Muğla Sıtkı Koçman University

EEE-2012 – Numerical Analysis

ASSIGNMENT

Ahmad Zameer Nazarı

220702706

18th-25th April 2024

Contents

Part 1

QUESTION

1) A The function $\boldsymbol{f(x) = e^{\frac{x}{2}} - 2x}$ has a root on $[\mathbf{0};\ \mathbf{2}]$. Find the root using

a) Bisection method.
b) False position method.

Write "Matlab" codes to solve the question. Use four significant figures. End the program after approximate relative error is below 0.01%.

Add your codes to your solution. Explain which method finds the solution faster.

ANSWER

Introduction

Bisection and False-Position Methods, both are grouped under bracketing methods. They each require two initial root estimations, that form an interval surrounding the actual root. With the knowledge that a function changes sign around the root, tests are repeatedly made at subsequent root estimations, that lead to the interval narrowing down to approximate the root.

In Bisection, each subsequent estimate is the midpoint of the current interval, while in False Position method a slightly complicated process is implemented. Function values are used to extrapolate a line that estimates the root closer to the bracket with function value closer to zero. This causes the

Main

The code is initiated at `main.m`. it includes initial values with some other things, before the actual bracketing algorithms are called. The interval is specified by `bnd_low` and `bnd_up`. Prespecified percent tolerance by `tol_lvl` and max number of iterations by `iter_max`.

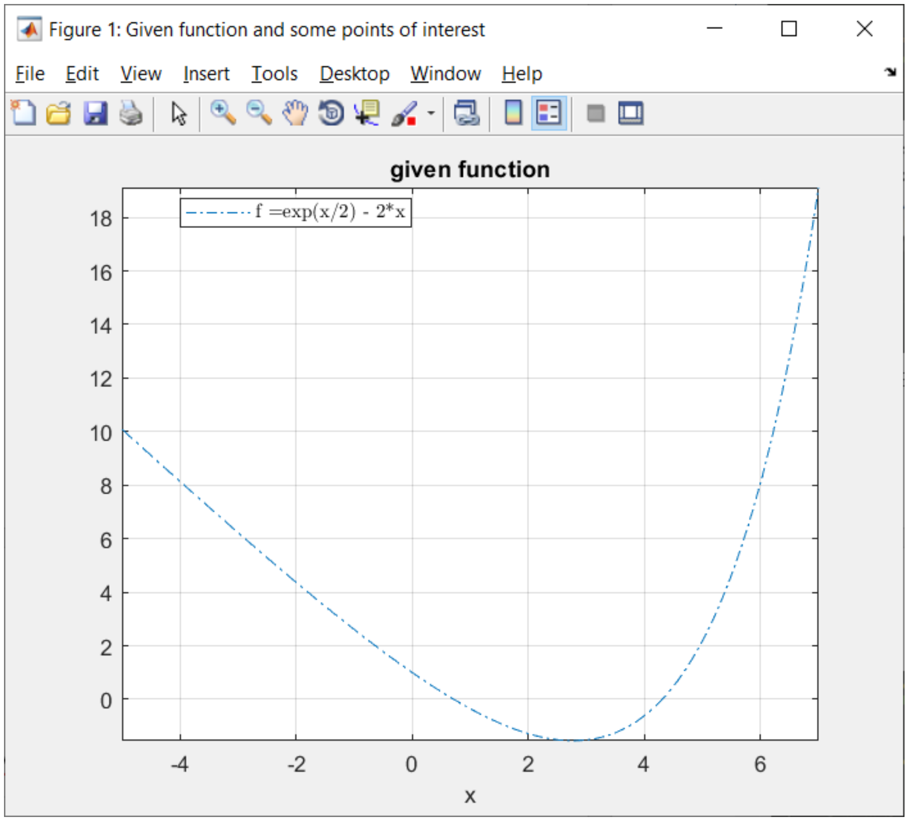
```matlab

clc;clear;close all


%assign interval and some intial values to apply bracketting algorithms on
bnd_low = 0;
bnd_up = 2;
tol_lvl=1e-2;
iter_max=20;


%assign funciton to handle for easier manipulation
syms x
f = @(n) my_f(x);


%plot the function before applying the algorithm

fprintf('Given Function: f(x)= %s\n',f(x))
fprintf('Provided tolerance level and interval: %2.0E, [%3.2f,%3.2f]\n\n',tol_lvl,bnd_low,bnd_up)

n=linspace(bnd_low-5,bnd_up+5,100);
figure('Name', 'Given function and some points of interest');
plot(n,my_f(n),'-.'), grid
axis([bnd_low-5 bnd_up+5 -inf, inf])
xlabel('x');
l=legend(strcat('f = ',char(f(x))),'Location', 'best');
set(l, 'Interpreter', 'latex')
title('given function')



%applying the bisection algorithm on given function in given interval
bisectionM(x, f, bnd_low, bnd_up, tol_lvl, iter_max);



%applying the false position method on given function in given interval
fprintf("\n\n\n")
falsePositionM(x, f, bnd_low, bnd_up, tol_lvl, iter_max);
```

The function itself is actually stored in a separate function script which can be used to input any other function as well to run the algorithm on.

```matlab
function y = my_f(x)
% input function here
y = exp(x./2)-(2.*x);
end
```

The function is first plotted to give a visual idea of the function under examination.

For the given function, it is given below. Two roots can be seen in the vicinity of x=3. But the provided interval excludes the latter root.



Then the first method `bisectionM` is called with all the relevant info input as its argument. Afterwards the given function is solved by calling the `falsePositionM`.

Bisection Method

It begins by some initializations. Where `Ea` is approximated percent error, `R` is the estimated root at each iteration. And the lower and upper bounds have been renamed to `L` and `U` respectively for convenience.

```
function R = bisectionM(x, f, bnd_low, bnd_up, Es, k_max)

fprintf('\n***INITIATING BISECTION METHOD***\n\n')

tic % start counting time

% rename some variables
L=bnd_low;
U=bnd_up;

% initializing some parameters
k=1;
Ea=[];
R=[];

```

```
% form table header in output showing each values at each iteration
disp('iter    L          R          U          f(L)        f(R)        f(U)        test      Ea')
figure('Name', 'Iterations'); hold on


% undergo bisection until termination criteria are met, and root is found, hopefully.
while k<k_max

    R_old=R;
    R=(L+U)/2;

    if(R~=0)
        Ea=abs((R-R_old)/R)*100;
    end

    test=subs(f,L)*subs(f,R);

    % form table in output showing each values at each iteration
    fprintf('%3i %10.4f %10.4f %10.4f %10.4f %10.4f %10.4f %10.4f %10.4f\n',...
        k,L,R,U,subs(f,L),subs(f,R),subs(f,U), test, Ea)

    if(test<0)
        U=R;
    elseif(test>0)
        L=R;
    else
        Ea=0;
    end
    if(Ea<Es)
        fprintf('prespecified percent tolerance passed: %2.4E\n',Ea);
        break;
    end
    if(k>=k_max)
        fprintf('max iterations reached: %3i\n',k_max);
        break;
    end

```

The main algorithm is the while loop which will run until max iterations is reached.

R is determined as the midpoint of `L` and `U`. the variable test determines sign change. And the subsequent `if-elseif-else` chain statements compare results and act accordingly.

The approximate error is determined at each iteration, but included under an `if`-check in case the root x value is ever zero.

Two termination criteria are included: when the approximate error `Ea` passes the prespecified tolerance, and when the algorithm iterates longer than is allowed.

Before the loop and during it, print statements are included to form a table of values at each iteration in the output window.

Also included in the loop is some instructions to plot `x` and `f(x)` values at each iteration. To see how much they oscillate before settling down on the final root estimate.

Time is measured until the end of the loop.

```
    %plot x, f values at each iteration
    itr_Z(k+1) = R; %#ok<AGROW>
    itr_f(k+1)=subs(f,R); %#ok<AGROW>

    subplot(2,1,1)
    line([k k+1], [itr_Z(k),itr_Z(k+1)]),
    ylabel('x'),legend('x_k'),xlim([0,k])
    subplot(2,1,2)
    line([k k+1], [itr_f(k), itr_f(k+1)]),
    ylabel('f(x)'),legend('f(x_k)'),xlim([0,k])
    %draw iteration plot as its iterating; disable to improve performance
    %drawnow;

    k=k+1;

end

fprintf('\n***PROCESS FINISHED***\n')
fprintf("time elapsed: %g seconds.\n", toc) % stop counting time
```

A final plot is made of the function with the estimated root marked.

```
%display results
fprintf('\nRoot estimated: x= %10.4f\n',R)
fprintf('where: f(x) = %10.10f\n',subs(f,R))



%plot found solution by the algorithm
figure('Name', 'Solution via Bisection Method');
hold on
title('Root via Bisection Method')
axis([bnd_low bnd_up -2 2]);
xlabel('x'), grid, legend
fplot(subs(f), 'DisplayName', 'f') %plot f
plot(R,subs(f,R),'Marker','diamond', 'Color', 'k'); %plot root point
text(R,subs(f,R)+0.2,num2str(R,4),'Color', 'k') %plot root value
hold off
l=legend('f', 'root');
set(l, 'Interpreter', 'latex')
```

When the algorithm is run, the result is displayed in the output window shown in the next page. In the value table we can see that at each iteration, `R` converges at a particular value and `f(R)` approaches zero. The relative error reduces down below the tolerance specified, thereby terminating the process. The root approximated, up to 4 significant figures is at

$$x = 0.7148$$

The function value at this point is not entirely zero, but within the limits specified it is the best approximation.

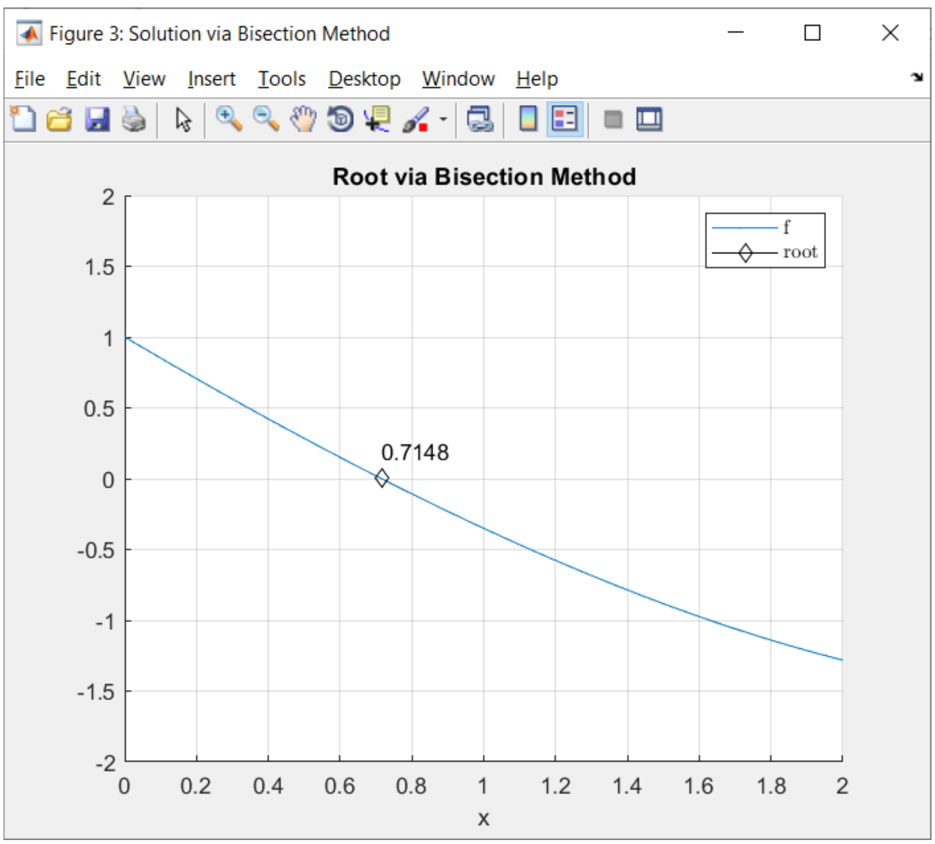
We take note of the iterations, and time taken to reach this value:

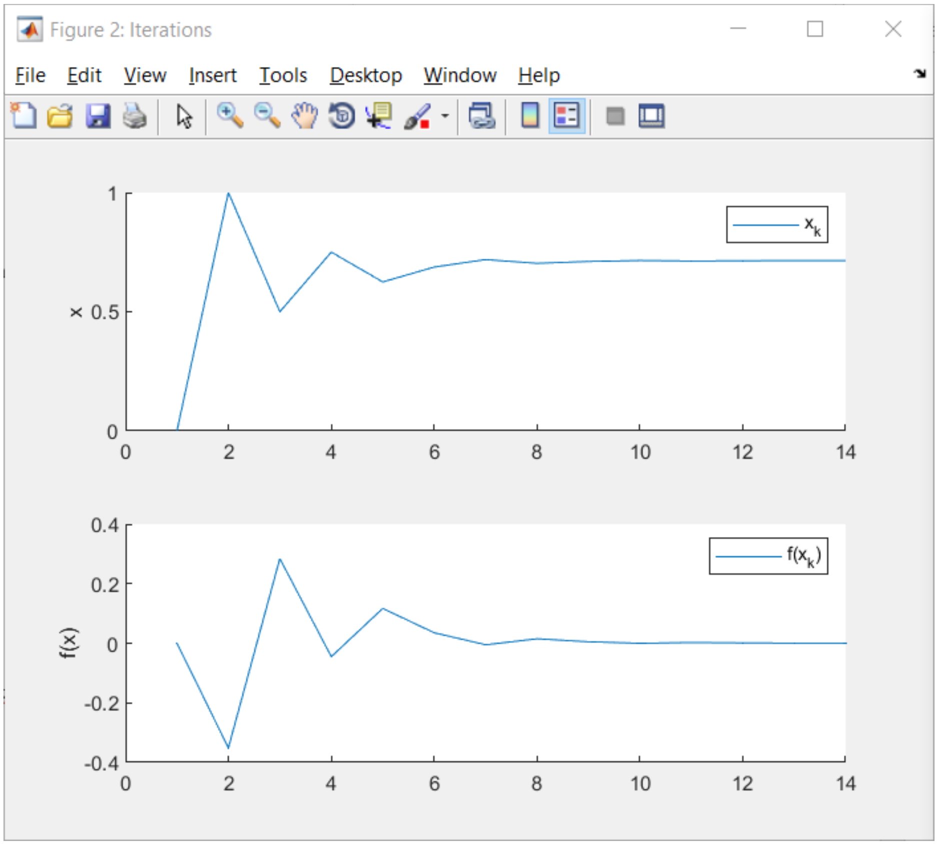
$$k = 15$$

$$t = 1.22372\ s$$

Command Window

```
Given Function: f(x)= exp(x/2) - 2*x
Provided tolerance level and interval: 1E-02, [0.00,2.00]


***INITIATING BISECTION METHOD***

iter    L          R          U          f(L)       f(R)       f(U)       test       Ea
  1     0.0000     1.0000     2.0000     1.0000    -0.3513    -1.2817    -0.3513
  2     0.0000     0.5000     1.0000     1.0000     0.2840    -0.3513     0.2840   100.0000
  3     0.5000     0.7500     1.0000     0.2840    -0.0450    -0.3513    -0.0128    33.3333
  4     0.5000     0.6250     0.7500     0.2840     0.1168    -0.0450     0.0332    20.0000
  5     0.6250     0.6875     0.7500     0.1168     0.0352    -0.0450     0.0041     9.0909
  6     0.6875     0.7188     0.7500     0.0352    -0.0051    -0.0450    -0.0002     4.3478
  7     0.6875     0.7031     0.7188     0.0352     0.0150    -0.0051     0.0005     2.2222
  8     0.7031     0.7109     0.7188     0.0150     0.0050    -0.0051     0.0001     1.0989
  9     0.7109     0.7148     0.7188     0.0050    -0.0000    -0.0051    -0.0000     0.5464
 10     0.7109     0.7129     0.7148     0.0050     0.0025    -0.0000     0.0000     0.2740
 11     0.7129     0.7139     0.7148     0.0025     0.0012    -0.0000     0.0000     0.1368
 12     0.7139     0.7144     0.7148     0.0012     0.0006    -0.0000     0.0000     0.0684
 13     0.7144     0.7146     0.7148     0.0006     0.0003    -0.0000     0.0000     0.0342
 14     0.7146     0.7147     0.7148     0.0003     0.0001    -0.0000     0.0000     0.0171
 15     0.7147     0.7148     0.7148     0.0001     0.0000    -0.0000     0.0000     0.0085
prespecified percent tolerance passed: 8.5390E-03

***PROCESS FINISHED***
time elapsed: 1.22372 seconds.

Root estimated: x=     0.7148
where: f(x) = 0.0000298134
```





False-Position Method

Being a bracketing method as well, the algorithm for the False-Position method is the same, except for the root estimation formula. It is no longer simply the midpoint of the interval but based on a line joining the two intervals. The False-Position formula is:

$$x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)}$$

This facilitates better root estimates.

When the algorithm is run, the result is shown in the next page. The same root position is found, but at much better speed. Where:

$$k = 6$$

$$t = 0.437848\ s$$

It converged at the root much faster. At 2nd iteration already, the approximate error jumped down to 20% then 1.7% in the next.

```
%run algorithm loop until termination criteria are met, and root is found, hopefully.
while k<k_max


    R_old=R;
    R=double(U-((fU*(L-U))/(fL-fU)));
    fR=subs(f,R);

    if(R~=0)
        Ea=abs((R-R_old)/R)*100;
    end

    test=fL*fR;

    % form table in output showing each values at each iteration
    fprintf('%3i %10.4f %10.4f %10.4f %10.4f %10.4f %10.4f %10.4f %10.4f\n',...
        k,L,R,U,fL,fR,fU, test, Ea)

    if(test<0)
        U=R;
        fU=subs(f,U);

    elseif(test>0)
        L=R;
        fL=subs(f,L);

    else
        Ea=0;
    end
    if(Ea<Es)
        fprintf('prespecified percent tolerance passed: %2.4E\n',Ea);
        break;
    end
    if(k>=k_max)
        fprintf('max iterations reached: %3i\n',k_max);
        break;
    end

```

```
***INITIATING FALSE-POSITION METHOD***

iter    L          R          U          f(L)       f(R)       f(U)       test       Ea
  1     0.0000     0.8765     2.0000     1.0000    -0.2030    -1.2817    -0.2030
  2     0.0000     0.7286     0.8765     1.0000    -0.0177    -0.2030    -0.0177    20.3047
  3     0.0000     0.7159     0.7286     1.0000    -0.0014    -0.0177    -0.0014     1.7686
  4     0.0000     0.7149     0.7159     1.0000    -0.0001    -0.0014    -0.0001     0.1447
  5     0.0000     0.7148     0.7149     1.0000    -0.0000    -0.0001    -0.0000     0.0118
  6     0.0000     0.7148     0.7148     1.0000    -0.0000    -0.0000    -0.0000     0.0010
prespecified percent tolerance passed: 9.5755E-04

***PROCESS FINISHED***
time elapsed: 0.437848 seconds.

Root estimated: x=     0.7148
where: f(x) = -0.0000007788
>>
```

From the value table we observe that at every iteration L remains unchanged, hence interval is narrowed in this region. It remains stuck as the lower bracket throughout. By adding a snippet that prevents this, this method might find root even faster.

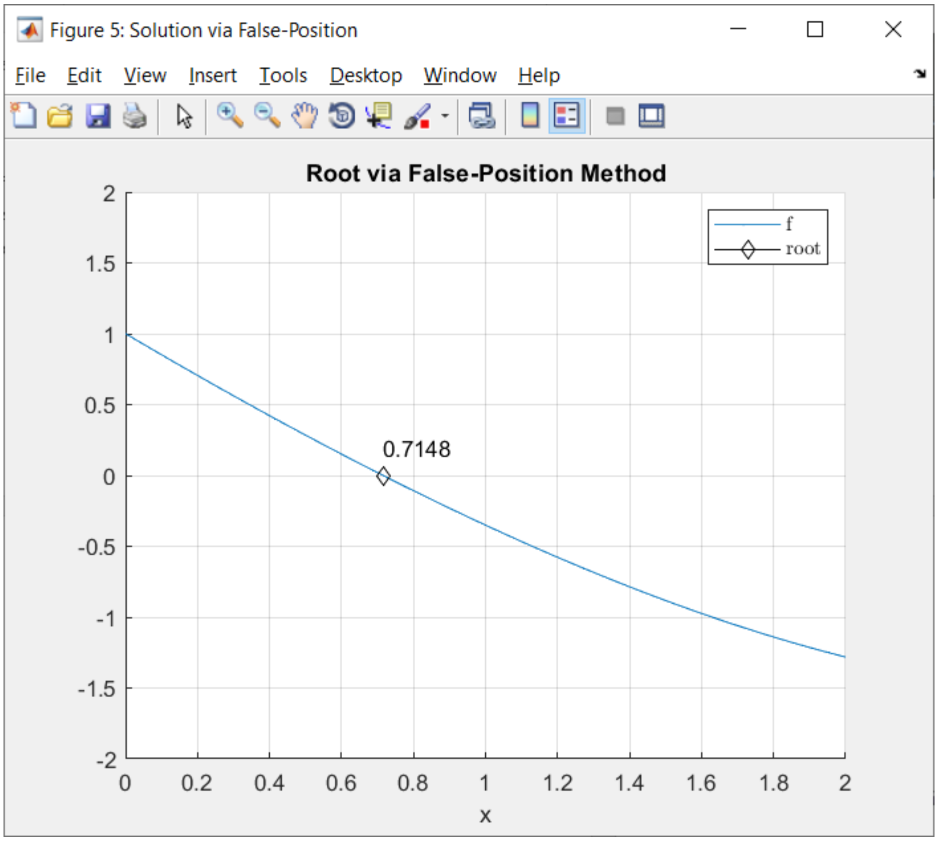


Figure 5: Solution via False-Position
File Edit View Insert Tools Desktop Window Help
Root via False-Position Method
f
root
0.7148
x

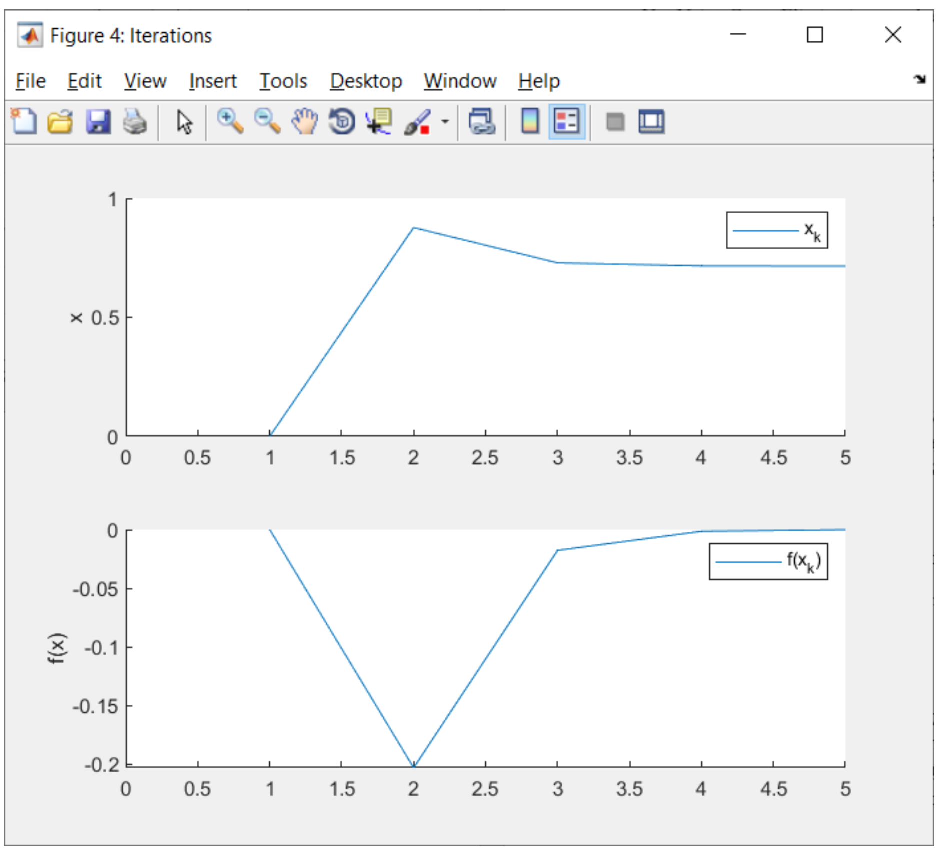


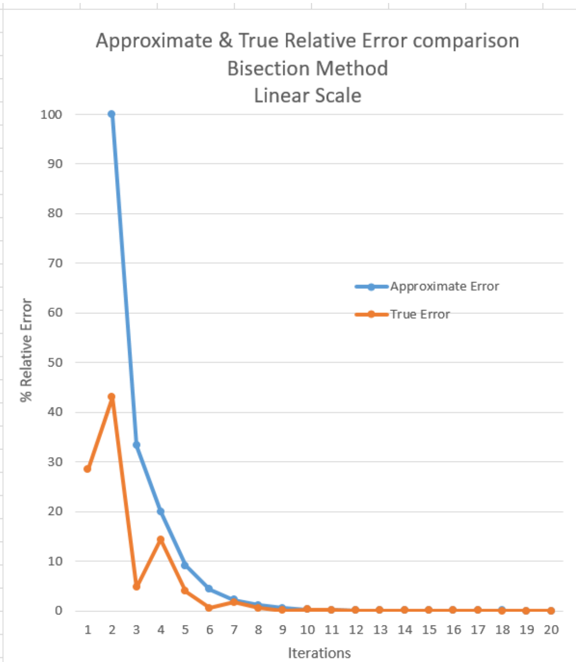
Figure 4: Iterations
File Edit View Insert Tools Desktop Window Help
x
x_k
f(x)
$f(x_k)$

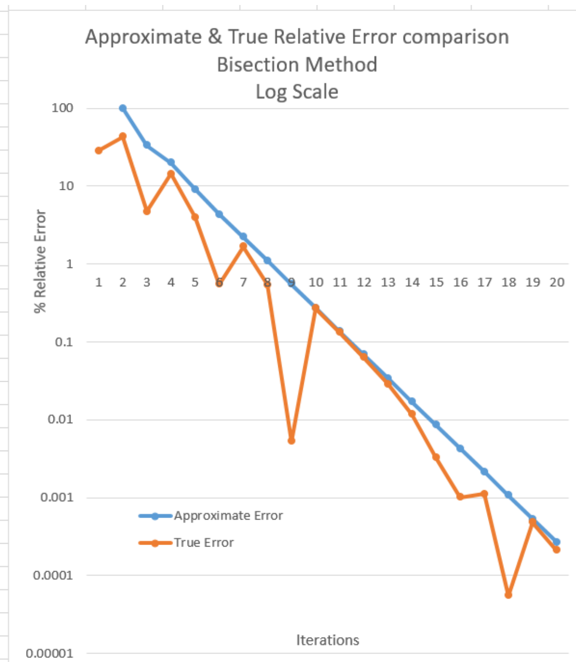
Excel Results

$$f(x) = e^{\frac{x}{2}} - 2x \qquad x = 0.714806\ldots$$

iteration	l	r	u	f(l)	f(r)	f(u)	f(l)*f(r)	f(u)*f(r)	Ea	Et
1	0.0000	1.0000	2.0000	1.0000	-0.3513	-1.2817	-0.3513	0.4502		28.5194
2	0.0000	0.5000	1.0000	1.0000	0.2840	-0.3513	0.2840	-0.0998	100.0000	42.9612
3	0.5000	0.7500	1.0000	0.2840	-0.0450	-0.3513	-0.0128	0.0158	33.3333	4.6925
4	0.5000	0.6250	0.7500	0.2840	0.1168	-0.0450	0.0332	-0.0053	20.0000	14.3690
5	0.6250	0.6875	0.7500	0.1168	0.0352	-0.0450	0.0041	-0.0016	9.0909	3.9718
6	0.6875	0.7188	0.7500	0.0352	-0.0051	-0.0450	-0.0002	0.0002	4.3478	0.5487
7	0.6875	0.7031	0.7188	0.0352	0.0150	-0.0051	0.0005	-0.0001	2.2222	1.6613
8	0.7031	0.7109	0.7188	0.0150	0.0050	-0.0051	0.0001	0.0000	1.0989	0.5441
9	0.7109	0.7148	0.7188	0.0050	0.0000	-0.0051	0.0000	0.0000	0.5464	0.0053
10	0.7109	0.7129	0.7148	0.0050	0.0025	0.0000	0.0000	0.0000	0.2740	0.2687
11	0.7129	0.7139	0.7148	0.0025	0.0012	0.0000	0.0000	0.0000	0.1368	0.1315
12	0.7139	0.7144	0.7148	0.0012	0.0006	0.0000	0.0000	0.0000	0.0684	0.0631
13	0.7144	0.7146	0.7148	0.0006	0.0003	0.0000	0.0000	0.0000	0.0342	0.0289
14	0.7146	0.7147	0.7148	0.0003	0.0001	0.0000	0.0000	0.0000	0.0171	0.0118
15	0.7147	0.7148	0.7148	0.0001	0.0000	0.0000	0.0000	0.0000	0.0085	0.0033
16	0.7148	0.7148	0.7148	0.0000	0.0000	0.0000	0.0000	0.0000	0.0043	0.0010
17	0.7148	0.7148	0.7148	0.0000	0.0000	0.0000	0.0000	0.0000	0.0021	0.0011
18	0.7148	0.7148	0.7148	0.0000	0.0000	0.0000	0.0000	0.0000	0.0011	0.0001
19	0.7148	0.7148	0.7148	0.0000	0.0000	0.0000	0.0000	0.0000	0.0005	0.0005
20	0.7148	0.7148	0.7148	0.0000	0.0000	0.0000	0.0000	0.0000	0.0003	0.0002

BISECTION METHOD

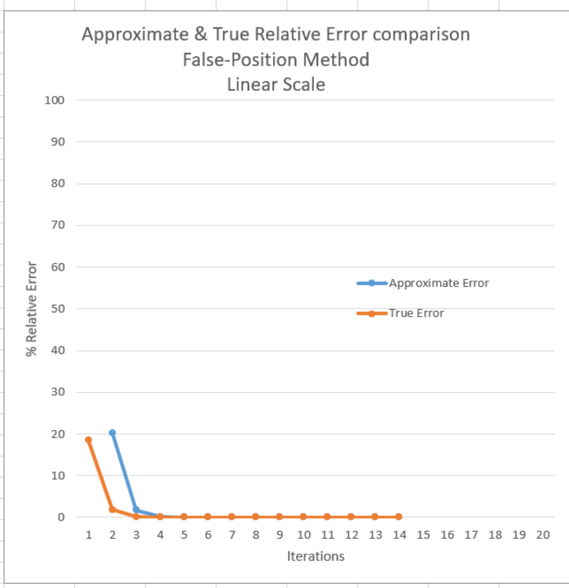


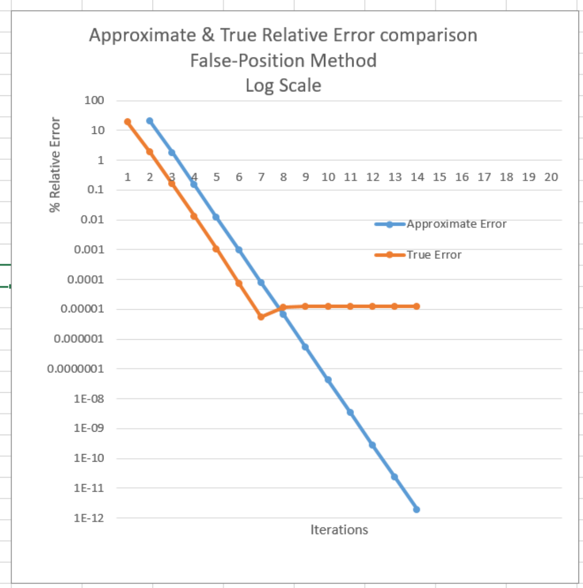


$$f(x) = e^{\frac{x}{2}} - 2x \qquad x = 0.714806 \ldots$$

iteration	l	r	u	f(l)	f(r)	f(u)	f(l)*f(r)	f(u)*f(r)	Ea	Et
1	0.0000	0.8765	2.0000	1.0000	-0.2030	-1.2817	-0.2030	0.2602		18.4507
2	0.0000	0.7286	0.8765	1.0000	-0.0177	-0.2030	-0.0177	0.0036	20.3047	1.8923
3	0.0000	0.7159	0.7286	1.0000	-0.0014	-0.0177	-0.0014	0.0000	1.7686	0.1572
4	0.0000	0.7149	0.7159	1.0000	-0.0001	-0.0014	-0.0001	0.0000	0.1447	0.0128
5	0.0000	0.7148	0.7149	1.0000	0.0000	-0.0001	0.0000	0.0000	0.0118	0.0010
6	0.0000	0.7148	0.7148	1.0000	0.0000	0.0000	0.0000	0.0000	0.0010	0.0001
7	0.0000	0.7148	0.7148	1.0000	0.0000	0.0000	0.0000	0.0000	0.0001	0.0000
8	0.0000	0.7148	0.7148	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
9	0.0000	0.7148	0.7148	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
10	0.0000	0.7148	0.7148	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
11	0.0000	0.7148	0.7148	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
12	0.0000	0.7148	0.7148	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
13	0.0000	0.7148	0.7148	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
14	0.0000	0.7148	0.7148	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
15	0.7148	#DIV/0!	0.7148	0.0000	#DIV/0!	0.0000	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!
16	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!
17	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!
18	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!
19	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!
20	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!

FALSE-POSITION METHOD





Part 2

QUESTION:

Use Naive Gauss Elimination to solve the following equation system.

$$\begin{aligned} 8x_1+2x_2-2x_3&=-2\\ 10x_1+2x_2+4x_3&=4\\ 12x_1+2x_2+2x_3&=6 \end{aligned}$$

Write "Matlab" codes to solve the question. Use four significant figures. Don't forget to check if the system is ill conditioned. Check if the system requires pivoting.

Add your codes to your solution. Does the system have unique solution? Explain.

ANSWER:

Introduction

Naive Gauss Elimination is a systematic algorithm for algebraic elimination of unknowns. The equations form a system of unknowns in a matrix form. A is the coefficient matrix, B the matrix formed from the R.H.S of the equations system. X is the solution matrix.

$$\boldsymbol{A\cdot X=B}$$

A and B combine to give the augmented matrix $[\boldsymbol{A}|\boldsymbol{B}]=\boldsymbol{Aug}.$

This algorithm consists of the two general steps of elimination, i.e. forward elimination and backward substitution.

Basic Algorithm

In my attempt at coding the algorithm, the first lines include initial values. First of all, the two matrices `A` and `B`, that form the linear equation systems are input. They are concatenated to form the augmented matrix `Aug.` matrix size `n` is determined for latter uses. And a zero matrix `X` is initialized to add the solution to.

```
clc;clear;close all

% Change Numerical Format to decimal (short) or rational (rat) representation
format short

% GIVEN MATRICES
fprintf("GIVEN MATRICES\n");
A= [8 2 -2;10 2 4;12 2 2]              % Coefficient Matrix A
B= [-2;4;6]                            % equation RHS Column Matrix B


% INITIALIZING SOME VARIABLES
n = length(B);                         % Matrix Size n
X=zeros(n,1);                          % Zero Matrix in size nx1 to store solutions
Aug=[A B];                             % Concatenating A and B to form Augmented Matrix
fprintf("\nMatrix Size: n= %i\n", n);
fprintf("\nResulting Augmented Matrix\n"); Aug
```

The output of this section is as follows.

```
GIVEN MATRICES

A =

     8     2    -2
    10     2     4
    12     2     2


B =

    -2
     4
     6


Matrix Size: n= 3

Resulting Augmented Matrix

Aug =

     8     2    -2    -2
    10     2     4     4
    12     2     2     6
```

The actual algorithm begins now. Forward Elimination is the first stage. In the innermost loop a `factor` is determined that is to be multiplied to each entries of the pivot row, and later subtracted from the subsequent row entries, hence eliminating entry below the pivot element. Elimination proceeds as levels, until the augmented matrix achieves an upper triangular form.

```
% FORWARD ELIMINATION LOOP
fprintf("\n\n\n\n\n***INITIATING FORWARD ELIMINATION STAGE***\n");
for k=1:n-1                               % for each kth column from 1 to n-1.

    fprintf("\n\n\n     ELIMINATION LEVEL %i of %i: \n", k, n-1);
    for i=k+1:n                           % for each ith row from k+1 till n. i.e. start from 2nd row till last
        factor = Aug(i,k)/Aug(k,k);       % determine the factor for each ith row. i.e. a_(i,k) divided by pivot element
        fprintf("\nelimination factor for level %i,row %i: %s", k, i, strtrim(rats(factor)));
        fprintf("\nAugmented Matrix after row elimination:\n");
        Aug(i,:) = Aug(i,:) - factor*Aug(k,:)     % undergo elimination in each entry in ith row
    end

end

% FINAL RESULTING AUGMENTED MATRIX AFTER ELIMINATION
fprintf("\n\n\n***END OF ELIMINATION STAGE. RESULTING AUGMENTED MATRIX IN UPPER TRIANGULAR FORM***\n");
Aug
```

```
***INITIATING FORWARD ELIMINATION STAGE***



     ELIMINATION LEVEL 1 of 2:

elimination factor for level 1,row 2: 5/4
Augmented Matrix after row elimination:

Aug =

    8.0000    2.0000   -2.0000   -2.0000
         0   -0.5000    6.5000    6.5000
   12.0000    2.0000    2.0000    6.0000


elimination factor for level 1,row 3: 3/2
Augmented Matrix after row elimination:

Aug =

    8.0000    2.0000   -2.0000   -2.0000
         0   -0.5000    6.5000    6.5000
         0   -1.0000    5.0000    9.0000



     ELIMINATION LEVEL 2 of 2:

elimination factor for level 2,row 3: 2
Augmented Matrix after row elimination:

Aug =

    8.0000    2.0000   -2.0000   -2.0000
         0   -0.5000    6.5000    6.5000
         0         0   -8.0000   -4.0000



***END OF ELIMINATION STAGE. RESULTING AUGMENTED MATRIX IN UPPER TRIANGULAR FORM***

Aug =

    8.0000    2.0000   -2.0000   -2.0000
         0   -0.5000    6.5000    6.5000
         0         0   -8.0000   -4.0000
```

The 2nd stage, Backward Substitution then begins. The `nth` variable is first determined as can easily be done from the upper triangular form. Then as apparent from the name of this stage, the nth variable just found is input in the previous row, to find the `(n-1)th` variable. And so on, until the solution matrix `X` is completely determined.

```
% BACKWARD SUBSTITUTION LOOP
fprintf("\n\n\n\n\n***INITIATING BACKWARD SUBSTITUTION STAGE***\n");

X(n)= Aug(n, n+1)/Aug(n,n);
fprintf("\n solution %i: %4.4f", n, X(n,1));    % nth solution
%X(n,1)

for k=n-1:-1:1                          % for each (n-1)th row backward to 1.
    X(k) = (Aug(k, n+1)-Aug(k,k+1:n)*X(k+1:n))/Aug(k,k);    % undergo back substitution
    fprintf("\n solution %i: %4.4f", k, X(k,1));
end


% FINAL SOLUTION MATRIX
fprintf("\n\n***END OF SUBSTITUTION STAGE. RESULTING SOLUTION***\n");
X
```

```
***INITIATING BACKWARD SUBSTITUTION STAGE***

 solution 3: 0.5000
 solution 2: -6.5000
 solution 1: 1.5000

***END OF SUBSTITUTION STAGE. RESULTING SOLUTION***

X =

    1.5000
   -6.5000
    0.5000

>>
```

Hence, for this particular case, through this algorithm, the solution found is as given above, where:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1.500 \\ -6.500 \\ 0.5000 \end{bmatrix}$$

System Singularity Check

The method being "Naïve", there aren't many error detection built in it. But to add a few, one would be to check if the system is singular. A singular system has a determinant of zero. And the determinant of an upper triangular matrix is most easily found by the product of its principle diagonal. So the following snippet could be added for the check:

```
% DETERMINE DETERMINANT TO CHECK FOR SINGULARITY. SINGULAR IF DET=0
fprintf("\n\n\nDIAGONAL ENTRY DETERMINANT CHECK FOR SINGULARITY");
det=1;
for p=1:n
    det=det*Aug(p,p);
end
det
if(det==0)
    fprintf("\nDETERMINANT IS ZERO. SYSTEM IS SINGULAR. ALGORITHM TERMINATED\n");
    return
end
```

But for our given matrix, this is not the case:

```
DIAGONAL ENTRY DETERMINANT CHECK FOR SINGULARITY
det =

    32
```

Ill-Condition System Check

A system is ill-conditioned if a wide range of solutions approximately satisfies it. Or in other words small changes in the system result in large changes in the solution.

One sign of an ill-conditioned system is its determinant being very close to zero. Which with the determinant check discussed above rules out the possibility of both the system being singular and ill-conditioned. And exactly as the result above shows for our given case, the given system is far from being ill-conditioned.

Another method used specifically for the determination of ill-condition systems is to change the coefficient values of the system by a small amount, and find solutions multiple times. Widely differing solution values shows ill-condition.

To achieve this, I have added a routine for adding slight perturbations to the matrix before the other two loop stages commence. the routine runs `d_max` times, that is initially specified. A random small perturbation matrix called `delta` is randomly generated, then added to `B` column matrix, eventually modifying the augmented matrix `Aug`.

```
% INITIATING MAIN LOOP. OUTER LOOP TO FIND MULTIPLE SAMPLES OF SOLUTION
% WITH SLIGHT PERTURBATION IN INITIAL CONDITION

for d=1:d_max                               % running algorithm d_max times with small changes as a check for ill conditionness


    % ADDING PERTURBATION TO SYSTEM TO CHECK FOR ILL-CONDITIONNESS
    fprintf("\n\n\n\n\n***DELTA PERTURBATION LEVEL %i OF TOTAL %i***\n\n", d, d_max);

    for b=1:n
        delta(b,1) = rand;                  % generating random normalized delta perturbation values at each d iteration
        B_d{d}(b,1) = B(b,1)+delta(b,1);    % modifying set B with them.
    end

    fprintf("where perturbation matrix:"); delta
    fprintf("modified augmented matrix: "); Aug_d{d} = [A B_d{d}]



    % FORWARD ELIMINATION LOOP
    fprintf("\n\n\n\n\n***INITIATING FORWARD ELIMINATION STAGE***\n");
    for k=1:n-1                             % for each kth column from 1 to n-1....

    % FINAL RESULTING AUGMENTED MATRIX AFTER ELIMINATION
    fprintf("\n\n\n***END OF ELIMINATION STAGE. RESULTING AUGMENTED MATRIX IN UPPER TRIANGULAR FORM***\n");
    Aug_d{1,d}




    % BACKWARD SUBSTITUTION LOOP
    fprintf("\n\n\n\n\n***INITIATING BACKWARD SUBSTITUTION STAGE***\n");

    X{d}(n,1)= Aug_d{d}(n, n+1)/Aug_d{d}(n,n);
    fprintf("\n solution %i: %4.4f", n, X{d}(n,1));   % nth solution

    for k=n-1:-1:1                          % for each (n-1)th row backward to 1....


    % FINAL SOLUTION MATRIX
    fprintf("\n\n***END OF SUBSTITUTION STAGE. RESULTING SOLUTION***\n");
    X{1,d}
end


fprintf("\n\n\n***END OF ALL DELTA PERTURBATION SAMPLES***\n\n\n\n\n\n\n");
```

```
***DELTA PERTURBATION LEVEL 1 OF TOTAL 10***

where perturbation matrix:
delta =

    0.0105
    0.1602
    0.1357

modified augmented matrix:
ans =

    8.0000    2.0000   -2.0000   -1.9895
   10.0000    2.0000    4.0000    4.1602
   12.0000    2.0000    2.0000    6.1357
```

For each perturbation level, each of the matrices, B and Aug, with the differing delta perturbations are stored as cell arrays B_d and Aug_d storing each B and Aug.

Variables - B_d

B_d

10x1 cell

	1	2
1	[-1.9895;4.1602;6.1357]	
2	[-1.8108;4.0183;6.1817]	
3	[-1.8980;4.1230;6.0632]	
4	[-1.9845;4.1701;6.0289]	
5	[-1.9259;4.1245;6.1995]	
6	[-1.8965;4.1981;6.0453]	
7	[-1.9204;4.1393;6.0129]	
8	[-1.8505;4.0841;6.1623]	
9	[-1.9241;4.0638;6.1972]	
10	[-1.8564;4.0826;6.0197]	
11		

Variables - Aug_d

B_d Aug_d

10x1 cell

	1	2
1	*3x4 double*	
2	*3x4 double*	
3	*3x4 double*	
4	*3x4 double*	
5	*3x4 double*	
6	*3x4 double*	
7	*3x4 double*	
8	*3x4 double*	
9	*3x4 double*	
10	*3x4 double*	
11		

They're actually 1x10 cell array. Here transposed to 10x1 for better display.

Variables - Aug_d{1, 6}

Aug_d Aug_d{1, 6}

Aug_d{1, 6}

	1	2	3	4	5
1	8	2	-2	-1.8965	
2	0	-0.5000	6.5000	6.5688	
3	0	0	-8	-4.2474	
4					

In the figure right above for example the Aug matrix for the 6th perturbation level is shown.

In the figure to the right we can see the resulting X cell array holding all the solutions at each perturbation.

Notice the values don't differ by much, reinforcing

the fact that the system is well-conditioned

Variables - X

B_d Aug_d X

10x1 cell

	1	2
1	[1.5095;-6.5111;0.5218]	
2	[1.5399;-6.6068;0.4582]	
3	[1.4802;-6.3598;0.5101]	
4	[1.4664;-6.3208;0.5370]	
5	[1.5344;-6.6038;0.4969]	
6	[1.4545;-6.2355;0.5309]	
7	[1.4601;-6.2772;0.5233]	
8	[1.5211;-6.5277;0.4820]	
9	[1.5485;-6.6743;0.4818]	
10	[1.4688;-6.3031;0.5002]	
11		
12		

I have also added a section at the end that aggregates all the solution and finds their mean:

```
% FINDING AVERAGE SOLUTION FROM ALL DELTA PERTURBATION SAMPLES

X_d=zeros(n,1);                          % initializing average nx1 solution matrix

for m=1:d
    for r=1:n
        X_d(r,1)=X_d(r,1)+X{1,m}(r,1);    %find mean of each solution across d samples
    end
end

X_m=X_d./d;

fprintf("\n\nRESULTING MEAN SOLUTION\n"); X_m
```

As can be seen, the mean solution X_m, is close the actual solutions obtained earlier.

```
***END OF ALL DELTA PERTURBATION SAMPLES***




RESULTING MEAN SOLUTION

X_m =

    1.4983
   -6.4420
    0.5042

>>
```

Partial Pivoting

Most of the problems encountered with the Naïve Gauss method occurs when the pivot element is either zero or close to zero. When it is close to zero, there is a large difference between it and the other elements which can lead to round off errors during normalization for instance. Partial Pivoting attempts to tackle it, by preventing small coefficients be pivot elements, by exchanging rows.

Before each forward elimination step, a search is made in the pivot column for the largest coefficient (`big`), then that row (`p`) is brought to the top and the largest coefficient made the pivot element instead.

```

% FORWARD ELIMINATION LOOP
fprintf("\n\n\n\n\n***INITIATING FORWARD ELIMINATION STAGE***\n");
for k=1:n-1                                % for each kth column from 1 to n-1.


    % PARTIAL PIVOTING:
    fprintf("\n\n\n     PARTIAL PIVOTING LEVEL %i of %i: \n", k, n-1);
    [big, p] = max(abs(Aug(k:n,k)));   % finding max entry at kth column. big = max entry, p = which row it is at
    fprintf("\n bigger entry is  %4.4f of  row %i: \n", big, p+k-1);
    L = Aug(k,:);                          % storing row to be replaced in L
    Aug(k,:) = Aug(p+k-1,:);               % replacing row L with B
    Aug(p+k-1,:) = L;                      % placing stored row L where it should be
    fprintf("\nresulting augmented matrix after exchange:"); Aug


    fprintf("\n\n\n     ELIMINATION LEVEL %i of %i: \n", k, n-1);
    for i=k+1:n                            % for each ith row from (k+1)th column till n. i.e. start from 2nd row till last...


end

% FINAL RESULTING AUGMENTED MATRIX AFTER ELIMINATION
fprintf("\n\n\n***END OF ELIMINATION STAGE. RESULTING AUGMENTED MATRIX IN UPPER TRIANGULAR FORM***\n");
Aug

```

```
Aug =

     8     2    -2    -2
    10     2     4     4
    12     2     2     6





***INITIATING FORWARD ELIMINATION STAGE***


     PARTIAL PIVOTING LEVEL 1 of 2:

 bigger entry is  12.0000 of  row 3:

resulting augmented matrix after exchange:
Aug =

    12     2     2     6
    10     2     4     4
     8     2    -2    -2



     ELIMINATION LEVEL 1 of 2:
```

```
Aug =

   12.0000    2.0000    2.0000    6.0000
         0    0.3333    2.3333   -1.0000
         0    0.6667   -3.3333   -6.0000



     PARTIAL PIVOTING LEVEL 2 of 2:

 bigger entry is  0.6667 of  row 3:

resulting augmented matrix after exchange:
Aug =

   12.0000    2.0000    2.0000    6.0000
         0    0.6667   -3.3333   -6.0000
         0    0.3333    2.3333   -1.0000



     ELIMINATION LEVEL 2 of 2:

elimination factor for level 2,row 3: 1/2
Augmented Matrix after row elimination:

Aug =

   12.0000    2.0000    2.0000    6.0000
         0    0.6667   -3.3333   -6.0000
         0         0    4.0000    2.0000



***END OF ELIMINATION STAGE. RESULTING AUGMENTE
```

However, with our given matrix, it is very clear that pivoting is not needed! The coefficients are relatively close in magnitudes, and far from zero.

The result obtained with pivoting is the same:

```
***INITIATING BACKWARD SUBSTITUTION STAGE***

 solution 3: 0.5000
 solution 2: -6.5000
 solution 1: 1.5000

***END OF SUBSTITUTION STAGE. RESULTING SOLUTION***

X =

    1.5000
   -6.5000
    0.5000

>>
```

One way to determine whether pivoting is needed or not before actually commencing it, is adding a scaling routine, where the coefficients are compared and scaled. It is done before actual elimination. And if pivoting is included too, it is done based on the scaled values, and the actual matrix coefficients remain unaltered.

```
% FORWARD ELIMINATION LOOP
fprintf("\n\n\n\n\n***INITIATING FORWARD ELIMINATION STAGE***\n");


% SCALING
for k=1:n
    S(k) = max(abs(Aug(k,1:n)));      % determine the largest entry in kth row. store in S
end
fprintf("SCALING: Largest entry in each row:"); S


for k=1:n-1                           % for each kth column from 1 to n-1.

    for a=k:n
        d(a) = abs(Aug(a,k)/S(a));    % normalize kth column by largest entry
    end
    fprintf("SCALING: scaled column"); d

    % PARTIAL PIVOTING:
    fprintf("\n\n\n     PARTIAL PIVOTING LEVEL %i of %i: \n", k, n-1);
    [big, p] = max(d(k:n));   % finding max entry at kth column. big = max entry, p = which row it is at
    fprintf("\n bigger entry is  %4.4f of  row %i: \n", big, p+k-1);
    L = Aug(k,:);                     % storing row to be replaced in L
    Aug(k,:) = Aug(p+k-1,:);          % replacing row L with B
    Aug(p+k-1,:) = L;                 % placing stored row L where it should be
    fprintf("\nresulting augmented matrix after exchange:"); Aug


    % SCALING
    for l=k:n
        S(l) = max(abs(Aug(l,1:n)));  % determine the largest entry in kth row. store in S
    end
    fprintf("SCALING: Largest entry in each row:"); S


    fprintf("\n\n\n     ELIMINATION LEVEL %i of %i: \n", k, n-1);
    for i=k+1:n                       % for each ith row from (k+1)th column till n. i.e. start from 2nd row till
end

% FINAL RESULTING AUGMENTED MATRIX AFTER ELIMINATION
fprintf("\n\n\n***END OF ELIMINATION STAGE. RESULTING AUGMENTED MATRIX IN UPPER TRIANGULAR FORM***\n");
Aug
```

```
***INITIATING FORWARD ELIMINATION STAGE***
SCALING: Largest entry in each row:
S =

     8    10    12

SCALING: scaled column
d =

    0.2500    0.2000    0.1667




     PARTIAL PIVOTING LEVEL 1 of 2:

 bigger entry is  0.2500 of  row 1:

resulting augmented matrix after exchange:
Aug =

     2     8    -2    -2
     2    10     4     4
     2    12     2     6

SCALING: Largest entry in each row:
S =

     8    10    12
```

all source code available:

https://github.com/az-yugen/Numerical-Analysis