an expression like " `u^n` " the programmer's intent to compute μ^n .

This perverse phenomenon, `fibon`(n) more accurate without than with compensation for error μ–u on computer systems whereon Matlab computes `u^n` less accurately, occurs mainly for uncommonly big values n . Most applications of Fibonacci numbers F_n require them to be single `double` numbers computed exactly. For how big an $n \le 78$ can `fibon`(n) do that?

The next table exhibits the errors F_n - `fibon`(n) for five different computations: with or without extra-precise arithmetic used or simulated during exponentiation, with or without a compensating correction for the error μ–u , and by the additive recurrence, all for $0 \le n \le 82$.

Errors F_n – `Fibon`(n) Five Ways

Extra-Precise?	Corrected for $\mu - u$?	n = 0:70	71	72	73	74	75	76	77	78	79	80	81	82
No	No	0	0	0	0	0	0	-1	-1	0	-3	-9	-10	-7
	Yes	0	-1	-1	-2	-3	-6	-9	-15	-24	-41	-73	-106	-183
Yes	No	0	1	1	2	3	5	9	14	24	39	59	102	161
	Yes	0	0	0	0	0	0	0	-1	0	1	-5	-2	-7
Additive Recurrence :		0	0	0	0	0	0	0	0	0	-1	-1	-2	-7

From this table and the earlier graphs of the errors in `fibon`(n) come the following conclusions:

- The most accurate floating-point Fibonacci numbers F_n are computed by the additive recurrence unless n is huge, in which case the recurrence is too slow.
- For very large n or for a scattering of values n , the best choice is that version of `fibon`(n) that compensates for the error μ–u , run only on computer systems on which Matlab computes exponentials `u^n` accurately aided perhaps by extra-precisely accumulated multiplications and squares. But on other computer systems with inaccurate exponentials this version of `fibon`(n) is too inaccurate.
- For very large n or for a scattering of values n , the second-best choice is that version of `fibon`(n) that does not compensate for the error μ–u , run only on computer systems on which Matlab computes exponentials `u^n` inaccurately by repeated multiplications and squarings each rounded to `double`. But on other computer systems with accurate exponentials this version of `fibon`(n) is too inaccurate.

These conclusions pose two dilemmas:

The first dilemma confronts anyone who wishes to distribute `fibon`(n) , perhaps as part of a larger piece of software. Which version should be distributed? Distributing two versions is ill-advised because it practically ensures that the wrong version will be installed fairly often and then poor results will be blamed on the software. Devising a new version of `fibon`(n) that disables compensation for μ–u whenever a brief test of Matlab's exponentiation exposes excessive inaccuracy is ill-advised because a new release of Matlab may invalidate the test.

This dilemma is unnecessary. Matlab could compute exponentials accurately without extra-precise arithmetic by using not a nondescript **C** compiler's math library but algorithms promulgated with 4.2 BSD Berkeley UNIX in the early 1980s and easily available on the World-Wide-Web in the Freely Distributed Math Library `fdlibm`.

A second dilemma confronts anyone who would incorporate the computation of F_n for huge n into a benchmark that tests a computer system's floating-point for accuracy as well as speed. An uncompensated algorithm could easily lead benchmark users to a faulty conclusion reminiscent of the Stopped Clock Paradox: Because a stopped mechanical clock is exactly right twice a day, it is deemed more accurate than a running clock, which is never quite right.

Example 2 . The Zeros of a Real Quadratic:

Given real coefficients a, b, c should the zeros x_1 and x_2 of the real quadratic $a·x^2 - 2·b·x + c$ be computed at least about as accurately as the data [a, b, c] deserve? Of course! The question seems silly until you try to figure out how much accuracy the data deserve; it’s not obvious. A programmer could reasonably decide instead to compute the zeros about as quickly as possible and hope that they are accurate enough. To what extent does the phrase “ about as quickly as possible ” oblige the programmer to produce different programs each optimized for a different family of computers? A programmer could reasonably decide instead to write one program nearly optimal for his own computer, or for most of the computers to which he intends to promulgate his program, and hope it runs fast enough and accurately enough on all other computers. If it runs rather slower or rather less accurately on a few other computers than on his, where should blame fall? In other words, if something has to change, should it be his program, or those few other computers? Before resolving this moral dilemma let’s see what happens in our second example.

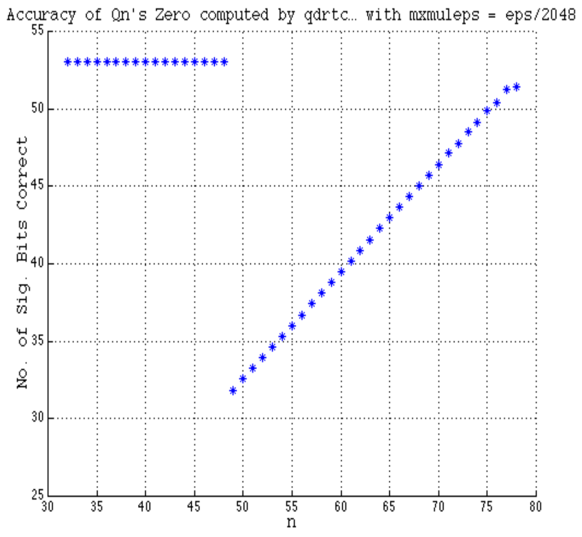
Our second example, Matlab function `qdrtc0`, has almost as simple a program as possible:

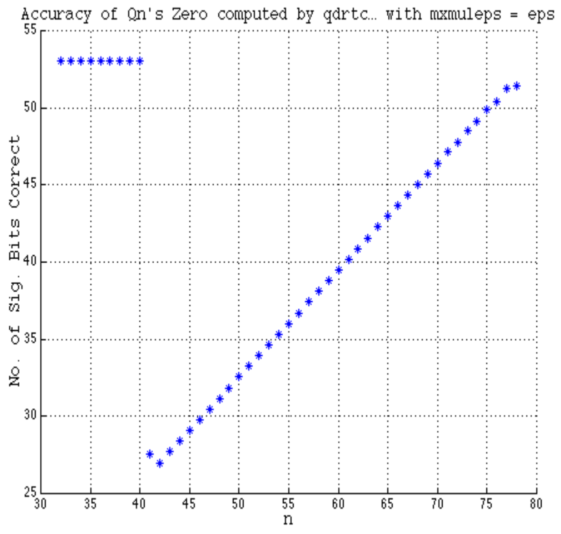
```
function  [x1, x2] = qdrtc0(a, b, c)
%  [x1, x2] = qdrtc0(a, b, c)  computes the zeros  x1  and  x2  of the real
%  quadratic  a*x^2 - 2*b*x + c  almost as accurately as a short simple Matlab
%  program can.  The zeros are ordered:   |x1| < |x2|  unless these differ by
%  less than a few  ULPs.  To keep it simple the program mishandles premature
%  over/underflow and infinite or complex coefficients but otherwise handles
%  infinite zeros correctly.                        (C) W. Kahan 29 June 1998

if  b == 0 ,  x1 = sqrt(-c/a) ;  x2 = -x1 ;  %...  Zeros with opposite signs.
else  d = [b, -a]*[b; c] ;                  %...  =  Discriminant  b^2 - a*c .
      r = sqrt(d) ;
      if  d > 0 ,
         if  b < 0 ,  s = b - r ;  else  s = b + r ;  end
         x1 = c/s ;  x2 = s/a ;              %...  distinct real zeros.
      else  x1 = b/a-r/a ;  x2 = conj(x1) ;  %...  complex or equal zeros.
end,  end                                    %...  End  qdrtc0
```

This program’s assignment … `d = [b, –a]*[b; c]` is slightly tricky; by exploiting extra-precise accumulation of the matrix product, on computers so endowed, it obtains the crucial discriminant $d = b^2 – a·c$ more accurately on these computers than on others.

To compare this program’s behavior on different computers we must generate exactly the same test data [a, b, c] on all of them, and generate the correct results [x1, x2] accurately too. Ideal test data should resemble “ typical ” data; consequently small integer data should be avoided because they are atypical of data most often encountered in practice and generate atypically small rounding errors during the first few arithmetic operations. However, this ideal is so difficult to reconcile with exact data and correct results obtained accurately that we use integer data anyway; they are drawn from large Fibonacci numbers F_n generated exactly up to F_{78} . For $32 ≤ n ≤ 78$ we use the quadratics $Q_n(x) := F_n·x^2 - 2 F_{n-1}·x + F_{n-2}$ to test `qdrtc0` . The zeros of $Q_n(x)$ are accurately $x1_n = 1/2 + (F_{n–3}/2 - \sqrt{(–1)^n})/F_n$ and $x2_n = 1/2 + (F_{n–3}/2 + \sqrt{(–1)^n})/F_n$.

Differences between $x1_n$ and the smaller zeros computed by `qdrtc0` on diverse computers have been converted into assessments of accuracy measured in sig. bits and plotted on graphs. Instances when `qdrtc0` alleges that $x1_n$ and $x2_n$ are complex though they are actually real and distinct, or *vice-versa*, are plotted as symbols “ o ” instead of “ * ” on these graphs.

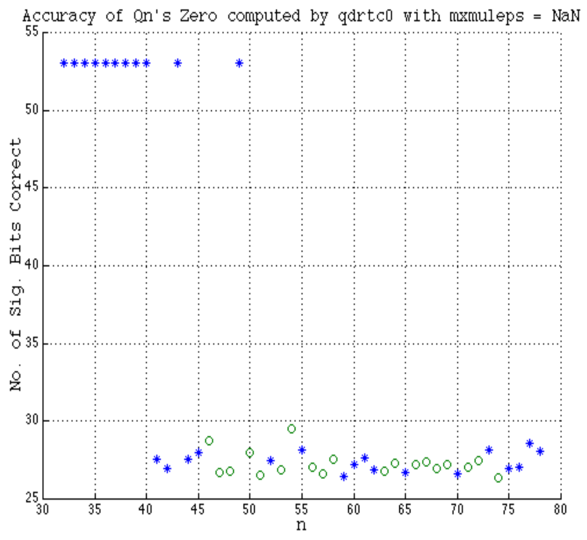


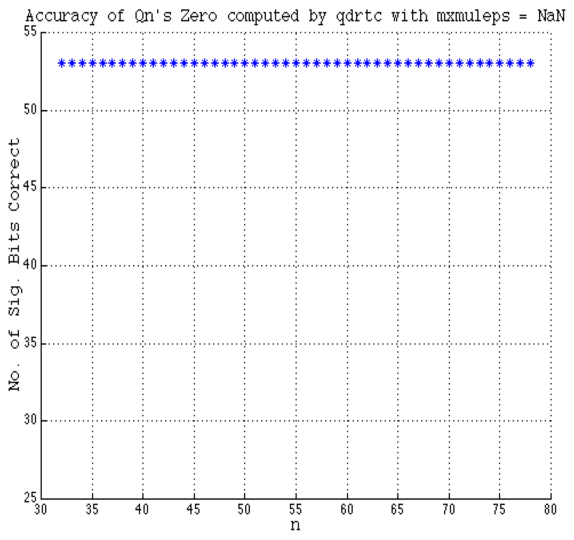


The first graph exhibits the accuracy obtained from Matlab 3.5, 4.2 or 5.x on 680x0-based Macintoshes, or Matlab 3.5 or 4.2 on x86/Pentium-based PCs or clones, on all of which `mxmuleps` = `eps`/2048 . The worst accuracy shown, about 32 sig. bits at n = 49 , is also provably the worst that can arise from any data, not just Fibonacci numbers, on these systems. The worst cases arise when a quadratic’s zeros are nearly coincident, in which cases the discriminant $d = b^2 - a \cdot c$ mostly cancels. These are the cases that benefit most from the 11 extra bits carried during `qdrtc0`’s computation of `d = [b, -a]*[b; c]` as a matrix product.

Similar worst cases yield only about 27 sig. bits, as the second graph shows at n = 41 to 43 , on a computer that lacks those extra bits, for instance a RISC-based computer like a Sun SPARC, SGI MIPS, DEC Alpha or HP PA-RISC-1, or an Intel-based PC running Matlab 5.x, on all of which `mxmuleps` = `eps` . This poorer accuracy is to be expected from machines that round to 53 sig. bits; error analysis proves that up to half the sig. bits carried may be lost when zeros are nearly coincident.

Something bizarre happens on such a computer if it has a Fused MAC that is used during matrix multiplication. This happens on the IBM RS/6000 and PowerPC, Apple Power Mac, SGI MIPS R8000 and HP PA-RISC-2, the computers on which `mxmuleps` returns NaN . If their FMAC is used naively to compute the discriminant as a scalar product `d = [b, -a]*[b; c]` then, besides delivering only about 27 sig. bits in worst cases, such computers can deliver complex roots when they should be real and distinct, and *vice-versa*. This anomaly shows up as “ o ” instead of “ * ” on the first of the next two plots.





The left-hand graph’s `o`s show that, besides delivering only about 27 sig. bits in worst cases, the aforementioned computers with FMACs can deliver complex roots when they should be real and distinct, and *vice-versa*. This can’t happen on other computers; how does a FMAC do it?

Computers without a FMAC approximate discriminant $d = b^2 - a·c$ by either {{b·b} – {a·c}} or {«b·b» – «a·c»} neither of which, if nonzero, can have a sign opposite to sign(d) because rounding is a monotonic nondecreasing function. With an FMAC, d is approximated by one of {{b·b} – a·c} or {b·b – {a·c}} which often have opposite nonzero signs when d mostly cancels. Similarly `[1-eps, eps-1]*[1+eps; eps+1]` is positive if Matlab evaluates scalar products left-to-right, negative if right-to-left, using a FMAC; otherwise this expression is zero.

As it happens, Matlab evaluates left-to-right on computers with a FMAC so their anomalous results can be repaired, without degrading results on other computers, by another trick:

First replace `qdrtc0`’s line
```
else  d = [b, -a]*[b; c] ;                 %...  =  Discriminant  b^2 - a*c .
```
by the tricky line
```
else  d = [b, -a]*[b; c] - [b, -b]*[b; b] ; %...  =  Discriminant  b^2 - a*c .
```
Then wherever the name “`qdrtc0`” appears replace it by “`qdrtc`”, and finally replace the word “`almost`” by “`about`” in the comments.

On computers without a FMAC, the new program `qdrtc` gets the same results as `qdrtc0` got; on computers with a FMAC, `qdrtc` always gets impeccable results, at least 51 sig. bits correct out of 53 , as is illustrated by this example’s last graph above on the right-hand side.

We have just made a silk purse out of a sow’s ear. It won’t happen again.

How much opprobrium is deserved by a programmer whose program eschews the tricks in `qdrtc` and therefore delivers results sometimes less accurate than they could have been? Perhaps it’s the other way round; perhaps program `qdrtc` deserves opprobrium for making some computers and the latest versions of Matlab look worse than the others. How do you feel about that?

Example 3. Iterative Refinement for Linear Systems:

The method most often used to solve a given linear system $\mathbf{A}\cdot\mathbf{z} = \mathbf{b}$ for $\mathbf{z} = \mathbf{A}^{-1}\cdot\mathbf{b}$ is called " Triangular Factorization ": First a matrix factorization $\mathbf{A} = \mathbf{L}\cdot\mathbf{U}$ is computed; here $\mathbf{U}$ is an upper-triangular matrix and $\mathbf{L}$ is a unit lower-triangular matrix with rows permuted to reflect pivotal row exchanges. Then $\mathbf{L}\cdot\mathbf{c} = \mathbf{b}$ is solved for $\mathbf{y}$ by forward substitution, and $\mathbf{U}\cdot\mathbf{x} = \mathbf{c}$ is solved for $\mathbf{x}$ by backward substitution. This is roughly how Matlab gets `x = A\b` . Because rounding errors intervene, $\mathbf{x}$ merely approximates the exact solution $\mathbf{z}$, sometimes poorly. No matter how poorly $\mathbf{x}$ approximates $\mathbf{z}$, the method's residual $\mathbf{r} = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ usually so nearly cancels out that it is overwhelmed by rounding errors accumulated during its own computation.

Occasionally, for unobvious technical reasons, the method's residual $\mathbf{r}$ is rather bigger than usual, and then $\mathbf{x}$ is a rather poorer approximation than usual. Such occasions are not frequent enough to weigh upon our minds, nor are they rare enough to ignore. Instances will be presented shortly. On such occasions the simplest and cheapest remedy is Iterative Refinement :

- Factor $\mathbf{A} = \mathbf{L}\cdot\mathbf{U}$ and solve $\mathbf{L}\cdot\mathbf{c} = \mathbf{b}$, $\mathbf{U}\cdot\mathbf{x} = \mathbf{c}$ for $\mathbf{x}$ as usual; in Matlab compute
  `[L,U] = lu(A) ;  x = U\(L\b) ; .`
- Obtain residual $\mathbf{r} = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$. Its accuracy is crucial so use extra precision where available;
  in Matlab compute `r = [A, b]*[x; -1]` with one matrix multiplication.
- Solve $\mathbf{L}\cdot\mathbf{d} = \mathbf{r}$, $\mathbf{U}\cdot\Delta\mathbf{x} = \mathbf{d}$ for $\Delta\mathbf{x}$, re-using factors $\mathbf{L}$ and $\mathbf{U}$ obtained earlier.
- The refined solution $\mathbf{y} = \mathbf{x} - \Delta\mathbf{x}$ should approximate $\mathbf{z}$ better than $\mathbf{x}$ did, we hope.

This process can be iterated: replace $\mathbf{x}$ by $\mathbf{y}$ and repeat the last three steps; but we shall not do so here since the first refined solution $\mathbf{y}$ is usually about as good as it's ever going to get.

Our objective here is to assess the benefit extra-precise accumulation confers upon a once refined solution $\mathbf{y}$. We assess it twice: We compare errors $\mathbf{z}-\mathbf{y}$ and $\mathbf{z}-\mathbf{x}$, and we compare residuals $\mathbf{r} = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ and $\mathbf{s} = \mathbf{A}\cdot\mathbf{y} - \mathbf{b}$. Ideally the refined solution $\mathbf{y}$ should have a smaller error than $\mathbf{x}$ has, and a smaller residual too, except for the effect of roundoff during refinement. But what do we mean by "smaller" vector errors and residuals? We could mean $\|\mathbf{s}\| < \|\mathbf{r}\|$ for some vector norm $\|\dots\|$ but there are infinitely many of these and they are arbitrarily different; which would we choose? "Smaller" could mean " $|\mathbf{s}| < |\mathbf{r}|$ elementwise " but this is too much to ask if, say, an element of $\mathbf{r}$ vanishes by accident. (Such accidents occur surprisingly often.) Accuracy much better than the data deserve adds little to a computation's value; how do we take this into account? Meaningful assessments require careful thought to which the next digression is devoted.

How accurately does $\mathbf{A}\cdot\mathbf{z} = \mathbf{b}$ deserve to be solved? Robert Skeel's answer to this question in the late 1970s is now accepted widely: We suppose, for a known and sufficiently tiny nonnegative matrix $\partial\hat{\mathbf{A}}$, that $\mathbf{A}$ can be considered practically indistinguishable from every perturbed matrix $\mathbf{A} + \partial\mathbf{A}$ with $|\partial\mathbf{A}| \le \partial\hat{\mathbf{A}}$ elementwise; then we call $\partial\hat{\mathbf{A}}$ the " Uncertainty " in $\mathbf{A}$. (For simplicity's sake $\mathbf{b}$'s uncertainty is ignored here because it would add little to our conclusions.) When $\mathbf{A}$'s elements are obtained from floating-point computations with uncorrelated rounding errors $\mathbf{A}$'s uncertainty $\partial\hat{\mathbf{A}}$ cannot be much smaller elementwise than $\mathtt{eps}\cdot|\mathbf{A}|$ and is probably bigger, except for elements of $\mathbf{A}$ computed exactly because they are small integers or half-integers etc.

$\mathbf{A}$'s uncertainty induces uncertainty in a residual $\mathbf{r} = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ which must be deemed practically indistinguishable from $\mathbf{r} + \partial\mathbf{r} = (\mathbf{A} + \partial\mathbf{A})\cdot\mathbf{x} - \mathbf{b}$ whenever $|\partial\mathbf{A}| \le \partial\hat{\mathbf{A}}$ elementwise. Since $|\partial\mathbf{r}| \le |\partial\hat{\mathbf{A}}|\cdot|\mathbf{x}|$ for all such $\partial\mathbf{A}$ and no smaller elementwise bound is valid, we regard $|\partial\hat{\mathbf{A}}|\cdot|\mathbf{x}|$ as the uncertainty $\mathbf{r}$ inherits from $\mathbf{A}$'s uncertainty. It provides a natural unit by which to measure how big $\mathbf{r}$ is: define $ß(\mathbf{x}) = \max(|\mathbf{r}|/(|\partial\hat{\mathbf{A}}|\cdot|\mathbf{x}|) \text{ elementwise})$, and deem $\mathbf{r}$ to be negligible whenever $ß(\mathbf{x}) \le 1$. Otherwise $\log_2 ß(\mathbf{x})$ counts the number of sig. bits by which $\mathbf{r}$ is bigger than negligible, perhaps because of roundoff. Likewise for $\log_2 ß(\mathbf{y}) = \max(|\mathbf{A}\cdot\mathbf{y} - \mathbf{b}|/(|\partial\hat{\mathbf{A}}|\cdot|\mathbf{y}|) \text{ elementwise})$.

Similarly, a natural unit by which to measure the error $\mathbf{z}–\mathbf{x}$ in an approximation $\mathbf{x}$ to $\mathbf{z}$ is the uncertainty inherited by $\mathbf{z} = \mathbf{A}^{-1}\cdot\mathbf{b}$ from $\mathbf{A}$'s uncertainty $\partial\hat{\mathbf{A}}$. Ideally this inherited uncertainty would be a tight outer bound for the region $\mathbf{z} + \partial\mathbf{z} = (\mathbf{A} + \partial\mathbf{A})^{-1}\mathbf{b}$ sweeps out as $\partial\mathbf{A}$ runs through *all* matrices constrained by $|\partial\mathbf{A}| \le \partial\hat{\mathbf{A}}$. This region cannot be circumscribed simply (it can have disconnected unbounded components) unless $\partial\hat{\mathbf{A}}$ is so tiny that no $\mathbf{A} + \partial\mathbf{A}$ with $|\partial\mathbf{A}| \le \partial\hat{\mathbf{A}}$ can be singular or too nearly so. Consequently a simple and usable overestimate for $\mathbf{z}$'s uncertainty is available only if $\partial\hat{\mathbf{A}}$ is sufficiently tiny, the tinier the better. In this case $\partial\mathbf{z} = -\mathbf{A}^{-1}\cdot\partial\mathbf{A}\cdot(\mathbf{z} + \partial\mathbf{z})$ implies first $|\partial\mathbf{z}| \le |\mathbf{A}^{-1}|\cdot|\partial\mathbf{A}|\cdot(|\mathbf{z}| + |\partial\mathbf{z}|) \le |\mathbf{A}^{-1}|\cdot\partial\hat{\mathbf{A}}\cdot(|\mathbf{z}| + |\partial\mathbf{z}|)$ and then $|\partial\mathbf{z}| \le \partial\mathbf{u}(\mathbf{z})$ elementwise, where we define

$$\partial\mathbf{u}(\mathbf{z}) = \left(\mathbf{I} - (|\mathbf{A}^{-1}|\cdot\partial\hat{\mathbf{A}})\right)^{-1}\cdot(|\mathbf{A}^{-1}|\cdot\partial\hat{\mathbf{A}})\cdot|\mathbf{z}| \ .$$

$\partial\mathbf{u}(\mathbf{z})$ is our overestimate of $\mathbf{z}$'s uncertainty. It is valid if and only if matrix $\left(\mathbf{I} - (|\mathbf{A}^{-1}|\cdot\partial\hat{\mathbf{A}})\right)^{-1}\cdot(|\mathbf{A}^{-1}|\cdot\partial\hat{\mathbf{A}})$ has no negative elements (we check this by computing the matrix); this is what " $\partial\hat{\mathbf{A}}$ is sufficiently tiny " means. And then $\partial\mathbf{u}(\mathbf{z})$ can be proved to overestimate $\mathbf{z}$'s inherited uncertainty almost always only slightly.

Having in hand an adequate estimate $\partial\mathbf{u}(\mathbf{z})$ for the uncertainty $\mathbf{z} = \mathbf{A}^{-1}\cdot\mathbf{b}$ inherits from $\mathbf{A}$'s uncertainty $\partial\hat{\mathbf{A}}$, we define $\mu(\mathbf{x}) = \max(|\mathbf{x}–\mathbf{z}|/\partial\mathbf{u}(\mathbf{z})$ elementwise$)$ to gauge how the error $\mathbf{z}–\mathbf{x}$ compares with $\mathbf{z}$'s uncertainty, and deem the error negligible when $\mu(\mathbf{x}) \le 1$. Otherwise $\log_2\mu(\mathbf{x})$ counts the number of sig. bits by which the error in $\mathbf{x}$ is bigger than negligible. Of course $\mu(\mathbf{x})$ costs more to compute than does $\max(|\mathbf{x}–\mathbf{z}|/|\mathbf{z}|$ elementwise$)$, the obvious measure of $\mathbf{x}$'s relative error, but does not becomes problematic when an element of $\mathbf{z}$ vanishes accidentally.

Note how the constructs $\partial\mathbf{u}(\mathbf{z})$, $\mu(\mathbf{x})$ and $ß(\mathbf{x})$ distinguish respectively *ill-condition* from *numerical instability*: Solving $\mathbf{A}\cdot\mathbf{z} = \mathbf{b}$ for $\mathbf{z}$ is an ill-conditioned problem, in view of $\mathbf{A}$'s uncertainty $\partial\hat{\mathbf{A}}$**,** to the extent that $\partial\mathbf{u}(\mathbf{z})$ is bigger than negligible compared with $\mathbf{z}$. A program that produces an approximate solution $\mathbf{x}$ is numerically unstable, in view of the uncertainty $\partial\hat{\mathbf{A}}$**,** to the extent that $\mu(\mathbf{x})$ and/or $ß(\mathbf{x})$ far exceed 1 . Triangular factorization is severely unstable on rare occasions. We wish to see how well a pass of iterative refinement copes with such occasions, reducing $\mu(\mathbf{x}) >> 1$ to $\mu(\mathbf{y}) \le 1$ and $ß(\mathbf{x}) >> 1$ to $ß(\mathbf{y}) \le 1$ we hope, and with minimal arbitrariness in our measures of error vectors and residuals.

By positing a known uncertainty $\partial\hat{\mathbf{A}}$ in $\mathbf{A}$, we have avoided the choice of an arbitrary vector norm $||\mathbf{z}–\mathbf{x}||$ by which to measure error in $\mathbf{x}$. Still, some arbitrariness persists in the choice of $\partial\hat{\mathbf{A}}$. Since $ß(\mathbf{x})$ and $\mu(\mathbf{x})$ are monotone nondecreasing functions of every element of $\partial\hat{\mathbf{A}}$, increasing $\partial\hat{\mathbf{A}}$ makes the computed vectors $\mathbf{x}$, $\mathbf{r}$, $\mathbf{y}$ and $\mathbf{s}$ look better without changing them. For the most conservative appraisal of these vectors we need $\partial\hat{\mathbf{A}}$ to be as small as possible but not entirely $\mathbf{0}$. We have chosen $\partial\hat{\mathbf{A}} = \mathtt{eps}\cdot|\mathbf{A}|$ because it seems nearly minimal; uncertainty much smaller than that seems implausible though it cannot be ruled out. This thought will be revisited later.

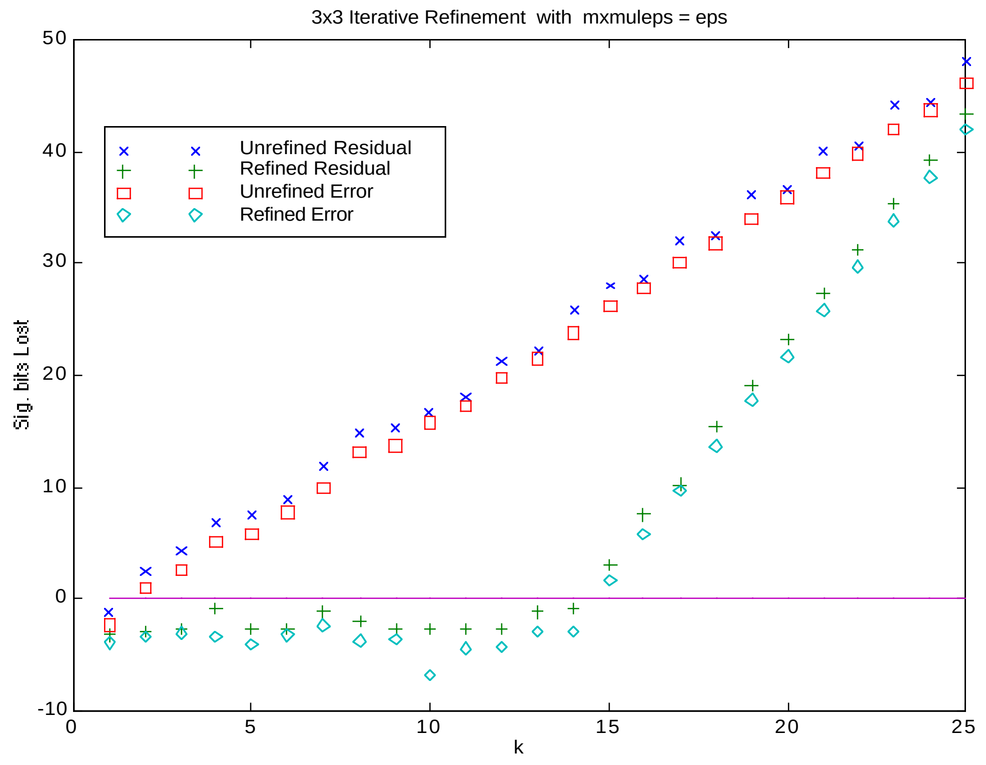
The foregoing digression has explained how $\log_2 ß(\mathbf{x})$ and $\log_2\mu(\mathbf{x})$ measure the number of sig. bits by which roundoff worsens the residual $\mathbf{r} = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ and the error $\mathbf{x}–\mathbf{z}$ respectively of a computed approximation $\mathbf{x}$ to $\mathbf{z} = \mathbf{A}^{-1}\cdot\mathbf{b}$ beyond their uncertainties inherited from the data. Next come applications to concrete instances.

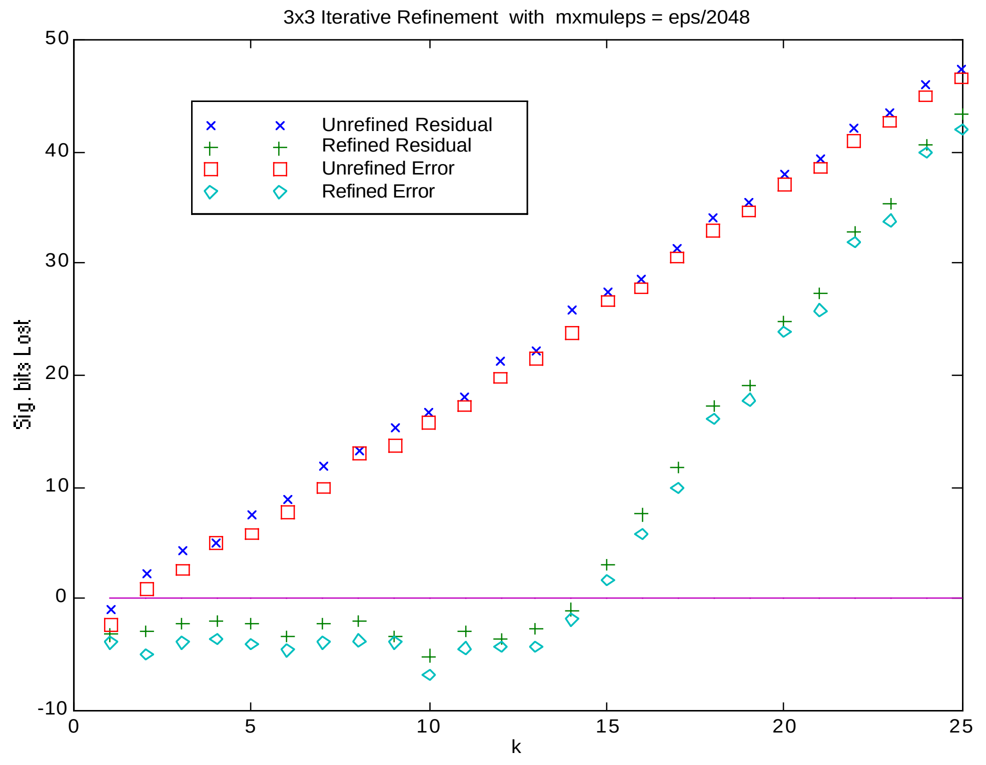
Our first sequence, for $k = 1, 2, 3, \dots$ or 25 , of instances are 3-by-3 systems $\mathbf{A}\cdot\mathbf{z} = \mathbf{b}$ with

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 1 \\ 1 & \frac{s}{4^k} & \frac{s}{4^k} \\ 1 & \frac{s}{4^k} & \frac{(s-1)}{4^k} \end{bmatrix} \quad \text{for } 1/2 < s < 1 , \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 2^k \\ -2^{1-k} \\ 2^{-k} \end{bmatrix} .$$

Here $s = 0.7$ arbitrarily. These systems are well-conditioned in the sense that changes smaller than an ULP in every element of $\mathbf{A}$ and $\mathbf{b}$ change no element of $\mathbf{z}$ by more than a few ULPs; *i.e.*, $\partial\mathbf{u}(\mathbf{z})$ amounts to at most a few ULPs of $\mathbf{z}$. But as k increases, Matlab's `x = A\b` loses almost $2k$ sig. bits, as if the systems became very ill-conditioned; look at the graphs:

Assessments of Residuals ($\log_2 ß$) and Errors ($\log_2 \mu$) for our 3-by-3 Systems:





(In these plots, lower is better.)

The graphs' [] marks plot $\log_2\mu(\mathbf{x})$ which shows how much the error in the unrefined solution `x = A\b` exceeds its uncertainty $\partial\mathbf{u}(\mathbf{z})$ inherited from $\partial\hat{\mathbf{A}} = \texttt{eps}\cdot|\mathbf{A}|$. Were $\partial\mathbf{u}(\mathbf{z})$ unknown, ill-condition could be blamed mistakenly for the large error. Indeed, measured by any `norm(…)` customarily offered by Matlab, the system would seem ill-conditioned for large k ; but such a norm is inappropriate for this system, and its implicit use by Matlab's triangular factorization causes $A_{11} = 2$ to be chosen as the first pivot. Any other choice would have avoided numerical harm. Rescaling the system, multiplying the first row down by 2^{-k} and the others up by 2^k , would have avoided harm too. But how could we be expected to know all that? There is a way.

The graphs' X marks plot $\log_2ß(\mathbf{x})$ which shows how much the unrefined solution's residual $\mathbf{r} = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ exceeds its uncertainty $\texttt{eps}\cdot|\mathbf{A}|\cdot|\mathbf{x}|$ inherited from $\partial\hat{\mathbf{A}} = \texttt{eps}\cdot|\mathbf{A}|$. This measure is easy to compute; if too big it indicates unmistakably that triangular factorization has malfunctioned. The simplest remedy, albeit not foolproof, is iterative refinement. One pass replaces **x** by a refined approximation **y** .

The graphs of $\log_2ß(\mathbf{y})$, plotted with + signs, and $\log_2\mu(\mathbf{y})$, plotted with ◊ marks, show how one pass of refinement pushes both residuals and errors below their uncertainties for $1 \le k \le 14$. Sufficiently repeated refinement would achieve the same effect for $15 \le k \le 25$, beyond which refinement is futile because roundoff damages the triangular factors too severely.

Bad scaling is not the only cause of damage to triangular factors, but the foregoing results are typical of all of them to the following extent: When an excessively big $ß(\mathbf{x})$ indicates excessive damage, iterative refinement counteracts most of it unless the damage is too severe, which is very rare. With that rare exception, the once-refined **y** has $ß(\mathbf{y}) < ß(\mathbf{x})$ and usually $ß(\mathbf{y}) \le 1$, which seems about as small as the data deserves. In short, the refined solution **y** usually has a negligible residual $\mathbf{s} = \mathbf{A}\cdot\mathbf{y} - \mathbf{b}$, so it looks pretty good. Moreover, this behavior is practically independent of whether the residual is accumulated extra-precisely, as is evident from the graphs.

An inescapable conclusion presents itself:

> Since testing for $ß(\mathbf{x}) >> 1$ and subsequent iterative refinement cost relatively little, they ought to be the default or at least a convenient option whenever a statement like Matlab's `x = A\b` is executed.

This is the conclusion Robert Skeel drew in the late 1970s, and it is accepted widely nowadays. But it is not quite right. Although the preceding paragraph is quite right, it omits a consideration. Can you see what has been overlooked? If not, the next sequence of instances will surprise you.

For n = 3, 4, 5, … or 18 let $\mathbf{A} = \mathbf{R}\cdot\mathbf{R}'$ where **R** is the leading n-by-n submatrix of

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & \dots \\ 0 & -1 & -2 & -3 & -4 & -5 & \dots \\ 0 & 0 & 1 & 3 & 6 & 10 & \dots \\ 0 & 0 & 0 & -1 & -4 & -10 & \dots \\ 0 & 0 & 0 & 0 & 1 & 5 & \dots \\ 0 & 0 & 0 & 0 & 0 & -1 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix} ,$$

whose columns come from Pascal's triangle. It is not hard to prove by induction that $\mathbf{R}^{-1} = \mathbf{R}$.

Consequently $\mathbf{A}^{-1} = \mathbf{R}'\cdot\mathbf{R}$; it is a well-known nonnegative integer matrix `pascal(n)` whose skew diagonals are also drawn from Pascal's triangle. For instance, when n = 6 ,

$$\mathbf{A} = \begin{bmatrix} 6 & -15 & 20 & -15 & 6 & -1 \\ -15 & 55 & -85 & 69 & -29 & 5 \\ 20 & -85 & 146 & -127 & 56 & -10 \\ -15 & 69 & -127 & 117 & -54 & 10 \\ 6 & -29 & 56 & -54 & 26 & -5 \\ -1 & 5 & -10 & 10 & -5 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{A}^{-1} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 3 & 6 & 10 & 15 & 21 \\ 1 & 4 & 10 & 20 & 35 & 56 \\ 1 & 5 & 15 & 35 & 70 & 126 \\ 1 & 6 & 21 & 56 & 126 & 252 \end{bmatrix} .$$

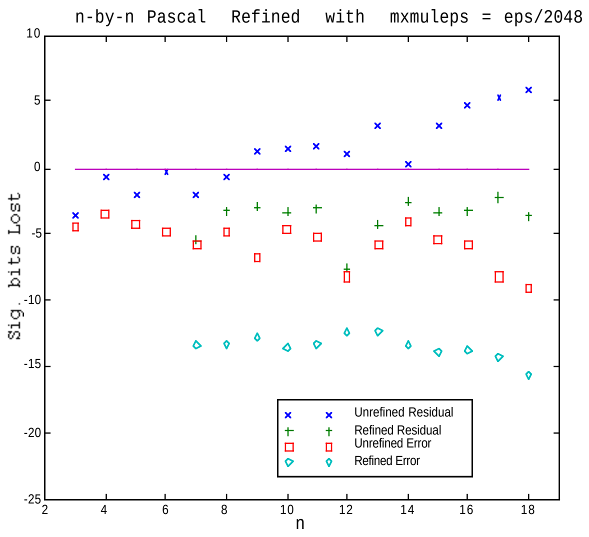
In short, **A** and $\mathbf{A}^{-1}$ are known matrices of integers some of which grow huge as n gets big.

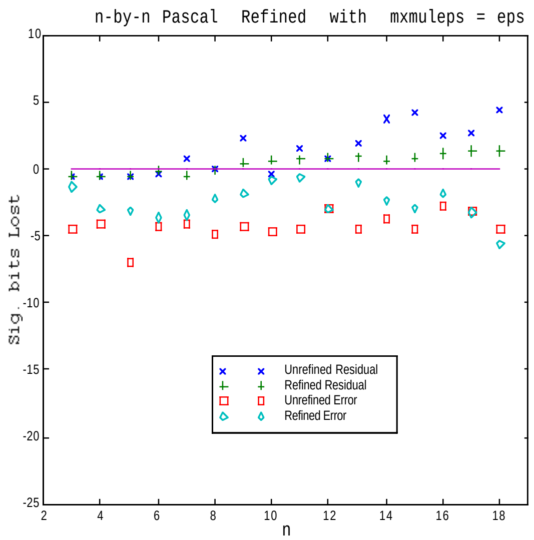
Next compute column n-vector **z** by dividing the eigenvector of **A** belonging to its biggest eigenvalue by this vector's last element and rounding all quotients to nearest integers. Then $\mathbf{b} = \mathbf{A}\cdot\mathbf{z}$ is computed, and suffers no rounding errors unless n > 20 . For example, when n = 6 ,

$\mathbf{z} = [\,-2 \;\; 8 \;\; -14 \;\; 12 \;\; -6 \;\; 1\,]'$ and $\mathbf{b} = [\,-629 \;\; 2667 \;\; -4634 \;\; 4098 \;\; -1837 \;\; 333\,]'$.

As n increases, the elements in the middle of **z** and **b** grow orders of magnitude bigger than those near the vectors' ends, but the first element of **z** is always ±2 and its last always 1 .

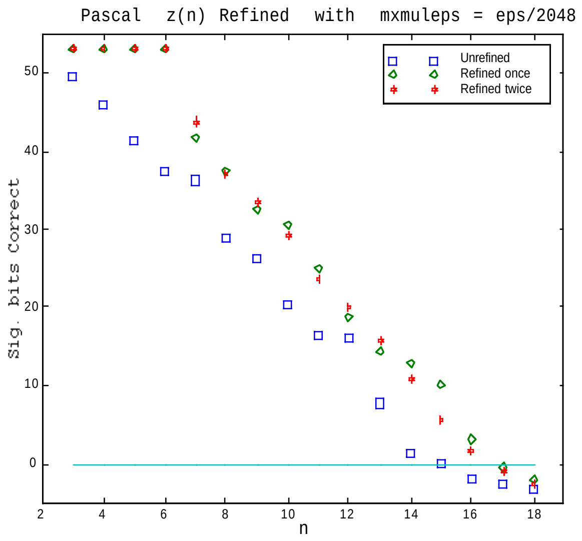
Rounding errors begin when `x = U\(L\b)` is computed, and continue when **x** is refined to get **y** . Plotted below are assessments $\log_2\mu(\mathbf{x})$ ([]) and $\log_2\mu(\mathbf{y})$ (◊) of their respective errors **z–x** and **z–y** , and assessments $\log_2ß(\mathbf{x})$ (x) and $\log_2ß(\mathbf{y})$ (+) of their respective residuals $\mathbf{r} = \mathbf{A}\cdot\mathbf{x}-\mathbf{b}$ and $\mathbf{s} = \mathbf{A}\cdot\mathbf{y}-\mathbf{b}$, indicating the excess numbers (beyond inherited uncertainties) of sig. bits lost to roundoff both with and without extra-precise accumulation of residuals. In these plots lower is better because it indicates less loss (or more gain) of accuracy.

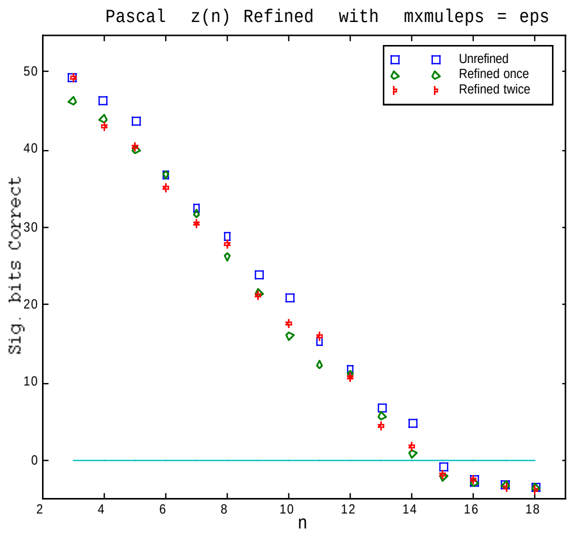




So far as the accuracy and residual of Matlab's unrefined `x = U\(L\b)` go, there is little to choose between results computed with or without extra-precise arithmetic of which Matlab's triangular factorization appears to make little use. Neither is there much to choose between the residuals of the refined **y** computed with or without extra-precisely accumulated residuals; both ways attenuate residuals adequately. Evidently **y** is more accurate than **x** by roughly ten sig. bits with extra-precise accumulation, but surprisingly *less* accurate without. Also surprising is how far all errors, refined or not, fall below the uncertainty allegedly inherited from the data.

The degradation of accuracy by iterative refinement without extra-precise accumulation is worse than surprising; it is perplexing. Is this phenomenon an artifact of the complicated way we have chosen to assess accuracy? No. Exhibited below are the accuracies, in sig. bits, of the last (n^{th}) element of the unrefined **x** ([]) , of the once refined **y** (◊) , and of a twice refined approximation (∗) to $z_n = 1$, computed with and without extra-precise accumulation. In these plots, higher is better, and accuracy is limited to the 53 sig. bits of `double`’s precision.





Again, one pass of iterative refinement improves accuracy by roughly ten sig. bits with extra-precise accumulation, degrades accuracy without. A second pass doesn’t change things much.

The foregoing brief summary is contradicted occasionally when flukes of roundoff produce extraordinarily small errors. In the left-hand plot, once-refined results **y** are perfect when $3 \le n \le 6$ and extraordinarily accurate when $n = 15$; unrefined results **x** are extraordinarily accurate when $n = 7$ or $n = 12$. Similar flukes are less pronounced in the right-hand plot. Trying to restrain rounding errors to smooth curves is like trying to herd cats.

The plots on this page reveal something that the previous page’s plots concealed: The Pascal systems **A**·**z** = **b** are ill-conditioned. Accuracy lost to roundoff when `x = U\(L\b)` is computed can range from roughly 3n–4 sig. bits using Skeel’s kind of assessment to roughly 4n–6 sig. bits using a customary norm. With Skeel’s kind of assessment, all 53 sig. bits of accuracy are lost when **z**’s uncertainty ∂**u**(**z**) cannot be computed, which happens for $n > 18$. In the graphs on this page all 53 sig. bits are lost when $n > 15$. The different assessments are not so much inconsistent as they are reflective of the unreasonableness of attempts to capture ill-condition in a single number divorced from the provenance of the data and the purpose to which the result will be put. For example, as n becomes large, the last element (1) of **z** becomes orders of magnitude more sensitive than the first element (±2) to roundoff-like perturbations in the data **[A, b]** ; a single number can’t convey two such different sensitivities.

In short, iterative refinement almost always improves residuals (ß ; x —› +) but it can worsen solutions’ errors (µ ; [] —› ◊) if applied to ill-conditioned systems without accumulating residuals extra-precisely. With this extra-precision, refinement usually shrinks solutions’ errors by factors like $1/2^{10}$, unless fewer than 10 sig. bits were lost to roundoff. (For an example whose errors shrink far beyond that see http://http.cs.berkeley.edu/~wkahan/Cantilever.ps .) However, *the benefits of extra-precise accumulation are hidden if data are presumed uncertain by an ULP or more*, because then solutions’ inherited uncertainties will be deemed to exceed their errors so much as to make smaller errors seem worthless. Such presumptions can be too presumptuous.

Treacherous Mathematics:

We based our estimate $\partial\mathbf{u}(\mathbf{z})$, of the uncertainty that the solution $\mathbf{z}$ of $\mathbf{A}\cdot\mathbf{z} = \mathbf{b}$ inherits from uncertainty in the data $[\mathbf{A}, \mathbf{b}]$, upon a plausible but dubious premise. We presumed the elements of $\mathbf{A}$ to be uncorrelatedly uncertain by roughly an ULP in each element because an ULP seemed nearly minimal. They aren't and it isn't. Integer elements need not be uncertain at all, unless they are so big that they must be rounded off to fit into the `double` format's 53 sig. bits.

Our n-by-n Pascal systems $\mathbf{A}\cdot\mathbf{z} = \mathbf{b}$ have integer elements known exactly, none too big. At $n = 15$ none of those integers occupies more than 37 of the 53 sig. bits available, yet all sig. bits are lost in Matlab's computed last element of $\mathbf{x}$. Therefore the difference between iterative refinement with and without extra-precise accumulation, almost ten sig. bits, is the difference between something and nothing revealed about a mathematically well-defined last element of $\mathbf{z}$.

More generally, systems $\mathbf{A}\cdot\mathbf{z} = \mathbf{b}$ often have data $[\mathbf{A}, \mathbf{b}]$ in which many elements are known exactly or are correlated, thereby inducing less uncertainty into $\mathbf{z}$ than our estimate $\partial\mathbf{u}(\mathbf{z})$, sometimes far less than that. Oversized estimates of $\partial\mathbf{u}(\mathbf{z})$ distract us from the purpose of our computation, which is to obtain results about as accurate as the data deserve and at a tolerable price. What good is a $\partial\mathbf{u}(\mathbf{z})$ that does not reveal how much (in)accuracy the data deserves?

It exposes how an explanation evolved into an exculpation, into a plausible excuse for inaction.

In the 1950s almost no programmers knew how to distinguish numerical instability from ill condition. In other words, the distinction between an algorithm's hypersensitivity to its own internal rounding errors, and a true solution's hypersensitivity to roundoff-like errors in its problem's data, as contributors to wrongly computed solutions, was obscure. By the 1960s the distinction had been clarified through " Backward Error Analyses," which showed how results produced by many important numerical computations (but not all) suffer from their internal rounding errors little more than if their data had first been altered by roundoff-like perturbations and then these computation had been carried out exactly, albeit to a wrong conclusion if the true solution is hypersensitive to perturbations in data.

For example, the result $\mathbf{x}$ produced by Matlab's `x = A\b` to approximate $\mathbf{z} = \mathbf{A}^{-1}\mathbf{b}$ almost always satisfies exactly some slightly perturbed equation $(\mathbf{A} + \partial\mathbf{A})\cdot(\mathbf{x} + \partial\mathbf{x}) = \mathbf{b} + \partial\mathbf{b}$ in which none of $\|\partial\mathbf{A}\|/\|\mathbf{A}\|$, $\|\partial\mathbf{x}\|/\|\mathbf{x}\|$ or $\|\partial\mathbf{b}\|/\|\mathbf{b}\|$ can be arbitrarily bigger than `eps` , though $\mathbf{x}$ can differ from $\mathbf{z}$ arbitrarily if $\mathbf{A}$ is very ill-conditioned. Here the norms $\|\ldots\|$ must be chosen to suit the error-analyst if not the application. Moreover, no such backward analysis can be valid in general if the vectors $\mathbf{b}$, $\mathbf{z}$ and `x` are replaced by matrices $\mathbf{B}$, $\mathbf{Z} = \mathbf{A}^{-1}\mathbf{B}$ and `X = A\B` , because different columns of $\mathbf{X}$ may require different perturbations $\partial\mathbf{A}$, very different if $\mathbf{A}$ is ill-conditioned. The appearance of $\mathbf{A}^{-1}$ here is irrelevant; the error-analysis of most matrix products $\mathbf{Z} = \mathbf{C}\cdot\mathbf{B}$ is afflicted similarly.

Backed by flawed backward error-analyses, the vendors of floating-point hardware and software used to respond to a complaint about a badly wrong result by averring that their handiwork could not reasonably be required to do any better since it had delivered a result scarcely worse than if the data had been perturbed negligibly. To the complainant that response sounded like this:

> " We disliked your problem, so we solved one with very slightly different data for you; since you can't know your problem's data exactly you have no grounds for complaint."

This is as irksome as Lily Tomlin's response in her radio/TV skit about a switchboard operator:

> " We don't care. We don't have to. We're the Phone Company."

Error-analysts irked by such responses began in the late 1960s to reconsider the foundations of backward error-analysis. The foundations seemed to rest upon choices of norms $\|\ldots\|$ that led to conclusions error-analysts desired, not necessarily choices that served software users' needs. For example, the error-analyst would measure error with a norm $\|\mathbf{z}-\mathbf{x}\|$ that weighted all elements equally, whereas a user might need $\|\mathbf{D}\cdot(\mathbf{z}-\mathbf{x})\|$ for a diagonal matrix $\mathbf{D}$ of scale factors, say reciprocals of the elements of $\mathbf{z}$. Attempts to reconcile those choices led to better error-analyses and algorithms whose results are independent of scaling to a considerable degree; an example is Skeel's analysis and iterative refinement, and many more examples have turned up over the past two decades. The better algorithms (available in ScaLAPACK but not yet built into Matlab) cost little more than the old and produce accurate results over a greatly widened range of input data, thereby vindicating the better analyses and reducing the incidence of complaints. Are all remaining complaints groundless?

Regardless of complaints about inaccuracies, there is something curious about the function $\mu(\mathbf{x})$ by which we measured the error $\mathbf{z}-\mathbf{x}$ in Matlab's approximation `x = A\b` to $\mathbf{z} = \mathbf{A}^{-1}\cdot\mathbf{b}$. This function compares that error with $\partial\mathbf{u}(\mathbf{z})$, the lowest available estimate of the uncertainty induced in $\mathbf{z}$ by a backward error-analyst's roundoff-like perturbation $\partial\mathbf{A}$ in the data. Except when $ß(\mathbf{x}) >> 1$ (betraying an atypically big residual $\mathbf{r} = \mathbf{A}\cdot\mathbf{x}-\mathbf{b}$ due to bad scaling or some other pathology), that error usually falls at least an order of magnitude below the alleged uncertainty. This disparity is visible in the plots of $\log_2\mu(\mathbf{x})$ ([]) above for n-by-n Pascal matrices $\mathbf{A}$.

More pronounced disparities,— errors orders of magnitude smaller than alleged uncertainties,— occur with other systematically ill-conditioned matrices $\mathbf{A}$. (I prefer to leave the distinction between *systematic* and *accidental* ill-condition to the reader's imagination rather than digress to define it here.) These disparities cannot be explained away by pseudo-probabilistic reasoning, as if accumulations of rounding errors, like random variables with mean zero, should tend to cancel out on average. The disparities persist for dimensions n so small that too few rounding errors are committed to comport with that explanation. On the other hand, to dismiss such persistent disparities as statistical flukes is to disregard a Law of Statistics:

If you're lucky (or unlucky) *all* the time, it isn't luck.

If the disparate smallness of errors, compared with uncertainties, has a probabilistic explanation, it probably goes like this: Elements of the backward error-analyst's roundoff-like perturbation $\partial\mathbf{A}$, an inverse image in the data of the computation's internal rounding errors, are correlated so strongly that in the derivation of uncertainty $\partial\mathbf{u}(\mathbf{z})$ the assertion " $\partial\mathbf{z} = -\mathbf{A}^{-1}\cdot\partial\mathbf{A}\cdot(\mathbf{z} + \partial\mathbf{z})$ implies first $|\partial\mathbf{z}| \le |\mathbf{A}^{-1}|\cdot|\partial\mathbf{A}|\cdot(|\mathbf{z}| + |\partial\mathbf{z}|)$ " becomes a gross overestimate.

In short, the allegedly minimal uncertainty $\partial\mathbf{u}(\mathbf{z})$ is too often too much too big. Consequently we have no satisfactory way to characterize the accuracy that Matlab's computed `x = A\b` deserves divorced from the provenance of its data and the purposes that the result will serve. Attempts to convert an error's explanation, provided by backward error-analysis, into an excuse for such an error are misguided. They turn an oversized alleged uncertainty $\partial\mathbf{u}(\mathbf{z})$ into a cosmetic that makes a numerical algorithm look good even when its result is poor. Worse, they promote indifference to errors smaller than that uncertainty thereby making light of the difference between refinement with and without extra-precisely accumulated residuals, as if accuracy well beyond that uncertainty were never worthwhile. In fact, sometimes it is and sometimes it isn't.

How Much is Accuracy Worth?:

How do you put a price upon something of which you know neither how much you have nor how much you need? Accuracy is like that. Without specific and often tedious details, only these three generalities about accuracy are worth uttering:

1• Accuracy can't be appraised meaningfully out of context, which resembles a spider web.
2• The intransitivity of accuracy can undermine the utility of software packages like Matlab.
3• Greater accuracy has greater value principally to the extent that it diminishes risk.

Each of these utterances will be explained briefly in turn.

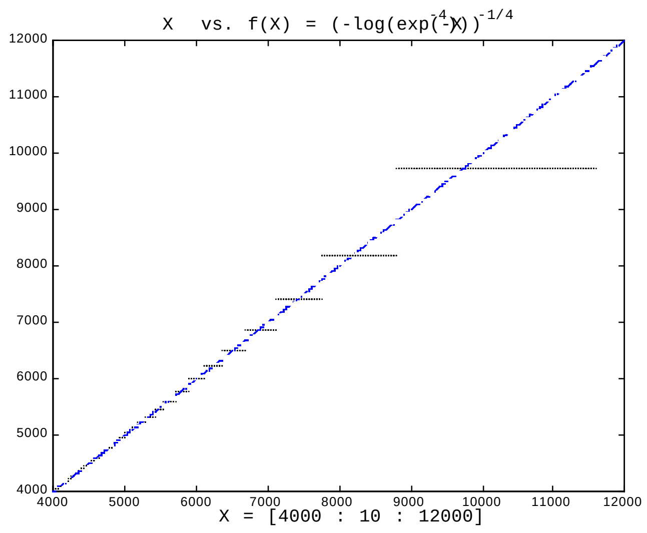
1• Approximate computation is sometimes likened to a chain no stronger than its weakest link, but with strength replaced by accuracy. This analogy is misleading. In fact many a computation, if interrupted in the middle, may appear to have produced utterly inaccurate intermediate results none of them close to what would have been computed in the absence of roundoff, and yet resuming this computation and completing it produces a result of exemplary accuracy. Many matrix computations are like this. They transform the given problem through a sequence of problems, each with very nearly the same solution, to one of a distinguished family of problems whose solution is obvious. Roundoff alters the path through the problems' sequence; substantial alterations are tolerable provided the path ends soon enough and near enough to the solution of the given problem. In general the accuracies of intermediate results can be assessed properly only in the context of mathematically coherent relationships they are intended to maintain. The same goes for the "final" result insofar as it is an intermediate result in a larger context.

Approximate computation is like a web whose strands are mathematical relationships, among computational variables and constants, that connect these as strongly as the relationships are accurate. The art of computation consists partly in finding such a web, and partly in determining which of its strands can be weakened, by how much, and with what advantage to performance, while maintaining adequate strength overall in the web's connection of output results to input data. Matlab and similar software systems provide their users with pieces of web ready-made.

2• " The intransitivity of accuracy " is a mathematically hifalutin way of saying this: Suppose `g(y)` and `h(x)` are programmed floating-point implementations of real functions G(y) and H(x) respectively with fairly high accuracy. Any reasonable interpretation of " accuracy " is acceptable here. Then the composed program `f(x) = g(h(x))` may approximate the composed function F(x) = G(H(x)) inaccurately in every useful sense**!** Thus, accuracy need not necessarily survive functional composition; sometimes this chain is so much weaker than its weakest link that it is better likened to a strand in a cobweb.

For example, take $G(y) = (-\log(y))^{-1/4}$ for $0 < y < 1$ and $H(x) = \exp(-x^{-4})$ for $x > 1$, so $F(x) = G(H(x)) = x$. As x runs up to `(4/eps)`$^{1/4} \approx 11585.2375\ldots$, composed program `f(x) = g(h(x))` loses all resemblance to F(x) despite that programs `h(x) = exp(-x^(-4))` and `g(y) = (-log(y))^(-1/4)` are both accurate to almost their last sig. bits. As computed by Matlab 5.x on a Mac Quadra 950 (68040), the graphs of F(x) and `f(x)` are plotted on the next page except for $x > 11585.2375\ldots$ whereon their difference is infinite. The trouble is caused by the singularity of G(y) at y = 1 . If not for a rounding error no bigger than half an ULP of y , the singularity would have been smoothed away by y = H(x) as x approaches infinity.

F(x) = x vs. $f(x) = (-\log(\exp(-x^{-4})))^{-1/4}$



Only rarely is accuracy lost so severely to intransitivity; otherwise numerical software would be impossible. Some kinds of accuracy are more vulnerable than others to this kind of loss; most of Matlab’s operations fall into the more vulnerable category. The explanation is subtle:

If programs `g(y)` and `h(x)` are accurate in the naive sense that `g`(y) = (G + ∂G)(y) and `h`(x) = (H + ∂H)(x) for acceptably tiny perturbations ∂G and ∂H , then it follows that program `f`(x) = (G + ∂G)((H + ∂H)(x)) suffers from two perturbations of which only one, ∂H(x) , is capable of causing `f`(x) to lose accuracy severely, and does so only when y = H(x) is too near a singularity of G(y) . Otherwise `f`(x) cannot be degraded by singularities of H(x) .

Compare that with what happens when `h`(x) = (H + ∂H)(x+∂x) is accurate only in the sense of backward error-analysis, now for acceptably tiny perturbations ∂H and ∂x . Now program `f`(x) = (G + ∂G)((H + ∂H)(x+∂x)) can be degraded by the singularities of H(x) as well as the singularities of G(y) . When x and y are not scalars but vectors or matrices something worse happens: Now ||∂H|| becomes acceptably tiny to the backward error-analyst if it is no bigger than ||H(x+∂x) – H(x)|| , and now `f`(x) is degraded by the singularities of both H(x) and G(y) though their composition F(x) = G(H(x)) may have no singularity at all; if so, `f`(x) is truly numerically unstable.

In short, if programs are accurate only in the sense of backward error-analysis, composing them tends to lose accuracy to intransitivity rather more often than if the programs were accurate in the obvious naive sense. This is bad news for Matlab most of whose matrix operations are accurate only in a backward sense. For several of these operations accuracy better than that is so costly to achieve that, before we demand it, we should perform a cost/benefit analysis. Alas, the cost must be borne initially by Matlab’s authors, The Math Works Inc., in the hope that customers will appreciate the benefit enough to pay extra for it. Can customers quantify benefit that way?

Do Matlab’s customers appreciate how easily they combine its operations and functions, or do they take it for granted? Do customers rely upon the accuracies of combined operations and functions, taking them for granted too? Perhaps it is unlikely that, starting with 53 sig. bits, so many would get lost as to leave too few, fewer than a dozen, to produce smooth graphs; but the graph above shows how quickly it can happen. Reliability is a delicate issue, but crucial. By improving operations’ accuracy we enhance their reliability, which promotes their ease of use.

How much?

3• Accuracy is not like Virtue, which is its own and often sole reward. Numerical accuracy is valuable only to the extent that it helps us avoid inaccuracy severe enough to cause loss or pain.

If severe inaccuracy occurs, it occurs because numerical errors that seemed small when they were committed were amplified too highly while propagating through subsequent computation. High amplification is always due to a singularity in some function evaluated during computation subsequent to the error. Most often this amplification is roughly inversely proportional to the distance from that function's argument to its nearest singularity. This singularity may be an accident, not a property of the final function that is the purpose of the whole computation but arising out of a latter part of the computation whose singularity would have been smoothed out by the earlier part had small errors not intervened. Numerical instability is always due to such accidental singularities not intrinsic to the desired function but arising from the procedures chosen to compute it

For example, the computation of `x = A\b` by triangular factorization entails divisions by numbers called " pivots." Rounding errors get amplified by factors roughly proportional to the divisions' quotients. Prevention of unnecessarily big quotients is the motivation behind pivotal exchanges without which arbitrarily tiny pivots can arise accidentally, causing numerical instability even if the data **A** is far from singular. With pivotal exchanges, rounding errors can get amplified greatly only if **A** is nearly singular; in fact the amplification factor turns out to be about

$$\mathrm{cond}(\mathbf{A}) = \|\mathbf{A}\|\cdot\|\mathbf{A}^{-1}\| = \|\mathbf{A}\|/\|\mathbf{A} - (\text{the singular matrix nearest } \mathbf{A})\| .$$

Now suppose there is some tolerance T beneath which errors in the final result will be deemed negligible. In the space of all admissible input data (real and complex) there are points, curves, surfaces, hypersurfaces … on which the factors by which errors get amplified become infinite; these loci are inverse images of the aforementioned singularities. Surrounding these loci are balls, tubes, slabs, sheets … inside which some rounding errors get amplified to intolerable sizes in the final result, outside which none can be amplified intolerably. Let's call these balls, tubes, slabs, sheets … " pejorative " to signify computation's intolerable degradation by data inside them. Their size depends upon the tolerance T and also upon the precision of floating-point arithmetic, which constrains the sizes of rounding errors. Insofar as increasing precision diminishes rounding errors it diminishes the sizes of pejorative regions too. How much?

For most computations, pejorative balls, tubes, slabs, sheets … surround hypersurfaces of dimension one less than that of the space of all admissable input data, and the amplification factors are roughly inversely proportional to the distance from the data to the nearest hypersurface if it is near enough. Therefore, if an increase of arithmetic precision by K sig. bits diminishes rounding errors by a factor roughly $1/2^K$, that increase also diminishes the thickness and volume of pejorative regions by roughly the same factor. This factor applies to most computations with real data and arithmetic; with complex data and arithmetic the volume of pejorative regions tends to shrink by a factor $1/4^K$. Then, provided the population of admissible data sets is distributed not too nonuniformly in the neighborhoods of pejorative regions, the final result's error becomes intolerable (exceeds tolerance T) for a smaller proportion of this population, a proportion shrunk by roughly the same factor $1/2^K$ for real, $1/4^K$ for complex data. Since data sets are discrete rather than distributed continuously, a law of diminishing returns will set in as K gets so big that further increases liberate no more data sets from inside the shrinking pejorative regions.

In short, an increase of K sig. bits in an intermediate result's accuracy usually attenuates by a factor roughly $1/2^K$ the incidence of numerical embarrassment attributable to the use later of that result. For most computations, and for a fixed population of data sets (stored to a precision that does not change), an increase of K sig. bits in arithmetic's precision usually attenuates by a factor roughly $1/2^K$ the incidence of numerical embarrassment attributable to the use later of that computation's result. The factor $1/2^K$ is for real arithmetic and data; for complex it is $1/4^K$. This is how increased accuracy pays off in diminished risk until it gets down to a level below which risk cannot be decreased further merely by increasing accuracy some more.

When K = 11 the reduction factors are below $5/10^4$ and $3/10^7$. How much is that decrease in risk worth? This is a personal question, as must be evident from news reports of drunk driving, speeding and unused seat-belts on our highways. Multiplying a calamity's cost by its probability would estimate a premium to be paid for insurance against that risk. Multiplying one guess by another is futile, so I shall cite my own experience instead.

Finding and fixing my mistakes and others' consumes ten to a hundred times as much of my time as is spent figuring out how to compute the solution to a problem. At least half of those mistakes arise from underestimated rounding errors, which lurk as invisible as *E. coli* until the harm they have done attracts notice. Unlike some other people, I never trust a numerical result computed by just one method without corroboration, so I find discrepancies that others, more trusting, may overlook. If I could become more trusting, or if the incidence of computational malfunction due to rounding errors could be reduced by an order of magnitude, I could spend more time with my grandchildren.

. . . .

We should not be surprised if Computer Science students prefer a sliver under the fingernail to compulsory study of Numerical Analysis. It's a horrible subject. We have just been asked to believe paradoxical assertions about a composed program `f(x) = g(h(x))` that implements a composed function F(x) = G(H(x)) :

• If `h`(x) approximates H(x) badly, `f`(x) may still approximate F(x) well.

• If `h`(x) and `g`(y) approximate H(x) and G(y) well, `f`(x) may still approximate F(x) badly.

We are assured that improvements in accuracy will usually reduce the risk of inaccuracy; and we are told how much reduction to expect, but not how bad the risks are nor how to ascertain when whatever might go wrong has gone wrong. What is a conscientious programmer to do?

The best policy is to strive for as much accuracy as possible without unduly degrading speed. This policy will not avoid all dilemmas. For instance, the conscientious programmer knows that iterative refinement is probably the best way to reduce the size of an overly big residual so that a computed solution won't look so bad. However, just when ill condition most jeopardizes that solution's accuracy, iterative refinement is likely to worsen it on machines that cannot or do not accumulate residuals extra-precisely. And ill condition is seldom obvious. To refine, or not to refine? That is the question, a dilemma for the conscientious programmer.

Matlab can alleviate this kind of dilemma for the overwhelming majority of computers on desktops by turning on extra-precise arithmetic for matrix multiplications and exponentiations whenever a computer has it. What about benighted computers that lack it? Tough.

Example 4. A Simple Geometry Problem:

If you believe that the weighty analyses above are for other people with weighty computations, not for people like yourself with computations too short and simple to be so terribly vulnerable to roundoff, this example is for you.

Given are the coordinates of a point $\mathbf{y}$, and the equations $\mathbf{b}'\cdot\mathbf{x} = ß$ and $\mathbf{p}'\cdot\mathbf{x} = \pi$ of two planes that intersect in some line £ in Euclidean 3-space. Sought are the coordinates of the point $\mathbf{z}$ on £ nearest $\mathbf{y}$. The following neat text-book formula solves this problem:

$$\mathbf{z} = (\mathbf{v}\cdot\mathbf{v}'\cdot\mathbf{y} + \mathbf{v}\times(\mathbf{p}\cdot ß - \mathbf{b}\cdot\pi))/(\mathbf{v}'\cdot\mathbf{v}) \quad \text{wherein } \mathbf{v} = \mathbf{p}\times\mathbf{b}.$$

Here $\mathbf{v}\times\mathbf{u}$ stands for the cross-product that Matlab calls `cross(v, u)`. Its elements are expressions like `v(2)*u(3)–v(3)*u(2)` that are vulnerable to cancellation when computed in the obvious way, as `cross` does. For our purposes the cross-product is better expressed as a matrix-vector product `crs(v)*u` where `crs(v)` is the skew matrix

```
crs( [v1; v2; v3] ) =  [  0, -v3,  v2 ;
                          v3,   0, -v1 ;
                         -v2,  v1,   0  ] .
```

Besides running slightly faster, `crs(v)*u` differs from `cross(v, u)` in the following ways: On computers that accumulate matrix products extra-precisely `crs(v)*u` is more accurate. On computers with a FMAC `crs(v)*u` commits six rounding errors instead of nine but violates the identity $\mathbf{v}\times\mathbf{u} = -\mathbf{u}\times\mathbf{v}$ slightly. On other computers there is no difference. Now the formula to compute $\mathbf{z}$ becomes simply

```
    v = crs(p)*b ;   z = ([v, crs(v)]*[ v'*y ; [p, b]*[ß; –π] ])/(v'*v) ;
```

in Matlab’s notation.

The numerical data for our problem consists of $\mathbf{y} := [1; -1; 1]$ and, for $n = 1, 2, 3, \dots, 50$,

$$[\mathbf{p}', \pi] := [F_{n+4}, F_{n+3}, F_{n+2}, F_{n+1}]\cdot u_n, \quad [\mathbf{b}', ß] := [F_{n+3}, F_{n+2}, F_{n+1}, F_n]\cdot v_n$$

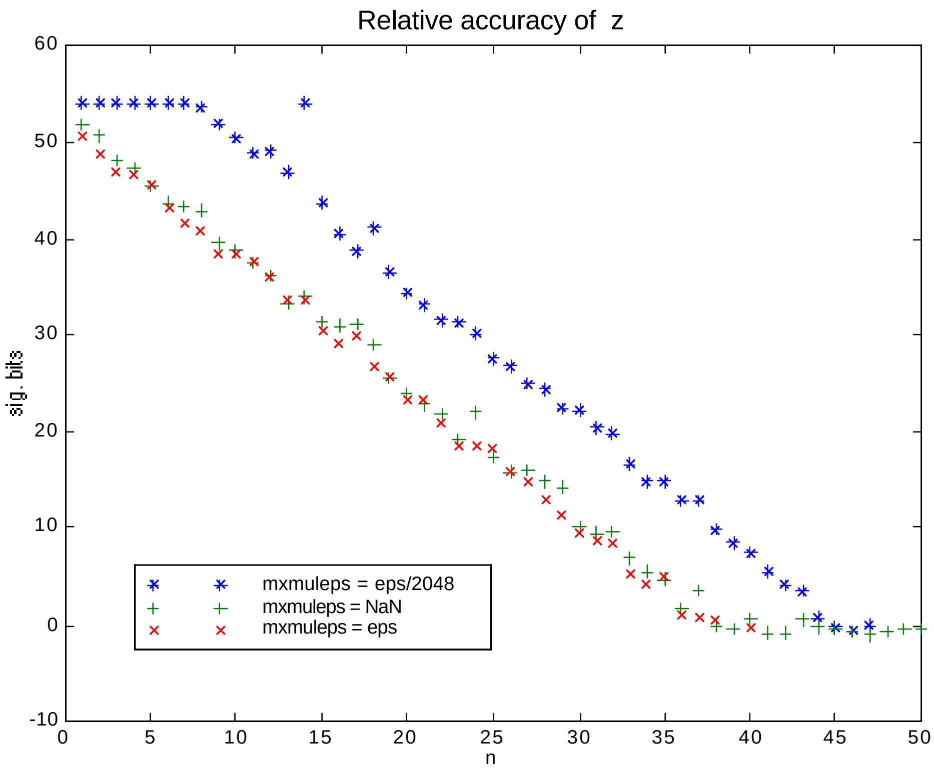
where each F_k is a Fibonacci number, and the multipliers $u_n \approx 5/3$ and $v_n \approx 4/5$ are rounded short in such a way that their products with F_k’s are nonintegers computed exactly:

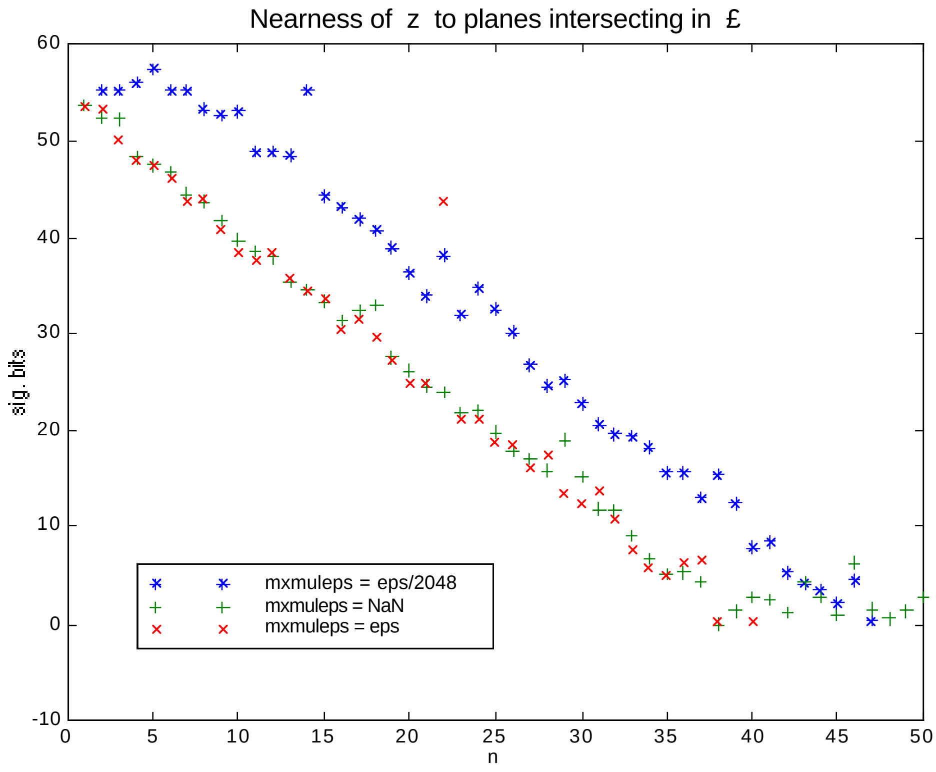
$$u_n := 1 + 2\cdot(\{F_{n+4} + 1/3\} - F_{n+4}), \quad v_n := 1 + (\{F_{n+3} - 1/5\} - F_{n+3}).$$

As n increases the line £ stays unchanged as does the correct solution $\mathbf{z} = [1/3; 2/3; -4/3]$, but roundoff gradually degrades the computation of $\mathbf{z}$ as the two planes that intersect in £ become more nearly parallel; the angle between them is approximately $0.195/2.618^n$.

How much accuracy does $\mathbf{z}$ deserve? Intuition may suggest that $\mathbf{z}$, as the location of a function’s minimum, deserves to lose half the sig. digits carried by the arithmetic. Alternatively, imagine the line segment joining $\mathbf{y}$ to $\mathbf{z}$ to be an elastic band stretched tight around £ and $\mathbf{y}$, and unable to resist infinitesimal displacements of $\mathbf{z}$ along £. But intuition and imagination rarely accord with roundoff’s effect upon a formula. The accuracy of $\mathbf{z}$ actually depends primarily upon the accuracy with which £ can be located, and this depends upon the accuracies of $\mathbf{v} = \mathbf{p}\times\mathbf{b}$ and of $\mathbf{p}\cdot ß - \mathbf{b}\cdot\pi$. Thus the computation’s several earliest rounding errors dominate the rest.

Plotted against n below are the relative accuracies of the computed solution vectors $\mathbf{z}$ and also how close they are to the given planes through £, all measured in sig. bits, as computed on systems with different values of `mxmuleps`. The nearness of $\mathbf{z}$ to both planes matters because excessive departure from those planes can introduce intolerable geometrical inconsistencies into subsequent computation. For a given error tolerance, angles between planes can be roughly 1/2048 smaller with extra-precise accumulation than without. This is the bottom line.


Relative accuracy of z
sig. bits
n
mxmuleps = eps/2048
mxmuleps = NaN
mxmuleps = eps
60
50
40
30
20
10
0
-10
0
5
10
15
20
25
30
35
40
45
50


Nearness of z to planes intersecting in £
sig. bits
n
mxmuleps = eps/2048
mxmuleps = NaN
mxmuleps = eps
60
50
40
30
20
10
0
-10
0
5
10
15
20
25
30
35
40
45
50

A Proposal for Modest Modifications to Matlab :

The first proposal is to turn extra-precise accumulation back on for exponentiations and matrix multiplications on computers capable of the extra accuracy at negligible cost. Also, at least on these computers (and perhaps on all), offer iterative refinement as an option to `/` and `\`. This proposal does not require that Matlab's users be allowed to declare extra-precise `longdouble` variables, nor that Matlab evaluate all expressions extra-precisely. Only atomic operations built into Matlab's kernel need exploit `longdouble`, and only when accuracy is enhanced at no great loss in speed, so that users of these atomic operations (which a user could not easily replace by his own revised versions, unlike .m files) would enjoy more reliable results. Foremost among these atomic operations are matrix multiplications for full and sparse matrices, real and complex. Now many atomic operations give slightly different results on different computers anyway; why not get different results that are better results whenever the hardware affords the opportunity?

Complacency about accuracy may be traceable to false presumptions like this: Matlab statement `P = X*Y` intended to compute a matrix product $\mathbf{P} = \mathbf{X}\cdot\mathbf{Y}$ produces instead $\mathbf{P}$ satisfying an equation $\mathbf{P}+\partial\mathbf{P} = (\mathbf{X}+\partial\mathbf{X})\cdot(\mathbf{Y}+\partial\mathbf{Y})$ in which perturbations $\partial\mathbf{P}$, $\partial\mathbf{X}$, $\partial\mathbf{Y}$ are widely presumed to amount to at most an ULP or two elementwise in their respective matrices $\mathbf{P}$, $\mathbf{X}$, $\mathbf{Y}$. This is true for special cases, as when $\mathbf{X}$ or $\mathbf{Y}$ is a vector, but false in general; the smallest perturbations that satisfy such an equation can be orders of magnitude bigger than an ULP or two. Extra-precise accumulation reduces those perturbations by enough to make that presumption almost always valid.

Matlab is written in **C** and is therefore unable to do more than **C** compilers allow to be done. Microsoft's **C** does not support the `longdouble` format on PCs. This does not excuse Matlab's decision to eschew `longdouble` ; Borland's **C** supports it, and Intel's VTUNE 2.5 optimizes object code that uses it on Pentiums. On computers that lack `longdouble` it is treated just like `double` by **C** compilers. Also, different compilers have different Math. libraries of different qualities; if a compiler's `pow(y, x)` computes y^x less accurately than `fdlibm`'s does and not much faster, then Matlab should replace the poor program by a good one.

Although exception handling has not been discussed in this document, the second proposal is to provide access to the flags and modes mandated on all computers that conform to IEEE Standard 754 for Floating-Point Arithmetic. The flags would permit the detection of, and subsequent compensation for, …

- Invalid Operations that now generate NaNs which can get lost during comparisons,
- Divisions-by-Zero and other operations that generate Infinity exactly from finite operands,
- Overflows that generate Infinities as often poor approximations for huge finite numbers,
- Underflows which can sometimes produce misleading results even though they are Gradual,
- Inexact Operations whose results may be rounded unexpectedly or undesirably.

The modes would allow the direction of rounding to be altered at least for algebraic operations if not also transcendental functions. The principal value of directed roundings is that their effect upon computational modules (functions and operations) often exposes a module's numerical instability, helping thus to isolate the more likely sources of error in results suspected of gross inaccuracy due to roundoff. In short, the modes and flags permit programs and their users more easily to detect and diagnose difficulties that are otherwise practically inscrutable.

Finally, on computers that offer extra-precise arithmetic, provide the IEEE's recommended modes to control its precision; these would have made this work a lot easier.

Ruminations and Conclusions:

The quality of a computer system depends upon more than just its speed and memory capacities. That quality or the lack of it is difficult to conceal completely; it seeps through even software designed to hide it. Does it come to the notice of the marketplace? If not, there is no economic incentive to design better systems nor deterrent to designing worse.

> A mason was chastised for wasting 16% of his time smoothing all six sides of a stone block destined for the outer wall of a mediæval cathedral. “One of those sides will face inside the wall where it cannot be seen; who will know or care if you leave it rough?” asked his critic. “God will know” was the reply.

Usually the inner compulsion to build better than barely good enough benefits us all regardless of whether we share it. Computer arithmetic is different; if a programmer cannot depend upon the presence of better arithmetic hardware he must presume its absence and program accordingly, leaving better qualities unexploited. Something worse than that has happened to Matlab. In the overwhelming majority of computer systems on which it runs, better arithmetic quality has been built into the hardware, but Matlab 5.x has turned it off. Why?

The current fad in software requires it to get the same result everywhere, on all computers. This is a good idea in most respects. However, computer arithmetics vary; they always have and they always will despite conformity to standards that enforce a considerable degree of uniformity. Variations, from one computer to another, in the relative speeds of different arithmetic operations and memory accesses already suffice to induce a programmer and an optimizing compiler to alter numerical algorithms in ways that will best suit the target computer. Such alterations would affect ideally only an algorithm’s speed; often its roundoff is affected too. The availability of a Fused MAC or extra precision may induce further alterations, or not, as we have seen.

Therefore computed results will vary, from one computer to another, unless that variation is inhibited by some conscious effort. This effort is tantamount to a choice: From the diverse results that diverse computers might deliver naturally, one result is chosen for all computers to deliver no matter how unnatural it may be for some of them. What principles should guide the choice of this distinguished result?

One principle is “Majority rule”: Choose the result that most computer systems get, if there is one. Another principle is “Strive for Excellence”: Choose the best result, if there are good ones. Neither of these principles explains Matlab’s past choices so well as “Expedience” does.

Actually, Matlab hasn’t made all its choices yet because it does get different results for matrix products whenever it uses the Fused MAC, or when it sums scalar products in different orders on different machines. However, future versions of Matlab could, for all we know, be designed to get results identical on all computer systems though slower on some. Do you think this would improve Matlab ? If so, which distinguished result should Matlab choose?

If I have a vote, I prefer that Matlab be designed to get about as good a result as can be gotten from the underlying hardware at nearly its full speed even if this result varies from one computer arithmetic to another. Then people who buy better arithmetics will usually get better results, and better results will help make computing more reliable instead of merely more uniform.

For Further Reading:

A case study "Miscalculating Area and Angles of a Needle-like Triangle" posted on the web at `http://http.cs.berkeley.edu/~wkahan/Triangle.ps` makes the case for extra-precise arithmetic by analyzing formulas for elementary trigonometric problems taught in high schools; these formulas are numerically unstable and yet they continue to be taught in lieu of simple but little known stable formulas. The formulas illustrate misconceptions about numerical analysis that continue to be taught in colleges and enshrined in programming languages.

Matlab's users may be less interested in triangles than in matrix computations. Several recent publications by SIAM (Society for Industrial & Applied Mathematics, Philadelphia) are devoted to numerical linear algebra and its error-analysis. J.W. Demmel's 419 page book *Applied Numerical Linear Algebra* (1997) has an up-to-date 20-page Bibliography of 271 items. Its items [10] and [34] are the *Users' Guide* s to LAPACK (1995) and ScaLAPACK (1997) wherein algorithms better than some of Matlab's can be found. N.J. Higham's book *Accuracy and Stability of Numerical Algorithms* 2d. ed. (2002) provides the best coverage of a broader range of topics.

None of these works is light reading. None say much about the use of extra-precise arithmetic with only 11 more sig. bits than `double`, partly because experience with that arithmetic is limited for want of compiler support. At this time, Borland's **C** and **C**++ compilers are the only commercially significant compilers I know to support fully all three floating-point formats, `float` and `double` and `long double`, on Intel x86/Pentium-based PCs and clones under Microsoft's Windows. The gcc compiler under Linux does less well. Other compilers and programming languages are crippled by floating-point misconceptions similar to those that afflict the new language Java™ ; see " How Java's Floating-Point Hurts Everyone Everywhere," `http://www.berkeley.edu/~wkahan/JAVAhurt.pdf` .

Acknowledgments:

Thanks are due first to Dr. Cleve Moler of The MathWorks Inc. for keeping me up to date with versions of Matlab. Dr. Ali Sazegari of Apple Computer Inc. has made it possible for me to work on both a Quadra 950 and Power Mac 8500 at the same time. Intel has supplied Pentium-based PCs running Windows NT to the Univ. of Calif. at Berkeley; Hewlett-Packard has supplied hp 715/64s running HPUX to the Computer Science Dept. Without comparing results from different software and different computers, how likely would anyone discover that some of them are wrong? Consequently I am grateful most of all that Matlab does not get identical results everywhere.

Appendix: The Program `divulge.m` :

This program divulges interesting details about the way Matlab uses floating-point arithmetic. Following the program are a few examples of its output.

```
function  [c,v,d,lr,meps,y2n,y2xx,epd] = divulge(machine)
%  [c,v,d,lr,meps,y2x,y2xx,epd] = divulge('machine')  divulges some of  Matlab's
%  arithmetic properties.  The string  'machine'  is echoed for the record.
%    c = computer  as reported by  Matlab
%    v = version  and  d = release date    as reported by  Matlab.
%    lr = direction on which scalar products are summed.
%    meps = mxmuleps ,  q.v.
%    y2n  assesses the ULP accuracy of  (Real)^(Integer)  exponentiation,  and
%    y2xx = 0  unless  (Real)^(Integer)  accumulates products extra-precisely.
%    epd = 1  if extra-precision exists but is discarded from most expressions,
%        = 0  if extra precision is turned off or does not exist,
%        = -1  if extra precision exists but is turned off  ( on a  PC ).
%                                                     (C)  W. Kahan,  4 Aug. 1998
disp(['Matlab  divulges its arithmetic when run on a  ', machine ]) ;
c = computer;
disp(['Matlab  says it is running on a  ', c]) ;
[v, d] = version ;
disp(['Matlab''s  version  is  ', v, '  released on  ', d]) ;
x = ( [eps, 9, 9]*[eps; 9; -9] ~= 0 ) ;
if  x ,
         lr = 'right-to-left';
    else
         lr = 'left-to-right' ;
    end
disp([' Scalar product  a''*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)']) ;
disp(['          is summed from  ', lr, ' .']) ;
y = [1-eps,  eps-1]*[1+eps; eps+1] ;
meps = mxmuleps ;
if isnan(meps),
        me = ' is NaN ' ;  y = (y==0)|((y<0)~=x) ;
   elseif  meps==eps
        me = ' = eps' ;
   else
        me = [' = eps / ',num2str(eps/meps)] ;
   end
disp([' mxmuleps', me, ' .' ]) ;
if y ,
    disp([' Something about the previous two statements is wrong.']) ;
   end
x = ones(128,1)*([1:2:127]/65536 + 1) ;
N = [65501:2:65755]'*ones(1,64) ;
x3N = x.^(3*N) ;
y2n = max(max(abs( ( x3N - (x.*x.*x).^N )./x3N )))/eps ;
Y2N = num2str(y2n) ;
disp([' y^N  errors as big as  ', Y2N, ' ULPs  have been seen.']) ;
x = 0.5 + rand(32,32) ;
N = round(256*rand(32,32)) + 7 ;
y2xx = any(any(  x.^(2*N) - (x.*x).^N  )) ;
if  y2xx,  not = '' ;
    else,  not = 'not' ;  end
disp([' y^N  is ', not, ' accumulated extra-precisely.']) ;
```

```
y = 1 + eps ;  d = 1 - eps/2 ;
x = 1 + y*eps/2 ;  epd = (x == 1) ;
x = y + d*eps/2 ;  epd1 = (x ~= y) ;
if  (epd~=epd1),
    disp([' Something about the next statement is wrong.']) ;
  end
if  epd,
    epds = 'discarded from most (sub)expressions' ;
  else
    if v(1) > int2str(3) ,   t = realmin ;
         else                t = 0.5^1022 ;  end
    z = ( t*(2 - 2*eps) )/(2 - eps) ;
    if  z < t ,  epds = 'turned off or do not exist' ;
         else    epds = 'turned off' ;  epd = -1 ;  end
  end
disp([' Extra-precise digits are ', epds, '.']) ;
```

```
divulge('Mac Quadra 950')
Matlab  divulges its arithmetic when run on a  Mac Quadra 950
Matlab  says it is running on a  MAC2
Matlab's  version  is  5.0.0.4075  released on  Dec 13 1996
 Scalar product  a'*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
          is summed from  left-to-right .
 mxmuleps = eps / 2048 .
 y^N  errors as big as  0.99903 ULPs  have been seen.
 y^N  is  accumulated extra-precisely.
 Extra-precise digits are discarded from most (sub)expressions.


divulge('PowerMac 8500')
Matlab  divulges its arithmetic when run on a  PowerMac 8500
Matlab  says it is running on a  MAC2
Matlab's  version  is  5.2.0.3084  released on  Jan 26 1998
 Scalar product  a'*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
          is summed from  left-to-right .
 mxmuleps is NaN  .
 y^N  errors as big as  0.99942 ULPs  have been seen.
 y^N  is  accumulated extra-precisely.
 Extra-precise digits are turned off or do not exist.


» divulge('Pentium II under WinNT/DOS')
Matlab  divulges its arithmetic when run on a  Pentium II under WinNT/DOS
Matlab  says it is running on a  PC386
Matlab's  version  is  3.5m  released on  Aug 12 1992
 Scalar product  a'*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
          is summed from  left-to-right .
 mxmuleps = eps / 2048 .
 y^N  errors as big as  0.9942 ULPs  have been seen.
 y^N  is  accumulated extra-precisely.
 Extra-precise digits are discarded from most (sub)expressions.
```

```
>> divulge('hp-715')
Matlab  divulges its arithmetic when run on a  hp-715
Matlab  says it is running on a  HP700
Matlab's  version  is  5.0.0.4064  released on  Nov 15 1996
 Scalar product  a'*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
          is summed from  left-to-right .
 mxmuleps = eps .
 y^N  errors as big as  52101.6199 ULPs  have been seen.
 y^N  is not accumulated extra-precisely.
 Extra-precise digits are turned off or do not exist.
```

```
divulge('Pentium II under WinNT')
Matlab  divulges its arithmetic when run on a  Pentium II under WinNT
Matlab  says it is running on a  PCWIN
Matlab's  version  is  4.2c.1  released on  Oct 03 1994
 Scalar product  a'*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
          is summed from  left-to-right .
 mxmuleps = eps / 2048 .
 y^N  errors as big as  5.214e+004 ULPs  have been seen.
 y^N  is not accumulated extra-precisely.
 Extra-precise digits are discarded from most (sub)expressions.
```

```
divulge('Pentium under Windows 98')
Matlab  divulges its arithmetic when run on a  Pentium under Windows 98
Matlab  says it is running on a  PCWIN
Matlab's  version  is  5.2.0.3084  released on  Jan 17 1998
 Scalar product  a'*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
          is summed from  right-to-left .
 mxmuleps = eps .
 y^N  errors as big as  52101.6199 ULPs  have been seen.
 y^N  is not accumulated extra-precisely.
 Extra-precise digits are turned off.
```

Compared with previous versions of Matlab on Wintel PCs, version 5 sums matrix products in the opposite direction, computes exponentials inaccurately, and no longer accumulates matrix products extra-precisely (`mxmuleps = eps`), and all because it left the hardware’s extra digits turned off.

. . . .

Addendum (July 2003): Matlab 6.5 on PCs reverts to summing matrix products left-to-right, as in version 4, and provides a way to re-enable extra-precise accumulation of matrix products:

```
>>  system_dependent(‘setprecision’, 64)
```

On Wintel PCs the default is “53” rather than “64”. Another alternative is “24”, which rounds at least matrix product accumulation to 24 sig. bits. Rounding direction is influenced by …

```
>>  system_dependent(‘setround’, r#)
```

rounding *Up* if `r# = inf`, *Down* if `r# = –inf`, *To Zero* if `r# = 0`, *To Nearest* if `r# = 0.5` or `‘nearest’` (the default). All this comes from a footnote on p. 55 of Mike Overton’s book *Numerical Computing with IEEE Floating Point Arithmetic* (2001, SIAM, Philadelphia). I wonder: what else can `system_dependent(…)` do?

Here are results from `divulge` when it ran in Matlab 6.5 on an IBM PC under Windows 2000:

```
divulge('Win2000pc')
Matlab  divulges its arithmetic when run on a  Win2000pc
Matlab  says it is running on a  PCWIN
Matlab's  version  is  6.5.0.180913a (R13)  released on  Jun 18 2002
 Scalar product  a'*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
          is summed from  left-to-right .
 mxmuleps = eps .
 y^N  errors as big as  0.83399 ULPs  have been seen.
 y^N  is  accumulated extra-precisely.
 Extra-precise digits are turned off or do not exist.


system_dependent('setprecision', 64)
divulge('Win2000pc64')
Matlab  divulges its arithmetic when run on a  Win2000pc64
Matlab  says it is running on a  PCWIN
Matlab's  version  is  6.5.0.180913a (R13)  released on  Jun 18 2002
 Scalar product  a'*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
          is summed from  left-to-right .
 mxmuleps = eps / 2048 .
 y^N  errors as big as  0 ULPs  have been seen.
 y^N  is  accumulated extra-precisely.
 Something about the next statement is wrong.
 Extra-precise digits are turned off or do not exist.


system_dependent('setprecision', 24)
divulge('Win2000pc24')
Matlab  divulges its arithmetic when run on a  Win2000pc24
Matlab  says it is running on a  PCWIN
Matlab's  version  is  6.5.0.180913a (R13)  released on  Jun 18 2002
 Scalar product  a'*b = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
          is summed from  left-to-right .
 Has precision been altered?  Why do the following differ? ...
AnULPofOne =
     0
Eps =
     2.22044604925031e-016
 Now  mxmuleps  cannot be trusted.
 mxmuleps is NaN  .
 Something about the previous two statements is wrong.
 y^N  errors as big as  2.609722e+013 ULPs  have been seen.
 y^N  is  accumulated extra-precisely.
 Extra-precise digits are discarded from most (sub)expressions.
```

Deciphering the results above for Win2000pc64 and Win2000pc24 will take some time.