THE FACTORIZE OBJECT for solving linear systems

Copyright 2011, Timothy A. Davis, University of Florida.

davis@cise.ufl.edu

http://www.cise.ufl.edu/~davis

This is a demonstration of the FACTORIZE object for solving linear systems and least-squares problems, and for computations with the matrix inverse and pseudo-inverse.

Contents

Rule Number One: never multiply by the inverse, inv(A)

Use backslash or a matrix factorization instead (LU, CHOL, or QR).

Rule Number Two: never break Rule Number One

However, the problem with Rule Number One is that it can be hard to figure out which matrix factorization to use and how to use it. Using LU, CHOL, or QR is complicated, particularly if you want the best performance. BACKSLASH (MLDIVIDE) is great, but it can't be reused when solving multiple systems (`x=A\b` and `y=A\c`). Its syntax doesn't match the use of the inverse in mathematical expressions, either.

The goal of the FACTORIZE object is to solve this problem ...

Don't let that INV go past your eyes; to solve that system, FACTORIZE!

How to use BACKSLASH solve $Ax = b$

First, let's create a square matrix A and a right-hand-side b for a linear system $Ax = b$. There are many ways to solve this system. The best way is to use `x=A\b`. The residual r is a vector of what's left over in each equation, and its norm tells you how accurately the system was solved.

```
format compact ;
A = rand (3)
b = rand (3,1)
x = A\b
r = b-A*x ;
norm (r)
```

```
A =
    0.2396    0.5056    0.2656
    0.5514    0.9184    0.6897
    0.4524    0.9631    0.5616
b =
    0.8563
    0.8662
    0.9045
x =
   15.3396
    0.6867
  -11.9222
ans =
   6.4737e-16
```

BACKSLASH versus INV ... let the battle begin

The backslash operation `x=A\b` is mathematically the same as `x=inv(A)*b`. However, backslash is faster and more accurate since it uses a matrix factorization instead of multiplying by the inverse. Even though your linear algebra textbook might write $x = A^{-1}b$ as the solution to the system $Ax = b$, your textbook author never means for you to compute the inverse.

These next statements give the same answer, so what's the big deal?

```
S = inv(A) ;
x = S*b
x = A\b
```

```
x =
   15.3396
    0.6867
  -11.9222
x =
   15.3396
    0.6867
  -11.9222
```

The big deal is that you should care about speed and you should care even more about accuracy. BACKSLASH relies on matrix factorization (LU, CHOL, QR, or other specialized methods). It's faster and more reliable than multiplying by the inverse, particularly for large matrices and sparse matrices. Here's an illustration of how pathetic `inv(A)*b` can be.

```
A = gallery ('frank',16) ; xtrue = ones (16,1) ; b = A*xtrue ;

x = inv(A)*b ; norm (b-A*x)
x = A\b      ; norm (b-A*x)
```

```
ans =
    0.0619
ans =
   1.7764e-15
```

The performance difference between BACKSLASH and INV for even small sparse matrices is striking.

```
load west0479 ;
A = west0479 ;
n = size (A,1)
b = rand (n,1) ;
tic ; x = A\b ; toc
norm (b-A*x)
tic ; x = inv(A)*b ; toc
norm (b-A*x)
```

```
n =
   479
Elapsed time is 0.002238 seconds.
ans =
   2.7491e-10
Elapsed time is 0.070899 seconds.
ans =
   2.4049e-09
```

What if you want to solve multiple systems? Use a matrix factorization. But which one? And how do you use it? Here are some alternatives using LU for the sparse west0479 matrix, but some are faster than others.

```
tic ; [L,U]     = lu(A) ; x1 = U \ (L \ b)         ; t1=toc ; nz1=nnz(L+U);
tic ; [L,U,P]   = lu(A) ; x2 = U \ (L \ P*b)       ; t2=toc ; nz2=nnz(L+U);
tic ; [L,U,P,Q] = lu(A) ; x3 = Q * (U \ (L \ P*b)) ; t3=toc ; nz3=nnz(L+U);

fprintf ('1: nnz(L+U): %5d time: %8.4f resid: %e\n', nz1,t1, norm(b-A*x1));
fprintf ('2: nnz(L+U): %5d time: %8.4f resid: %e\n', nz2,t2, norm(b-A*x2));
fprintf ('3: nnz(L+U): %5d time: %8.4f resid: %e\n', nz3,t3, norm(b-A*x3));
```

```
1: nnz(L+U): 16151 time:   0.0034 resid: 1.426123e-10
2: nnz(L+U): 15826 time:   0.0084 resid: 1.472569e-10
3: nnz(L+U):  3703 time:   0.0035 resid: 1.161913e-10
```

LU and LINSOLVE are fast and accurate but complicated to use

A quick look at "help lu" will scroll off your screen. For full matrices, `[L,U,p] = lu (A,'vector')` is fastest. Then for the forward/backsolves, use LINSOLVE instead of BACKSLASH for even faster performance. But for sparse matrices, use the optional 'Q' output of LU so you get a good fill-reducing ordering. But you can't use 'Q' if the matrix is full. But LINSOLVE doesn't work on sparse matrices.

But ... Ack! That's getting complicated ...

Here's the best way to solve $Ax = b$ and $Ay = c$ when A is full and unsymmetric:

```
n = 1000 ;
A = rand (n) ;
b = rand (n,1) ;
c = rand (n,1) ;
tic ; [L,U,p] = lu (A, 'vector') ; LUtime = toc

tic ; x = U \ (L \ b (p,:)) ;
      y = U \ (L \ c (p,:)) ; toc

tic ; opL = struct ('LT', true) ;
      opU = struct ('UT', true) ;
      x = linsolve (U, linsolve (L, b(p,:), opL), opU) ;
      y = linsolve (U, linsolve (L, c(p,:), opL), opU) ; toc
```

```
LUtime =
    0.1188
Elapsed time is 0.019615 seconds.
Elapsed time is 0.008751 seconds.
```

INV is easy to use, but slow and inaccurate

Oh bother! Using LU and LINSOLVE is too complicated. You just want to solve your system. Let's just compute `inv(A)` and use it twice. Easy to write, but slower and less accurate ...

```
S = inv (A) ;
x = S*b ; norm (b-A*x)
y = S*c ; norm (c-A*y)
```

```
ans =
   1.4614e-11
ans =
   1.2540e-11
```

Sometimes using the inverse seems inevitable. For example, your textbook might show the Schur complement formula as $S = A - BD^{-1}C$. This can be done without `inv(D)` in one of two ways: SLASH or BACKSLASH (MRDIVIDE or MLDIVIDE to be precise).

`inv(A)*B` and `A\B` are mathematically equivalent, as are `B*inv(A)` and `B/A`, so these three methods give the same results (ignoring computational errors, which are worse for `inv(D)`). Only the first equation looks like the equation in your textbook, however.

```
A = rand (200) ; B = rand (200) ; C = rand (200) ; D = rand (200) ;

tic ; S1 = A - B*inv(D)*C ; toc ;
tic ; S2 = A - B*(D\C) ;    toc ;
tic ; S3 = A - (B/D)*C ;    toc ;
```

```
Elapsed time is 0.011946 seconds.
Elapsed time is 0.009852 seconds.
Elapsed time is 0.009212 seconds.
```

So the winner is ... nobody

BACKSLASH: mostly simple to use (except remember that Schur complement formula?). Fast and accurate ... but slow if you want to solve two linear systems with the same matrix A.

LU, QR, CHOL: fast and accurate. Awful syntax to use. Drag out your linear algebra textbook if you want to use these in MATLAB. Whenever I use them I have to derive them from scratch, even though I **wrote** most of the sparse factorizations used in MATLAB!

INV: slow and inaccurate. Wins big on ease-of-use, though, since it's a direct plug-in for all your nice mathematical formulas.

No method is best on all three criterion: speed, accuracy, and ease of use.

Is there a solution? Yes ... keeping reading ...

The FACTORIZE object to the rescue

The FACTORIZE method is just as easy to use as INV, but just as fast and accurate as BACKSLASH, LU, QR, CHOL, and LINSOLVE.

`F = factorize(A)` computes the factorization of A and returns it as an object that you can reuse to solve a linear system with `x=F\b`. It picks LU, QR, or Cholesky for you, just like BACKSLASH.

`S = inverse(A)` is simpler yet. It does NOT compute `inv(A)`, but factorizes A. When multiplying `S*b`, it doesn't mulitply by the inverse, but uses the correct forward/backsolve equations to solve the linear system.

```
n = 1000 ;
A = rand (n) ;
b = rand (n,1) ;
c = rand (n,1) ;

tic ;                        x = A\b ; y = A\c ; toc
tic ; S = inv(A) ;           x = S*b ; y = S*c ; toc
tic ; F = factorize(A) ;     x = F\b ; y = F\c ; toc
tic ; S = inverse(A) ;       x = S*b ; y = S*c ; toc
```

```
Elapsed time is 0.312721 seconds.
Elapsed time is 0.404185 seconds.
Elapsed time is 0.156734 seconds.
Elapsed time is 0.163260 seconds.
```

Least-squares problems

Here are some different methods for solving a least-squares problem when your system is over-determined. The last two methods are the same.

```
A = rand (1000,200) ;
b = rand (1000,1) ;

tic ; x = A\b            ; toc, norm (A'*A*x-A'*b)
tic ; x = pinv(A)*b      ; toc, norm (A'*A*x-A'*b)
tic ; x = inverse(A)*b   ; toc, norm (A'*A*x-A'*b)
tic ; x = factorize(A)\b ; toc, norm (A'*A*x-A'*b)
```

```
Elapsed time is 0.075459 seconds.
ans =
   2.2721e-12
Elapsed time is 0.110536 seconds.
ans =
   1.7740e-12
Elapsed time is 0.044961 seconds.
ans =
   3.2002e-12
Elapsed time is 0.044590 seconds.
ans =
   3.2002e-12
```

FACTORIZE is better than BACKSLASH because you can reuse the factorization for different right-hand-sides. For full-rank matrices, it's better than PINV because it's faster (and PINV fails for sparse matrices).

```
A = rand (1000,200) ;
b = rand (1000,1) ;
c = rand (1000,1) ;

tic ;                  ; x = A\b ; y = A\c ; toc
tic ; S = pinv(A)      ; x = S*b ; y = S*c ; toc
tic ; S = inverse(A)   ; x = S*b ; y = S*c ; toc
tic ; F = factorize(A) ; x = F\b ; y = F\c ; toc
```

```
Elapsed time is 0.134460 seconds.
Elapsed time is 0.113886 seconds.
Elapsed time is 0.047543 seconds.
Elapsed time is 0.046980 seconds.
```

Underdetermined systems

The under-determined system $Ax = b$ where A has more columns than rows has many solutions. `x=A\b` finds a basic solution (some of the entries in x are zero). `pinv(A)*b` finds a minimum 2-norm solution, but it's slow. QR factorization will do the same if A has full rank. That's what the `factorize(A)` and `inverse(A)` methods do.

```
A = rand (200,1000) ;
b = rand (200,1) ;

tic ; x = A\b            ; toc, norm (x)
tic ; x = pinv(A)*b      ; toc, norm (x)
tic ; x = inverse(A)*b   ; toc, norm (x)
tic ; x = factorize(A)\b ; toc, norm (x)
```

```
Elapsed time is 0.094211 seconds.
ans =
    2.7463
Elapsed time is 0.114486 seconds.
ans =
    0.5211
Elapsed time is 0.046506 seconds.
ans =
    0.5211
Elapsed time is 0.046187 seconds.
ans =
    0.5211
```

Computing selected entries in the inverse or pseudo-inverse

If you want just a few entries from the inverse, it's still better to formulate the problem as a system of linear equations and use a matrix factorization instead of computing `inv(A)`. The FACTORIZE object does this for you, by overloading the subsref operator.

```
A = rand (1000) ;

tic ; S = inv (A)     ; S (2:3,4), toc
tic ; S = inverse (A) ; S (2:3,4), toc
```

```
ans =
   -0.0790
    0.0515
```

```
Elapsed time is 0.396958 seconds.
ans =
   -0.0790
    0.0515
Elapsed time is 0.158975 seconds.
```

Computing the entire inverse or pseudo-inverse

Rarely, and I mean RARELY, you really do need the inverse. More frequently what you want is the pseudo-inverse. You can force a factorization to become a plain matrix by converting it to double. Note that `inverse(A)` only handles full-rank matrices (either dense or sparse), whereas `pinv(A)` works for all dense matrices (not sparse).

The explicit need for `inv(A)` (or `S=A\eye(n)`, which is the same thing) is RARE. If you ever find yourself multiplying by the inverse, then you know one thing for sure. You know with certainty that you don't know what you're doing.

```
A = rand (500) ;
tic ; S1 = inv (A) ;            ; toc
tic ; S2 = double (inverse (A)) ; toc
norm (S1-S2)

A = rand (500,400) ;
tic ; S1 = pinv (A)             ; toc
tic ; S2 = double (inverse (A)) ; toc
norm (S1-S2)
```

```
Elapsed time is 0.059431 seconds.
Elapsed time is 0.076613 seconds.
ans =
   1.2206e-12
Elapsed time is 0.260396 seconds.
Elapsed time is 0.097643 seconds.
ans =
   1.4181e-14
```

Update/downdate of a dense Cholesky factorization

Wilkinson considered the update/downdate of a matrix factorization to be a key problem in computational linear algebra. The idea is that you first factorize a matrix. Next, make a low-rank change to A, and patch up (or down...) the factorization so that it becomes the factorization of the new matrix. In MATLAB, this only works for dense symmetric positive definite matrices, via cholupdate. This is much faster than computing the new factorization from scratch.

```
n = 1000 ;
A = rand (n) ;
A = A*A' + n*eye (n) ;
w = rand (n,1) ; t = rand (n,1) ; b = rand (n,1) ;
F = factorize (A) ;

tic ; F = cholupdate (F,w,'+') ; x = F\b ; toc
tic ; y = (A+w*w')\b ;      toc
norm (x-y)

tic ; F = cholupdate (F,t,'-') ; x = F\b ; toc
tic ; y = (A+w*w'-t*t')\b ; toc
norm (x-y)
```

```
Elapsed time is 0.030108 seconds.
Elapsed time is 0.094033 seconds.
ans =
   3.2704e-17
Elapsed time is 0.030737 seconds.
Elapsed time is 0.099725 seconds.
ans =
   3.3827e-17
```

Caveat Executor

One caveat: If you have a large number of very small systems to solve, the object-oriented overhead of creating and using an object can dominate the run time, at least in MATLAB R2011a. For this case, if you want the best performance, stick with BACKSLASH, or LU and LINSOLVE (just extract the appropriate formulas from the M-files in the FACTORIZE package).

Hopefully the object-oriented overhead will drop in future versions of MATLAB, and you can ignore this caveat.

```
A = rand (10) ; b = rand (10,1) ; F = factorize (A) ;

tic ; for k = 1:10000, x = F\b ; end ; toc

tic ; for k = 1:10000, x = A\b ; end ; toc

[L,U,p] = lu (A, 'vector') ;
```

```
opL = struct ('LT', true) ;
opU = struct ('UT', true) ;
tic ;
for k = 1:10000
    x = linsolve (U, linsolve (L, b(p,:), opL), opU) ;
end
toc
```

```
Elapsed time is 1.337981 seconds.
Elapsed time is 0.205922 seconds.
Elapsed time is 0.097137 seconds.
```

Summary

So ... don't use INV, and don't worry about how to use LU, CHOL, or QR factorization. Just install the FACTORIZE package, and you're on your way. Assuming you are now in the `Factorize/` directory, cut-and-paste these commands into your command window:

```
 addpath (pwd)
 savepath
```

And remember ...

Don't let that INV go past your eyes; to solve that system, FACTORIZE!