Do MATLAB's `lu(…)`, `inv(…)`, `/` and `\` have *Failure Modes* ?

These exercises are designed to exhibit and explain the perverse behavior of MATLAB's inverse-related operations `lu(…)`, `inv(…)`, `/` and `\` upon certain matrices. This perverse behavior occurs only very rarely and afflicts other matrix-handling software like LINPACK and LAPACK too. As three examples will illustrate, the behavior is perverse for three reasons:

- It causes gross inaccuracy which could often be avoided by altering algorithms slightly.
- It can happen to otherwise unexceptionable matrices about which nothing is wrong.
- The gross inaccuracy may be merely an artifact of the norm by which it is assessed.

The last point means that gross inaccuracy may go away, *often with no change to computed results,* when the norm, by which variations are deemed "negligible" or not, is replaced by another equally plausible norm. This compounds perversity with perplexity.

A First Example of Failure

A parameter h is given accurate to at least ten sig. dec.; say h := 1000000.0000 . Then matrices

$$A := \begin{bmatrix} -2 & 1 & 1 \\ 1 & h^{-2} & h^{-2} \\ 1 & h^{-2} & 0 \end{bmatrix} \quad \text{and} \quad b := \begin{bmatrix} 0 \\ 0 \\ h^{-1} \end{bmatrix} \quad \text{and later} \quad Y := \begin{bmatrix} h^{-1} & 0 & 0 \\ 0 & h & 0 \\ 0 & 0 & h \end{bmatrix}$$

are constructed, and the linear system $A \cdot z = b$ is to be solved for z at least about as accurately as it is determined by the given data. One approximation to z is MATLAB's `x = A\b` : we may expect it to be accurate enough since MATLAB carries 53 sig. bits, worth more than fifteen sig. dec., and we need only ten. However, just to check up on the computation's accuracy, we also compute matrices $YAY := Y \cdot A \cdot Y$ and $Yb := Y \cdot b$ and then solve $YAY \cdot vx = Yb$ for vx by computing MATLAB's `vx = YAY\Yb` , from which $Yvx := Y \cdot vx$ is obtained. Ideally we expect $Yvx \approx x \approx z$ to within a few rounding errors, but something else happened on my Macintosh:

$$x = \begin{bmatrix} 0 \\ 1000022.1222095 \\ -1000022.1222095 \end{bmatrix} \quad \text{differs from} \quad Yvx = \begin{bmatrix} 0 \\ 1000000 \\ -1000000 \end{bmatrix} \quad \text{in the worst way,}$$

by too little to be obvious but by too much to tolerate. What results do you get? Which, if either, is correct? You can solve $A \cdot z = b$ in your head. Why did MATLAB get it wrong?

Iterative Refinement is a way to improve (usually) the computed solution x of an equation. First compute a residual $r := b - A \cdot x$ as accurately as you can at a tolerable price; then re-use the triangular factorization of A that occurred during the solution of $A \cdot x = b$ to solve $A \cdot \Delta x = r$ for Δx , though this too will be computed only approximately. Finally replace x by $x + \Delta x$ to enhance its accuracy. The process may be *iterated* (repeated) until the *Law of Diminishing Returns* renders further iteration futile. Yvx did not change, but MATLAB changed x to first

$$x = \begin{bmatrix} 0 \\ 1000000 \\ -1000000 \end{bmatrix} \quad \text{and subsequently} \quad x = \begin{bmatrix} 0 \\ 1000000.00000001 \\ -1000000.00000001 \end{bmatrix} .$$

What results do you get on your computer? How do you explain them? What if h = 100000000 ?

Hereunder is the script that delivered the foregoing results in MATLAB 5.2 on my old Macintosh:

```
%  Matlab Script  badscale.m  demonstrates how much
%  scaling a linear system can affect computed results.
h = 1000000.0000  % ...  the parameter,
A = [-2,1,1; 1, h^-2, h^-2; 1, h^-2, 0]
b = [0; 0; 1/h]  % ...  b = A*z ;  solve for  z .
x = A\b  % ...  approximates the solution  z .
Y = diag([1/h, h, h])
YAY = Y*A*Y
Yb = Y*b  % ...  scaled system  Yb = YAY*vz .
vx = YAY\Yb  % ...  approximates scaled solution  vz .
x_Yvx = [x, Y*vx]  % ...  compare solutions.
%  Iterative refinement to improve accuracies:
r = [b, A]*[1; -x]  % ...  r = b - A*x  but computed
%   more accurately on a  Mac Quadra 950 in Matlab 5.2.
dx = A\r  % ...  correction to  x .
Yr = [Yb, YAY]*[1; -vx]
dvx = YAY\Yr
x = x+dx  % ...  updated  x .
vx = vx + dvx  % ...  updated scaled  x .
x_Yvx = [x, Y*vx]  % ...  compare solutions again.
r = [b, A]*[1; -x]  % ...  iterate refinement.
dx = A\r
x = x + dx
x_Yvx = [x, Y*vx]  % ...  compare solutions again.
```

This script doesn't mention MATLAB's `norm(..., p)` for $p = 1, 2$ or ∞. Denote one by $\|\ldots\|$.

Diagonal scaling that changes A to YAY can be interpreted as a change of norm from $\|x\|$ to $[\![x]\!] := \|Y^{-1}\cdot x\|$ for vectors x in the domain of A which, since both A and YAY are symmetric, is best regarded as the matrix of a linear map from column vectors x to the *Dual-space* of rows $w' = (A\cdot x)'$. Dual-spaces and norms are explained at length in course notes posted at
<www.cs.berkeley.edu/~wkahan/MathH110/NORMlite.pdf> .
The dual of column-norm $\|x\|$ is row-norm $\|w'\| := \max_{x\neq o} |w'\cdot x|/\|x\|$; the dual of $[\![x]\!]$ is $[\![w']\!] := \max_{x\neq o} |w'\cdot x|/[\![x]\!] = \|w'\cdot Y\|$. The *Operator Norm* $\|A\| := \max_{x\neq o} \|(A\cdot x)'\|/\|x\|$ is induced by $\|x\|$; and similarly $[\![A]\!] := \max_{x\neq o} [\![(A\cdot x)']\!]/[\![x]\!] = \max_{x\neq o} \|(A\cdot x)'\cdot Y\|/\|Y^{-1}\cdot x\| = \|YAY\|$. Now observe that A is far farther from singular when perturbations are gauged by $[\![\ldots]\!]$ than by $\|\ldots\|$.

Is $[\![\ldots]\!]$ more appropriate than $\|\ldots\|$ to gauge perturbations in our data? If so, our data $[A, b]$ will be misconstrued as *Ill-Conditioned* by MATLAB, whose programmers had $\|\ldots\|$ in mind. One way to diminish the necessity for mind-reading is to employ iterative refinement routinely.

In general, the numerical solution of a system $B\cdot z = c$ of linear equations is influenced implicitly by the choices of three norms: one for matrices like B, one for columns like c in the target space of B, and one for columns like z in the domain of B. All are denoted by the overloaded symbol $\|\ldots\|$ though they may be very different. Scaling that replaces B by, say, $T^{-1}\cdot B\cdot Y$ can be reinterpreted as changing those norms to $\|T^{-1}\cdot B\cdot Y\|$, $\|T^{-1}\cdot c\|$ and $\|Y^{-1}\cdot z\|$ respectively. Iterative refinement is a way to diminish the generally unknown influences of the norms' choices. Sometimes they affect a computed result a lot. We wish they wouldn't. Sometimes they don't.

A Second Example of Failure

Perversity is elicited next by two families of n-by-n matrices $\mathbf{B}_x$ and $\mathbf{C}_x$ dependent upon a real parameter x restricted henceforth to two values x = 0 and x = 1 . Inverses $\mathbf{B}_x^{-1}$ and $\mathbf{C}_x^{-1}$ can be computed directly, quickly and accurately from short formulas without invoking `inv(…)` . Its perverse behavior will become apparent as n grows past the number of significant bits carried by the matrix operations' binary floating-point arithmetic. 53 sig. bits are carried by MATLAB on practically all commercially significant computers nowadays, so we shall let n grow past 53 .

When n = 6 , $\mathbf{B}_x := \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & 1-x \end{bmatrix}$. In general, $\mathbf{B}_x$ has –1 below its diagonal, 1 on its diagonal, 0 above the diagonal, except for 1 in the last column whose bottom-right-most element $\mathbf{B}_{x(n,n)} = 1–x$. Other matrices needed below will be constructed with the aid of two n-rows $\mathbf{t}' := [\, 2, 4, 8, \ldots. 2^{n-2}, 2^{n-1}, 2^{n-1} \,]$ and $\mathbf{y}' := [\, 1, 2, 4, 8, \ldots, 2^{n-3}, 2^{n-2}, 1 \,]$ (here the prime ' in $\mathbf{y}'$ denotes an array's transpose) and their associated diagonal matrices $\mathbf{T} := \mathrm{Diag}(\mathbf{t})$ and $\mathbf{Y} := \mathrm{Diag}(\mathbf{y})$, plus one more n-row $\mathbf{e}' := [\, 0, 0, 0, \ldots. 0, 1 \,]$. Now it turns out (and you should confirm) that $\mathbf{B}_0^{-1} = \mathbf{Y}\cdot\mathbf{B}_0'\cdot\mathbf{T}^{-1}$ and $\mathbf{B}_x^{-1} = \mathbf{B}_0^{-1} + x\cdot\mathbf{B}_0^{-1}\cdot\mathbf{e}\cdot\mathbf{e}'\cdot\mathbf{B}_0^{-1}/(1 - x/2^{n-1})$.

The sensitivity of $\mathbf{B}^{-1}$ to infinitesimal perturbations $d\mathbf{B}$ in $\mathbf{B}$ can be inferred from the derivative $d(\mathbf{B}^{-1}) = -\mathbf{B}^{-1}\cdot d\mathbf{B}\cdot\mathbf{B}^{-1}$ and is customarily gauged by a *Condition Number* $\kappa(\mathbf{B}) := \|\mathbf{B}^{-1}\|\cdot\|\mathbf{B}\|$ because $\|d(\mathbf{B}^{-1})\|/\|\mathbf{B}^{-1}\| \le \kappa(\mathbf{B})\cdot\|d\mathbf{B}\|/\|\mathbf{B}\|$ with equality possible for some aptly chosen $d\mathbf{B} \ne \mathbf{O}$. Thus $\kappa(\mathbf{B})$ is a kind of relative magnification factor for the worst perturbations. The choice of norm $\|\ldots\|$ appears at first not to matter much unless dimension n gets huge, far bigger than any value n that will be used in this exercise. We shall use $\|\ldots\| := \|\ldots\|_\infty$, which is …

$\|\mathbf{x}\|_\infty :=$ (the largest magnitude among the elements of column $\mathbf{x}$) $= \max_{\mathbf{y}'\ne\mathbf{o}'} \mathbf{y}'\cdot\mathbf{x}/\|\mathbf{y}'\|_\infty$;

$\|\mathbf{y}'\|_\infty := \max_{\mathbf{x}\ne\mathbf{o}} \mathbf{y}'\cdot\mathbf{x}/\|\mathbf{x}\|_\infty =$ (the sum of the magnitudes of the elements of row $\mathbf{y}'$) ;

$\|\mathbf{B}\|_\infty := \max_{\mathbf{x}\ne\mathbf{o}} \|\mathbf{B}\mathbf{x}\|_\infty/\|\mathbf{x}\|_\infty = \max_{\mathbf{y}'\ne\mathbf{o}'} \|\mathbf{y}'\mathbf{B}\|_\infty/\|\mathbf{y}'\|_\infty = \max \|\mathbf{b}'\|_\infty$ over rows $\mathbf{b}'$ of $\mathbf{B}$.

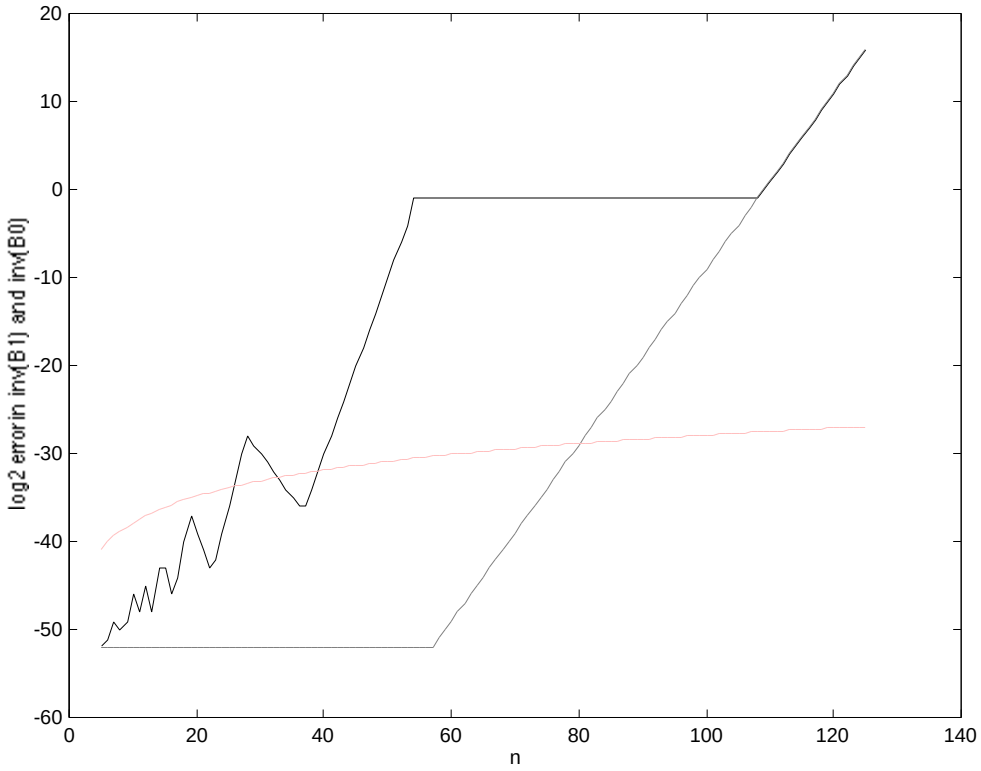
Were a real n-by-n matrix $\mathbf{B}$ chosen at random the expected value of its condition number $\kappa(\mathbf{B})$ would be not very different from n . In this respect our matrices $\mathbf{B}_x$ are not far from average:

$$\|\mathbf{B}_0\| = n ; \quad \|\mathbf{B}_0^{-1}\| = 1 ; \qquad \kappa(\mathbf{B}_0) = n ;$$
$$\|\mathbf{B}_1\| = n ; \quad \|\mathbf{B}_1^{-1}\| = 3/2 - 1/(2^n–2) ; \qquad \kappa(\mathbf{B}_1) = 3n/2 - n/(2^n–2) .$$

With these moderate condition numbers neither $\mathbf{B}_0$ nor $\mathbf{B}_1$ can be considered ill-conditioned. In consequence we might expect both $\mathbf{B}_0^{-1}$ and $\mathbf{B}_1^{-1}$ to be computed with satisfactory accuracy by MATLAB's `inv(…)` . This is so because, in general, we expect `inv(`$\mathbf{B}$`)` to differ in norm from $\mathbf{B}^{-1}$ by not much more than could $(\mathbf{B} + \Delta\mathbf{B})^{-1}$ for some roundoff-induced perturbation $\Delta\mathbf{B}$ no worse in norm than $\|\Delta\mathbf{B}\| \le \mathtt{eps}\cdot g\cdot n^{5/2}\cdot\|\mathbf{B}\|$ and almost always very much smaller. Here `eps` is a roundoff threshold like `eps` := 1.000…001 – 1 ; MATLAB's `eps` $= 2^{-52} \approx 2.22/10^{16}$. And g is a "pivot growth factor" rarely bigger than 8 and almost never bigger than 8n .

In short, we might reasonably expect $\| \texttt{inv}(\mathbf{B}_x) - \mathbf{B}_x^{-1} \|$ to be almost always far smaller than $18\cdot\texttt{eps}\cdot n^{7/2}$ and never to exceed it. But something else happens. Shown below are plots of …

$\log_2\| \texttt{inv}(\mathbf{B}_1) - \mathbf{B}_1^{-1} \|$: —— $\log_2\| \texttt{inv}(\mathbf{B}_0) - \mathbf{B}_0^{-1} \|$: -·-·-· $\log_2(18\cdot\texttt{eps}\cdot n^3)$: · · ·



These results were obtained from MATLAB 5.2 on a 68040-based Apple Quadra 950.

Why do the errors in $\texttt{inv}(\mathbf{B}_x)$ grow far bigger than might reasonably be expected as n increases towards 53 and beyond? Why is $\texttt{inv}(\mathbf{B}_1)$ so much worse than $\texttt{inv}(\mathbf{B}_0)$?

Here is an observation that hints at what is going wrong: $\|\mathbf{B}_1^{-1} - \mathbf{B}_0^{-1}\| = 1/2 + 1/(2^n–2)$ and $\|\mathbf{B}_1 - \mathbf{B}_0\| = 1$ but $\|\texttt{inv}(\mathbf{B}_1) - \texttt{inv}(\mathbf{B}_0)\| = 0$ just for $n > 53$. Why ? What about $\texttt{lu}(\mathbf{B}_x)$?

Now, to compound the foregoing perversity with perplexity, let $\mathbf{C}_x := \mathbf{T}^{-1}\cdot\mathbf{B}_x\cdot\mathbf{Y}$. It is computed exactly because multiplications by $\mathbf{T}$ and $\mathbf{Y}$ merely shift binary points (by adding integers to floating-point exponents) without altering binary floating-point numbers' sig. bits. For example,

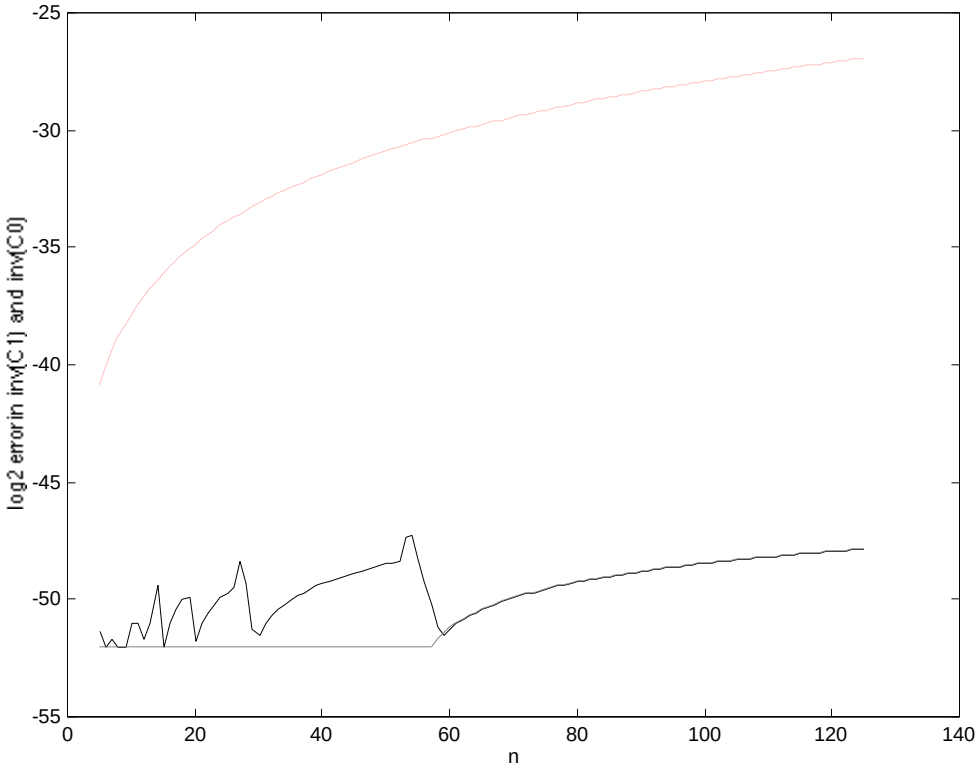
$$\text{if } n = 6\,, \quad \mathbf{C}_x = \begin{bmatrix} 16 & 0 & 0 & 0 & 0 & 16 \\ -8 & 16 & 0 & 0 & 0 & 8 \\ -4 & -8 & 16 & 0 & 0 & 4 \\ -2 & -4 & -8 & 16 & 0 & 2 \\ -1 & -2 & -4 & -8 & 16 & 1 \\ -1 & -2 & -4 & -8 & -16 & 1-x \end{bmatrix} /32\,.$$

Now $\mathbf{C}_x^{-1} = \mathbf{Y}^{-1}\cdot\mathbf{B}_x^{-1}\cdot\mathbf{T} = \mathbf{B}_0' + x\cdot\mathbf{B}_0'\cdot\mathbf{e}\cdot\mathbf{e}'\cdot\mathbf{B}_0'/(2^{n-1} - x)$ and consequently …

$$\|\mathbf{C}_0\| = 1\ ; \quad \|\mathbf{C}_0^{-1}\| = n\ ; \qquad \kappa(\mathbf{C}_0) = n\ ;$$
$$\|\mathbf{C}_1\| = 1\ ; \quad \|\mathbf{C}_1^{-1}\| = n/(1 - 2^{1-n})\ ; \qquad \kappa(\mathbf{C}_1) = n/(1 - 2^{1-n})\ :$$
$$\|\mathbf{C}_1 - \mathbf{C}_0\| = 2^{1-n}\ ; \qquad \|\mathbf{C}_1^{-1} - \mathbf{C}_0^{-1}\| = n/(2^{n-1} - 1)\ .$$

Like $\mathbf{B}_x$, matrices $\mathbf{C}_x$ have condition numbers near the average, n. Consequently when n gets so big that $\mathbf{C}_1$ barely differs from $\mathbf{C}_0$ in norm, so does $\mathbf{C}_1^{-1}$ differ little from $\mathbf{C}_0^{-1}$ in norm. And roundoff conforms to expectations in so far as $\mathtt{inv}(\mathbf{C}_x)$ differs little from $\mathbf{C}_x^{-1}$ in norm:

$\log_2\|\,\mathtt{inv}(\mathbf{C}_1) - \mathbf{C}_1^{-1}\,\|$: —— $\log_2\|\,\mathtt{inv}(\mathbf{C}_0) - \mathbf{C}_0^{-1}\,\|$: -·-·-· $\log_2(18\cdot\mathtt{eps}\cdot n^3)$: · · ·



But, just as $\mathbf{C}_x^{-1} = \mathbf{Y}^{-1}\cdot\mathbf{B}_x^{-1}\cdot\mathbf{T}$, so does $\mathtt{inv}(\mathbf{C}_x) = \mathbf{Y}^{-1}\cdot\mathtt{inv}(\mathbf{B}_x)\cdot\mathbf{T}$ ***exactly*** including all rounding errors. In other words, every element of $\mathtt{inv}(\mathbf{C}_x)$ has exactly the same sig. bits as has the corresponding element of $\mathtt{inv}(\mathbf{B}_x)$; only their exponents differ, so one is as accurate as the other so far as sig. bits go. Why is $\mathtt{inv}(\mathbf{C}_x)$ so much more accurate than $\mathtt{inv}(\mathbf{B}_x)$ in norm?

Iterative Refinement often improves the accuracy of computed inverses, though at a cost far from negligible. If an approximation E to B^{-1} is obtained from triangular factors, they can be reused to enhance accuracy by solving $B\cdot\Delta E := (I - B\cdot E)$ for ΔE (approximately) and then updating E to $E + \Delta E$. What does this iterative refinement do for computed inverses above, and why?

Hereunder are six MATLAB programs used to get the foregoing results:

Y = diag(oy(n)) :

```
function  y = oy(n)
%  oy(n) = [1, 2, 4, 8, ..., 2^(n-3), 2^(n-2), 1 ]
if  (n < 2),  y = ones(1,n) ;  return,  end
y = [ cumprod([1, 2*ones(1,n-2)]), 1 ] ;
```

$\mathbf{T}^{-1}$ = diag(oti(n)) :

```
function  t = oti(n)
%  oti(n) = [ 1/2, 1/4, 1/8, ..., 2^(2-n), 2^(1-n), 2^(1-n) ]
t = cumprod( [ ones(1, n-1)*0.5, 1 ] ) ;
```

$\mathbf{B}_x$ = ob(x, n) :

```
function  B = ob(x, n)
%             [ +1   0   0   0   1 ]
%             [ -1  +1   0   0   1 ]
%  ob(x, 5) = [ -1  -1  +1   0   1 ] ,  for example.
%             [ -1  -1  -1  +1   1 ]
%             [ -1  -1  -1  -1  1-x]
B = eye(n) - tril(ones(n), -1) ;
B(:,n) = ones(n, 1) ;
B(n,n) = 1-x ;
```

$\mathbf{B}_x^{-1}$ = obi(x, n) :

```
function  Z = obi(x, n)
%  obi(x, n) = inv( ob(x, n) )  accurately without using  inv(...)
%  but using  ob.m,  oy.m  and  oti.m,  q.v.
B = ob(0, n)' ;
t = oti(n) ;  y = oy(n) ;
Z = (y'*t).*B ;  % = inv(b(0,n))
Z = Z + ((x/(1-x*t(n)))*Z(:,n))*Z(n,:) ;
```

$\mathbf{C}_x$ = oc(x, n) :

```
function  C = oc(x, n)
%  oc(x, n) = diag(oti(n))*ob(x, n)*diag(oy(n))
C = (oti(n)'*oy(n)).*ob(x,n) ;
```

$\mathbf{C}_x^{-1}$ = oci(x, n) :

```
function  Z = oci(x, n)
%  oci(x, n) = inv(oc(x, n))  accurately without
%  using  inv(...)  but using  ob.m,  q.v.
B = ob(0,n)' ;
Z = B + ((x/(2^(n-1) - x))*B(:,n))*B(n,:) ;
```

It is tempting to explain the drastically different appraisals of what are essentially the same errors in these programs as mere consequences of the uses of drastically different norms to gauge error. This explanation's relativism is mistaken. Instead, indications that the triangular factors of B_x of large dimension n would be corrupted by roundoff could and should have been noticed by `lu(…)` and `inv(…)`, and should have elicited a warning message and advice to *Change Column Order.* The next example will help to explain why column order is relevant.

A Third Example of Failure

The previous examples' matrices all had inverses that could be computed accurately enough from relatively simple formulas. Next suppose a numerically computed G approximates the inverse of a given matrix F whose inverse is unknown or (for this example) determinable from a formula too complicated to be worth computing accurately. How can the accuracy of G be gauged? In general, so far as is known nowadays, estimating the accuracy of G reliably costs at least about as much (time) again as was spent to compute G .

For instance, if we obtained G from MATLAB, say from `G = inv(F)` , we could compare it with `G1 = inv(G')'` and with `G2 = flipud(inv(fliplr(G)))` , say. Agreement proves nothing. Substantial differences would indicate that at least one is wrong, but which? Maybe all?

Better information may come from residuals $R := G\cdot F - I$ and $L := F\cdot G - I$. Both should vanish if G is exactly right, but that hardly ever happens. Usually at least one of $\|L\|$ and $\|R\|$ is tiny. Only in very ill-conditioned cases can $\|L\|$ and $\|R\|$ differ by many orders of magnitude:

Exercise: Show that $1/\kappa(F) \le \|L\|/\|R\| \le \kappa(F)$, and either bound can achieved.

Our third example will be well-conditioned with condition number $\kappa(\ldots) < 22$, so little will be lost by computing only $\|R\|$. It provides upper bounds for the relative error in G thus:

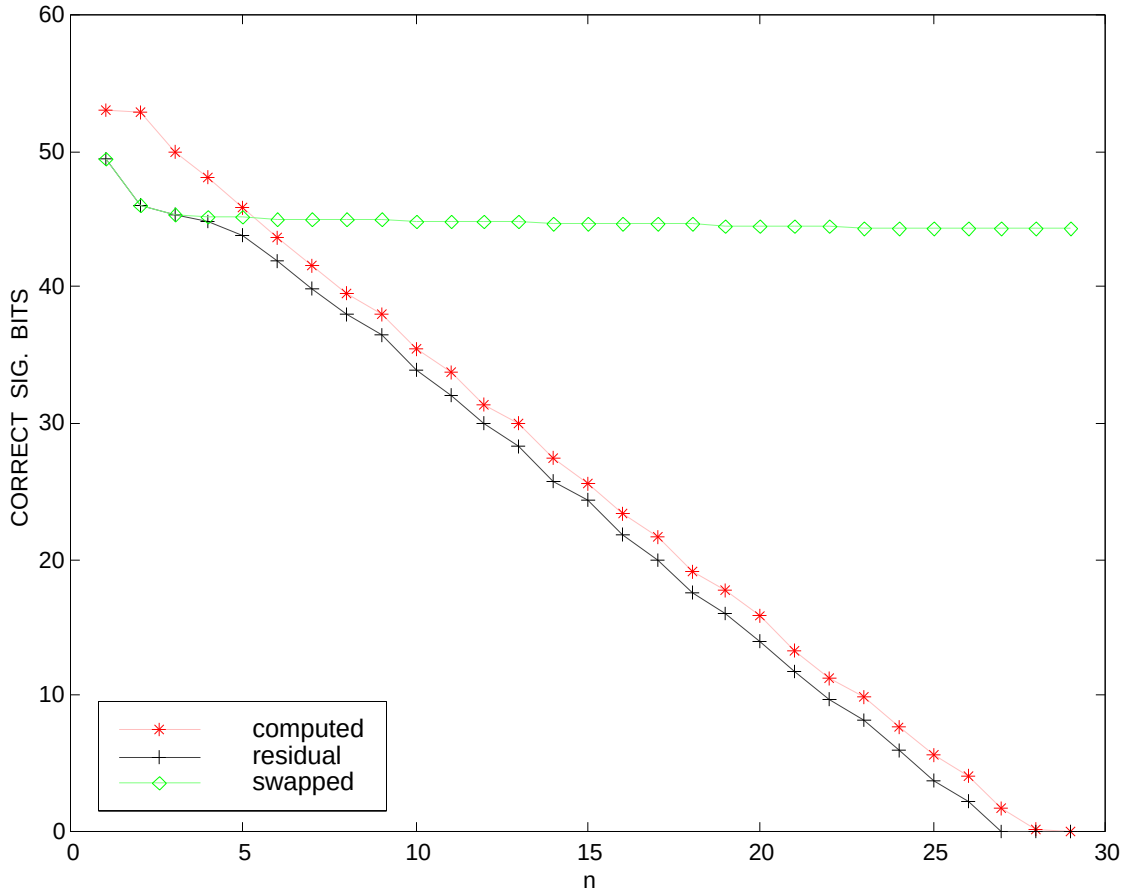
Exercise: Show that $\|G - F^{-1}\|/\|F^{-1}\| \le \|R\|$, and this bound is achievable. Show too that $\|G - F^{-1}\|/\|G\| \le \|R\|/(1 - \|R\|)$ provided $\|R\| < 1$.

This exercise's error bound will be used to assess the inaccuracy of an inverse computed one way, and to confirm its accuracy computed another way.

Our third example is a k·n-by-k·n matrix exemplified for n = 6 thus: $H := \begin{bmatrix} I & O & O & O & O & I \\ -M & I & O & O & O & I \\ O & -M & I & O & O & I \\ O & O & -M & I & O & I \\ O & O & O & -M & I & I \\ O & O & O & O & -M & I \end{bmatrix}$ in which I is a k-by-k identity and M is a k-by-k matrix whose every element is a fraction f . Particular values k := 4 and $f :=$ `1 - eps` $= 1 - 1/2^{52} \approx 0.9999999999999998$ were used to obtain the numerical results plotted below. After `iH = inv(H)` was computed, it was compared with Hi computed by applying `inv(…)` to the result of moving the last k columns of H to its front, and then moving the first k rows of this' inverse to its bottom. Nearly the same results were obtained by using instead `Hi = inv(H')'` or `Hi = flipud(inv(fliplr(H)))` .

The point is that MATLAB seems to dislike inverting H unless its columns are rearranged first.

In the plot below, the numbers of correct sig. bits are plotted against n . (The dimension of H is 4n .) The graph —+— shows the lower bound upon accuracy inferred from the residual of iH . The graph - - * - - shows the accuracy of iH compared with Hi , whose accuracy inferred from its residual is plotted in graph —<>— . Remember, the only difference between computed inverses iH and Hi is that Hi was computed by swapping rows after inverting the result of swapping columns of H . If no rounding errors occurred we would have Hi = iH .



Here is the MATLAB program used to generate `H = ok(n, 1-eps, 4)`:

```
function  H = ok(n, f, k)
%  ok(n, f, k)  is  nk-by-nk  with the form shown here for  n = 6 :
%      [ I                   I ]
%      [-M   I               I ]
%      [    -M   I           I ]  in which  I = eye(k)  and
%      [        -M   I       I ]       and  M = ones(k)*f .
%      [            -M   I   I ]
%      [                -M   I ]
%  If omitted,  k  defaults to  4  and  f  defaults to  1/2 .

if ( nargin < 3 ),  k = 4 ;  end
if ( nargin < 2 ),  f = 0.5 ;  end
u = ones(n-1,1) ;  I = eye(k) ;  M = ones(k)*f ;
J = diag(u, -1) ;  K = eye(n) ;  K(:,n) = [u; 1] ;
H = kron(K, I) - kron(J, M) ;
```

Why does MATLAB so dislike inverting H though its condition number never exceeds 22 ? At n = 29 MATLAB claimed that H was practically singular, but it is nowhere near singular. How well does Iterative Refinement overcome MATLAB's peculiar misbehavior?

How does the order of columns affect MATLAB's treatment of all the foregoing examples? Why?