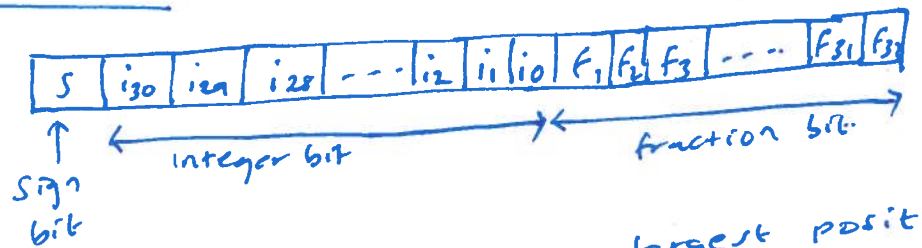
Name:Lawi Mwirigi
CLass : ECSE 343
Assignment 1.

Solutions.

Question 1.

a)



Find the smallest and the largest positive number that can be represented in this format.

i) smallest

$\underline{0}$ 000 - - - 0 . 000 - - - 01.

sign

$= 2^{-32} = 2.328306437 \times 10^{-10}$

ii) largest.

$\underline{0}$ 111 - - - . 1 . 111 - - - - 11

this is equal to

$(2^{31} - 1) + (1 - 2^{-32}) =$

$= 2.147483648 \times 10^{9}$

b) How does this compare to the largest and smallest positive number with 64-bit floating point representation?

64 bit floating point representation;

Smallest $= 2 \times 10^{-308}$

largest $= 2 \times 10^{308}$

when we compare the 64 bit floating point representation with above we realize that 64 bit floating point representation is better than the one above because it can represent a larger range in a , while using the same number of bits.

c) Find the largest subnormal number that can be represented in 64-bit floating point representation?

0 000 0 --- 000 1111 --- 1111

$\uparrow$ sign $\longleftrightarrow$ exponent $\longleftrightarrow$ mantissa (52 bits)

$\approx 2.22507385 \times 10^{-308}$

Name:Lawi Mwirigi
CLass : ECSE 343
Assignment 1.

Solutions.

Question 2.

a) From the plot, we realize that the epsilon is increasing as M is increasing. Also if we ~~add~~ check my epsilon implementation I added count to know how many times we divide the epsilon by 2 to get the smallest representable number. From this we can learn that for our numbers from $1 \to 1000$ e.g. $\boxed{1 \to 7}$ ~~can~~ we obtain epsilon by dividing 53 times, thus they share same epsilon value. In conclusion, we can say different numbers of groups of numbers from $1 \to 1000$ share same epsilon values that why the there is a sudden ~~bum~~ jump followed by a constant epsilon value. e.g from $666 \to 1000$ they have same epsilon value, M is divided 52 times and so on.

b). The method I implemented for calculating epsilon gave me the same exact epsilon as the computer epsilon. Thus the graph is similar to that in a as expected.

Explanation for the code.

declared epsilon $= 2^{\text{round}(\log(m))}$ | 1.
then I am using a while loop until M+ epsilon is not greater than M, thus I have gotten the epsilon from the privious iteration, thus I multiply the current epsilon by 2.

c). i) The graph I obtained has sudden Spikes then after every spike there is a sort of exponential decay. As I highlight earlier, there are some numbers with similar epsilon, we can group these numbers like group $a_1, a_2, a_3 \ldots a_m$ where a_1 contains $M = 1 \ldots 7$, these numbers in a_1 have same epsilon. thus the relative error ~~of~~ of 1 is greater than the relative error of 7, because $1 < 7$, thus the decrease in relative error.

ii) Yes there is an upperbound for η_m.

iii) Estimated upper bound $= \underline{2.22045 81 \times 10^{-16}}$

real upperbound from this function $= \max(\text{eps}(i)/i) = \underline{\underline{2.2204e^{-16}}}$

d) eps ('single') = 1.1921e-07

$1 = 1.0000\ldots0 \times 2^0$

the next number is

$1.0000\ldots1 \times 2^0$

the difference is

epsilon $= 2^{1-23}$

$$\boxed{K = 2^{-23} = 1.1921e^{-0.7}}$$

2a) Question 2 Matlab implementation.

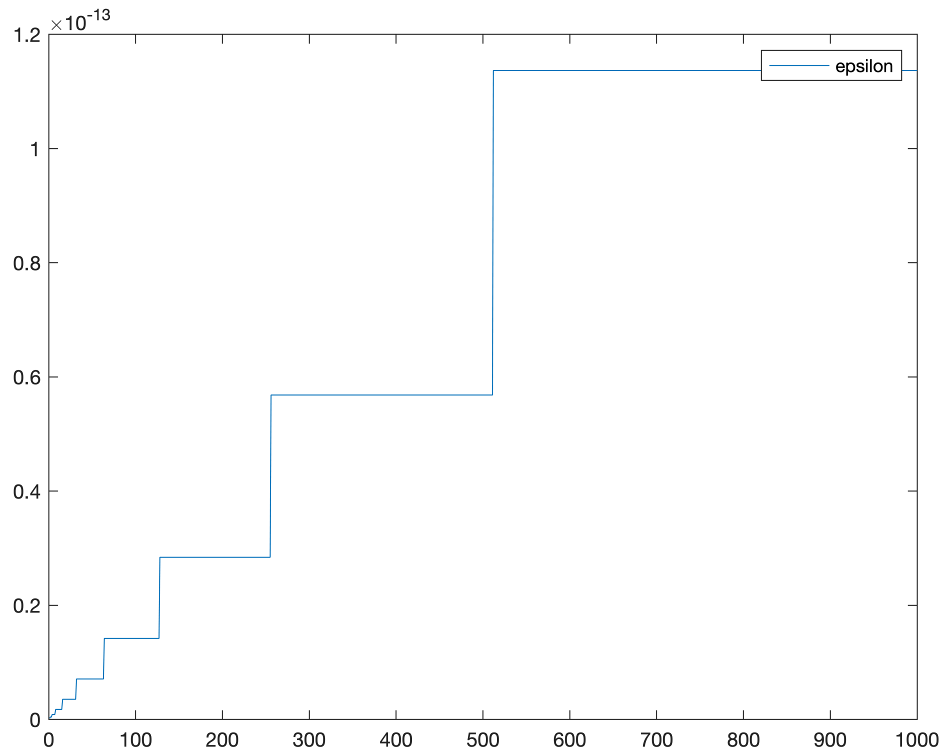
```
%Part a
v = 1:1000 % vector
```

```
v = 1×1000
     1     2     3     4     5     6     7     8     9    10    11    12    13 ···
```

```
b = length(v)
```

```
b = 1000
```

```
B = 1;
for i=1:b
    B(i)=  eps(i);
end
plot(v,B)
legend('epsilon');
```




```
eps(43)
```

```
ans = 7.1054e-15
```

```
ep(43)
```

```
ans = 7.1054e-15
```

```
%from the plot we realize as M is increasing the epsilon is increasing.the
%reason why epsilon is constant(same for some m values) is  sometimes  that for cases 2
%where x is an odd number then the epsilon remains constant.
```

2c)

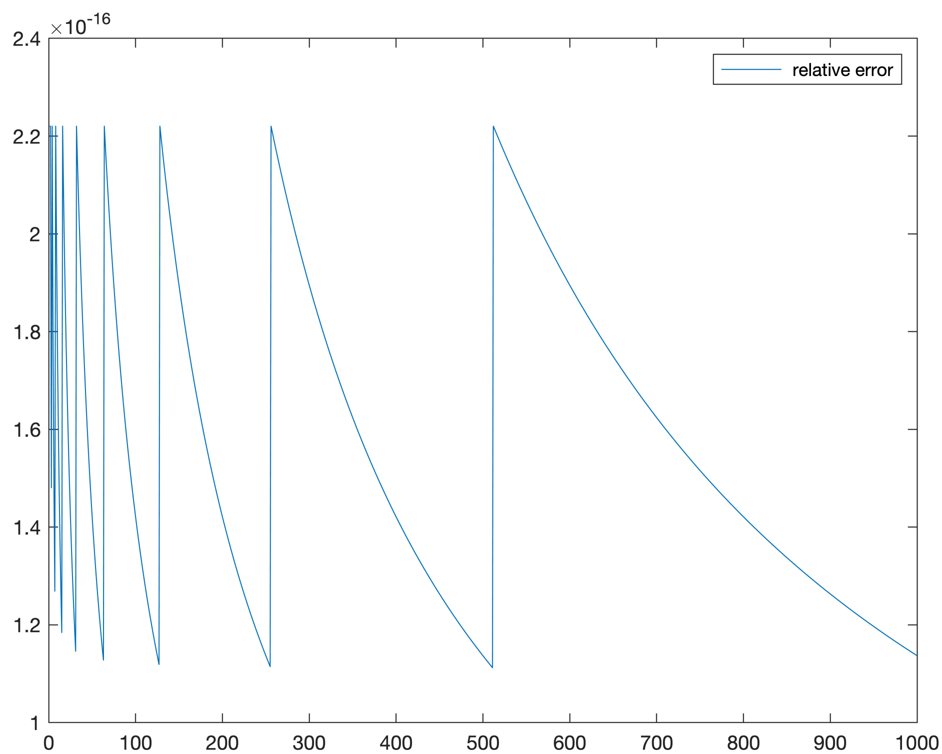
```
%Part c

Q = 1:1000 % vector
```

```
Q = 1×1000
     1     2     3     4     5     6     7     8     9    10    11    12    13 ···
```

```
b = length(Q)
```

```
b = 1000
```

```
B = 1;
vec=[];
for i=1:b
    B(i)=  eps(i)/i;
P=max(B);
end
plot(Q,B)
legend(' relative error');
```




```
upperbound = P
```

```
upperbound = 2.2204e-16
```

```
%for this question we learn that from question a there were some numbers
%with similar epsilon thus we can group these numbers in groups like group
%a1, b1, c1 etc i.e a group of numbers with similar epsilon numbers . Now
%for every group let say a1 it has numbers from 1 to 16 , thus since their
%epsilon is the same if we find the relative error , 1 will have the
%largest relative error while 16 will have the lowest relative error
%because it has a bigger denominator.
```

2d)

```
%part d

eps("single")
```

```
ans = single

    1.1921e-07
```

```
%1=1.0000......0*2^0
%the next number is 1.0000........1*2^0
%the difference is epsolon =2^-23
k = 2^-23
```

```
k = 1.1921e-07
```

2b)

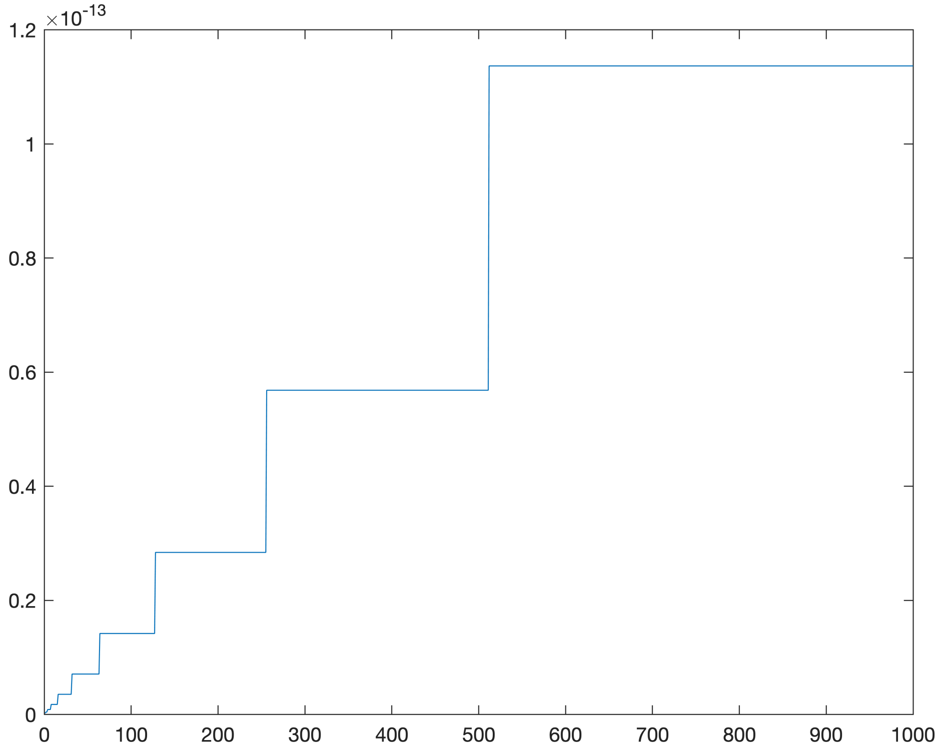
```
%part b

V = 1:1000 % vector
```

```
V = 1×1000
     1     2     3     4     5     6     7     8     9    10    11    12    13 ···
```

```
a = length(V)
```

```
a = 1000
```

```
for i=1:a
    M(i)=  ep(i);
end
plot(V,M)
```




```
ep(10)
```

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

```
eps(10)
```

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

```
%the graph we obtained here is similar to the one we obtained in part a .
%Thus similar explanation.
```

my function.

```
function eplison = ep(m)
 % your code goes here
 eplison = 2^round(log(m));
 count = 1;
while(m+eplison)> m
 eplison=eplison/2;
 count = count + 1;
 end
 eplison = 2 * eplison ;
 count;
 m;
end
```

Name:Lawi Mwirigi
CLass : ECSE 343
Assignment 1.

Solutions.

Question 3.

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{1}{2!}f''(x_0)h^2 + \frac{1}{3!}(x_0)f'''(x_0)h^3 + \cdots$$

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h}$$

a) $$\frac{f(x_0+h) - f(x_0)}{h} = \frac{f(x_0)}{1!} + \left[\frac{hf''(x_0)}{2!} + \frac{h^2}{3!}\frac{f'''(x_0)}{} + \cdots\right]$$

$$\text{truncation error} = \boxed{\frac{h}{2!}f''(x_0) + \frac{h^2}{3!}f'''(x_0) + \cdots}$$

$$\approx \frac{hf''(x_0)}{2!}$$

b) From my graph we notice that for large values of h the error is smooth because it represents the truncation error which is ~~smooth~~ linear while for smaller values it is more erratic because the round off error becomes dominant.

c) I obtained the minimum error ~~at~~ in my graph when the y value is lowest at $h = 0.00000001322084$.
$= 1.322084 \times 10^{-8}$.

$\boxed{\text{min abs error (y)} = 2.0614e^{-8}}$

$$f'(x_0) = \frac{f(x_0+h) + e_1 - f(x_0) - e_2}{h} - \frac{f''(x_0) \times h}{2}$$

$$= \frac{f(x_0+h) - f(x_0)}{h} + \frac{e_1 - e_2}{h} - \frac{f''(x_0)h}{2}$$

thus the resulting error

is $\text{error} = \frac{e_1 - e_2}{h} - \frac{f''(x_0)h}{2}$

$$|\text{error}| \leq \frac{e_1 - e_2}{h} + \frac{f''(x_0)h}{2}$$

$$|\text{error}| \leq \frac{2\varepsilon}{h} + \frac{f''(x_0)h}{2}$$

$$\frac{d}{dh}(|\text{error}|) \leq \frac{d}{dh}\left(\frac{2\varepsilon}{h} + \frac{f''(x_0)h}{2}\right)$$

$$0 \leq \frac{2\varepsilon}{h^2} + \frac{f''(x_0)}{2}$$

$$\frac{2\varepsilon}{h^2} = \frac{f''(x_0)}{2} \qquad h = \sqrt{\frac{4\varepsilon}{f''(x_0)}}$$

$$4\varepsilon = h^2 f''(x_0)$$

$\boxed{\cancel{h = 3.6380\,e^{-12}}}$ from the computer.

read from my computer min abs error = $\boxed{2.0614e^{-8}}$ Note, $\boxed{h = 1.3219856 \times 10^{-8}}$

d) truncation error = $\boxed{\frac{f'''(x_0)h^2}{3!} - \frac{f''''(x_0)h^3}{4!} + \cdots}$

truncation error = $\frac{f'''(x_0)h^2}{3!}$

e) optimal value of h where I obtain the minimum error

$h = ?$ ⇒ when abs error is smallest.

$y = \min(\text{absolute error}) = \boxed{6.1628\,e^{-12}}$ $h=$

f) The differences formula that gives a smaller absolute error is the central difference method, which gives us a small absolute value of $\boxed{6.1628\,e^{-12}}$ while the other in b gives us $\boxed{2.0614e^{-8}}$

Question 3b.

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

x0 = pi/4;


% write the true value of the derviavtive of tan(x) evaluated at x0
TrueDeriv = sec(x0)*sec(x0)
```

```
TrueDeriv = 2.0000
```

```
% compute the numerical deriative of the given function using eq. (2.2) of assignment
% for all the values of h

dydx_approx = (tan(x0+h)-tan(x0))./h
```

```
dydx_approx = 1×100
         0         0         0         0         0         0         0         0 ···
```

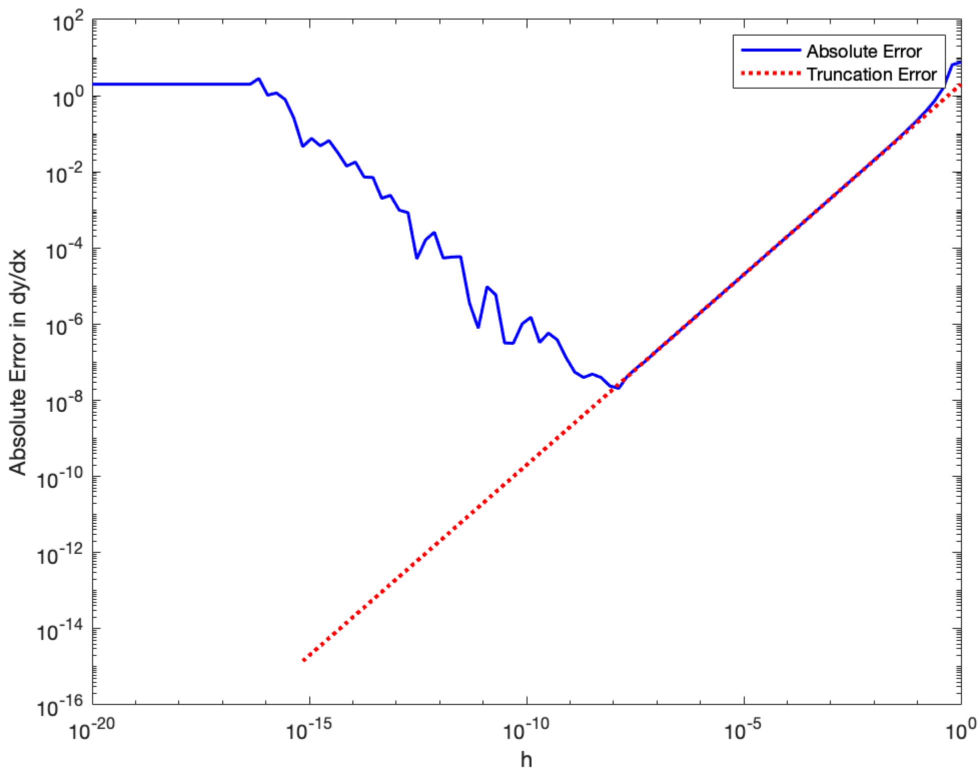
```
abserror=abs(TrueDeriv-dydx_approx)
```

```
abserror = 1×100
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000 ···
```

```
% compute the truncation error using the expression you obtained in Question 2 part a)
truncation_error = 2*sec(x0)*sec(x0)*tan(x0)*h/factorial(2)
```

```
truncation_error = 1×100
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000 ···
```

```
% compute the absolute error . It is already done for you.
% Absolute error is the difference between the True value of derivative and
% estimated value of the derivative.
Absolute_error = abs(TrueDeriv - dydx_approx);

clf
figure(1)
loglog(h,Absolute_error,'b','Linewidth',1.5)
hold on
figure(1)
loglog(h(25:100),truncation_error(25:100),'r:','Linewidth',2)

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

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




```
%b i)
%Explanation

%For large values of h the error is smooth because it represents the
%trauncation error which is linear while for small values it is more
%erratic the round off error becomes dominant leading to a
%more erratic behavior of the total error
```

```
minabs=min(Absolute_error)
```

```
minabs = 2.0614e-08
```

```
h=sqrt((4*eps(min(Absolute_error)))/2*sec(x0)*sec(x0)*tan(x0))
```

```
h = 3.6380e-12
```

Question 3(d).

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

x0 = pi/4;

% write the true value of the derviavtive of tan(x) evaluated at x0
TrueDeriv = sec(x0)^2;

% compute the numerical deriative of the given function using eq. (2.2) of assignment
% for all the values of h
dydx_approx =(tan(x0+h)-tan(x0-h))./(2*h)
```

```
dydx_approx = 1×100
         0         0         0         0         0         0         0         0 ···
```

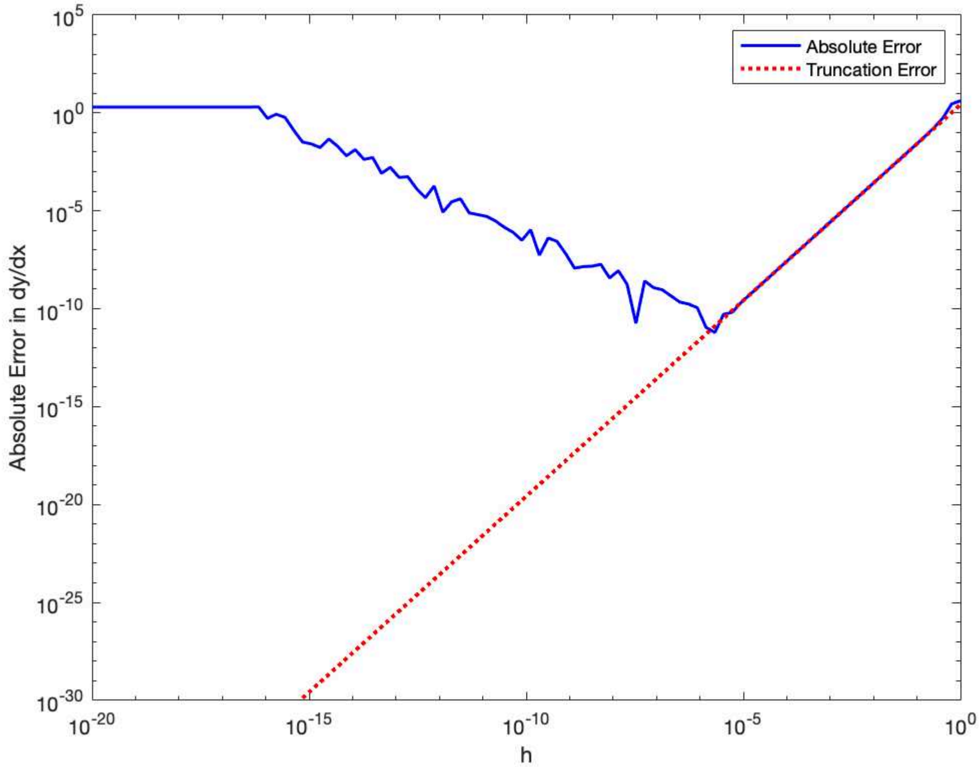
```
abserror=abs(TrueDeriv-dydx_approx)
```

```
abserror = 1×100
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000 ···
```

```
% compute the truncation error using the expression you obtained in Question 2 part a)
truncation_error = (4*sec(x0)^2*tan(x0)^2+2*sec(x0)^4)*(h.*h)/factorial(3)
```

```
truncation_error = 1×100
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000 ···
```

```
% compute the absolute error . It is already done for you.
% Absolute error is the difference between the True value of derivative and
% estimated value of the derivative.
Absolute_error = abs(TrueDeriv - dydx_approx);

clf
figure(1)
loglog(h,Absolute_error,'b','Linewidth',1.5)
hold on
figure(1)
loglog(h(25:100),truncation_error(25:100),'r:','Linewidth',2)

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

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




```
%TE = ((x0)*f"'(x0)*h^3)/factorial(3)
ymin=min(Absolute_error)
```

```
ymin = 6.1628e-12
```

Name:Lawi Mwirigi
CLass : ECSE 343
Assignment 1.

Solutions.

Question 4.

a) code implemented using gaussian elimination.

b) LU factorization implemented using partial pivoting to give us L, U, P.

c) Forward and backward substitution

d)
$$\underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{x} = \begin{bmatrix} -284.2171 \\ -1.2393 \\ 14.2857 \\ 3.3929 \end{bmatrix}$$

$$Ax = \begin{bmatrix} 40.0000 \\ 59.6387 \\ 82.9343 \\ 206.4182 \end{bmatrix} \neq \begin{bmatrix} 40 \\ 52 \\ 18 \\ 93 \end{bmatrix}$$

e)
$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 908.4554 \\ 4.24 \\ -20.7698 \\ -6.2389 \end{bmatrix} \qquad Ax = PAx = \begin{bmatrix} 40.0000 \\ 52.0000 \\ 18.0000 \\ 95.0000 \end{bmatrix} = \begin{bmatrix} 40 \\ 52 \\ 18 \\ 95 \end{bmatrix}$$

f) The solution obtained in e is more accurate than the one obtained in d. thus LU factorization by partial pivoting is more accurate. Gaussian elimination is unstable

Question 4.

Use this .mlx file to write the code for LU decompositin. Use the format of the functions provided below.

Note that in the mlx script the function need to be located at the end of the file.

```
%d)%Test the code here
A = [1e-16 50 5 9;
    0.2 5 7.4  5;
    0.5 4 8.5 32;
    0.89 8 11 92];

B = [40; 52; 18; 95];
[L, U] = LU_decompositon(A)
```

```
L = 4x4
10^15 ×
    0.0000         0         0         0
    2.0000    0.0000         0         0
    5.0000    0.0000    0.0000         0
    8.9000    0.0000    0.0000    0.0000
U = 4x4
10^17 ×
    0.0000    0.0000    0.0000    0.0000
         0   -1.0000   -0.1000   -0.1800
         0         0   -0.0000    0.0000
         0         0         0    0.0000
```

```
y = forward_sub(L,B)
```

```
y = 4x1
10^16 ×
    0.0000
   -8.0000
   -0.0000
    0.0000
```

```
X = backward_sub(U,y)
```

```
X = 4x1
 -426.3256
   -1.0469
   13.1250
    2.9688
```

```
verify = A*X
```

```
verify = 4x1
   40.0000
   21.4692
  -10.7878
   29.6952
```

```
%e)% compute X with LU decomposition implement in part b)
[L, U, P] = LU_rowpivot(A)
```

```
L = 4x4
    1.0000         0         0         0
```

```
    0.0000    1.0000         0         0
    0.2247    0.0640    1.0000         0
    0.5618   -0.0099    0.5143    1.0000
U = 4×4
    0.8900    8.0000   11.0000   92.0000
         0   50.0000    5.0000    9.0000
         0    0.0000    4.6079  -16.2506
         0   -0.0000         0  -11.2393
P = 4×4
     0     0     0     1
     1     0     0     0
     0     1     0     0
     0     0     1     0
L = 4×4
    1.0000         0         0         0
    0.0000    1.0000         0         0
    0.2247    0.0640    1.0000         0
    0.5618   -0.0099    0.5143    1.0000
U = 4×4
    0.8900    8.0000   11.0000   92.0000
         0   50.0000    5.0000    9.0000
         0    0.0000    4.6079  -16.2506
         0   -0.0000         0  -11.2393
P = 4×4
     0     0     0     1
     1     0     0     0
     0     1     0     0
     0     0     1     0
```

```
y = forward_sub(L,B)
```

```
y = 4×1
   40.0000
   52.0000
    5.6809
   70.1208
```

```
X = backward_sub(U,y)
```

```
X = 4×1
  908.4554
    4.2400
  -20.7698
   -6.2389
```

```
verify=P*A*X
```

```
verify = 4×1
   40.0000
   52.0000
   18.0000
   95.0000
```

4a)

Part (a): Implement your LU decomposition **without** pivoting here.

```
function [L, U] = LU_decompositon(A)
L = eye(4);
U = A;
n = length(A);
```

```
 for j=1:n-1
   for i=j+1:n
       L(i,j)=U(i,j)/U(j,j);
       U(i,j:n)=U(i,j:n) - L(i,j)*U(j,j:n);
    end
   end
 end
```

4b)

Part (b): Implement your LU decomposition using **partial pivoting (row pivoting)** here.

```
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
[n,n]=size(A);
L=eye(n); P=L; U=A;
for k=1:n
    [pivot q]=max(abs(U(k:n,k)));
    q=q+k-1;
    if q~=k
        temp=P(k,:);
        P(k,:)=P(q,:);
        P(q,:)=temp;

        temp=U(k,:);
        U(k,:)=U(q,:);
        U(q,:)=temp;

        if 2<=k
            temp=L(k,1:k-1);
            L(k,1:k-1)=L(q,1:k-1);
            L(q,1:k-1)=temp;
        end
    end
    for j=k+1:n
        L(j,k)=U(j,k)/U(k,k);
        U(j,:)=U(j,:)-L(j,k)*U(k,:);
    end

end
L
U
P
end
```

4c)

Part (c): Implement your forward and backward substitution algorithms here.

```
function y=forward_sub(L,b)
n=length(b);
for i=2:n
    y(1,1)=b(1)/L(1,1);
    y(i,1)=(b(i)-L(i,1:i-1)*y(1:i-1,1))./L(i,i);
```

```
end
end

%% MATLAB code for backward_sub

function x=backward_sub(U,y)
n=length(y);
for i=n-1:-1:1
    k=0;
    x(n)=y(n)/U(n,n);
    for j=(i+1):n
        k=k+U(i,j)*x(j);
    end
    x(i)=(y(i)-k)/U(i,i);
end
x=x';
end
```