Matthew Lacaire
UID: 205288439
CEE/MAE M20
November 7, 2020

Homework 5

a. Introduction

In this multi-part problem, I use the bisection and fixed point iteration methods to approximate the roots of a function. I utilize the function handle capabilities of MATLAB in the process, and also analyze the limits of the fixed point iteration process.

b. Models and Methods

In part (a), I implement the bisection method, which takes a function to analyze, lower and upper bounds, and a given tolerance. This method is based on having answers of different sign, and systematically decreasing the distance between them so that a root is found numerically. I also use the fixed point iteration method, which uses the concept of the fixed point (g(x) = x) to estimate the value of roots in a given range.

c. Calculations and Results

I first begin discussion of the main file. In my main file, hw5_205288439_p1.m, I provide all of the necessary information to actually use functions. Code is below:

```
%% Bisection Method
b = 2;
a = 1;
tol = 1e-5;
f = @fun1;

[xRoot, numIter] = biSection(f,a,b,tol);

fprintf('Approx. root is %.10f, done in %.0f iterations\n',
xRoot, numIter)

%% Fixed point iteration

maxIter = 100;

f2 = @fun2;
x0 = 3;
[xStar, xRoot2] = fixPoint(f2, x0, tol, maxIter);

fprintf('Approx. root is %.10f, with fixed point %.5f\n',
xRoot2, xStar)
```

In the first section, I provide everything necessary to run the yet-unmentioned biSection function: b, a, tolerance, and the given function. Note that the function is called fun1, but I use the function handle to assign fun1 to simply *f*. Note that instead of using this call, I could have created a one-line *anonymous function* in this main file instead of calling to a different file. In keeping with homework requirements, I kept fun1 (and fun2 later) separate.

In the second part, I again use the handle to reference the "fun2" as f2. I also provide an x0 (initial guess), tolerance (from above), and maxIter (max. Allowable iterations).

In both of these sections, I print out the outputs. Namely, in the first bisection portion, I print the approx. root and #of iterations required. In the second portion, I print the approximate root along with the original function's fixed point.

Fun1 and fun2 are simple to describe, because they are just functions that correspond to f(x) = 1+0.5*sin(x)-x, and g(x) = 1+0.5*sin(x), respectively. Here's the fun1 code, which I believe is self-explanatory after what I've explained:

```
function fx = fun1(x)

fx = 1 + 0.5*sin(x) - x;

End
```

Note that as required, fx is the output, and x is the input. Fun2 is almost identical.

Now, I begin discussing the actual programs that I wrote. Here's the code for the biSection function:

```
function [xRoot, numIter] = biSection(f, a, b, tol)


nMax = (log(b-a) - log(tol) / log(2)); %nMax calculated w/given
formula.
xRoot = 0;  %instantiating xRoot.
numIter = 1; %Initial number of iterations.
continueLoop = true; %flag variable set to true.
while (numIter <= nMax && continueLoop)
    xRoot = (a+b)/2;
```

```
    if (f(xRoot) == 0  || (b-a)/2 < tol) %If function of x value
(xRoot) has output 0
        ...or the tolerance is satisfied, we stop the loop.
        fprintf('%f', xRoot);
        continueLoop = false; %Ending while loop
    end
    numIter = numIter + 1; %Adding one more to numIter per loop.
    if (sign(f(xRoot)) == sign(f(a)))
        a = xRoot;
    else
        b = xRoot;
    end
end

numIter = numIter - 1;
```

This function takes in f (our function handle), a, b, and allowable tolerance. For consistency, I subtract one from *numIter* considering that it begins at 1 before the first iteration has even begun. I also needed to instantiate a true bool flag variable *continueLoop* to ensure the loop will stop when conditions are reached. I instantiate xRoot (output of root) to zero, so the program recognizes that it exists as a variable.

The program first tries the midpoint of a and b as the xRoot, in line with "bisecting" the given domain. Then, it checks to see if this is actually zero by running f(xRoot), with f being the function handle we declared in the main code. It also checks to see if the bounds (a,b) divided by two are smaller than the tolerated difference from the actual zero. Lastly, the program checks to see if the sign of f(xRoot) equals the sign of f(a). If so, the function "redefines" the domain a-b, assigning xRoot to a (the new left bound). If signs are different, the function sets the new b to xRoot. This ensures that the program will always have endpoints of different signs. Then, the program continues through the loop again until tolerance is satisfied or the result is exact. Then, finally, the program ceases the loop eventually and returns the approximate (or exact, depending on which condition was satisfied) result along with the number of iterations.

The logic behind the fixPoint is different. The code is below:

```
function [xStar, xRoot] = fixPoint(f2, x0, tol, maxIter)
%Declaring function that returns

iter = 1; %Iteration tracking variable.
```

```
x = zeros(length(maxIter)); %Initializing array of zeros for
xRoot values.
y = zeros(length(maxIter)); %Initializing array of zeros for
xStar values (the fixed point of original f(x)).
x(1) = x0; %Initializing first elements to initial guess.
y(1) = x0;
while(abs(f2(x(iter) - x(iter)) > tol && iter < maxIter)) %While
loop with conditions ensuring
    ...iteration continues until max iterations or accuracy is
achieved.
    x(iter+1) = f2(x(iter)); %Assigning f2(x) to x(n+1). This is
for the roots case.
    y(iter+1) = f2(y(iter)) - y(iter); %This is the same, with
y(iter) subtracted to find fixed point of f(x), not root.
    iter = iter + 1; %increasing iteration by one.
end

xRoot = x(maxIter); %Assigning final value (when loop stops) to
final (most accurate) value.
xStar = y(maxIter); %Same for fixed point.
```

Here, the function also has a counting variable "iter". I instantiate two vector of length maxIter, which fill with the calculated values of g(x) (which is fun2), and f(x) (fun1). These are x and y respectively.

Then, I assign the initial condition to x(1) and y(1). While the error between f2 (which we called earlier in the main function; fun1 is f2), of a given array value and the actual array value is large enough, we keep iterating and tallying the number of iterations.

In short, the only thing that this loop does is check to see if we've exceeded the number of allowed iterations or if the error is small enough to stop the loop. If not, we calculate the next array value of f1 or f2 (input x) as the output of the function of the last input x. Note that f1 isn't actually mentioned here, but it's the same thing as f2 minus x! Thus, in the line

```
y(iter+1) = f2(y(iter)) - y(iter);
```

I'm actually implicitly using f1 to calculate the fixed point of the function, not xRoot.

Then, acknowledging that the loop stops abruptly when iterations are maxed or error is minimized, I take the absolute last value in the x and y arrays as xRoot and xStar (zero and fixed point, respectively).

Here are my outputs for both:
biSection, with a = 1, b = 2, tol = 10e-5:

```
Approx. root is 1.4987030029, done in 16 iterations
```

fixPoint, with x0 = 0, tol same, and maxIter = 100:

```
Approx. root is 1.4987011335, with fixed point 0.65162
```

d. Discussion

Here I address the last question in the homework, part (iii) and explain my reasoning. First, I change g(x) to gx = 3 + 2*sin(x), and compute with x0 = 3:

```
Approx. root is 4.6838231311, with fixed point 1.04058
```

With x0 = 0:

```
Approx. root is 1.0008159523, with fixed point 2.74042
```

Visually, neither of these are correct. This is because the g(x) of the function (set so that the g(x) = x = y) exceeds a certain slope as it approaches the suspected fixed point value (root). Consider the equation sourced online:

$$\alpha - x_{n+1} = g'(c_n)(\alpha - x_n)$$

Alpha is the expected root value, with xn+1 the next iterated x value. g'(cn) is the slope at the test point (xn). The equation itself comes from arguments about error decrease, culminating in the fact that:

$$\frac{\alpha - x_{n+1}}{\alpha - x_n} = g'(c_n)$$

This is why the slope appears even though it was never actually a part of the algorithm. I understand the proof, but find it tedious to write it all down.

If and only if the value g'(cn) has an absolute value less than one does the solution converge. In other words, per each iteration, the term on the right hand side shrinks as a

result, minimizing error. Visually, the slope of the equation y = 3+2*sin(x) has a slope of absolute value more than one close to the true fixed point.

Also, it's worth also mentioning the shortcomings of the bisection method. It seems like it wouldn't work if we had bounds of the same sign at the beginning (repeated roots of a polynomial, etc.). Also, the chosen a, b must be sufficiently small. If there's a complicated oscillation or something between a and b, the program seems like it would struggle or frankly be unable to assign accurate results to the x root.