ENM 502 : Homework Assignment 1

Submitted by :Ramalingam Kailasham, kailr@seas.upenn.edu

Objective : To test and compare the performance of user-written LU Decomposition code against MATLAB's intrinsic **lu** command.

Procedure

1. The given pseudo-code was coded in MATLAB.
2. The code was found to run after minor tweaks and debugging.
3. The code was tested on several systems of Ax = b to confirm that the solution obtained matched the one obtained by direct solving, i.e x = inv(A)*b.
4. Certain minor subroutines were coded to answer the questions asked in tasks (a) to (d). All the codes used for obtaining the answer have been included in the Appendix section. They carry detailed comments regarding their function and the meanings of the various variables used therein.
5. The power-law fitting in figures 1-4 was carried out using MATLAB's Curve Fitting Toolbox.

The following table shows the variation of decomposition time for the two decomposition codes; for different matrix types.

Time taken for Hilbert Matrices (in seconds) by user's code	**Time taken for Random Matrices (in seconds) by user's code**	**Size of Matrix, n**	**Time taken for Random Matrices (in seconds) by MATLAB code**	**Time taken for Hilbert Matrices (in seconds) by MATLAB code**
0.0016	0.0015	50	3.984×10^{-5}	3.713×10^{-5}
0.0131	0.0115	100	1.506×10^{-4}	1.539×10^{-4}
0.1559	0.1713	250	7.048×10^{-4}	5.880×10^{-4}
1.2670	1.2574	500	0.0024	0.0025

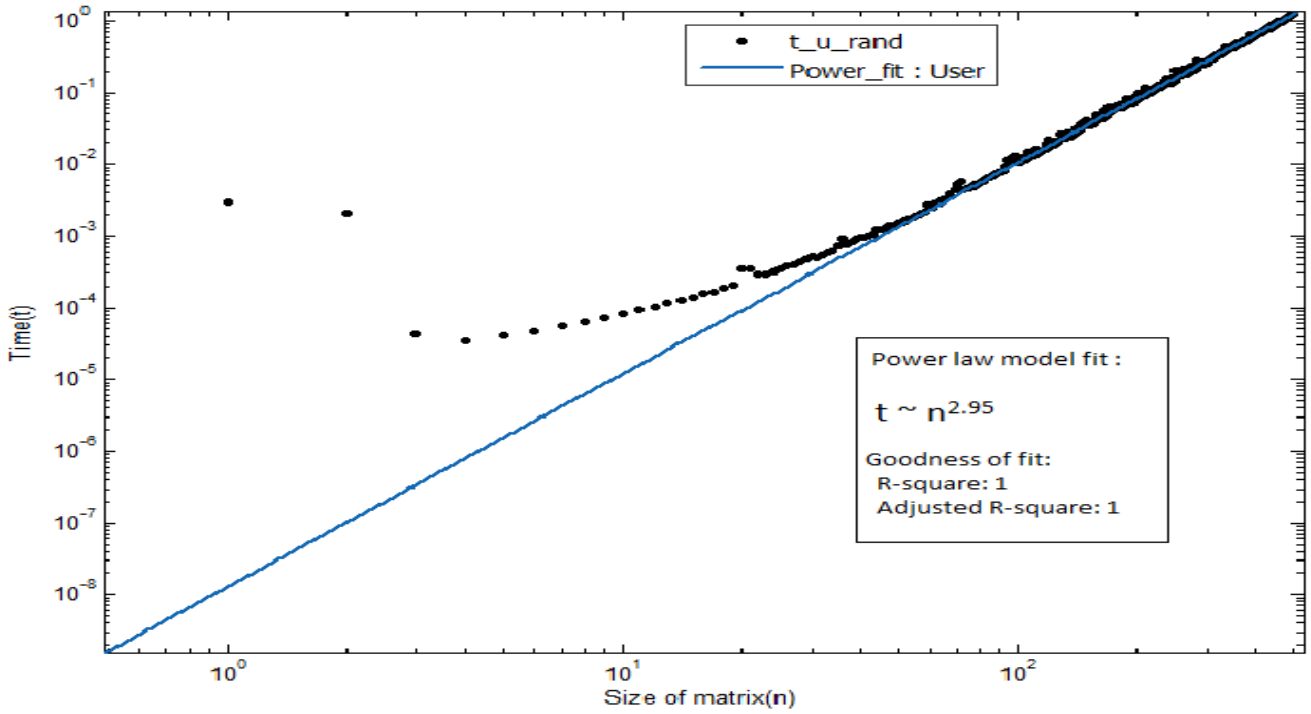

Figure 1 : Variation of time with size of random matrix for user-written LU decomposition code

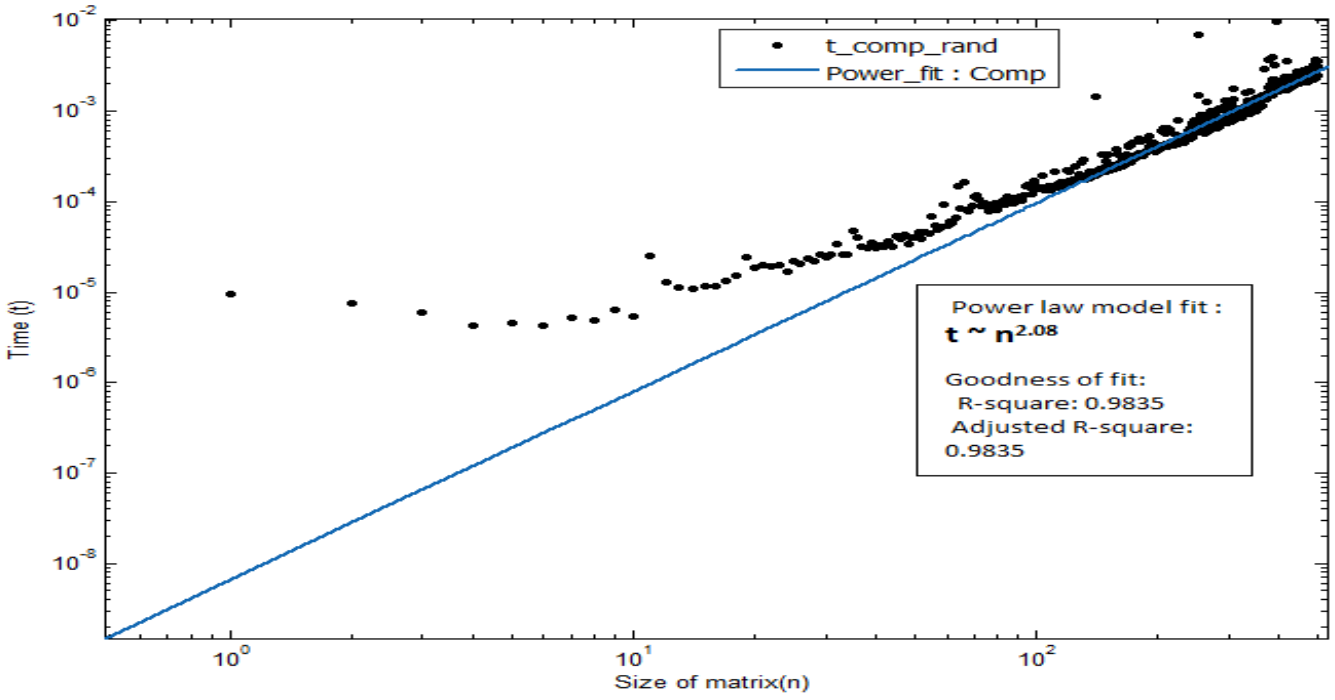

Figure 2 : Variation of time with size of random matrix for MATLAB's LU code

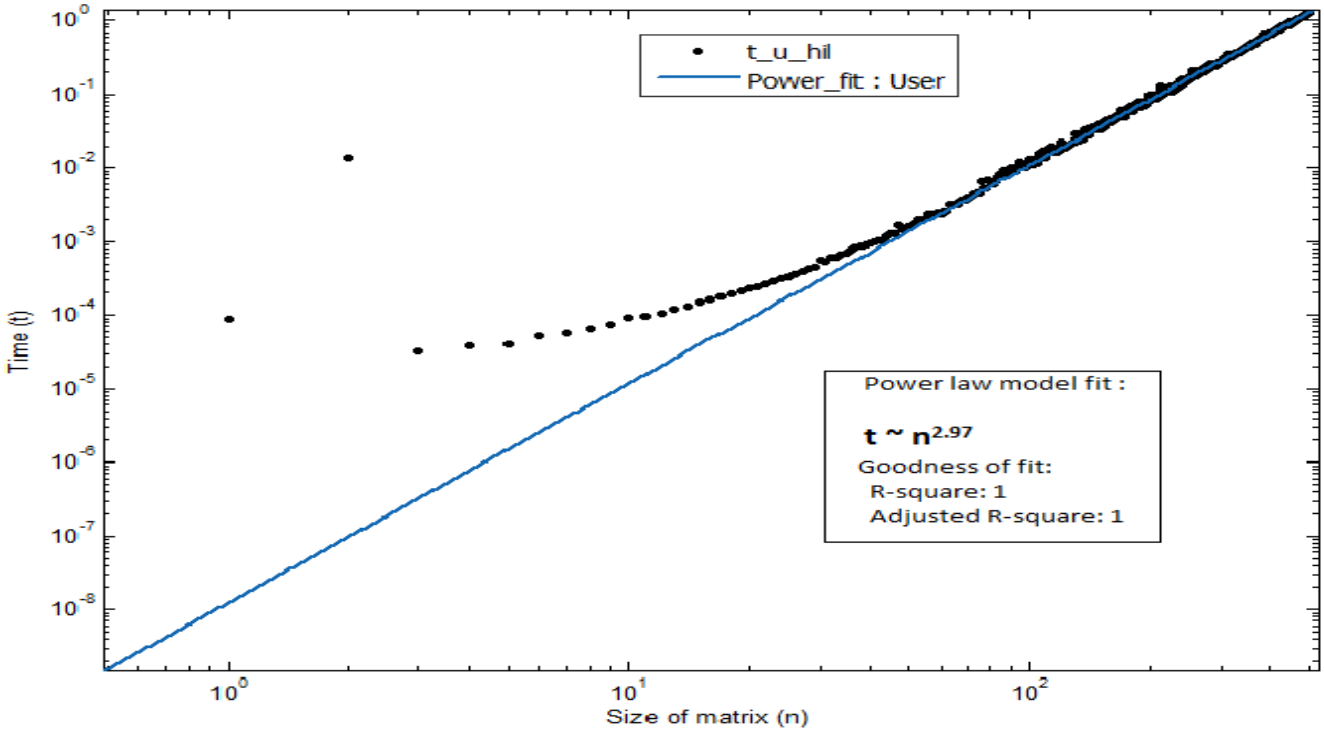

Figure 3 : Variation of time with size of Hilbert matrix for user-written LU decomposition code

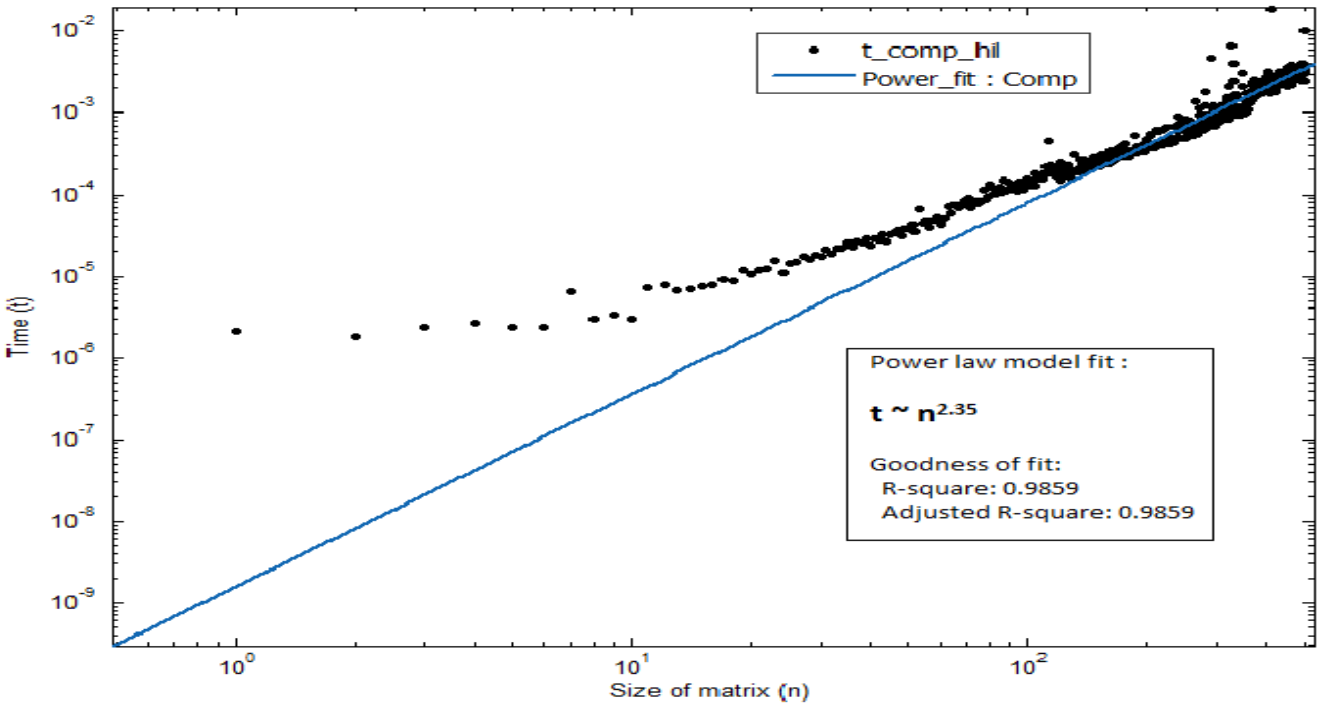

Figure 4 : Variation of time with size of Hilbert matrix for MATLAB's LU code

Discussion of plots and comments

Figure 1 : A power-law dependence of 2.95 was obtained between the decomposition time and size of matrix. This is in close agreement with the expected n^3 dependence. The n^3 doesn't strictly account for the cost of pivoting but we see from the plots that pivoting does not contribute significantly to the cost of LU decomposition.

Figure 2: MATLAB's intrinsic code is cheaper (in computational terms) than the user-written code with a power-law dependence of 2.08. It is expected that MATLAB's code is optimized to handle several different types of matrices and figure out the best possible method. For example : if the random matrix turns out to be positive definite AND symmetric, the computer might be following the Cholesky algorithm, which is cheaper than the Doolittle algorithm.

Figure 3: For Hilbert matrices, a power-law dependence of 2.97 was obtained. This suggests that Hilbert matrices are slightly more difficult to decompose than random matrices. As noted later in this paper, Hilbert matrices do not need pivoting. However, their condition number increases very quickly with size and this could be the reason for the higher power.

Figure 4: MATLAB's intrinsic code takes more time to factor Hilbert matrices as compared to random matrices. A power-law dependence of 2.35 is obtained. This is still cheaper compared to the user-written LU code. The reason for this has been noted above, in the discussion of Figure 2.

The points that do not fall on the straight line are the "outliers". They denote the background noise in the system (also called latency) when the time taken to decompose the matrix is nearly of the same order as the time taken for program overheads (calling functions, passing arguments, etc)

(b) The code is not reliable with pivoting turned off. However, Hilbert matrices are properly conditioned in the sense that they don't need pivoting. This is because a_{ii} is the largest element in the i^{th} row.

So the code works reliably well for higher order Hilbert matrices with pivoting turned off.

The same cannot be said about random matrices and the code cannot be used, in general, to decompose random matrices without pivoting.

(c) The condition number of a Hilbert matrix increases with n, much more rapidly than that of a random matrix.

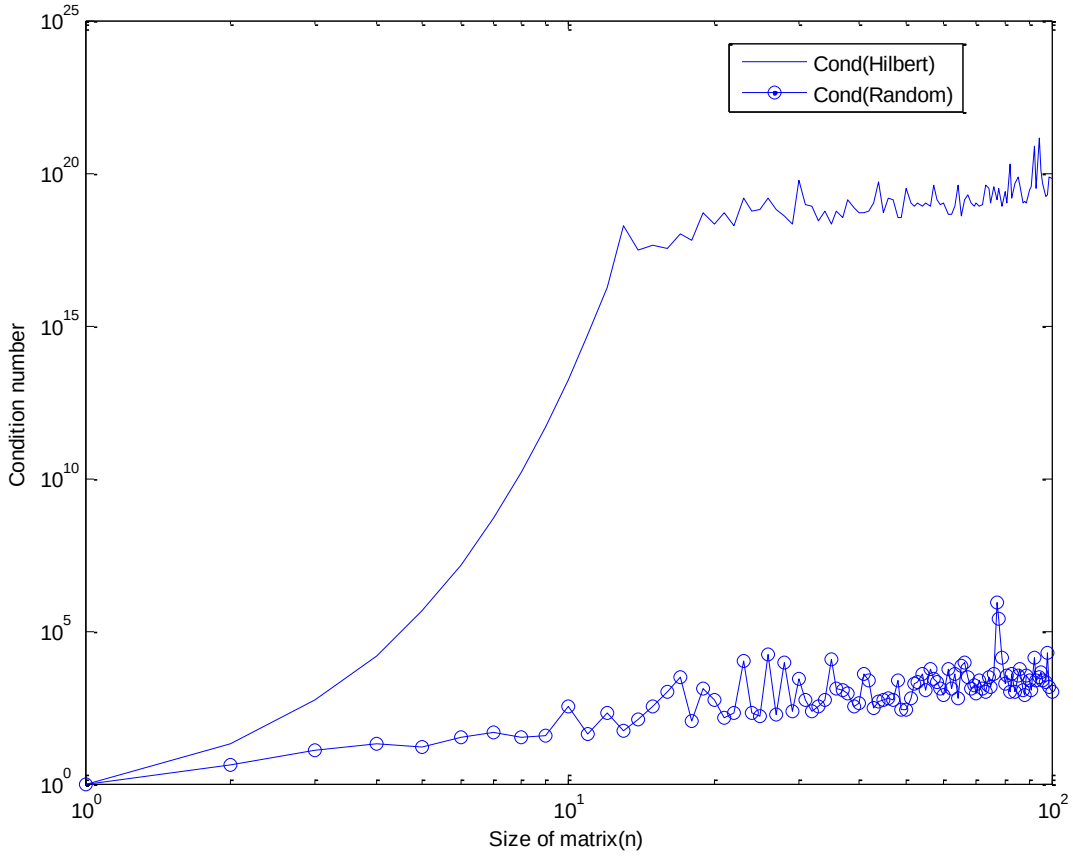

Figure 5 : Variation of condition number with matrix size for different matrix types

Using the default setting of double precision, MATLAB can factor matrices with condition numbers as high as 10^{10} with accuracy. Higher condition number matrices can still be factored but the answers won't be necessarily accurate (limited by machine precision).

(d) A random matrix A[25 x 25] and a random vector b[25] was set up. The performance of the solver and the behavior of the solution 'x' was analysed with respect to increasing n. Here 'n' refers to the power to which A is raised in the equation $A^n x = b$.

As seen from part (a), the solver time increases only with the size of the matrix. Since A^n is the same size as A, the solver time doesn't vary appreciably with n.

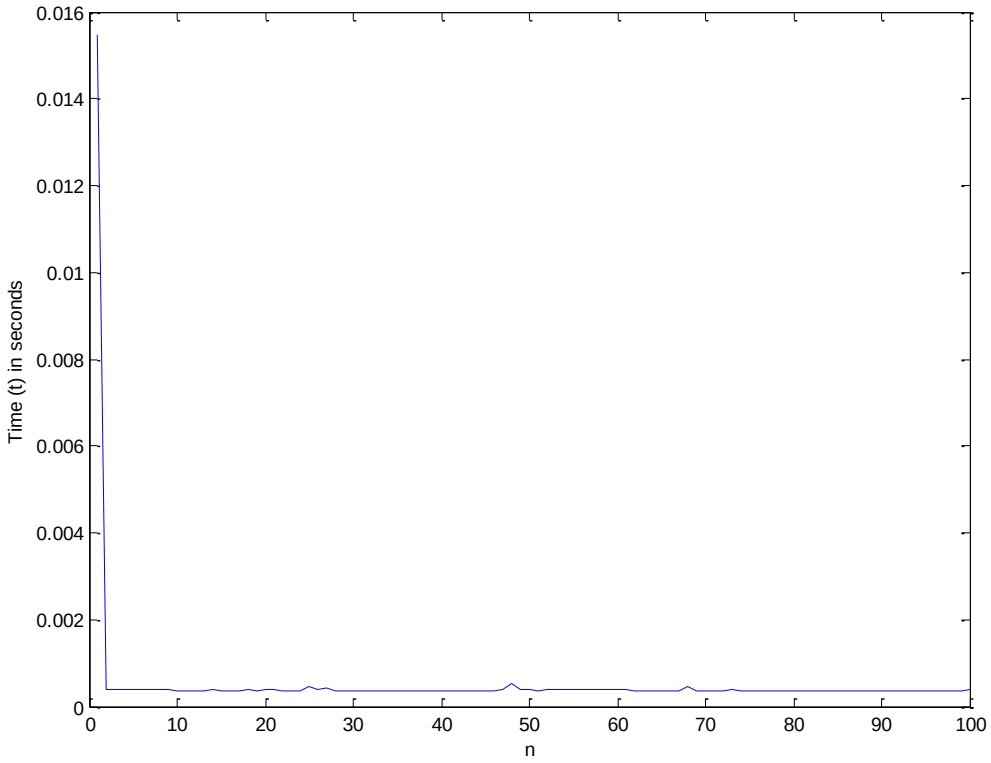

Figure 6 : Variation of solver time with n for the system $A^n x = b$

However, with increasing 'n' and constant b, the value of x has to decrease so as to maintain the equality $A^n x = b$. This is clear from the following table that shows the value of x for several n values.

n	**Order of x values**
1	10^{0}
10	10^{-5}
50	10^{-93}
100	10^{-203}

APPENDIX

The driver routine – Ludecomp.m

```
%%%%%%%%%%%%%%%%%%%%%%% INPUT PARAMETERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%A - Co-efficient array i.e the A in Ax = b                                    %
%b - RHS of the equation Ax = b                                               %
%n - dimension of A                                                           %
%tol - Tolerance value as decided by the user. 10e-6 will suffice for         %
%      normal operations and commonly encountered matrices (lower tol         %
%      value(10e-30) needed for Herbert matrices.                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%% OUTPUT PARAMETERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%sol - A [1 x n] array that contains the solution x of Ax=b                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [sol] = Ludecomp(A,b,n,tol)
O = zeros(1,n);   %Array of size 1 x n that keeps track of row interchanges
S = zeros(1,n);   %Array of size 1 x n that keeps track of the largest
element in each row
er = 0;
[er,O,S,A] = Decompose(A,n,tol,O,S);
if er~=-1
    sol = Substitute(A,O,n,b);
else disp('Tolerance limit exceeded');
     sol = zeros(1,n);
end
end
```

Decompose.m

```
%%%%%%%%%%%%%%%%%%%%%%% INPUT PARAMETERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%A - Co-efficient array i.e the A in Ax = b                                    %
%O - Array of size [1 x n] that keeps track of row interchanges               %
%S - Array of size [1 x n] that keeps track of the largest element in each%
%    row                                                                      %
%n - dimension of A                                                           %
%tol - tolerance value. As entered by the user.                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%% OUTPUT PARAMETERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%flag - set equal to 0 if tolerance value is not violated. -1 otherwise   %
%Order- Contains the same information as in O.                                %
%Si   - Contains the same information as in S.                                %
%A_Changed - Matrix A after it has been LU Decomposed                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [flag, Order, Si, A_changed] = Decompose(A,n,tol,O,S)
flag=0;
for i=1:n
```

```
    O(i) = i;
    S(i) = abs(A(i,1));
    for j=2:n
        if abs(A(i,j)) > S(i)
            S(i) = abs(A(i,j)); %stores the largest element in each row
        end
    end
end
for k=1:(n-1)
    O = Pivot(A,O,S,n,k); % Keeps track of row-interchanges during pivoting
    if abs(A(O(k),k)/S(O(k))) < tol
       flag = -1;
       disp(A(O(k),k)/S(O(k)));
       Order = O;
       Si = S;
       A_changed = A;
       return;
    end
    for i= (k+1) : n
        factor = A(O(i),k)/A(O(k),k);
        A(O(i),k) = factor;
        for j = (k+1) : n
            A(O(i),j)=A(O(i),j) - factor*A(O(k),j);
        end
    end
end
if abs(A(O(k),k)/S(O(k))) < tol
    flag = -1;
    disp(A(O(k),k)/S(O(k)));
else flag = 0;
end
Order = O;
Si = S;
A_changed = A;
end
```

Pivot.m

```
%%%%%%%%%%%%%%%%%%%%%%% INPUT PARAMETERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%A - Co-efficient array i.e the A in Ax = b                                    %
%O - Array of size [1 x n] that keeps track of row interchanges               %
%S - Array of size [1 x n] that keeps track of the largest element in each%
%    row                                                                      %
%n - dimension of A                                                           %
%k - passed in as a parameter. Keeps track of the row which is being          %
%     checked for pivoting.                                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%% OUTPUT PARAMETERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% sol - A [1 x n] array that contains information about the order of rows %
%        after pivoting.                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [sol] = Pivot(A,O,S,n,k)

p=k;
```

```
big = abs(A(O(k),k)/S(O(k))); % each row element scaled by didviding by the
                              %  largest element in the row.
for ii = (k+1) : n
    dummy = abs(A(O(ii),k)/S(O(ii)));
    if dummy > big   %if the scaled value in a subsequent row is bigger than
        big = dummy; % the scaled value of the previous row, then pivot!
        p = ii;
    end
end
dummy = O(p);
O(p) = O(k);
O(k) = dummy;
sol = O;
end
```

Substitute.m

```
%%%%%%%%%%%%%%%%%%%%%% INPUT PARAMETERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%A - The LU decomposed version of the A in Ax = b                           %
%O - Array of size [1 x n] that contains info about row interchanges        %
%n - dimension of A                                                        %
%b - RHS of the equation Ax = b                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%% OUTPUT PARAMETERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%sol - A [1 x n] array that contains the solution x of Ax=b                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [sol] = Substitute (A,O,n,b)
x = zeros(1,n);
%This part of the code takes care of forward substitution
for i =2:n
    sum = b(O(i));
    for j = 1 : (i-1)
        sum = sum - A(O(i),j)*b(O(j));
    end
    b(O(i)) = sum;
end
%End of forward substitution routine
x(n) = b(O(n))/A(O(n),n);

%This part does the backward substitution
for i = (n-1) : -1 : 1
    sum = 0;
    for j = (i+1) : n
        sum = sum + A(O(i),j)*x(j);
    end
    x(i) = (b(O(i)) - sum)/A(O(i),i);
end
%End of backward substitution
sol = x;

end
```

Test_Random.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to test the performance of the inbuilt and user-defined        %
% LU codes on Random matrices of different sizes.                          %
% Takes the maximum size of matrix that needs to be tested as an argument %
% and returns the following :                                              %
%
% C_No      - Condition number of the matrix being solved                  %
% time_user - Time taken by the user-written code to solve the system Rx=b%
% time_comp - Time taken by the inbuilt LU function to solve the same
%             system Rx = b                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [C_No, time_user, time_comp] = Test_Random(n)
time_comp = zeros(1,n);
time_user = zeros(1,n);
S = zeros(1,n);
O = zeros(1,n);
flag = 0;
for i=1:n
    flag = 0;
    R = randi((4*i),i);
    b= [1:i]';
    o = [1:i];
    %tic
    tic                 %begin stopwatch to track computer's LU code
    B = lu(R);
    time_comp(i) = toc; %end stopwatch to track computer's LU code

    tic                 %begin stopwatch to track Kailash's LU code
    [flag,O,S,A] = Decompose(R,i,10e-50,O,S);
    time_user(i) = toc; %end stopwatch to track Kailash's LU code
    C_No(i)=cond(R);

end
end
```

Test_Hilbert.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to test the performance of the inbuilt and user-defined        %
% LU codes on Hilbert matrices of different sizes.                         %
% Takes the maximum size of matrix that needs to be tested as an argument %
% and returns the following :                                              %
%
% C_No      - Condition number of the matrix being solved                  %
% time_user - Time taken by the user-written code to solve the system Hx=b%
% time_comp - Time taken by the inbuilt LU function to solve the same
%             system Hx = b                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [C_No, time_user, time_comp] = Test_Hilbert(n)
time_comp = zeros(1,n);
time_user = zeros(1,n);
S = zeros(1,n);
O = zeros(1,n);
for i=1:n
    H = hilbert(i);
    b= [1:i]';
    [L,U,P]=lu(H);
    o = [1:i];
    tic                 %begin stopwatch to track computer's LU code
    B = lu(H);
    time_comp(i) = toc; %end stopwatch to track computer's LU code

    tic                 %begin stopwatch to track Kailash's LU code
    [flag,O,S,A] = Decompose(H,i,10e-50,O,S);
    time_user(i) = toc; %end stopwatch to track Kailash's LU code
    C_No(i)=cond(H);

end
end
```

Test_HigherPower.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function to test the performance of user-written LU code on the system  %
% (A^n)x = b. A an b are randomly generated at the beginning. The exponent%
% 'n' is continually varied and the behavior of x is observed.            %
%  A system of 25 equations is studied.                                   %
% The function returns the following :                                    %
% x    - A [25 x n] array that contains the solution to (A^n)x = b for    %
%        each value of n.
% time - Time taken by the user-written code to solve the system(A^n)x = b%
%        for each value of n.                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [x, time] = Test_HigherPower(n)
x = zeros(25,n);
time = zeros(1,n);
A = randi(12,25);
b = [1:25]';
B = eye(25);        %creates an identity matrix of size 25 x 25
for i=1:n
    B=B*A;          %effectively, B = A^n
    tic             %begin stopwatch to track Kailash's LU code
    x(:,i)= Ludecomp(B,b,25,10e-30);
    time(i) = toc; %end stopwatch to track Kailash's LU code
end
end
```