lecture_11

February 22, 2017

```
In [18]: %plot --format svg

In [19]: setdefaults
```

1 LU Decomposition

1.0.1 efficient storage of matrices for solutions

Considering the same solution set:

$y = Ax$

Assume that we can perform Gauss elimination and achieve this formula:

$Ux = d$

Where, U is an upper triangular matrix that we derived from Gauss elimination and d is the set of dependent variables after Gauss elimination.

Assume there is a lower triangular matrix, L, with ones on the diagonal and same dimensions of U and the following is true:

$L(Ux - d) = Ax - y = 0$

Now, $Ax = LUx$, so $A = LU$, and $y = Ld$.

$2x_1 + x_2 = 1$

$x_1 + 3x_2 = 1$

$$\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

f21=0.5

A(2,1)=1-1 = 0

A(2,2)=3-0.5=2.5

y(2)=1-0.5=0.5

$$L(Ux - d) = \begin{bmatrix} 1 & 0 \\ 0.5 & 1 \end{bmatrix} \left(\begin{bmatrix} 2 & 1 \\ 0 & 2.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} \right) = 0$$

```
In [3]: A=[2,1;1,3]
        L=[1,0;0.5,1]
        U=[2,1;0,2.5]
        L*U

        d=[1;0.5]
        y=L*d
```

```
A =

   2   1
   1   3

L =

   1.00000   0.00000
   0.50000   1.00000

U =

   2.00000   1.00000
   0.00000   2.50000

ans =

   2   1
   1   3

d =

   1.00000
   0.50000

y =

   1
   1
```

1.1 Pivoting for LU factorization

LU factorization uses the same method as Gauss elimination so it is also necessary to perform partial pivoting when creating the lower and upper triangular matrices.

Matlab and Octave use pivoting in the command

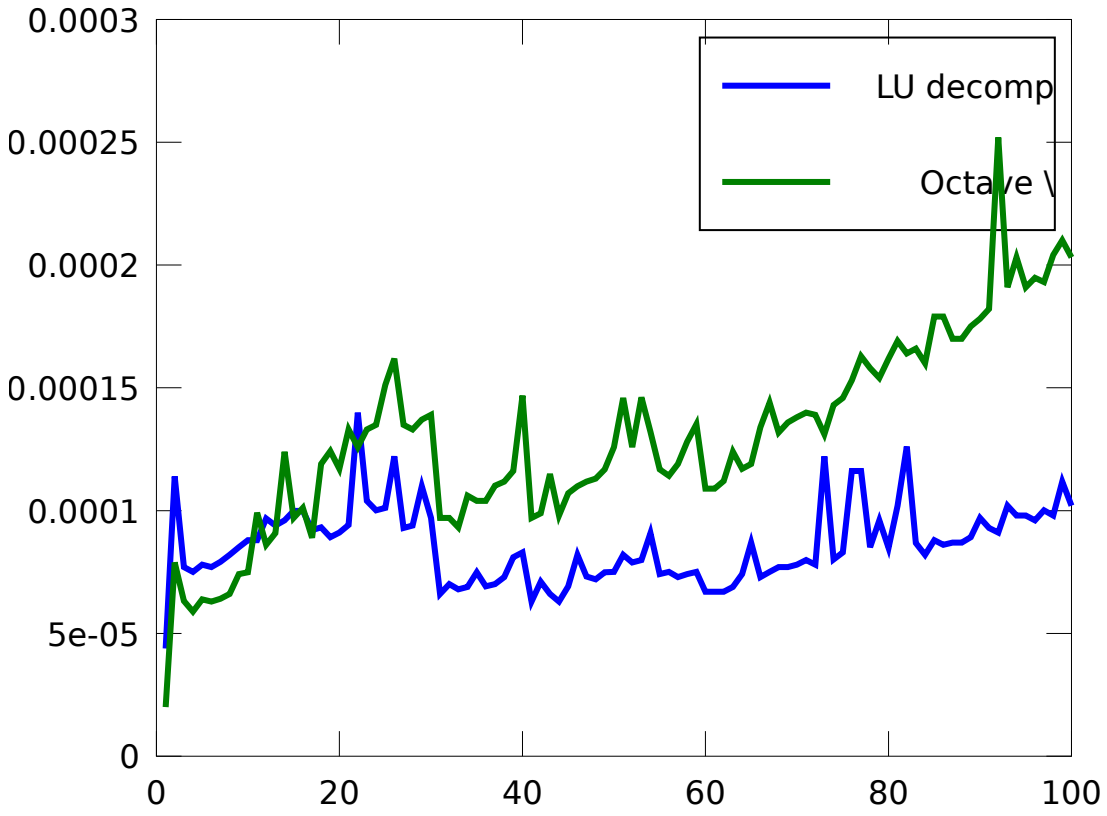
`[L,U,P]=lu(A)`

```
In [4]: help lu

'lu' is a built-in function from the file libinterp/corefcn/lu.cc

 -- Built-in Function: [L, U] = lu (A)
 -- Built-in Function: [L, U, P] = lu (A)
 -- Built-in Function: [L, U, P, Q] = lu (S)
 -- Built-in Function: [L, U, P, Q, R] = lu (S)
 -- Built-in Function: [...] = lu (S, THRES)
```

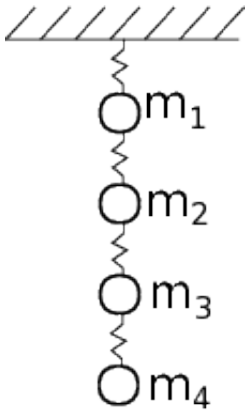
```
-- Built-in Function: Y = lu (...)
-- Built-in Function: [...] = lu (..., "vector")
    Compute the LU decomposition of A.

    If A is full subroutines from LAPACK are used and if A is sparse
    then UMFPACK is used.

    The result is returned in a permuted form, according to the
    optional return value P.  For example, given the matrix 'a = [1, 2;
    3, 4]',

         [l, u, p] = lu (A)

    returns

         l =

           1.00000  0.00000
           0.33333  1.00000

         u =

           3.00000  4.00000
           0.00000  0.66667

         p =

           0  1
           1  0

    The matrix is not required to be square.

    When called with two or three output arguments and a spare input
    matrix, 'lu' does not attempt to perform sparsity preserving column
    permutations.  Called with a fourth output argument, the sparsity
    preserving column transformation Q is returned, such that 'P * A *
    Q = L * U'.

    Called with a fifth output argument and a sparse input matrix, 'lu'
    attempts to use a scaling factor R on the input matrix such that 'P
    * (R \ A) * Q = L * U'.  This typically leads to a sparser and more
    stable factorization.

    An additional input argument THRES, that defines the pivoting
    threshold can be given.  THRES can be a scalar, in which case it
    defines the UMFPACK pivoting tolerance for both symmetric and
    unsymmetric cases.  If THRES is a 2-element vector, then the first
    element defines the pivoting tolerance for the unsymmetric UMFPACK
```

```
     pivoting strategy and the second for the symmetric strategy.  By
     default, the values defined by 'spparms' are used ([0.1, 0.001]).

     Given the string argument "vector", 'lu' returns the values of P
     and Q as vector values, such that for full matrix, 'A (P,:) = L *
     U', and 'R(P,:) * A (:, Q) = L * U'.

     With two output arguments, returns the permuted forms of the upper
     and lower triangular matrices, such that 'A = L * U'.  With one
     output argument Y, then the matrix returned by the LAPACK routines
     is returned.  If the input matrix is sparse then the matrix L is
     embedded into U to give a return value similar to the full case.
     For both full and sparse matrices, 'lu' loses the permutation
     information.

     See also: luupdate, ilu, chol, hess, qr, qz, schur, svd.

Additional help for built-in functions and operators is
available in the online version of the manual.  Use the command
'doc <topic>' to search the manual index.

Help and information about Octave is also available on the WWW
at http://www.octave.org and via the help@octave.org
mailing list.
```

In [22]:
```
% time LU solution vs backslash
t_lu=zeros(100,1);
t_bs=zeros(100,1);
for N=1:100
    A=rand(N,N);
    y=rand(N,1);
    [L,U,P]=lu(A);

    tic; d=L\y; x=U\d; t_lu(N)=toc;

    tic; x=A\y; t_bs(N)=toc;
end
plot([1:100],t_lu,[1:100],t_bs)
legend('LU decomp','Octave \\')
```



Consider the problem again from the intro to Linear Algebra, 4 masses are connected in series to 4 springs with K=10 N/m. What are the final positions of the masses?



Springs-masses

The masses haves the following amounts, 1, 2, 3, and 4 kg for masses 1-4. Using a FBD for each mass:

$m_1 g + k(x_2 - x_1) - kx_1 = 0$
$m_2 g + k(x_3 - x_2) - k(x_2 - x_1) = 0$
$m_3 g + k(x_4 - x_3) - k(x_3 - x_2) = 0$

$m_4 g - k(x_4 - x_3) = 0$

in matrix form:

$$\begin{bmatrix} 2k & -k & 0 & 0 \\ -k & 2k & -k & 0 \\ 0 & -k & 2k & -k \\ 0 & 0 & -k & k \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} m_1 g \\ m_2 g \\ m_3 g \\ m_4 g \end{bmatrix}$$

```
In [24]: k=10; % N/m
         m1=1; % kg
         m2=2;
         m3=3;
         m4=4;
         g=9.81; % m/s^2
         K=[2*k -k 0 0; -k 2*k -k 0; 0 -k 2*k -k; 0 0 -k k]
         y=[m1*g;m2*g;m3*g;m4*g]
```

```
K =

   20  -10    0    0
  -10   20  -10    0
    0  -10   20  -10
    0    0  -10   10

y =

    9.8100
   19.6200
   29.4300
   39.2400
```

This matrix, K, is symmetric.

`K(i,j)==K(j,i)`

Now we can use,

1.2 Cholesky Factorization

We can decompose the matrix, K into two matrices, U and U^T, where

$K = U^T U$

each of the components of U can be calculated with the following equations:

$u_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2}$

$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj}}{u_{ii}}$

so for K

```
In [25]: K
```

```
K =

   20  -10    0    0
  -10   20  -10    0
    0  -10   20  -10
    0    0  -10   10
```

```
In [26]: u11=sqrt(K(1,1))
         u12=(K(1,2))/u11
         u13=(K(1,3))/u11
         u14=(K(1,4))/u11
         u22=sqrt(K(2,2)-u12^2)
         u23=(K(2,3)-u12*u13)/u22
         u24=(K(2,4)-u12*u14)/u22
         u33=sqrt(K(3,3)-u13^2-u23^2)
         u34=(K(3,4)-u13*u14-u23*u24)/u33
         u44=sqrt(K(4,4)-u14^2-u24^2-u34^2)
         U=[u11,u12,u13,u14;0,u22,u23,u24;0,0,u33,u34;0,0,0,u44]

u11 =  4.4721
u12 = -2.2361
u13 = 0
u14 = 0
u22 =  3.8730
u23 = -2.5820
u24 = 0
u33 =  3.6515
u34 = -2.7386
u44 =  1.5811
U =

   4.47214  -2.23607   0.00000   0.00000
   0.00000   3.87298  -2.58199   0.00000
   0.00000   0.00000   3.65148  -2.73861
   0.00000   0.00000   0.00000   1.58114
```

```
In [27]: U'*U

ans =

   20.00000  -10.00000    0.00000    0.00000
  -10.00000   20.00000  -10.00000    0.00000
    0.00000  -10.00000   20.00000  -10.00000
    0.00000    0.00000  -10.00000   10.00000
```

In [37]:
```
% time solution for Cholesky vs backslash
t_chol=zeros(1000,1);
t_bs=zeros(1000,1);
for i=1:1000
    tic; d=U'*y; x=U\d; t_chol(i)=toc;
    tic; x=K\y; t_bs(i)=toc;
end
fprintf('average time spent for Cholesky factored solution = %e+/-%e',mean(t_chol),std(

fprintf('average time spent for backslash solution         = %e+/-%e',mean(t_bs),std(t_
```

```
average time spent for Cholesky factored solution = 1.623154e-05+/-1.166726e-05
average time spent for backslash solution         = 1.675844e-05+/-1.187234e-05
```