Assignment 2

COSC2500
Ryan White
44990392

7th of September 2020

R2.1

Firstly, the code used is shown in Appendix 0.1. Using such code, the first two roots of $f_1(x) = \cos(x)$ were found to be

Method	Root 1	Root 1 Error	Root 2	Root 2 Error
Bisection	1.570844263694518	-4.793689 $\times 10^{-5}$	4.712436917284311	-4.793689 $\times 10^{-5}$
Fixed Point Iteration	1.570791601024261	4.72577 $\times 10^{-6}$	N/A	N/A
Newton's Method	1.570796326795488	-5.9152683 $\times 10^{-13}$	4.712388980912051	-5.2736127 $\times 10^{-10}$

Table 1: Comparison of Values and Error for Root-Finding Methods of f_1

Worth noting is that, no matter which initial point was chosen, the second root for the fixed point iteration method could not be found. Initial values corresponded to either the first ($x \approx 1.57$) or third ($x \approx 7.85$) roots, but never the second. For the first positive root, all three methods were exceptionally forgiving in the choice of bounds or initial values. The code given in Appendix 0.1 shows that the fixed point iteration method only needed 3 iterations to yield a root that would be spot on under single precision. The bisection method needed 15 iterations to yield a root that was actually less accurate than the fixed point iterative method, and Newton's method also only needed 3 iterations to give the most accurate answer (by far). These methods had the same number of iterations for the second root also. It was found that for Newton's method, as long as the derivative of the function at the initial value had the same sign as the function at the next root, it would find it very quickly. If not, it would find the previous root if it exists. As such, choosing an initial value was reasonably simple.
For the function $f_2(x) = (x-1)^3(x+2)$, the two roots were found to be

Method	Root 1	Root 1 Error	Root 2	Root 2 Error
Bisection	-2.000015258789063	-1.5258789063 $\times 10^{-5}$	0.999984741210938	1.525878 $\times 10^{-5}$
Fixed Point Iteration	-2	0	0.948868930398334	5.11310696 $\times 10^{-2}$
Newton's Method	-2.000000018912795	-1.8912795 $\times 10^{-8}$	0.982189756765293	1.781024323 $\times 10^{-2}$

Table 2: Comparison of Values and Error for Root-Finding Methods of f_2

Once again, the bisection method took 15 steps, while Newton's method this time took 5 for the first root at $x = -2$. This time, however, Newton's method wasn't entirely accurate for the second root, likely due to this root being at a stationary point where the derivative switches signs. That said, it still got this calculation in 8 steps; more than half of that of the bisection method at 15 iterations again. With the second root, the fixed point iteration method was very particular in it's initial value, with it only giving reasonable results for initial values very close to the true root. For this result, the initial value was 0.95.

R2.2

For the function $f(x) = \left(x - \frac{7}{11}\right)^2 \left(x + \frac{3}{13}\right) e^{-x^2}$, the maxima and minima are shown in tables 3 and 4 respectively.

Method	Maxima 1	Error	Maxima 2	Error
Golden Section Search	0.053215590630520	-7.2769063 $\times 10^{-4}$	1.481226263118113	-2.15626311 $\times 10^{-3}$
Successive Parabolic Interpolation	0.052487843514822	5.6485178 $\times 10^{-8}$	1.479073143747886	-3.14374 $\times 10^{-6}$
Newton's Method	0.052467823265657	2.007673 $\times 10^{-5}$	1.478730209499440	3.397905 $\times 10^{-4}$

Table 3: Maxima and Corresponding Error for the Function $f(x)$

Method	Minima 1	Error	Minima 2	Error
Golden Section Search	-1.123172263453982	-2.79773654 $\times 10^{-3}$	0.637287570313157	-9.2393394 $\times 10^{-4}$
Successive Parabolic Interpolation	-1.125966585858277	-3.41414 $\times 10^{-6}$	0.636363636363636	0
Newton's Method	-1.125407648977172	-5.6235102 $\times 10^{-4}$	0.636454131721944	-9.049535 $\times 10^{-5}$

Table 4: Minima and Corresponding Error for the Function $f(x)$

As can be seen in Appendix 0.2, the Golden Section Search (GSS) and Successive Parabolic Interpolation (SPI) methods utilised all 10 steps for finding each stationary point, while Newton's Method used at most 2 iterations, making it computationally favourable. That said, the SPI method consistently yielded the least error by far, with Newton's method beating out the GSS method. In terms of ease in choosing initial points, the GSS method was the easiest as any interval (provided that there was only one stationary point in the bounds) worked in finding a stationary point. In contrast to it's accuracy, the SPI method was extremely temperamental in finding starting values, and often small bounds were required to find the desired stationary points.

R2.3

The code given in Appendix 0.3 attempts to calculate the equilibrium position (to a maximum error given by `z_precision = 1e-4`) along an axis subject to a force field given by the vector `a`. By default, the function attempts to use Newton's method for finding such an equilibrium position. Line 15 of the code checks to see whether an initial guess was given, and, if not, it gives an initial guess of `z = 0`.
If Newton's method isn't used (i.e. `newton = false` on line 9), then the bisection method is used. This method doesn't require an initial guess, and forms it's own bounds based on the size of the T-matrix used. If the force at each point in the 4 radii chosen by the size of the T-matrix is greater than 0, then the `if` statement from line 79 to 81 breaks the code with an error stating that no bisection starting points can be found, and so there are no equilibrium points. If starting points were found, the bisection search from line 85 onwards seems to work to a suitable degree.
Having included both methods is desirable under certain limiting conditions, such as the force not being differentiable (and so Newton's method wouldn't be applicable), or if no initial guesses could be made (with the force at $z = 0$ being asymptomatic, for example). So while newton's method is preferred due to computational simplicity and quick calculation time, the bisection method is desirable under the conditions that Newton's method cannot be calculated.

Appendices

0.1 Root Finding Functions Matlab Code

```
format long
clear all

f = @(x)cos(x);   %define function
f_dash = @(x) -1.* sin(x);   %define derivative of function
g = @(x)cos(x) + x;   %function used for fixed point iteration

bi_root = bisect(f, 0, pi, 0.00005)
fp_root = fpi(g, 0, 3)
new_root = newton(f, f_dash, 1, 0.00005)

function xc = bisect(f, a, b, tol)
    %Bisection Method
    %Input: function handle f; a,b such that f(a)*f(b) <0
    %       and tolerance tol
    %Output: Approximate solution xc
    %Source: Sauer, 3rd Edition
    if sign(f(a))*sign(f(b)) >= 0
        error('f(a)f(b)<0 not satisfied!') % ceases execution
    end
    while (b - a) / 2 > tol
        c = (a + b)/2;
        fa = f(a);
        fc = f(c);
        if fc == 0 %c is a solution, done
            break
        end
        if sign(fc) * sign(fa) > 0 %a and c make the new interval
            a = c;
        else %c and b make the new interval
            b = c;
        end
    end
    xc = (a + b) / 2; %new midpoint is best estimate
end
function xc = fpi(g, x0, k)
    %Fixed-Point Iteration
    % Input: function handle g, starting guess x0, number of iteration steps k
    % Output: Approximate solution xc
    %Source: Sauer, 3rd Edition
    x(1) = x0;
    for i = 1:k
        x(i + 1) = g(x(i));
    end
    xc = x(k + 1);
end
function xc = newton(f, f_dash, x0, tol)
    %Newton's Method
    %Input: function handle f, function handle derivative of f f_dash,
    %initial value x0, and maximum error tol
    %Output: Approximate solution xc
```

```
    %Source: Sil, Rohit (1)
    x(1) = x0;
    i = 1;
    while abs(f(x(i))) > tol;
        x(i+1) = x(i) - f(x(i)) / f_dash(x(i));
        i = i + 1;
    end
    xc = x(i);
end
```

0.2 Stationary Point Finding Functions Matlab Code

```
format long
clear all

f = @(x) (x - 7/11).^2 .* (x + 3/13) .* exp(-1 .* x.^2);
f_dash = @(x)(exp(-1.*x.^2).*(175 - 3572.*x + 4369.*x.^2 + 3278.*x.^3 - 3146.*x.^4))./1573;
f_ddash = @(x) (2.*(-1786 + 4194.*x + 8489.*x.^2 - 10661.*x.^3 - 3278.*x.^4 +
    3146.*x.^5))./(1573.*exp(x.^2));

gss_est = gss(f, -2, -0.5, 10)
spi_est = spi(f, -2, -1, 0, 10)
new_est = newton(f_dash, f_ddash, -1, 0.005)

function y = gss(f, a, b, k)
    % Golden section search for minimum of f(x)
    % Start with unimodal f(x) and minimum in [a, b]
    % Input: function f, interval [a, b], number of steps k
    % Output: approximate minimum y
    % Source: Sauer, 3rd Edition
    gss_count = 0;
    g = (sqrt(5)-1) / 2;
    x1 = a + (1 - g) * (b - a);
    x2 = a + g * (b - a);
    f1 = f(x1); f2 = f(x2);
    for i = 1:k
        gss_count = gss_count + 1;
        if f1 < f2   %if f(x1) < f(x2), replace b with x2
            b = x2; x2 = x1; x1 = a + (1 - g) * (b - a);
            f2 = f1; f1 = f(x1);  %single function evaluation
        else   %otherwise, replace a with x1
            a = x1; x1 = x2; x2 = a + g * (b - a);
            f1 = f2; f2 = f(x2);  %single function evaluation
        end
    end
    y = (a + b) / 2;
end
function y = spi(f, r, s, t, k)
    % Input: inline function f, initial guesses r, s, t, steps k
    % Output: approximate minimum x
    % Source: Sauer, 3rd Edition
    spi_count = 0;
    x(1) = r; x(2) = s; x(3) = t;
    fr = f(r); fs = f(s); ft = f(t);
```

```
    for i = 4:k + 3
        spi_count = spi_count + 1;
        x(i) = (r + s) / 2 - (fs - fr) * (t - r) * (t - s) / (2 * ((s - r) * (ft - fs) - (fs - fr)
            * (t - s)));
        t = s; s = r; r = x(i);
        ft = fs; fs = fr; fr = f(r);     %single function evaluation
    end
    y = x(end);
end

function xc = newton(df, ddf, x0, tol)
    %Newton's Method
    %Input: function handle derivative of f df, function handle double derivative of f ddf,
    %initial value x0, and maximum error tol
    %Output: Approximate solution xc
    %Source: Sil, Rohit (1) (slightly changed from source)
    x(1) = x0;
    i = 1;
    new_count = 0;
    while abs(df(x(i))) > tol
        x(i+1) = x(i) - df(x(i)) / ddf(x(i));
        i = i + 1;
        new_count = new_count + 1;
    end
    xc = x(i);
    new_count
end
```

0.3 R2.3 Sample Matlab Code

```
function z = find_eq(T,a,z)
% find_eq.m - find equilibrium position along z axis
%
% Usage:
% z = find_eq(T,a);
% z = find_eq(T,a,initial_guess);
% where T = T-matrix, a = field vector.

newton = true;

z_precision = 1e-4;
short_distance = 1e-5;
zpoints = 45;

if nargin < 3
       z = 0;
end

z_old = z + 10*z_precision;

if newton

% Newton's method
while abs(z-z_old) > z_precision
```

```
        a2 = translate_z(a,z);
        a3 = translate_z(a2,short_distance);

        p = T * a2;
        f1 = force_z(a2,p);

        p = T * a3;
        f2 = force_z(a3,p);

    dz = short_distance * f1/(f1-f2);

    z_old = z;

        z = z + dz;

end

else % end of if newton

% Bisection method

% Need initial guess

% Guess the radius
size_T = size(T);
radius = size_T(1)/(2*pi);

z = linspace(-radius,3*radius,zpoints);

fz = zeros(size(z));

for nz = 1:zpoints

        a2 = translate_z(a,z(nz));

    p = T * a2;
    fz(nz) = force_z(a2,p);

    if fz(nz) < 0
        z1 = z(nz-1);
        z2 = z(nz);
        f1 = fz(nz-1);
        f2 = fz(nz);
        break
        end

end

if f1 == 0
        z = z1;
    return
end

if nz == zpoints
        error('Starting points for bisection not found');
```

```
end

% Now the actual bisection search

while z2 - z1 > z_precision

       z3 = (z1+z2)/2;

    a2 = translate_z(a,z3);

       p = T * a2;
       f3 = force_z(a2,p);

       if f3 == 0
       z = z3;
       break
    end
       if f1*f3 < 0
       z1 = z3;
       f1 = f3;
       else
              z2 = z3;
       f2 = f3;
       end

end

z = z3;

end % end of if bisection (ie not newton)

return
```

References

[1] Rohit, Sil, 2019. *Newton Method using Matlab Code*, Matlab Answers, Accessed 06/09/2020 https://au.mathworks.com/matlabcentral/answers/442352-newton-method-using-matlab-code