ECSE 343: Numerical Methods in Engineering

Assignment 2

Due Date: 13th March 2023

Student Name: Chenyi Xu

Student ID: 260948311

Please type your answers and write you code in this .mlx script. If you choose to provide the handwritten answers, please scan your answers and include those in SINGLE pdf file.

Please submit this **.mlx** file along with a **PDF** copy of this file.

Note: You can write the function in the appendix section using the provided stencils.

In this assignment you may use the backslash operator, '\' to solve systems such as Ax=b.

Question 1: **Nonlinear Equations for univariate case.**

a) Bisection Method is used for finding the roots of a continuous function, $f(x)$, given endpoints; a, b with $f(a).f(b) < 0$. The interval $[a\ b]$ contains a root, x_r, because $f(a)$ and $f(b)$ have opposite signs.

Bisection method bisects the given interval at the midpoint, $c = \frac{a+b}{2}$ and then chooses a new interval based such that end of point of interval have opposite signs, i.e., if $f(a).f(c) < 0$, then the new interval is set to $[a\ c]$, else if $f(c).f(b) < 0$ then the interval is set to $[c\ b]$.

The above procedure is repeated until the following two conditions are simultaneously met,

- The function value at the interval is sufficiently small, i.e., $\|f(c)\|_2 < \epsilon_{\text{tolerance}}$
- The new interval is sufficiently small, i.e., $\|a-b\|_2 < \epsilon_{\text{tolerance}}$

Implement the Bisection Method in the cell below to find the root of $f(x) = x^4 - 2\,x^2 - 4$ in the interval $[-3\ \ 0]$ using $\epsilon_{\text{tolerance}} = 10^{-5}$. **Show the number of iterations the bisection method took to converge.**

Use the cell below to implement your code.

Note: There is no need to write the function for Bisection Method. However, if you wish to implement the function, use the appendix.

```
%implement your bisection method here
f=@(x)x^4-2*x^2-4
```

```
f = function_handle with value:
    @(x)x^4-2*x^2-4
```

```
a = -3;
b = 0;
tol = 10^(-5);

[c,counter]= bisectionMethod(f,a,b,tol)
```

```
c = -1.7989
counter = 22
```

```

```

b) **Newton-Raphson**

Bisection Method requires the given function, $f(x)$ to be continuous, Newton-Raphson requires $f(x)$ to be continuous and differentiable. While the Newton- Raphson method converges faster than the bisection method, however, Newton-Raphson method does not guarantee convergence. Newton-Raphson works by linearising the $f(x)$, at the given initial guess, $x^{(0)}$as shown below

$$f(x) = f(x^{(0)}) + f(x^{(0)})'(x - x^{(0)})$$

Setting the expression on the right to zero, provides an approximation to the root, and we obtain

$$x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f(x^{(0)})'}$$

Then, the function, $f(x)$, is again linearised using approximation, $x^{(1)}$as the initial guess, to obtain new approximation of the root. After k iterations the new approximation for the root becomes,

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f(x^{(k)})'}$$

The above procedure is repeated untill the following two convergence condtions are simultaneously met,

- The function value at new guess point is sufficiently small, i.e., $\|f(x^{(k+1)})\|_2 < \epsilon_{\text{tolerance}}$
- The difference between the two consecutive solutions is sufficiently small, i.e., $\|\Delta x^{(k+1)}\|_2 < \epsilon_{\text{tolerance}}$

where $\Delta x^{(k+1)} = x^{(k+1)} - x^{(k)}$.

Implement the **Newton-Raphson** Method in the cell below to find the root of $f(x) = x^4 - 2\,x^2 - 4$ use intial guess of $x^{(0)} = -3$ using $\epsilon_{\text{tolerance}} = 10^{-5}$. If the convergence is not reached in 100 iterations, quit the algorithm by displaying an error message indicating that Newton-Raphson failed to converge.

Plot the 2-Norm of difference between consecutive solutions $\|x^{(k+1)} - x^{(k)}\|_2$ for each iteration.

```
% write your code here
```

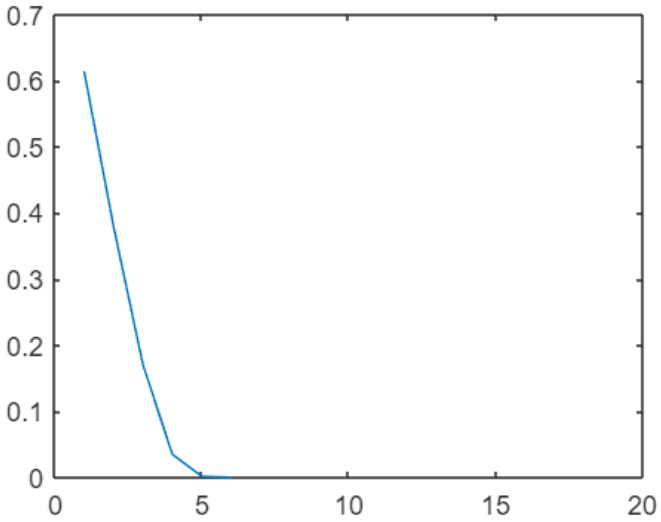
```
f=@(x)x^4-2*x^2-4;
fderivative = @(x)4*x^3 - 4*x;  % write the value of derivatives

nmax = 100;
x0 = -3;
tol = 10^(-5);

[x,deltaX]=newtonraphson(f,fderivative,x0,tol,nmax)
```

```
x = -1.7989
deltaX = 1×6
    0.6146    0.3798    0.1705    0.0348    0.0013    0.0000
```

```
plot(deltaX)
xlim ([0,20]);
```



Question 2: **Nonlinear Equations for N-variables.**

Newton-Raphson Method can be used to solve the system of nonlinear equations of N-variables, $[x_1, x_2, \cdots, x_n]$ shown below,

$$f_1(x_1, x_2, \cdots, x_N) = 0$$

$$f_2(x_1, x_2, \cdots, x_N) = 0$$

$$\vdots$$

$$f_N(x_1, x_2, \cdots, x_N) = 0$$

The above equations can be written as a vector valued function as shown below,

$$\boldsymbol{f}(\boldsymbol{X}) = \begin{bmatrix} f_1(x_1, x_2, \cdots, x_N) \\ f_2(x_1, x_2, \cdots, x_N) \\ \vdots \\ f_N(x_1, x_2, \cdots, x_N) \end{bmatrix}$$

where $\boldsymbol{X} = [x_1, x_2, \cdots, x_N]$ is the vector containing all the variables.

Following the idea of unidimensional Newton-Rapshson, we start with an initial guess $\boldsymbol{X}^{(0)}$ and we use this to linearise the function $\boldsymbol{f}(\boldsymbol{X})$ around $\boldsymbol{X}^{(0)}$

$$\boldsymbol{f}(\boldsymbol{X}) = \ f(\boldsymbol{X}^{(0)}) + \left(\frac{\partial}{\partial \boldsymbol{X}} \boldsymbol{f}(\boldsymbol{X}^{(0)})\right)(\boldsymbol{X} - \boldsymbol{X}^{(0)})$$

Setting the expresson on the left side to zero, we find the approximation of the solution vector. After k iterations the approximation for the solution vector can be written as

$$\boldsymbol{X}^{(k+1)} = \boldsymbol{X}^{(k)} - \left(\frac{\partial}{\partial \boldsymbol{X}} \boldsymbol{f}(\boldsymbol{X}^{(k)})\right)^{-1} \boldsymbol{f}(\boldsymbol{X}^{(k)})$$

where $\boldsymbol{f}(\boldsymbol{X}^{(k)})$ is the vector valued function evaluated at $\boldsymbol{X}^{(k)}$ and $\left(\frac{\partial}{\partial \boldsymbol{X}} \boldsymbol{f}(\boldsymbol{X}^{(k)})\right)$ is the jacobian evaluated at $\boldsymbol{X}^{(k)}$.

The structure of the Jacobian is given by,

$$\frac{\partial}{\partial \boldsymbol{X}} \boldsymbol{f}(\boldsymbol{X}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

The above procedure is repeated untill the following two convergence condtions are simultaneously met,

- The function value at new guess point is sufficiently small, i.e., $\|\boldsymbol{f}(\boldsymbol{X}^{(k+1)})\|_2 < \epsilon_{\text{tolerance}}$
- The difference between the two consecutive solutions is sufficiently small, i.e., $\|\boldsymbol{X}^{(k+1)} - \boldsymbol{X}^{(k)}\|_2 < \epsilon_{\text{tolerance}}$

Find a solution of the following system of three nonlinear equations using Newton-Raphson method.

$x_1^2 + x_2^2 + x_3^2 = 3$

$x_1^2 + x_2^2 - x_3 = 1$

$x_1 + x_2 + x_3 = 3$

Note that this can be expressed as

$$f(X) = \begin{bmatrix} x_1^2 + x_2^2 + x_3^2 - 3 \\ x_1^2 + x_2^2 - x_3 - 1 \\ x_1 + x_2 + x_3 - 3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \qquad \text{where } X = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T$$

a) Write a MATLAB function named *evaluateEquations(x)*, that evaluates the above equation at a given input vector. Use the provided X to check your implementation.

Use the framework of the function provided in the Appendix.

```
% X in this case is an example

X = [2;3;4];

f  = evaluateEquations(X)
```

```
f = 3×1
    26
     8
     6
```

b) Write a MATLAB function named *evaluateJacobian()*, that evaluates the Jacobian of the above equations. Use the provided X to check your implementation.

Use the framework of the function provided in the Appendix.

```
X = [2;3;4];
J  = evaluateJacobian(X)
```

```
J = 3×3
     4     6     8
     4     6    -1
     1     1     1
```

Newton-Raphson

c) Implement the Newton-Raphson method, use the function named *NewtonRaphson()* to write the code for this.

Start with intial guess, $X^{(0)} = [0 \quad 0 \quad 0]^T$ and then find a suitable intial guess. Plot the 2-Norm of difference between consecutive solutions $\|X^{(k+1)} - X^{(k)}\|_2$ for each iteration.

Use the framework of the function provided in the Appendix. In the *NewtonRaphson()* function use the methods implemented in part (a) and (b).

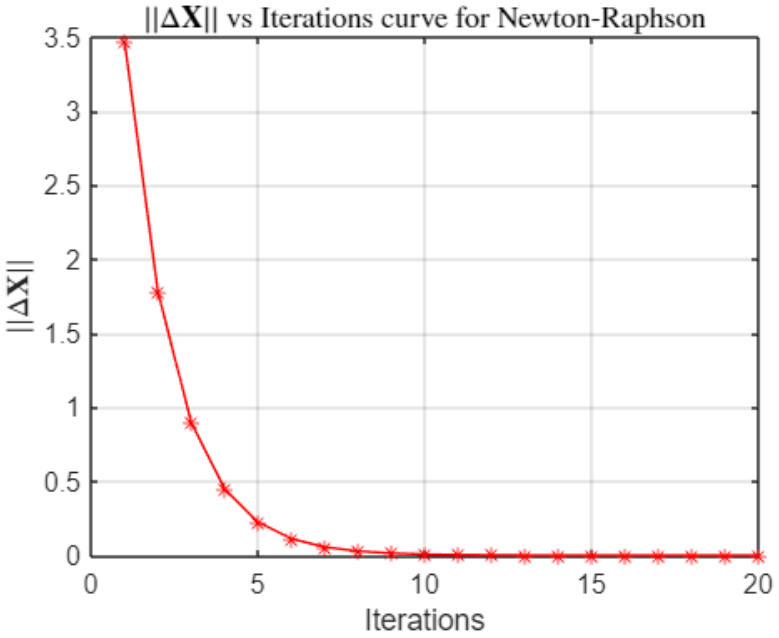
```
% write your code here

func =@evaluateEquations;
Jac =@evaluateJacobian;
Xguess = [0.6;0;0];
tol = 1e-5;
[X,deltaX] = NewtonRaphson(func,Jac,Xguess,tol)
```

```
X = 3×1
```

```
    1.0000
    1.0000
    1.0000
deltaX = 1×20
    3.4756    1.7816    0.8975    0.4488    0.2244    0.1122    0.0561    0.0280 ···
```

```
plot(deltaX,'r-*')
xlabel('Iterations')
ylabel(' $ || \Delta \textbf{X} || $ ', 'Interpreter', 'latex')
grid on
title(' $ || \Delta \textbf{X} || $  vs Iterations curve for Newton-Raphson', 'Interpreter', 'l
```



Newton-Raphson with Broyden's method to approximate Jacobian

d) Suppose you do not have access to the Jacobian of the nonlinear eqautions, use the Broyden's method to compute/ update the Jacobian.

At the $(k+1)^{th}$ iteration of Newton Raphson, the soltuion vector $\boldsymbol{X}$ is given by

$$X^{(k+1)} = X^{(k)} - \left(\frac{\partial}{\partial X} f(X^{(k)})\right)^{-1} f(X^{(k)})$$

let, $J_k = \left(\frac{\partial}{\partial X} f(X^{(k)})\right)$. Using the Broyden's update, J_k can be computed as

$$J_k = J_{k-1} + \frac{\left(f(X^{(k)}) - f(X^{(k-1)}) \right) - J_{k-1} \left(X^{(k)} - X^{(k-1)} \right)}{\| \left(X^{(k)} - X^{(k-1)} \right) \|_2^2} \left(X^{(k)} - X^{(k-1)} \right)^T$$

Write a MATLAB function named *NR_Broyden_Jacobian()*, that evaluates the Jacobian of the above equations.

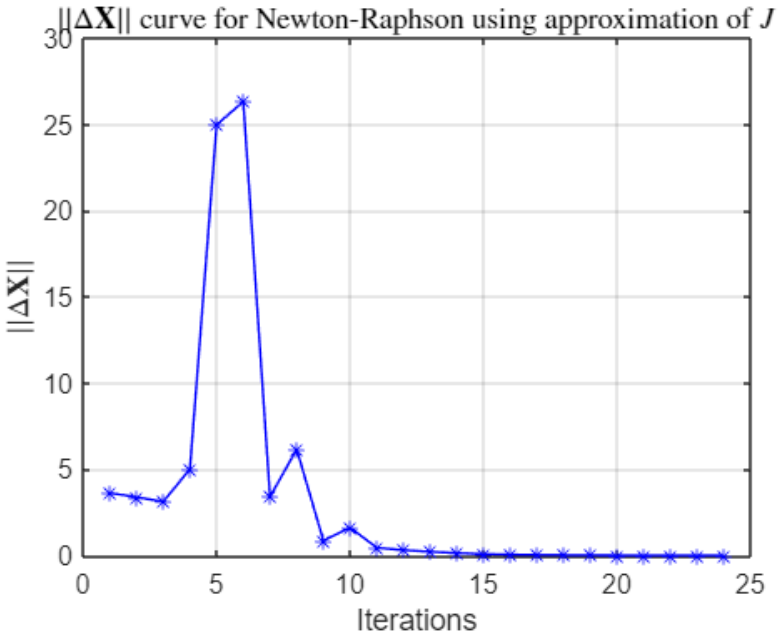
Use the initial guess, $X^{(0)}$ you used in part c). For the intial guess for Jacobian $J_0 = I_{n\times n}$, where $I_{n\times n}$ is the identity matrix and n is the number of equations.

Use the framework of the function provided in the Appendix

```
Xguess = [0.6;0;0];
func =@evaluateEquations;
[Xout,deltaXBroyden]  = NR_Broyden_Jacobian(func, Xguess,1e-5)
```

```
Xout = 3×1
    1.0013
    0.9987
    1.0000
deltaXBroyden = 1×24
    3.6248    3.3825    3.1189    5.0094   24.9756   26.3191    3.4066    6.1539 ···
```

```
plot(deltaXBroyden,'b-*')
xlabel('Iterations')
ylabel(' $ || \Delta \textbf{X} || $ ', 'Interpreter', 'latex')
grid on
title(' $ || \Delta \textbf{X} || $  curve  for Newton-Raphson using approximation of $J$', 'In
```



Newton-Raphson with Broyden's method to approximate Inverse of Jacobian

e) In part d) , you implemented the Broyden's method for updating the Jacobian, in this part use the Broyden's update for computing inverse of Jacobian. The inverse of Jacobian can be computed as

$$J_k^{-1} = J_{k-1}^{-1} + \frac{\left(X^{(k)} - X^{(k-1)} \right) - J_{k-1}^{-1} \left(f(X^{(k)}) - f(X^{(k-1)}) \right)}{\left(X^{(k)} - X^{(k-1)} \right)^T J_{k-1}^{-1} \left(f(X^{(k)}) - f(X^{(k-1)}) \right)} \left(\left(X^{(k)} - X^{(k-1)} \right)^T J_{k-1}^{-1} \right)$$

Write a MATLAB function named *NR_Broyden_InvJacobian()*, that evaluates the Jacobian of the above equations.

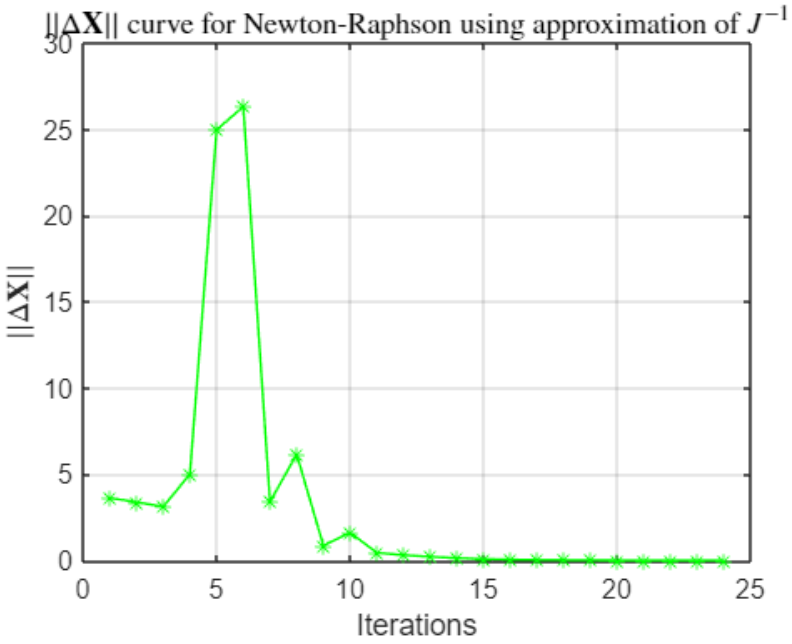
Use the initial guess, $X^{(0)}$ you used in part c). For the intial guess for inverse of Jacobian $J_0^{-1} = I_{n\times n}$, where $I_{n\times n}$ is the identity matrix and n is the number of equations.

Use the framework of the function provided in the Appendix.

```
Xguess = [0.6;0;0];
func =@evaluateEquations;
[Xout,deltaXBroyden]  = NR_Broyden_InvJacobian(func, Xguess,1e-5)
```

```
Xout = 3×1
    1.0013
    0.9987
    1.0000
deltaXBroyden = 1×24
    3.6248    3.3825    3.1189    5.0094   24.9756   26.3191    3.4066    6.1539 ···
```

```
plot(deltaXBroyden,'g-*')
xlabel('Iterations')
ylabel(' $ || \Delta \textbf{X} || $ ', 'Interpreter', 'latex')
grid on
title(' $ || \Delta \textbf{X} || $  curve  for Newton-Raphson using approximation of $J^{-1}$
```



f) Given a nonlinear system with n number of nonlinear equations, shown below,

$$f_1(x_1, x_2, \ldots, x_n) = 3x_1 - 2x_1^2 - 2x_2 + 1 = 0$$

$$f_i(x_1, x_2, \ldots, x_n) = 3x_i - 2x_i^2 - x_{i-1} - 2x_{i+1} + 1 = 0, \quad \text{for } 1 < i < n$$

$$f_n(x_1, x_2, \ldots, x_n) = 3x_n - 2x_n^2 - x_{n-1} + 1 = 0$$

where, x_1, x_2, ..., x_n are the variables.

The nonlinear vector of equations can written as,

$$\boldsymbol{f}(\boldsymbol{X}) = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_i(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The Matlab function to evalaute the above function is named *nlfunction()* and it is provided to you in appendix.

Suppose you do not have access to the jacobian, so you chose to solve the above nonlinear function using Broyden's methods you developed in part (d) and (e). The code cell below uses the Broyden's algorithms you implemented in paer d) and e) to solve the above nonlinear system for different size of the system ranging from $10 \leq n \leq 200$.

Run the cell below and explain the results in Figure Question 2 f.

```
clear all
func1 = @nlfunction;
c = 1
```

```
c = 1
```

```
N = 200
```

```
N = 200
```

```
time_BroydenIJ = zeros(N-10+1,1);
time_BroydenJ = zeros(N-10+1,1);
for I=10:N
Xguess = zeros(I,1);
brIJstart =tic ;
```

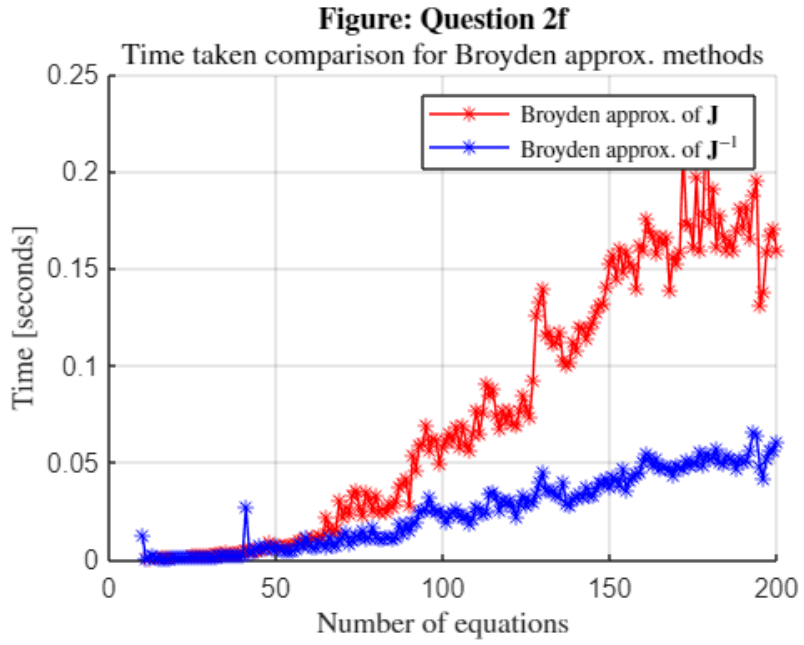
```
[XBrIJ,deltaXBroyden]  = NR_Broyden_InvJacobian(func1,Xguess,1e-6);
time_BroydenIJ(c) = toc(brIJstart) ;

brJstart =tic ;
[XBrJ,deltaXBroyden]  = NR_Broyden_Jacobian(func1,Xguess,1e-6);
time_BroydenJ(c) = toc(brJstart) ;

c =c +1;
end

figure()
clf
hold on
plot(10:N,time_BroydenJ,'r-*')
plot(10:N,time_BroydenIJ,'b-*')

legend('Broyden approx. of $\textbf{J}$' ,'Broyden approx. of $\textbf{J}^{-1}$','Interpreter',
title({'\textbf{Figure: Question 2f}','Time taken comparison for Broyden approx. methods'}, 'In
ylabel('Time [seconds]', 'Interpreter', 'latex')
xlabel('Number of equations', 'Interpreter', 'latex')
grid on
```



% Write your explanation here

X represents the number of equations and y represents the time aken for Broyden's algorithms of Jacobian and Broyden's algorithms of inverse Jacobian. The time taken for Broyden's algorithms for inverse Jacobian is less than Broyden's algorithms for Jacobian. This suggests that for larger systems of equations, using the Broyden

method with an approximation of the inverse Jacobian may be more computationally efficient than using the Broyden method with an approximation of the Jacobian.

Appendix

Function provided for Question 2 part e)

```
function [F] = nlfunction(x)
% x is a colmns vecotr.
% Evaluate the vector function

n = length(x);
F = zeros(n,1);
i = 2:(n-1);
F(i) = (3-2*x(i)).*x(i)-x(i-1)-2*x(i+1) + 1;
F(n) = (3-2*x(n)).*x(n)-x(n-1) + 1;
F(1) = (3-2*x(1)).*x(1)-2*x(2) + 1;
% % % % % Evaluate the Jacobian if nargout > 1
% % % % if nargout > 1
% % % %    d = -4*x + 3*ones(n,1); D = sparse(1:n,1:n,d,n,n);
% % % %    c = -2*ones(n-1,1); C = sparse(1:n-1,2:n,c,n,n);
% % % %    e = -ones(n-1,1); E = sparse(2:n,1:n-1,e,n,n);
% % % %    J = C + D + E;
% % % % end
end
```

Question 1) a)

```
function [x, iterations] = bisectionMethod(f,a,b,tol)

% inputs :
% f is  the function
% ( it is supplied as a handle for the nonlinear  fucniton f)
% in the Q1b it the fiunction f is already defined for you
% a and b: specify the interval
% tol is the desired tolerance

%psudo code
% check is f(a) * f(b) <0
% c = (a+b)/2
% find f(c)
%if f(a) * f(c) < 0, b = c, a= a
% if f(b) * f(c) <0 , a = b, b = c
%Above repeated untile 1. normal value of f(c) < tol
%normal value of (a-b) < tolerabce

if f(a)*f(b) > 0
```

```
    error('f(a) & f(b) must have opposite signs')
end

% intialization
c = (a + b)/2;
new_f = f(c);
iterations = 0;

while norm(new_f) >= tol || norm(a - b) >= tol
    c = (a + b)/2;
    new_f = f(c);
    iterations = iterations + 1;

    if f(a) * f(c) < 0
        b = c;
    end
    if f(b) * f(c) < 0
        a = c;
    end

end


x = c;

%outputs:
% x : is the solution of the nonlinear equation
% iterations is thenumber of iterations your method took to converge.

end
```

Question 1 b)

```
function [x,normdeltaX] = newtonraphson(f, fderiv, x0, tol, itermax)
% inputs :
% f is the function
% (it is supplied as a handle for the nonlinear function f)
% in Q1b, the function f is already defined for you

% fderiv is the derivative of the nonlinear function
% (it is supplied as a handle for the nonlinear function f)

% x0 is the initial guess
% tol is the desired tolerance
% itermax is the maximum number of iterations

% pseudocode
% initialization
% x_(k+1) = x0 - f(x0)/dferiv(x0)
%  if norm(f(x_(k+1)) < tol && change in x_(k+1) < tol, break loop
```

```
% in loop x_(k+1) = xk - f(xk)/dferiv(xk), update xk,
x = x0;
delta = 0;
normdeltaX = [];
for i = 1:itermax
    if i == 100
    error('Newton-Raphson failed to converge within 100 iterations.');
    end
    fx = f(x);
    df = fderiv(x);
    newX = x - fx/df;
    delta = newX - x;
    normdeltaX = [normdeltaX, norm(delta)];

    if norm(f(newX)) < tol && norm(delta) < tol
        break
    end

    x = newX;
end

% outputs:
% x: the solution of the nonlinear equation
% normdeltaX: the norm of deltaX at each iteration
end
```

Question 2) a)

```
function f  = evaluateEquations(x)

 f = zeros(3,1);
 % write your code here

 f(1) = x(1)^2 + x(2)^2 + x(3)^2 -3;
 f(2) = x(1)^2 + x(2)^2 - x(3) -1;
 f(3) = x(1) + x(2) + x(3) -3;

end
```

Question 2) b)

```
function J  = evaluateJacobian(x)
   J = zeros(3,3);

   % write your code here
   J(1,1) = 2*x(1);
   J(1,2) = 2*x(2);
```

```
    J(1,3) = 2*x(3);

    J(2,1) = 2*x(1);
    J(2,2) = 2*x(2);
    J(2,3) = -1;

    J(3,1) = 1;
    J(3,2) = 1;
    J(3,3) = 1;
end
```

Question 2) c)

```
function [Xout,normDeltaX]  = NewtonRaphson(func,Jac,Xguess,tol)
% func is the handle for evaluateEquations().. it is created for you
% Jac i is the handle for evaluateJacobian().. it is created for you

% Xguess is the initial Guess vector :
% tol is the desired tolerance

% Xout is the ouput vector
% normDeltaX stores the norm of the deltaX at each iteration

% write your code here

% Initialization
max_iter = 100;
x_new = Xguess;
x_current = Xguess;
normDeltaX = [];


for d = 1:max_iter
    J = Jac(x_current);
    x_new = x_current - inv(J)*func(x_current);
    normDeltaX = [normDeltaX, norm(x_new - x_current)];

    if norm(func(x_new)) < tol && norm(x_new - x_current) < tol
        break;
    end

    x_current = x_new;
end

% Output
Xout = x_new;
end
```

Appendix: Question 2 d)

```
function [Xout,normDeltaX]  = NR_Broyden_Jacobian(func, Xguess,tol)
% func is the handle for evaluateEquations().. it is created for you
% Xguess is a Column Vector , it is the initial Guess vector :
% tol is the desired tolerance

% Xout is the ouput vector
% normDeltaX stores the norm of the deltaX at each iteration


J_current = eye(size(Xguess,1)); %keep this line
% write your code here

%psudo code
% J_new = J_current + deltaJ
%where deltaJ =( deltaF - J_current *delta )/norm(delta)^2 *delta'
%deltaF = func(Xnew) - func(current)
%x_new = x_current - pinv(J)*func(x_current);
%initialization
max_iter = 100;
x_new = Xguess;
x_current = Xguess;
normDeltaX = []; % norm
J_new = eye(size(Xguess,1));
for k = 1 : max_iter
     if norm(func(x_new)) < tol && norm(x_new - x_current) < tol
        break;
     end
     x_new = x_current - inv(J_current)*func(x_current);
     delta_x = x_new - x_current;
     delta_F = func(x_new) - func(x_current);
     delta_J = (delta_F - J_current*delta_x)/norm(delta_x)^2 * delta_x';
     J_new = J_current + delta_J;

    normDeltaX = [normDeltaX, norm(delta_x)];

     %update fields
     x_current = x_new;
     J_current = J_new;


end
Xout = x_new;
end
```

Appendix: Question 2 e)

```
function [Xout,normDeltaX]  = NR_Broyden_InvJacobian(func, Xguess,tol)
%func is the handle for evaluateEquations().. it is created for you
% Xguess is a Column Vector , it is the initial Guess vector :
```

```
% tol is the desired tolerance

% Xout is the ouput vector
% normDeltaX stores the norm of the deltaX at each iteration

%J_inv_New = J_inv_current + delta_X * J_inv_current*
%delta_F/((delta_X')*J_inv_current*delta_F)*(delta_X' *J_inv_current)

J_inv_current = eye(size(Xguess, 1));
max_iter = 100;
x_new = Xguess;
x_current = Xguess;
delta_X = 0;
normDeltaX = [];

for k = 1:max_iter
    % Check if the desired tolerance has been reached
    if norm(func(x_new)) < tol && norm(x_new - x_current) < tol
        break;
    end

    % Update the guess
    x_new = x_current - J_inv_current * func(x_current);
    delta_X = x_new - x_current;
    delta_F = func(x_new) - func(x_current);
    normDeltaX = [normDeltaX, norm(delta_X)];

    % Update the inverse Jacobian using Broyden's update
    J_inv_New = J_inv_current + ((delta_X - J_inv_current * delta_F) / (delta_X' * J_inv_curren

    % Update fields
    x_current = x_new;
    J_inv_current = J_inv_New;
end

Xout = x_new;
end
```