ECSE 343 Assignment 1 Report
Grey Yuan (260857954)

Question 1

(a) Result obtained:

```
when n = 1 , the epsilon is 2.220446e-16
when n = 2 , the epsilon is 4.440892e-16
when n = 3 , the epsilon is 4.440892e-16
when n = 4 , the epsilon is 8.881784e-16
when n = 5 , the epsilon is 8.881784e-16
when n = 7 , the epsilon is 8.881784e-16
when n = 8 , the epsilon is 1.776357e-15
when n = 15 , the epsilon is 1.776357e-15
when n = 200 , the epsilon is 2.842171e-14
when n = 1022 , the epsilon is 1.136868e-13
when n = 1023 , the epsilon is 1.136868e-13
```

By observing the results we get in the figure above, we can see that as n increases the value for the epsilon which is the absolute error grows larger. This is because the nature of epsilon indicates that, as n is increasing, the difference between 1 and the next higher number that can be restored using floating point representation become smaller. However, as n increases, the power stored also gets larger. So, in floating point representation, the lowest number in the mantissa will remain the same. As the power become larger and larger, the error will become larger and larger.
We also notice in the results that some values the same while some are different. To explain this, we can show the calculation for n = 1, 2, and 3 as example:

$n=1$: # just greater than $1 = 1.000\ldots00\,|_{123\ldots51\,52} \times 2^0$

$\epsilon = 0.000\ldots00\,|\times 2^0 = 1.0\ldots0\times 2^{-52} = 2.2204\times 10^{-16}$

$n=2$: $\epsilon = 1.000\ldots001\times 2^1 - 1.000\ldots000\times 2^1 = 0.000\ldots001\times 2^1 = 2^{-51} = 4.44089\times 10^{-16}$

$n=3$: $\epsilon = 1.100\ldots001\times 2^1 - 1.100\ldots000\times 2^1 = 0.100\ldots001\times 2^1 = 2^{-51} = 4.44089\times 10^{-16}$

Therefore, we see ϵ is the same when the exponents of 2 (underlined) are the same.
All the statements above explain how the results are obtained.

(b) The function rel_ep(n):

```
function rel_eplison = rel_ep(n)
 rel_eplison = double(1);
 r = rel_eplison/2;
 while (n+rel_eplison) ~= n
 rel_eplison = rel_eplison/2;
 r = rel_eplison/2;
 end
 rel_eplison= rel_eplison*2/n;
%  r=r*2;
%  error=r/n;
end
```

We call this function with the same value with (a), then we get the result:

```
when n = 1 , the relative error is 2.220446e-16
when n = 2 , the relative error is 2.220446e-16
when n = 3 , the relative error is 1.480297e-16
when n = 4 , the relative error is 2.220446e-16
when n = 5 , the relative error is 1.776357e-16
when n = 7 , the relative error is 1.268826e-16
when n = 8 , the relative error is 2.220446e-16
when n = 15 , the relative error is 1.184238e-16
when n = 200 , the relative error is 1.421085e-16
when n = 1022 , the relative error is 1.112396e-16
when n = 1023 , the relative error is 1.111308e-16
```

In the code, the only difference from (a) is we divide the relative epsilon by the input n since we are calculating the relative error. Since the epsilon is increasing as n increases, dividing by n will make the ration between n and its error stay stable. That is why the exponent of e remains -16 in contrast to that in (a).

(c) With the knowledge about floating point representation, we know eps('single') should output eps_correct = 1*2^-23. To check if we get it correctly, we run them together and we get the following result:

```
ans = single
    1.1921e-07
eps_correct = 1.1921e-07
```

This is because a single type will have 1 bit of sign, 23 bits of mantissa, and 8 bits of exponent. The space of mantissa required for a 32-bit floating point number is 23. Therefore, eps('single') should give us 1*2^-23.

Question 2

(a) I implemented the function and used it to calculate sin(x0) where x0 is pi/4:

```
x0 = pi/4;
der = dydx(x0)
function der = dydx(x0)
%this fucntion computes the derivative of sinx(x)
% YOUR CODE GOES HERE.
der = cos(x0);
end
```

Then, the output is shown below:

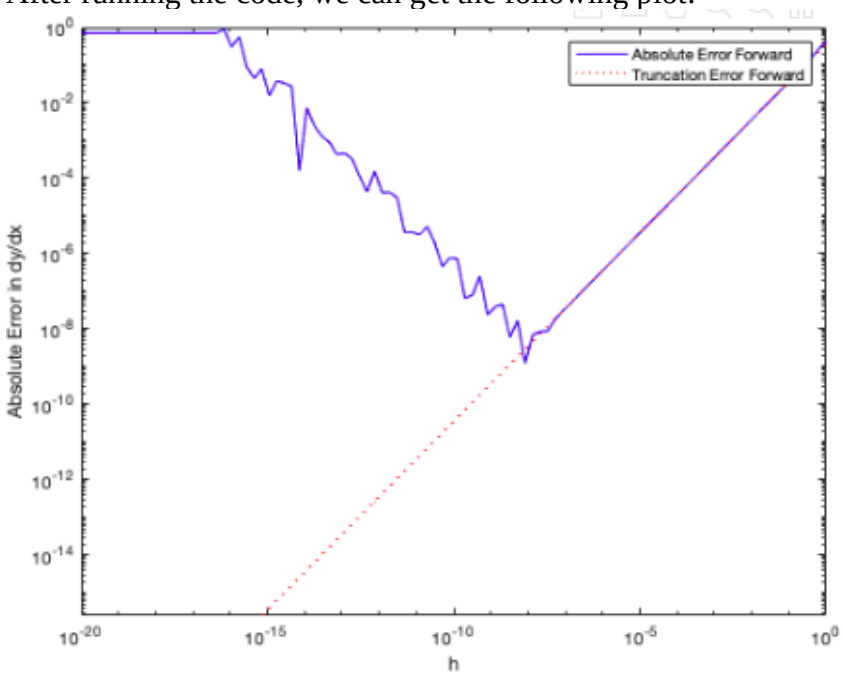
```
der = 0.7071
```

(b) I implemented the expression for the divided difference formula and truncation error for sin(x0) as shown below:

```
dydx_approx = (sin(x0+h)-sin(x0))./h   %IMPLEMENT equation (2.2) here
abserr = abs(derivative - dydx_approx);

truncation_error = sin(x0)*h/2   %Compute truncation_error
```

After running the code, we can get the following plot:



In the graph above, we can see the absolute value is erratic when h is small. This is because when h is small, when we implement the expression for divdied difference formula, there will be larger round-off error when doing (x0+h). In consequence, we would get x0 instead of x0+h as a result. Therefore, we may have higher or lower results depending on how small h is. If we have large h value, the round-off error effect will not be happening always, so we would get a relatively smoother graph.

(c) The point where the error is minimized is when h is 8.30218e^-9:

Justification:

$$f'(x) = \underbrace{\frac{f(x+h)-f(x)}{h}}_{\text{Approx.}} - \underbrace{\frac{f''(x)h}{2}}_{\text{truncation error}}$$

$$f(x+h) = \tilde{f}(x+h) + e_1$$

$$f(x) = \tilde{f}(x) + e_2$$

$$f'(x) = \frac{\tilde{f}(x+h)+e_1-(\tilde{f}(x)+e_2)}{h} - \frac{f''(x)}{2!}h$$

$$f'(x) = \underbrace{\frac{\tilde{f}(x+h)-\tilde{f}(x)}{h}}_{\text{derivative we can estimate}} + \underbrace{\frac{e_1-e_2}{h}}_{\text{round-off error}} - \underbrace{\frac{f''(x)h}{2!}}_{\text{truncation error}}$$

Error = True value − Estimated value

$$\text{Error} = \frac{e_1-e_2}{h} - \frac{f''(x)h}{2!}$$

we are trying to find optimal h to give us the smallest error.

$$|\text{Error}| \le \frac{e_1-e_2}{h} + \frac{f''(x)h}{2!}$$

max value of e_1 & e_2 is ϵ

max bond on e_1-e_2 is 2ϵ

$$|\text{Error}| \le \frac{2\epsilon}{h} + \frac{f''(x)h}{2!}$$

$$\frac{d|\text{Error}|}{dh} = 0 \Rightarrow \frac{d}{dh}\left(\frac{2\epsilon}{h} + \frac{f''(x)h}{2!}\right) = 0$$

$$-\frac{2\epsilon}{h^2} + \frac{f''(x)}{2!} = 0$$

$$\frac{2\epsilon}{h^2} = \frac{f''(x)}{2}$$

$$h = \sqrt{\frac{4\epsilon}{f''(x)}}$$

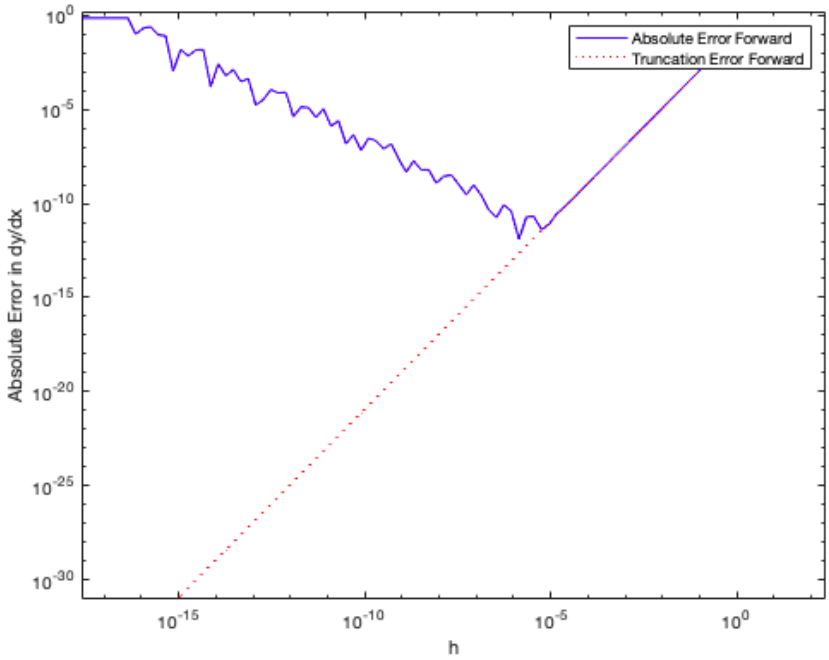
So, in our case, if we plug in f(x) and epsilon to the equation, h will minimize the error at approximately 8.30218e^-9.

(d) The expression for the truncation error is shown below:

```
dydx_approx = (sin(x0+h)-sin(x0-h))./(2*h)   %IMPLEMENT equation (2.4) here
abserr = abs(derivative - dydx_approx);

truncation_error = cos(x0)*h.*h/6   %Compute truncation_error
```

After runnig dydx_central_approx.mlx, we get the following plot:



(e) The central difference method would give a smaller abosolute error and the reason is followed:

It will reduce truncation error
we know the Forward difference formula: $f'(x) \simeq \frac{f(x+h)-f(x)}{h}$,
and its truncation error is $\frac{f''(x)h}{2!}$
Then will should find truncation error for the central difference method and see how its error is less.
central different formula: $f'(x) \simeq \frac{f(x+h)-f(x-h)}{2h}$

So, $f(x+h) \simeq f(x) + f'(x)(x+h-x) + \frac{f''(x)}{2!}(x+h-x)^2 + \frac{f'''(x)}{3!}(x+h-x)^3 + \cdots$ ①

$f(x-h) \simeq f(x) + f'(x)(x-h-x) + \frac{f''(x)}{2!}(x-h-x)^2 + \frac{f'''(x)}{3!}(x-h-x)^3 + \cdots$ ②

② - ①: $f(x+h) - f(x-h) \simeq f(x) - f(x) + f'(x)(x+h-x) - f'(x)(x-h-x)$
$+\frac{f''(x)}{2!}(x+h-x)^2 - \frac{f''(x)}{2!}(x-h-x)^2$
$+\frac{f'''(x)}{3!}(x+h-x)^3 - \frac{f'''(x)}{3!}(x-h-x)^3$

$\Rightarrow f(x+h) - f(x-h) \simeq f'(x)h + f'(x)h + \frac{f''(x)}{2!}h^2 - \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \frac{f'''(x)}{3!}h^3$

$f(x+h) - f(x-h) \simeq 2f'(x)h + 2\frac{f'''(x)}{3!}h^3$

$2hf'(x) \simeq f(x+h) - f(x-h) - \frac{2}{3!}f'''(x)h^3$

$f'(x) \simeq \frac{f(x+h)-f(x-h)}{2h} - \boxed{\frac{f''(x)h^2}{3!}}$ → truncation error

Generally, $\frac{f''(x)h^2}{3!}$ will be smaller than $\frac{f''(x)h}{2!}$, since $0<h<1$ causes $h^2<h$.
In our case, $f(x) = \sin(x\pi)$ so the difference between $f'''(x)$ and $f''(x)$ will not have large effect since they are bounded between -1 and 1.

Also, the central difference formula will reduce round-off error. Since we are using x0+h and x0-h instead of x0+h and x0 in our differential equation expression, we have reduce the possibility for round-off error to occur. The expression x0+h and x0-h are twice more seperated than x0+h and x0.

Question 3

(a) We first initialize L with identity matrix. Then we use the Gaussian Elimination algorithm introduced in class to compute A. After we get A, we use triu(A) to get the upper part of A and use tril(A,-1) to get the lower part replacing elements in the diagose with 1. The implementation of LU_decomposition(A) with Gaussian Elimination is shown below:

```
function [L, U] = LU_decomposition(A)
% L is lower triagular matrix
% U is upper triangular matrix
% YOUR CODE GOES HERE
s = size(A, 1); % Obtain number of rows in A
L = eye(s); %start L as an identity matrix
for k = 1 : s-1 % from column 1 to the second last column
    for l = k + 1 : s % from row below the pivot to the end
        A(l,k) = A(l,k)/A(k,k); %devide each entry under the pivot by the pivot
        A(l,k+1:end) = A(l,k+1:end)-A(l,k)*A(k,k+1:end);% gaussian elimination
    end
end
U = triu(A);% get the upper part in A
L = L + tril(A,-1);% add the lower part in A to L
end
```

(b) Based on (a), we need to pivot before we decompose the matrix to get L and U. . So, we need to find the maximum value in a column and change the max absolute value to the top. After finding what rows we need to do the swapping with max() and abs(), we swap them and then use X to store each P1,P2,P3… that we get from pivoting. Next, we get our new P and A by multiplying X in the front. The implementation of LU_rowpivot(A) with Gaussian Elimination is shown below:

```
function [L,U,P] = LU_rowpivot(A)
% L is lower triagular matrix
% U is upper triangular matrix
% P is the permutation matrix
% P*A = L*U
% YOUR CODE GOES HERE
s = size(A,1);
P = eye(s);
for k = 1:s-1 %for each column we need to find the max abs value and switch it with the pivot
    [~,n] = max(abs(A(k:end,k)));%find the max abs value in the column
    X = eye(s);
    temp=X(k, : );
    X(k, : )=X(n+k-1, : );
    X(n+k-1, : )=temp; % swap the pivot row and the max value row in P1
   P=X*P;
   A=X*A;
   % LU decomposition
   for l = k + 1 : s % from row below the pivot to the end
       A(l,k) = A(l,k)/A(k,k); %devide each entry under the pivot by the pivot
       A(l,k+1:end) = A(l,k+1:end)-A(l,k)*A(k,k+1:end);% gaussian elimination
   end
end
U = triu(A);% get the upper part in A
L = tril(A,-1)+eye(s);%get the lower part in A and add a identity matrix
end
```

(c) I used the formula introduced in class to implement my code.

1. Forward substitution:

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$y_1 = \frac{1}{l_{11}} b_1 \; ; \; y_2 = \frac{1}{l_{22}} (b_2 - l_{21} y_1) \; ; y_3 = \frac{1}{l_{33}} (b_3 - l_{31} y_1 - l_{32} y_2)$$

2. Backward substitution:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

$$x_3 = \frac{1}{u_{33}} y_3 \; ; \; x_2 = \frac{1}{u_{22}} (y_2 - u_{23} x_3) \; ; x_1 = \frac{1}{u_{11}} (y_1 - u_{12} x_2 - u_{13} x_3)$$

In forward substitution, we first need to get the number of rows in L. Also, intialize y with all elements equal to 0. Then we just translate the algorithm with a nested for loop to calculate y1,y2, and y3. It is very important to note not to flip b in this function, but flip before calling this function instead. The implementation of forward_sub(L,b) is shown below:

```
function y = forward_sub(L,b)
% y is the vector
% L is the lower triangular matrix
% % YOUR CODE GOES HERE
s = size(L,1);
y = zeros(s,1); %create verctor y with 0
for i = 1:s
    y(i,1) = b(i,1);
    cur = i;
    for j = 2:s
        if cur ~= 1
            cur = cur - 1;
            y(i) = y(i)-L(i,cur)*y(cur,1);
        end
    end
end
end
```

In backward substitution, we first get the row number in U and intialize the solution X with all elements equal to 0. The implementation of the formula is also similar but just in reverse order (from bottom to top) when computing X3, X2, and X1. The implementation of backward_sub(U,y) is shown below:

```
function X = backward_sub(U,y)
% X is the vector
% U is the upper triangular matrix
%    WRITE YOUR CODE HERE
s = size(U,1);
X = zeros(s,1); %create vecter X with 0
for i = s:-1:1 % opposite direction from bottom to top
    X(i,1) = y(i,1);
    cur = i;
    for j = 1:s
        if cur ~= s
            cur = cur + 1;
            X(i,1) = X(i,1)-U(i,cur)*X(cur,1);
        end
    end
    X(i,1) = X(i,1)/U(i,i);
end
end
```

(d) To use the LU decomposition without pivoting, we run the following code:

```
A = [1e-16 2 5 5; 0.2 1.6 7.4 5;0.5 4 8.5 5 ; 0.5002 8 11 97];
b = [400;5;18;95];
[L,U]=LU_decomposition(A)
% b = flip(b)
% [L,U,P]=LU_rowpivot(A)
y = forward_sub(L,b)
x = backward_sub(U,y)
```

We will get the following output:

```
L = 4×4
10^15 ×
      0.0000         0         0         0
      2.0000    0.0000         0         0
      5.0000    0.0000    0.0000         0
      5.0020    0.0000    0.0000    0.0000

y = 4×1
10^17 ×
       0.0000
      -8.0000
            0
      -0.0000

U = 4×4
10^16 ×
      0.0000    0.0000    0.0000    0.0000
           0   -0.4000   -1.0000   -1.0000
           0         0   -0.0000   -0.0000
           0         0         0    0.0000

x = 4×1
10^3 ×
      -2.2737
       0.2000
       0.0087
      -0.0087
```

(e) If we use the LU decomposition with row pivoting, we run the following code:

```
A = [1e-16 2 5 5; 0.2 1.6 7.4 5;0.5 4 8.5 5 ; 0.5002 8 11 97];
b = [400;5;18;95];
%[L,U]=LU_decomposition(A)
b = flip(b)
[L,U,P]=LU_rowpivot(A)
y = forward_sub(L,b)
x = backward_sub(U,y)
```

We will get the following output:

```
L = 4×4
      1.0000         0         0         0
      0.9996    1.0000         0         0
      0.3998    0.4000    1.0000         0
      0.0000   -0.5004    0.9378    1.0000

U = 4×4
      0.5002    8.0000   11.0000   97.0000
           0   -3.9968   -2.4956  -91.9612
           0         0    4.0000    3.0000
           0         0         0  -43.8308

y = 4×1
      95.0000
     -76.9620
      -2.2000
     363.5514

P = 4×4
     0     0     0     1
     0     0     1     0
     0     1     0     0
     1     0     0     0

x = 4×1
10^3 ×
      -1.6299
       0.2066
       0.0057
      -0.0083
```

(f) Obviously, the LU decomposition with pivoting will get us a more accurate result, since the whole point of using pivoting is to reduce the round-off error when dividing each elements under the pivot by 0 or a very small number. If we use a scientific calculator to solve the system, we get X = [-1629.931796;206.5590;5.6708;-8.2944] which is very close to what we got. So, we can see that LU decomposition with row poviting works very accurately on solving the matrix.

Appendices

Q1 code (A1Q1)

```
n = [1 2 3 4 5 7 8 15 200 1022 1023 1.3e6];


for a = 1:11
    fprintf("when n = %d , the epsilon is %d \n", n(a), ep(n(a)));
end


for a = 1:11
    fprintf("when n = %d , the relative error is %d \n", n(a),rel_ep(n(a)));
end


%clear eps
eps('single')
eps_correct = 1*2^-23
```

```
function eplison = ep(n)
 eplison = double(1);
 r = eplison/2;
 while (n+eplison) ~= n
 eplison = eplison/2;
 r = eplison/2;
 end
 eplison= eplison*2;
end


function rel_eplison = rel_ep(n)
 rel_eplison = double(1);
 r = rel_eplison/2;
 while (n+rel_eplison) ~= n
 rel_eplison = rel_eplison/2;
 r = rel_eplison/2;
```

```
 end
 rel_eplison= rel_eplison*2/n;
%  r=r*2;
%  error=r/n;
end
```

Q2 code (dydx_approx)

```
clear all
% 100 logarthmically spaced points between 10^0 and 10^-20
h = logspace(-20,0,100);


x0 = pi/4;
derivative= dydx(x0);


dydx_approx = (sin(x0+h)-sin(x0))./h   %IMPLEMENT equation (2.2) here
abserr = abs(derivative - dydx_approx);


truncation_error = sin(x0)*h/2   %Compute truncation_error




clf
figure(1)
loglog(h,abserr,'b')
hold on
figure(1)
loglog(h(25:100),truncation_error(25:100),'r:')


xlabel('h')
ylabel('Absolute Error in dy/dx')
```

```
legend('Absolute Error Forward','Truncation Error Forward')
```

```
x0 = pi/4;
der = dydx(x0)
function der = dydx(x0)
%this fucntion computes the derivative of sinx(x)
% YOUR CODE GOES HERE.
der = cos(x0);
end
```

Q2 code (dydx_central_approx)

```
clear all
% 100 logarthmically spaced points between 10^0 and 10^-20
h = logspace(-20,0,100);


x0 = pi/4;
derivative= dydx(x0);


dydx_approx = (sin(x0+h)-sin(x0-h))./(2*h)   %IMPLEMENT equation (2.4) here
abserr = abs(derivative - dydx_approx);


truncation_error = cos(x0)*h.*h/6   %Compute truncation_error




clf
figure(1)
loglog(h,abserr,'b')
hold on
figure(1)
```

```
loglog(h(25:100),truncation_error(25:100),'r:')


xlabel('h')
ylabel('Absolute Error in dy/dx')


legend('Absolute Error Forward','Truncation Error Forward')
```

```
function der = dydx(x)
%this fucntion computes the derivative of sinx(x)
% YOUR CODE GOES HERE.
der = cos(x);
end
```

Q3 code(A1Q3)

```
A = [1e-16 2 5 5; 0.2 1.6 7.4 5;0.5 4 8.5 5 ; 0.5002 8 11 97];
b = [400;5;18;95];
[L,U]=LU_decomposition(A)
% b = flip(b)
% [L,U,P]=LU_rowpivot(A)
y = forward_sub(L,b)
x = backward_sub(U,y)
```

```
function [L, U] = LU_decomposition(A)
% L is lower triagular matrix
% U is upper triangular matrix
% YOUR CODE GOES HERE
s = size(A, 1); % Obtain number of rows in A
```

```
L = eye(s); %start L as an identity matrix
for k = 1 : s-1 % from column 1 to the second last column
   for l = k + 1 : s % from row below the pivot to the end
      A(l,k) = A(l,k)/A(k,k); %devide each entry under the pivot by the pivot
      A(l,k+1:end) = A(l,k+1:end)-A(l,k)*A(k,k+1:end);% gaussian elimination
   end
end
U = triu(A);% get the upper part in A
L = L + tril(A,-1);% add the lower part in A to L
end


function [L,U,P] = LU_rowpivot(A)
% L is lower triagular matrix
% U is upper triangular matrix
% P is the permutation matrix
% P*A = L*U
% YOUR CODE GOES HERE
s = size(A,1);
P = eye(s);
for k = 1:s-1 %for each column we need to find the max abs value and switch it with the pivot
   [~,n] = max(abs(A(k:end,k)));%find the max abs value in the column
   X = eye(s);
   temp=X(k, : );
   X(k, : )=X(n+k-1, : );
   X(n+k-1, : )=temp; % swap the pivot row and the max value row in P1

   P=X*P;
   A=X*A;
   % LU decomposition
   for l = k + 1 : s % from row below the pivot to the end
      A(l,k) = A(l,k)/A(k,k); %devide each entry under the pivot by the pivot
      A(l,k+1:end) = A(l,k+1:end)-A(l,k)*A(k,k+1:end);% gaussian elimination
   end
```

```
end
U = triu(A);% get the upper part in A
L = tril(A,-1)+eye(s);%get the lower part in A and add a identity matrix
end


function y = forward_sub(L,b)
% y is the vector
% L is the lower triangular matrix
% % YOUR CODE GOES HERE
s = size(L,1);
y = zeros(s,1); %create verctor y with 0
for i = 1:s
    y(i,1) = b(i,1);
    cur = i;
    for j = 2:s
        if cur ~= 1
            cur = cur - 1;
            y(i) = y(i)-L(i,cur)*y(cur,1);
        end
    end
end
end


function X = backward_sub(U,y)
% X is the vector
% U is the upper triangular matrix
%    WRITE YOUR CODE HERE
s = size(U,1);
X = zeros(s,1); %create vecter X with 0
for i = s:-1:1 % opposite direction from bottom to top
    X(i,1) = y(i,1);
    cur = i;
    for j = 1:s
```

```
        if cur ~= s
            cur = cur + 1;
            X(i,1) = X(i,1)-U(i,cur)*X(cur,1);
        end
    end
    X(i,1) = X(i,1)/U(i,i);
end
end
```