PHYS 410 Homework 1

Gavin Pringle, 56401938

September 30, 2024

Introduction

In this homework assignment, two methods for solving nonlinear equations numerically via root finding are explored: bisection and Newton's method. In order to do this, two problems are provided. The first problem involves the 1-dimesional case where the roots of a nonlinear function are found using a hybrid algorithm that first employs bisection followed by Newton's method. The second problem involves the d-dimensional case where a nonlinear system is solved using a d-dimensional Newton iteration.

In both problems it is assumed the function and its derivative in problem 1 as well as the system of equations and its Jacobian in problem 2 are hard-coded as Matlab functions. Specifically, a polynomial of order 10 is provided for the first question and a nonlinear system of three variables is provided for the second question.

Problem 1 - Hybrid Algorithm

Review of Theory

Bisection, also referred to as binary search, is a method used for solving nonlinear equations of the form $f(x) = 0$ for the 1-dimensional case or $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ for the d-dimensional case. The bisection algorithm involves bisecting a search interval and checking whether the root is above or below the bisector. The search interval is then bisected again in the new interval where the root lies, and again it is determined whether the root is above or below the new bisector. This process then repeats until the root is determined to be in an interval of small enough tolerance.

For the 1-dimensional bisection algorithm, it is assumed that there is a root of $f(x) = 0$ in the interval $x_{min} \leq x \leq x_{max}$. From this, it follows that $f(x_{min})f(x_{max}) \leq 0$. As previously described, the interval $[x_{min}, x_{max}]$ which has width $\delta x_0 = x_{max} - x_{min}$ is successively divided into smaller intervals of width $\delta x_1 = \delta x_0/2$, $\delta x_2 = \delta x_0/4$, $\delta x_3 = \delta x_0/8$, ... each of contains the root which is checked using the condition $f({x_{min}}^{(n)})f({x_{max}}^{(n)}) \leq 0$. This process is continued until interval is suitably small, which is verified by checking the relative error given by the formula $\frac{|\delta x^{(n)}|}{|x^{(n+1)}|} \leq \epsilon$.

Newton's method is another method used for solving nonlinear equations of the form $f(x) = 0$ for the 1-dimensional case or $\vec{f}(\vec{x}) = \vec{0}$ for the d-dimensional case. Newton's method does not require an interval wherein the root lies but rather a "good" initial guess as to where the root is near. Whether or not the guess is "good" depends on the specific problem at hand.

In Newton's method, we let x^* be a root of $f(x)$. From the Taylor expansion,

$$0 \approx f(x^{(n)}) + (x^* - x^{(n)})f'(x^{(n)})$$

Letting $x^{(n+1)} = x^*$, we can rearrange to get

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

For the 1-dimensional Newton's method algorithm, the above equation can be successively applied to yield a closer and closer approximation to the true x^*. The process can again be stopped when a suitable precision is achieved, calculated using the formula

$$\frac{|\delta x^{(n)}|}{|x^{(n+1)}|} = \frac{|x^{(n+1)} - x^{(n)}|}{|x^{(n+1)}|} \leq \epsilon$$

Numerical Approach

Both of the 1-dimensional bisection algorithm and 1-dimensional Newton's method are used in this problem to create a hybrid algorithm that first employs bisection to an intermediate tolerance followed by Newton's method to a final tolerance in order to find the root of an arbitrary function was in the interval $[x_{min}, x_{max}]$. The algorithm was implemented as a function (as per the homework instructions):

```
function x = hybrid(f, dfdx, xmin, xmax, tol1, tol2)
```

where `x` is the returned value of the calculated root, `f` is the function for which the location of the root is sought, `dfdx` is the derivative of said function, `xmin` is the initial interval minimum, `xmax` is the initial interval maximum, `tol1` is the relative convergence criterion for bisection, and `tol2` is the relative convergence criterion for Newton iteration.

In order to test the function `hybrid(f, dfdx, xmin, xmax, tol1, tol2)`, the polynomial

$$f(x) = 512x^{10} - 5120x^9 + 21760x^8 - 51200x^7 + 72800x^6 - 64064x^5 + 34320x^4 - 10560x^3 + 1650x^2 - 100x + 1$$

and its derivative were both implemented as Matlab functions to pass as parameters. The online graphing calculator Desmos was used to graph the function in order to see where the roots are in order to find intervals for each root of $f(x)$ in the interval $[0, 2]$. For testing purposes `tol1` was set to 10^{-4} and `tol2` was set to 10^{-12} as per the homework instructions on relative precision. Refer to Appendix C for the testing Matlab script.

Implementation

The function `hybrid(f, dfdx, xmin, xmax, tol1, tol2)` first employs bisection via a while loop to a precision of `tol1`. The middle of the final bisection interval is passed as the starting point for the Newton's method algorithm which then computes the root to a precision of `tol2` via another while loop. A loop iteration counter is utilized for both the bisection and Newton's method loops that causes the function to immediately return `NaN` if either loop exceed 50 iterations.

Refer to Appendix A for the full Matlab script.

Results

Using the script `test.m` shown in Appendix C, the roots of $f(x)$ were computed using the hybrid algorithm as

```
roots(1) = 0.012311659405
roots(2) = 0.108993475812
roots(3) = 0.292893218813
roots(4) = 0.546009500260
roots(5) = 0.843565534960
roots(6) = 1.156434465042
roots(7) = 1.707106781183
roots(8) = 1.891006524190
roots(9) = 1.453990499747
roots(10) = 1.987688340584
```

Plugging these calculated roots into the original function we get the following output for $f(x)$ evaluated at the numerically computed roots, and we can confirm that they are all approximately zero to the desired relative precision:

```
function_at_roots =

    1.0e-08 *
                   0
   -0.000000177635684
                   0
   -0.000075317529991
   -0.000821387402539
    0.003998934516858
   -0.012423129192030
    0.074888362178172
   -0.004718003765447
   -0.112436282506678
```

Plotting the numerically computed roots on top of the function $f(x)$, we can again confirm they are accurate.

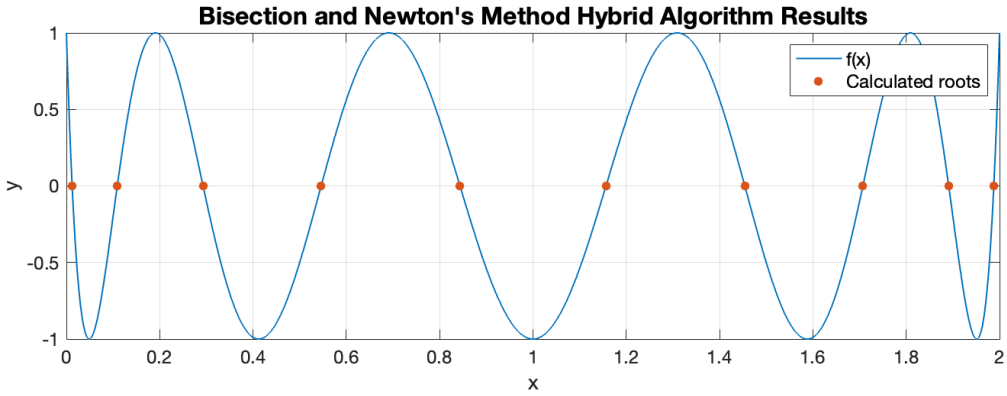

Figure 1: Numerically calculated roots overlaid on example function $f(x)$.

Problem 2 - D-dimesional Newton's Method

Review of Theory

Newton's method in d-dimensions follows the same process as Newton's method in one dimension, however the derivative $f'(x)$ is replaced with the Jacobian matrix of $\mathbf{f}(\mathbf{x})$. In order to solve $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, the multivariate Taylor series expansion is used:

$$\mathbf{0} \approx \mathbf{f}(\mathbf{x}^{(n)}) + \mathbf{J}[\mathbf{x}^{(n)}](\mathbf{x}^* - \mathbf{x}^{(n)})$$

where

$$\mathbf{J}_{i,j} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix}$$

Replacing $\mathbf{x}^*$ with $\mathbf{x}^{(n+1)}$:

$$\mathbf{0} \approx \mathbf{f}(\mathbf{x}^{(n)}) + \mathbf{J}[\mathbf{x}^{(n)}](\mathbf{x}^{(n+1)} - \mathbf{x}^{(n)})$$

Numerical approach

definition of all pertinent problem parameters

exposition of the requisite equations

methodology that is used to solve the problem

function signature

testing method

provided equations

Implementation

refer to appendix

The Jacobian matrix is

$$\begin{bmatrix} 2x & 4y^3 & 6z^5 \\ -yz^2 \sin\left(xyz^2\right) - 1 & -xz^2 \sin\left(xyz^2\right) - 1 & -2xyz \sin\left(xyz^2\right) - 1 \\ -2x - 2y + 2z & -2x + 2z & 2x + 2y + 3z^2 - 2z \end{bmatrix}$$

Figure 2: Calculated Jacobian matrix for the provided system of equations.

iteration counter

Results

console output

Conclusions

Briefly summarize your findings (again, more extensively in the case of projects). Discuss any particular problems you had with the homework/project. Include your statement concerning your use or non-use of generative AI here.

Appendix A - Hybrid Algorithm Code

```
%% Problem 1 - Hybrid algorithm

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% A hybrid algorithm that uses bisection and Newton's method
% to locate a root within a given interval [xmin, xmax]. If the
% number of iterations exceeds for either bisection or Newton's
% method returns 50, the function returns NaN.
%
% Arguments:
%  f:       Function whose root is sought (takes one argument).
%  dfdx:    Derivative function (takes one argument).
%  xmin:    Initial bracket minimum.
%  xmax:    Initial bracket maximum.
%  tol1:    Relative convergence criterion for bisection.
%  tol2:    Relative convergence criterion for Newton iteration.
% Returns:
%  x:       Estimate of root.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function x = hybrid(f, dfdx, xmin, xmax, tol1, tol2)
    overflow_counter = 0;
    MAX_ITERATIONS = 50;

    % Bisection:
    converged = false;
    fmin = f(xmin);
    while not(converged)
        xmid = (xmin + xmax)/2;
        fmid = f(xmid);
        if fmid == 0
            break
        elseif fmid*fmin < 0
            xmax = xmid;
        else
            xmin = xmid;
            fmin = fmid;
        end
        if (xmax - xmin)/abs(xmid) < tol1
            converged = true;
        end

        overflow_counter = overflow_counter + 1;
        if overflow_counter == MAX_ITERATIONS
            x = NaN;
            return;
        end
    end
    bisection_result = xmid;

    overflow_counter = 0;

    % Newton's method:
    converged = false;
    x = bisection_result;
    xprev = bisection_result;
    while not(converged)
        x = xprev - f(xprev)/dfdx(xprev);
```

```
57          if abs((x - xprev)/xprev) < tol2
58              converged = true;
59          end
60          xprev = x;
61
62          overflow_counter = overflow_counter + 1;
63          if overflow_counter == MAX_ITERATIONS
64              x = NaN;
65              return;
66          end
67      end
68
69  end
```

Appendix B - D-dimesional Newton's Method Code

```
%% Problem 2 - D-dimensional Newton iteration

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Newton's method for a d-dimensional space. If the number of
% iterations exceeds 50, the function returns NaN.
%
% Arguments:
%  f:    Function which implements the nonlinear system of
%        equations. Function is of the form f(x) where x is a
%        length-d vector, and which returns length-d column
%        vector.
%  jac: Function which is of the form jac(x) where x is a
%        length-d vector, and which returns the d x d matrix of
%        Jacobian matrix elements for the nonlinear system defined
%        by f.
%  x0:   Initial estimate for iteration (length-d column vector).
%  tol: Convergence criterion: routine returns when relative
%        magnitude of update from iteration to iteration is
%       <= tol.
% Returns:
%  x:    Estimate of root (length-d column vector).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function x = newtond(f, jac, x0, tol)
    overflow_counter = 0;
    MAX_ITERATIONS = 50;

    x = x0;
    res = f(x0);
    dx = jac(x0)\res;
    while rms(dx) > tol
        res = f(x);
        dx = jac(x)\res;
        x = x - dx;

        overflow_counter = overflow_counter + 1;
        if overflow_counter == MAX_ITERATIONS
            x = NaN;
            return;
        end
    end
end
```

Appendix C - Testing Code

```
%% Test script for Problem 1 and Problem 2

close all; clear; clc;

format long;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Test Script - Problem 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Example polynomial function given in problem 1 of Homework 1
% document.
%
% Arguments:
%  x:  Polynomial independent variable
% Returns:
%  example_f_out:  Function evaluated at x
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function example_f_out = example_f(x)
    example_f_out = 512*x^10 - 5120*x^9 + 21760*x^8 - 51200*x^7 + ...
    72800*x^6 - 64064*x^5 + 34320*x^4 - 10560*x^3 + 1650*x^2 - 100*x + 1;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Derivative of example polynomial function given in problem 1 of
% Homework 1 document.
%
% Arguments:
%  x:  Polynomial independent variable
% Returns:
%  example_dfdx_out:  Derivative evaluated at x
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function example_dfdx_out = example_dfdx(x)
    example_dfdx_out = 20*(-5 + 165*x - 1584*x^2 + 6864*x^3 - 16016*x^4 ...
    + 21840*x^5 - 17920*x^6 + 8704*x^7 - 2304*x^8 + 256*x^9);
end

% Root finding
roots = zeros([1,10]);

BS_tol = 1.0e-4;
NM_tol = 1.0e-12;

roots(1) = hybrid(@example_f, @example_dfdx, 0.0, 0.04, BS_tol, NM_tol);
roots(2) = hybrid(@example_f, @example_dfdx, 0.05, 0.15, BS_tol, NM_tol);
roots(3) = hybrid(@example_f, @example_dfdx, 0.23, 0.35, BS_tol, NM_tol);
roots(4) = hybrid(@example_f, @example_dfdx, 0.47, 0.6, BS_tol, NM_tol);
roots(5) = hybrid(@example_f, @example_dfdx, 0.77, 0.9, BS_tol, NM_tol);
roots(6) = hybrid(@example_f, @example_dfdx, 1.11, 1.22, BS_tol, NM_tol);
roots(7) = hybrid(@example_f, @example_dfdx, 1.65, 1.75, BS_tol, NM_tol);
roots(8) = hybrid(@example_f, @example_dfdx, 1.86, 1.90, BS_tol, NM_tol);
roots(9) = hybrid(@example_f, @example_dfdx, 1.4, 1.5, BS_tol, NM_tol);
roots(10) = hybrid(@example_f, @example_dfdx, 1.98, 2.0, BS_tol, NM_tol);

for i = 1:10
```

```
57      fprintf('roots(%d) = %.12f \n',i, roots(i));
58  end
59
60  function_at_roots = transpose(arrayfun(@example_f, roots))
61
62  % Plotting
63  xvec = linspace(0,2,10000);
64
65  fig = figure;
66  plot(xvec, arrayfun(@example_f, xvec), 'LineWidth', 1, 'DisplayName', 'f(x)
        ');
67  hold on;
68  scatter(roots, zeros([1,10]), 'filled', 'DisplayName', 'Calculated roots',
        'Color', 'r');
69  lgd = legend;
70  ax = gca;
71  fontsize(lgd,12,'points');
72  fontsize(ax,12,'points');
73  title('Bisection and Newton''s Method Hybrid Algorithm Results', 'FontSize'
        , 16);
74  xlabel('x');
75  ylabel('y');
76  grid on;
77
78  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
79  % Test Script - Problem 2
80  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
81
82  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
83  % Example nonlinear system given in problem 2 of Homework 1
84  % document.
85  %
86  % Arguments:
87  %  x:  Vector of length 3. x, y, z independent variables in the
88  %      system.
89  % Returns:
90  %  example_sys_out:  Column ector of length 3. f1, f2, f3
91  %      outputs of each function in the system.
92  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
93  function example_sys_out = example_sys(x)
94      example_sys_out = zeros(3,1);
95      example_sys_out(1) = x(1)^2 + x(2)^4 + x(3)^6 - 2;
96      example_sys_out(2) = cos(x(1)*x(2)*x(3)^2) - x(1) - x(2) - x(3);
97      example_sys_out(3) = x(2)^2 + x(3)^3 - (x(1) + x(2) - x(3))^2;
98  end
99
100 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
101 % Jacobian matrix of example nonlinear system given in problem 2
102 % of Homework 1 document.
103 %
104 % Arguments:
105 %  x:  Vector of length 3. x, y, z independent variables in the
106 %      System.
107 % Returns:
108 %  example_jac_out:  3x3 matrix. Entries of the Jacobian matrix
109 %      for f1(x,y,z), f2(x,y,z), f3(x,y,z).
110 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
111 function example_jac_out = example_jac(x)
```

```
112         example_jac_out(1,1) = 2*x(1);
113         example_jac_out(1,2) = 4*x(2)^3;
114         example_jac_out(1,3) = 6*x(3)^5;
115         example_jac_out(2,1) = -x(2)*x(3)^2*sin(x(1)*x(2)*x(3)^2) - 1;
116         example_jac_out(2,2) = -x(1)*x(3)^2*sin(x(1)*x(2)*x(3)^2) - 1;
117         example_jac_out(2,3) = -2*x(1)*x(2)*x(3)*sin(x(1)*x(2)*x(3)^2) - 1;
118         example_jac_out(3,1) = -2*x(1)-2*x(2)+2*x(3);
119         example_jac_out(3,2) = -2*x(1)+2*x(3);
120         example_jac_out(3,3) = 2*x(1)+2*x(2)+3*x(3)^2-2*x(3);
121     end
122
123     % Root finding
124     initial_guess = [-1.0; 0.75; 1.50];
125     NM_3D_tol = 1.0e-12;
126
127     solution = newtond(@example_sys, @example_jac, initial_guess, NM_3D_tol);
128
129     fprintf('x = %.12f \n', solution(1));
130     fprintf('y = %.12f \n', solution(2));
131     fprintf('z = %.12f \n', solution(3));
132
133     system_at_solution = example_sys(solution)
```