2018

Semester 2

Lab 3: Non Linear Equations

Connor McDowall
530913386
cmcd398

September 4, 2018

Contents

Listings

List of Figures

1 Questions

1.1 Task 2

1.1.1 Question 1

The newton method finds the roots for $f(x) = x^2 - 1$ and $f(x) = cos(x) + sin(x^2) - 0.5$ in four and seven iterations respectively. The newton method fails to find the root before the maximum number of iterations for $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. The newton method uses the derivative to calculate the new root value. If the derivative is too small, the new root estimated is significantly greater that the current iteration. This was the case for $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ as jumped from a negative function value to a large, positive function value. The combined method uses a combination of the bisection and newton method. The newton method is used first to find a new root estimate. If this estimate falls outside a root range, the bisection method is used instead to find the new root estimate, avoiding extreme leaps. Thereafter, the root bracket is updated. The method continues to iterate until a suitable root is found. The combined method found a root in 4 iterations for $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. (I have excluded the initial root estimates from my iterations).

1.1.2 Question 2

Use parallelization. Set up array of root estimates on a plausible range of values . Next, apply the desired root finding method on each element of the array The desired root finding method is based on the method's properties. Perform the method for each element simulataneously and select the best of the root estimates found. The best root estimates will be the global minima/maxima.

1.2 Task 4

1.2.1 Question 1

The intial root estimate is either too far away from the actual root or the derivative of the function is really small or zero.

1.2.2 Question 2

Use parallelization. Set up matrix of root estimates on a plausible range of values (two to n dimensions) where each column is a different combination of starting points and each row is a different variable in the function set. Next, apply the desired root finding method on each column of the matrix. The desired root finding method is based on the properties you wish to have. Perform the method for each column simulataneously and select the best of the root estimates found. The best root estimates will be the global minima/maxima. This will work for non linear functions with two to n dimensions.

2 Plots

The plots for both Tasks 2 and 4 are on the following pages.

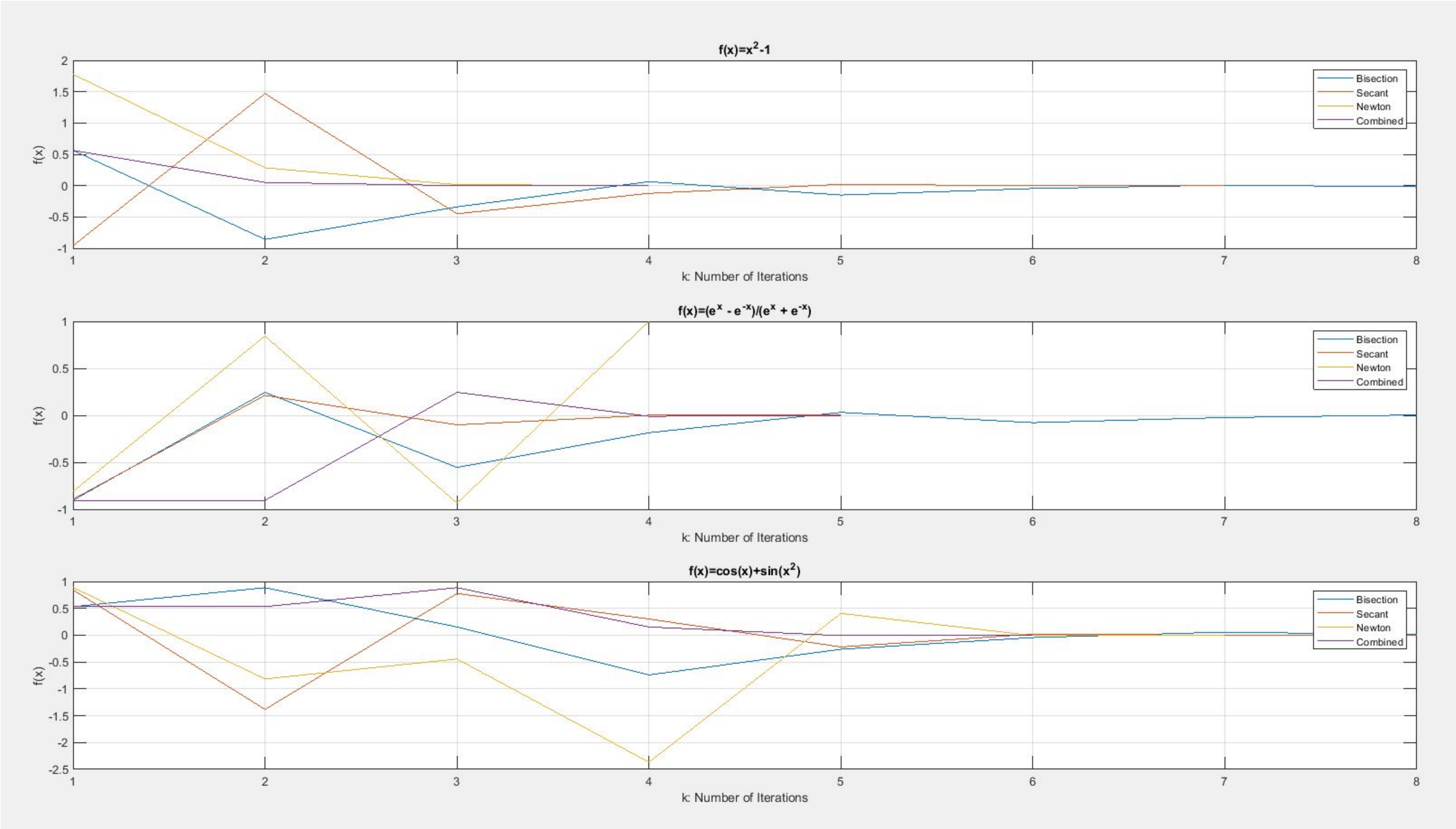

Figure 1: NLE Function Comparison

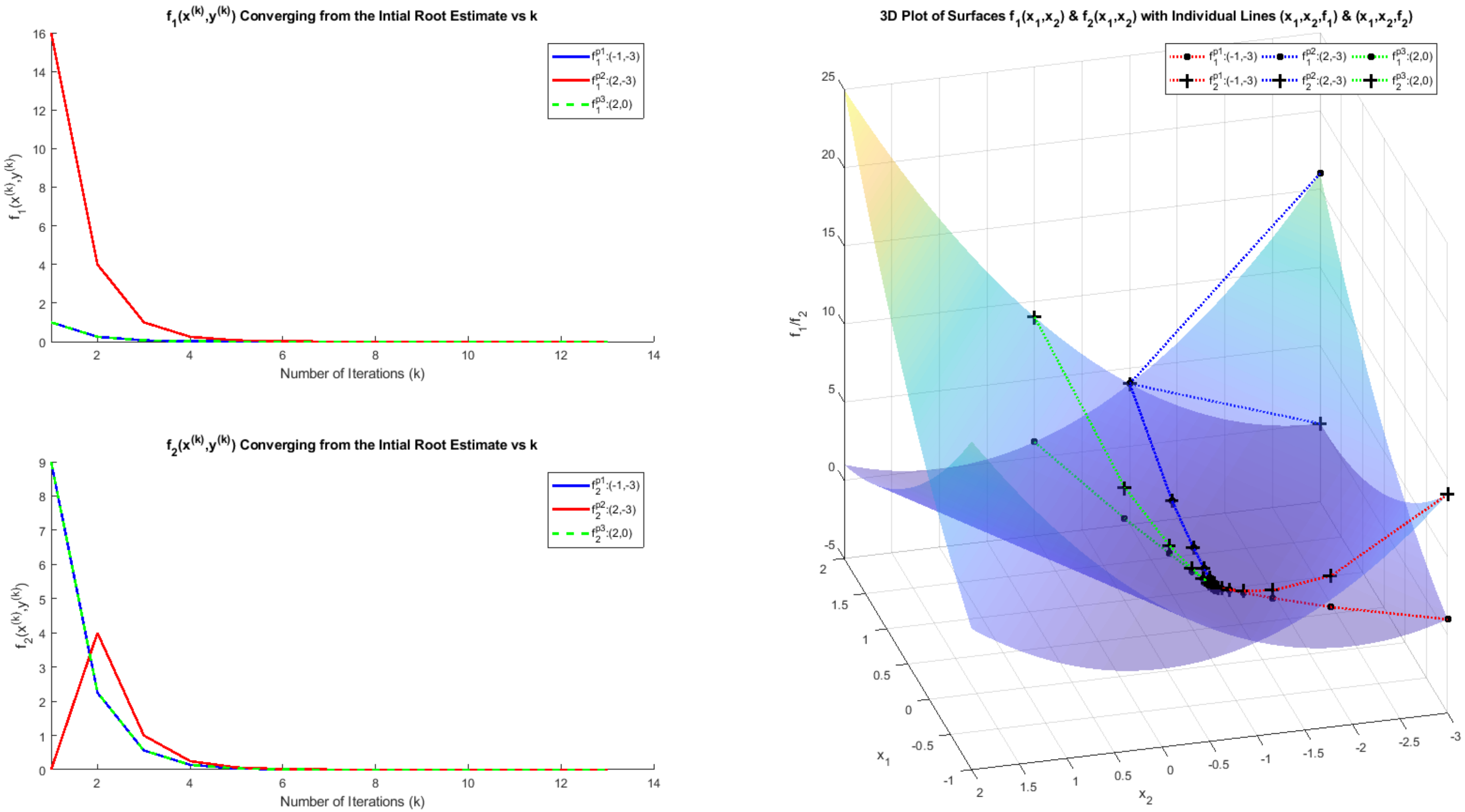

Figure 2: NLE Function Comparison

3 Code: Non Linear Equations

3.1 NLE Functions

Listing 1: Bisection

```
% Nonlinear equation root finding by the bisection method.
% Inputs
% f       : nonlinear function
% xl, xr  : initial root bracket
% nmax    : maximum number of iterations performed
% tol     : numerical tolerance used to check for root
% Outputs
% x       : one-dimensional array containing estimates of root

% Hint 1:
% Iterate until either a root has been found or maximum number of iterations has been reached

% Hint 2:
% Check for root each iteration, making use of tol

% Hint 3:
% Update the bracket each iteration

function x = Bisection(f, xl, xr, nmax, tol)
    % Initial array of root estimates, iteration and variable stores.
    x = [xl,xr];
    xbrac = x;
    n = 1;
    % Set iterative loop for the function
    while n < nmax + 1
        % Calculate the new root
        xnew = xbrac(1) + ((xbrac(2) - xbrac(1))/2);
        % Append the root estimate to the array
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) <= tol
           return
        % Calculate sign on the function with the new root
        % and find the new root bracket.
        elseif (f(xnew)*f(xbrac(1)))> 0
            % Reset the bracket with new LHS
            xbrac = [xnew,xbrac(2)];
        elseif (f(xnew)*f(xbrac(2)))> 0
            % Reset the bracket with new RHS
            xbrac = [xbrac(1),xnew];
        end
        % Increase iteration counter
        n = n + 1;
    end
```

```
    % Warn the user the maximum number of iterations have been
       performed
    disp('The maximum number of iterations have been performed
       without satisfying the required root finding condition');

end
```

Listing 2: Secant

```
% Nonlinear equation root finding by the secant method.
% Inputs
% f      : nonlinear function
% x0, x1 : initial root bracket
% nmax   : maximum number of iterations performed
% tol    : numerical tolerance used to check for root
% Outputs
% x      : one-dimensional array containing estimates of root

function x = Secant(f, x0, x1, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    x = [x0,x1];
    n = 1;
    k = 2;
    % Set iterative loop for the function
    while n <= nmax
        % Calculate the new root
        xnew = x(k) - (f(x(k))*(x(k)-x(k-1))/(f(x(k))- f(x(k-1)))
           );
        % Append the root estimate to the array
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) < tol
            return
        end
        % Increase iteration counter and k value
        n = n + 1;
        k = k + 1;
    end
    % Warn the user the maximum number of iterations have been
       performed
    disp('The maximum number of iterations have been performed
       without satisfying the required root finding condition');
```

Listing 3: Regula Falsi

```
% Nonlinear equation root finding by the Regula falsi method.
% Inputs
% f      : nonlinear function
% xl, xr : initial root bracket
% nmax   : maximum number of iterations performed
% tol    : numerical tolerance used to check for root
% Outputs
```

```
% x       : one-dimensional array containing estimates of root

function x = Regulafalsi(f, xl, xr, nmax, tol)
 % Initial array of root estimates, iteration and variable stores
    .
    x = [xl,xr];
    xbrac = x;
    n = 1;
    % Set iterative loop for the function
    while n <= nmax
        % Calculate the new root
        xnew = xbrac(2) - (f(xbrac(2))*(xbrac(2)-xbrac(1))/(f(
           xbrac(2))- f(xbrac(1))));
        % Append the root estimate to the array
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) < tol
           return
        % Calculate sign on the function with the new root
        % and find the new root bracket.
        elseif f(xnew)*f(xbrac(1))>0
            % Reset the bracket with new LHS
            xbrac = [xnew,xbrac(2)];
        elseif f(xnew)*f(xbrac(2))>0
            % Reset the bracket with new RHS
            xbrac = [xbrac(1),xnew];
        end
        % Increase iteration counter
        n = n + 1;
    end
    % Warn the user the maximum number of iterations have been
       performed
    disp('The maximum number of iterations have been performed
       without satisfying the required root finding condition');
```

Listing 4: Newton

```
% Nonlinear equation root finding by Newton's method
% Inputs
% f       : nonlinear function
% x0      : initial root estimate
% h       : step size for central difference formula
% nmax    : maximum number of iterations performed
% tol     : numerical tolerance used to check for root
% Outputs
% x       : one-dimensional array containing estimates of root

function x = Newton(f, x0, h, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    x = x0;
    n = 1;
    k = 1;
```

```
    % Set iterative loop for the function
    while n < nmax
        % Calculate the new root
        xnew = x(k) - (f(x(k)))/ ((f(x(k)+h) - f(x(k)-h))/(2*h));
        % Append the root estimate to the array
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) < tol
            return
        end
        % Increase iteration counter and k value
        n = n + 1;
        k = k + 1;
    end
    % Warn the user the maximum number of iterations have been performed
    disp('The maximum number of iterations have been performed without satisfying the required root finding condition');
```

Listing 5: Combined

```
% Nonlinear equation root finding by the combined binsection/Newton's method
% Inputs
% f       : nonlinear function
% xl, xr  : initial root bracket
% h       : step size for central difference formula
% nmax    : maximum number of iterations performed
% tol     : numerical tolerance used to check for root
% Outputs
% x       : one-dimensional array containing estimates of root

function x = Combined(f, xl, xr, h, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    x = [xl,xr];
    xbrac = x;
    n = 1;
    % Calculate the starting estimate
    xstart = xbrac(1) + (xbrac(2) - xbrac(1))/2;
    % Append the starting value
    x = [x,xstart];
    k = 3;
    % Set iterative loop for the function
    while n < nmax
        % Use newton method to calculate the new root estimate
        xnew = x(k) - (f(x(k)))/ ((f(x(k)+h) - f(x(k)-h))/(2*h));
        % Use if condition to check inside the bracket
        if (xnew < xbrac(1)) || (xnew > xbrac(2))
            % Use the bisection method to get a better estimate
            xnew = xbrac(1) + (xbrac(2) - xbrac(1))/2;
        end
        % Append the root estimate to the array
```

```
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) < tol
            return
        % Calculate sign on the function with the new root
        % and find the new root bracket.
        elseif f(xnew)*f(xbrac(1))>0
            % Reset the bracket with new LHS
            xbrac = [xnew,xbrac(2)];
        elseif f(xnew)*f(xbrac(2))>0
            % Reset the bracket with new RHS
            xbrac = [xbrac(1),xnew];
        end
        % Increase iteration counter and k count by one.
        n = n + 1;
        k = k + 1;
    end
    % Warn the user the maximum number of iterations have been performed
    disp('The maximum number of iterations have been performed without satisfying the required root finding condition');
```

Listing 6: Task 1

```
%% Task 1 - Bisection, Secant, Regula Falsi and Newton's Methods
% You do NOT need to modify this script

% clear workspace
clear
clc

% Initialisation
f = @(x) 2*x.^2-8*x+4; % function to evaluate
tol = 1.0e-4;          % tolerance for asserts
h = 1.0e-4;            % step size for numerical estimate of gradient
x0 = 0.0;              % initial interval left
x1 = 2.0;              % initial interval right
nmax = 50;             % maximum number of iterations

% Bisection method
xb = Bisection(f, x0, x1, nmax, tol);
assert(abs(f(xb(end))) < tol)
disp(['Bisection converged to root at x = ' num2str(xb(end))]);

% Secant method
xs = Secant(f, x0, x1, nmax, tol);
assert(abs(f(xs(end))) < tol)
disp(['Secant converged to root at x = ' num2str(xs(end))]);

% Regula Falsi method and verification
xrf = Regulafalsi(f, x0, x1, nmax, tol);
```

```
assert(abs(f(xrf(end))) < tol)
disp(['Regula Falsi converged to root at x = ' num2str(xrf(end))]);

% Newton's method and verification
xn = Newton(f, x0, h, nmax, tol);
assert(abs(f(xn(end))) < tol)
disp(['Newton converged to root at x = ' num2str(xn(end))]);

% Combined Bisection/Newton's method and verification
xc = Combined(f, x0, x1, h, nmax, tol);
assert(abs(f(xc(end))) < tol)
disp(['Combined Bisection/Newton converged to root at x = ' num2str(xc(end))]);
```

3.2 NLE Plotting

Listing 7: Task 2

```
%% Task 2 - Iterative Algorithm Comparison

% clear workspace
clear
clc

% Initialisation
tol = 1.0e-4;  % tolerance for asserts
h = 1.0e-4;    % step size for numerical estimate of gradient
nmax = 20;     % maximum number of iterations

% functions to test algorithms on
f1 = @(x) x.^2 - 1;                          % function 1
f2 = @(x) (exp(x)-exp(-x))./(exp(x)+exp(-x)); % function 2
f3 = @(x) cos(x)+sin(x.*x)-0.5;              % function 3

% initial root estimates for each function
% column 1: x0 for bisection, secant, regula falsi and combined methods
% column 2: x1 for bisection, secant, regula falsi and combined methods
% column 3: x0 for Newton's method
xint1 = ([-3.0,0.5,-3.0]);
xint2 = ([-5.,2.,1.1]);
xint3 = ([-2.0,1.5,-0.40]);

% function titles for plots
title1 = 'f(x)=x^2-1';
title2 = 'f(x)=(e^x - e^{-x})/(e^x + e^{-x})';
title3 = 'f(x)=cos(x)+sin(x^2)';

```

```
% set disp_func = false when you don't need to produce plot of
    functions
disp_func = false;
if disp_func
        x = linspace(-5.,5.,1000);
    figure(1), clf
    subplot(3,1,1)
    plot(x,f1(x))
    grid on, xlabel('x'), ylabel('f(x)'), title(title1)
    subplot(3,1,2)
    plot(x,f2(x))
    grid on, xlabel('x'), ylabel('f(x)'), title(title2)
    subplot(3,1,3)
    plot(x,f3(x))
    grid on, xlabel('x'), ylabel('f(x)'), title(title3)
end


%% find one root for each function using bisection, secant,
    newton's and combined methods
% Function 1
xB1 = Bisection(f1, xint1(1), xint1(2), nmax, tol);
xS1 = Secant(f1, xint1(1), xint1(2), nmax, tol);
xR1 = Regulafalsi(f1, xint1(1), xint1(2), nmax, tol);
xN1 = Newton(f1, xint1(3), h, nmax, tol);
xC1 = Combined(f1, xint1(1), xint1(2), h, nmax, tol);

% Function 2
xB2 = Bisection(f2, xint2(1), xint2(2), nmax, tol);
xS2 = Secant(f2, xint2(1), xint2(2), nmax, tol);
xR2 = Regulafalsi(f2, xint2(1), xint2(2), nmax, tol);
xN2 = Newton(f2, xint2(3), h, nmax, tol);
xC2 = Combined(f2, xint2(1), xint2(2), h, nmax, tol);

% Function 3
xB3 = Bisection(f3, xint3(1), xint3(2), nmax, tol);
xS3 = Secant(f3, xint3(1), xint3(2), nmax, tol);
xR3 = Regulafalsi(f3, xint3(1), xint3(2), nmax, tol);
xN3 = Newton(f3, xint3(3), h, nmax, tol);
xC3 = Combined(f3, xint3(1), xint3(2), h, nmax, tol);

%% individual plot for each function of f(x^k) vs k for each
    method
% i.e. each of the three plots (one per function) will have four
    lines, one for each method called.
figure(1), clf

% create top plot for function 1
% The intial root estimates have been excluded from the
    iterations.
subplot(3,1,1)
```

```
plot(1:length(xB1)-2,f1(xB1(3:length(xB1))))
hold on
plot(1:length(xS1)-2,f1(xS1(3:length(xS1))))
hold on
plot(1:length(xN1)-1,f1(xN1(2:length(xN1))))
hold on
plot(1:length(xC1)-2,f1(xC1(3:length(xC1))))
legend('Bisection','Secant','Newton','Combined')
xlim([1,8]);
grid on, xlabel('k: Number of Iterations'), ylabel('f(x)'), title
    (title1)

% create middle plot for function 2
subplot(3,1,2)
plot(1:length(xB2)-2,f2(xB2(3:length(xB2))))
hold on
plot(1:length(xS2)-2,f2(xS2(3:length(xS2))))
hold on
plot(1:length(xN2)-1,f2(xN2(2:length(xN2))))
hold on
plot(1:length(xC2)-2,f2(xC2(3:length(xC2))))
legend('Bisection','Secant','Newton','Combined')
xlim([1,8]);
grid on, xlabel('k: Number of Iterations'), ylabel('f(x)'), title
    (title2)

% create bottom plot for function 3
subplot(3,1,3)
plot(1:length(xB3)-2,f3(xB3(3:length(xB3))))
hold on
plot(1:length(xS3)-2,f3(xS3(3:length(xS3))))
hold on
plot(1:length(xN3)-1,f3(xN3(2:length(xN3))))
hold on
plot(1:length(xC3)-2,f3(xC3(3:length(xC3))))
legend('Bisection','Secant','Newton','Combined')
grid on, xlabel('k: Number of Iterations'), ylabel('f(x)'), title
    (title3)
xlim([1,8]);

% Save the plot
savefig('Task2Plot')
```

4 Systems of Non Linear Equations

4.1 Newton Two Variable

Listing 8: Newton Two Variables

```
% Nonlinear equation root finding in two dimensions using Newton's Method.
% Inputs
% func   : array of function handles for system of nonlinear equations
% x0     : vector of initial root estimates for each independent variable
% h      : step size for numerical estimate of partial derivatives
% nmax   : maximum number of iterations performed
% tol    : numerical tolerance used to check for root
% Outputs
% x      : two-dimensional array (two-row matrix) containing estimates of root

% Hint 1:
% Include the initial root estimate as the first column of x

% Hint 2:
% Use MATLAB in-built functionality for solving the matrix equation for vector of updates, delta

% Hint 3:
% Check for root each iteration, continuing until the maximum number of iterations has been reached

function x = Newton2Var(func, x0, h, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    % Initialise a storage array
    xstore = transpose(x0);
    x = xstore; % Vector of the most recent root variables
    % Set iterative loop for the function
    for i = 1:nmax
        % Calculate the function variables from the most recent iteration.
        f1 = func{1}(x);
        f2 = func{2}(x);

        %Calculate the derivatives for each of the points for the jacobian
        f1x1 = (func{1}(x + [h;0])- func{1}(x - [h;0]))/(2*h);
        f1x2 = (func{1}(x + [0;h])- func{1}(x - [0;h]))/(2*h);
        f2x1 = (func{2}(x + [h;0])- func{2}(x - [h;0]))/(2*h);
        f2x2 = (func{2}(x + [0;h])- func{2}(x - [0;h]))/(2*h);

        % Set Jacobian and f
        f = [f1;f2];
        J = [f1x1,f1x2;f2x1,f2x2];

        % Calculate the Delta
        del = -1*linsolve(J,f);
```

```

        % Find the new xvalues
        xnew = del + x;

        % Calculate the new f values
        fnew = [func{1}(xnew);func{2}(xnew)];
        %Use condition criteria to cancel out of the list
        if (abs(fnew(1))< tol) && (abs(fnew(2))< tol)
            % Add to the storage arrays
            xstore = [xstore,xnew];
            x = xstore;
            return
        end
        xstore = [xstore,xnew];
        x = xnew;
    end
    disp('The maximum number of iterations have been performed
       without satisfying the required root finding condition');
end
```

Listing 9: Task 3

```
%% Task 3 - System of Nonlinear Equations

% clear workspace
clear all
clc

% initialisation
tol = 1.0e-6;       % numerical tolerance
h = 1.0e-4;         % step size for central difference
nmax = 50;          % maximum number of iterations
x0 = [2,0];         % initial root estimate
func = {@f1, @f2}; % array of function handles

% set func_usage to false once you know how vector/array func
   works
func_usage = true;
if func_usage
    f1_initial = func{1}(x0);
    f2_initial = func{2}(x0);
end

% 2D Newton's method and verification
disp(['Newton2Var starting at point (x0,y0) = (' num2str(x0(1)),'
   ,',num2str(x0(2)),')']);
xn = Newton2Var(func, x0, h, nmax, tol);
disp([xn(1,end),xn(2,end)]);
assert(abs(func{1}([xn(1,end),xn(2,end)])) <= tol)
assert(abs(func{2}([xn(1,end),xn(2,end)])) <= tol)
disp(['Newton2Var converged to root at (x,y) = (' num2str(xn(1,
   end)),',',num2str(xn(2,end)),') in ',num2str(length(xn)),'
```

```
    iterations']);


% Functions to be used for testing out Newton2Var
function f = f1(x)
    f = x(1)*x(1)-2*x(1)+x(2)*x(2)+2*x(2)-2*x(1)*x(2)+1;
end
function f = f2(x)
    f = x(1)*x(1)+2*x(1)+x(2)*x(2)+2*x(2)+2*x(1)*x(2)+1;
end
```

4.2 Newton Two Variable Plotting

Listing 10: Task 4

```
%% Task 4

% clear workspace
clear all
clc

% initialisation
tol = 1.0e-6;       % numerical tolerance
h = 1.0e-4;         % step size for central difference
nmax = 50;          % maximum number of iterations
x0_p1 = [-1,-3];     % initial root estimate - point 1
x0_p2 = [2,-3];     % initial root estimate - point 2
x0_p3 = [2,0];      % initial root estimate - point 3
func = {@f1, @f2}; % array of function handles

% Two function two variable Newton's method for each starting
    location
xn_p1 = Newton2Var(func, x0_p1, h, nmax, tol);
xn_p2 = Newton2Var(func, x0_p2, h, nmax, tol);
xn_p3 = Newton2Var(func, x0_p3, h, nmax, tol);

%% start figure for algorithm visualisation
figure(1), clf
% Calculate all the values needed

% Iteration count 1
[~,c1] =size(xn_p1);
k1 = 1:c1;

% Call the first function for each iteration from the first
    starting point.
for i = 1:c1
    fp11(i) = func{1}([xn_p1(1,i),xn_p1(2,i)]);
end

%Iteration Count 2
```

```
[~,c2] =size(xn_p2);
k2 = 1:c2;

% Call the first function for each iteration from the second starting point.
for i = 1:c2
    fp21(i) = func{1}([xn_p2(1,i),xn_p2(2,i)]);
end

% Iteration count 3
[~,c3] =size(xn_p3);
k3 = 1:c3;

% Call the first function for each iteration from the third starting point.
for i = 1:c3
    fp31(i) = func{1}([xn_p3(1,i),xn_p3(2,i)]);
end

% Call the second function for each iteration from the first starting point.
for i = 1:c1
    fp12(i) = func{2}([xn_p1(1,i),xn_p1(2,i)]);
end

% Call the first function for each iteration from the second starting point.
for i = 1:c2
    fp22(i) = func{2}([xn_p2(1,i),xn_p2(2,i)]);
end

% Call the first function for each iteration from the third starting point.
for i = 1:c3
    fp32(i) = func{2}([xn_p3(1,i),xn_p3(2,i)]);
end


% Create all the labels to plot with
f1title = 'f_{1}(x^{(k)},y^{(k)}) Converging from the Intial Root Estimate vs k';
f2title = 'f_{2}(x^{(k)},y^{(k)}) Converging from the Intial Root Estimate vs k';
surftitle = '3D Plot of Surfaces f_{1}(x_1,x_2) & f_{2}(x_{1},x_{2}) with Individual Lines (x_1,x_2,f_1) & (x_1,x_2,f_2)';
f1xlabel = 'Number of Iterations (k)';
f1ylabel = 'f_{1}(x^{(k)},y^{(k)})';
f2xlabel = 'Number of Iterations (k)';
f2ylabel = 'f_{2}(x^{(k)},y^{(k)})';
surfxlabel = 'x_1';
surfylabel = 'x_2';
```

```
surfzlabel = 'f_1/f_2';

% Create axis labels
f1p1 = 'f_1^{p1}:(-1,-3)';
f1p2 = 'f_1^{p2}:(2,-3)';
f1p3 = 'f_1^{p3}:(2,0)';
f2p1 = 'f_2^{p1}:(-1,-3)';
f2p2 = 'f_2^{p2}:(2,-3)';
f2p3 = 'f_2^{p3}:(2,0)';
f1legend = {f1p1,f1p2,f1p3};
f2legend = {f2p1,f2p2,f2p3};
surflegend = {f1p1,f2p1,f1p2,f2p2,f1p3,f2p3};

% create top left plot for function 1
subplot(2,2,1)
hold on
plot(k1,fp11,'b','Linewidth',2)
hold on
plot(k2,fp21,'r','Linewidth',2)
hold on
plot(k3,fp31,'g--','Linewidth',2)
ylabel('Function 1 values')
xlabel('Number of Iterations (k)')
title(f1title)
legend(f1legend)
xlabel(f1xlabel)
ylabel(f1ylabel)
xlim([1,14]);

% create bottom left plot for function 2
subplot(2,2,3)
hold on
plot(k1,fp12,'b','Linewidth',2)
hold on
plot(k2,fp22,'r','Linewidth',2)
hold on
plot(k3,fp32,'g--','Linewidth',2)
ylabel('Function 2 values')
xlabel('Number of Iterations (k)')
title(f2title)
legend(f2legend)
xlabel(f2xlabel)
ylabel(f2ylabel)
xlim([1,14]);

% create right plot for 3d visualisation of 2d newton's method
subplot (2,2,[2 4])
% Plot both the functions
[X,Y] = meshgrid(-1:0.1:2,-3:0.1:2);
Z1 = X.*X-2.*X+Y.*Y+2.*Y-2.*X.*Y+1;
Z2 = X.*X+2.*X+Y.*Y+2.*Y+2.*X.*Y+1;
```

```
surf(X,Y,Z1);
hold on;
surf(X,Y,Z2);

% Improve plotting
alpha 0.4; % Make more transparent
rotate3d on; % Automatic switch on rotate feature
shading interp; % Change shading
view(255,25); % Specify view found by trial and error.

% Plot the lines on the surface
hold on
ob1 = plot3(xn_p1(1,:),xn_p1(2,:),fp11,':r.','Linewidth',2,'
    Markersize',20,'MarkerEdgeColor','k');
hold on
ob3 = plot3(xn_p2(1,:),xn_p2(2,:),fp21,':b.','Linewidth',2,'
    Markersize',20,'MarkerEdgeColor','k');
hold on
ob5 = plot3(xn_p3(1,:),xn_p3(2,:),fp31,':g.','Linewidth',2,'
    Markersize',20,'MarkerEdgeColor','k');
hold on
ob2 = plot3(xn_p1(1,:),xn_p1(2,:),fp12,':r+','Linewidth',2,'
    Markersize',10,'MarkerEdgeColor','k');
hold on
ob4 = plot3(xn_p2(1,:),xn_p2(2,:),fp22,':b+','Linewidth',2,'
    Markersize',10,'MarkerEdgeColor','k');
hold on
ob6 = plot3(xn_p3(1,:),xn_p3(2,:),fp32,':g+','Linewidth',2,'
    Markersize',10,'MarkerEdgeColor','k');
ob = [ob1,ob2,ob3,ob4,ob5,ob6];

% Plot labels
legend(ob,surflegend,'Location','northeast','NumColumns',3)
title(surftitle);
xlabel(surfxlabel);
ylabel(surfylabel);
zlabel(surfzlabel);
grid on;

% Functions to be used for testing out Newton2Var
function f = f1(x)
    f = x(1)*x(1)-2*x(1)+x(2)*x(2)+2*x(2)-2*x(1)*x(2)+1;
end
function f = f2(x)
    f = x(1)*x(1)+2*x(1)+x(2)*x(2)+2*x(2)+2*x(1)*x(2)+1;
end
```

4.3 Newton Multiple Variable

Listing 11: Newton Multiple Variables

```
% Nonlinear equation root finding in n dimensions using Newton's Method.
% Inputs
% n       : number of dimensions for Newton's method
% func    : array of function handles for system of nonlinear equations
% x0      : vector of initial root estimates for each independent variable
% h       : step size for numerical estimate of partial derivatives
% nmax    : maximum number of iterations performed
% tol     : numerical tolerance used to check for root
% Outputs
% x       : array (n-row matrix) containing estimates of root

% Hint 1:
% Include the initial root estimate as the first column of x

% Hint 2:
% Use MATLAB in-built functionality for solving the matrix equation for vector of updates, delta

% Hint 3:
% Check for root each iteration, continuing until the maximum number of iterations has been reached

function x = NewtonMultiVar(n, func, x0, h, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    % Initialise a storage array
    xstore = transpose(x0);
    x = xstore; % Vector of the most recent root variables
    n = 1;
    % Set iterative loop for the function
    while n < nmax
        % Calculate the current root estimate, used a nested for loop.
        for i = 1:length(func)
            f(i,1) = func{i}(x); % Vector of variables passed into the function call
        end
        % Initialise the size of the jacobian
        jacob = zeros(length(func):length(x));
        % Calculate the jacobian
        for i = 1:length(func)
            for j  = 1: length(x)
                % Create two small steps for the x values
                x(j) = x(j) + h;
                % Find the first part of the derivitive calculation.
                func1 = func{i}(x);
```

```
                % Do the second part of the derivitive calculation.
                x(j) = x(j) - 2.*h;
                % Find the first part of the derivitive calculation.
                func2 = func{i}(x);
                % Calculate the jacobian
                jacob(i,j) = ((func1 - func2)./(2.*h));
                % Correct the x value
                x(j) = x(j) + h;
            end
        end
        % Inverse the jacobian and get del
        del = -1.*(jacob\f);
        % Perform the necesary exit conditions
        % New del
        xnew = del + x;
        % Recalculate f with xnew values
        for i = 1:length(func)
            fnew(i,1) = func{i}(transpose(xnew)); % Vector of variables passed into the function call
        end
        % Test for both convergence and function call close to zero.
        if (prod((abs(fnew)<= tol)) == 1) && (prod((abs(del)<= tol))== 1)
            % Add to the store arrays
            xstore = [xstore,xnew];
            x = xstore;
            return
        end
        xstore = [xstore,xnew];
        x = xnew;
        % Increase iteration counter
        n = n + 1;
    end
    % Warn the user the maximum number of iterations have been performed
    disp('The maximum number of iterations have been performed without satisfying the required root finding condition');
end
```