**BME 313L: Introduction to Numerical Methods in Biomedical Engineering**

**Lab Report**

**Lab #3 Chapter 5. Roots: Bracketing Methods**

**Last name, First name: Liu, Vincent**

**EID: VL5649**

**Lab Section: 14035 (Tuesday 9:30-12:30)**

**Problem 1**

**< bisection method >** A majorproblem in the successful design of implantable issues is the availability of oxygen for respiring tissues, which is determined by the spatial access of tissues of blood capillaries that bring oxygen carrying red blood cells. A classic model in this field is the Krogh cylinder (Fournier, 1999). Imagine a tissue space with cells surrounding a cylindrical capillary. Oxygen and other metabolites arriving into the capillary axially due to the fresh flow of oxygenated blood will diffuse from the capillary radially toward the tissue, where they will be consumed by the cells. The solution of Krogh cylinder problem yields an expression for the critical distance into the tissue, beyond which no more solute is available, denoted by :

where .

Estimated parameters in human body are:

the metabolite tissue diffusivity 9x10-6 cm2/s

the blood plasma velocity 0.006 cm/s

capillary radius 0.0005 cm

capillary wall thickness 5x10-5 cm

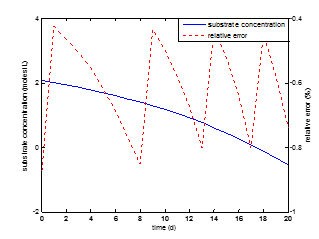
overall metabolite mass transfer rate 5.75x10-5 cm/s

6 μmole/cm3

0.01 μmole/(cm3 s)

Develop an M-file to generate plots of (μm) versus and approximate error versus from 0.001 cm to 0.1 cm in increments of 0.01 cm. Specify the maximum allowed iterations and desired relative error when calling ***bisect.m***, and explain how you determine these two parameters. Note that default values do not guarantee convergence.

The figure should (1) include two y-arises to indicate and respectively. (2) Include legend. (3) make blue solid line, red dashed line. An example figure is shown below.



Example figure with arbitrary data

**MATLAB code:**

z = .001:.01:.1; %creates vector

DT = 9e-6; %given parameters

V = .006;

rc = .0005;

tm = 5e-5;

K0 = 5.75e-5;

C0 = 6;

R0 = .01;

for i = 1:length(z) %loop for values of z

f=@(y,z) ((y .^2 \* log(y .^2)) - (y .^2) + 1 - (4\*DT\*C0/(R0\*((rc+tm)^2))) + ((4\*DT)\*(y .^2 - 1)\*z)/ (V\*(rc^2)) + (((2\*DT)\*(y .^2 - 1))/ (rc \* K0))); %function

[root,fx,ea] = bisect(f,4.9,10.5,.0001,50,z(i)); %outputs and parameters for bisect

rcrit = root .\* (rc + tm); %calculates rcrit

rCrit(i) = rcrit; %creates vector

error(i) = ea; %creates vector

end

[hAx,hLine1,hLine2] = plotyy(z,rCrit,z,error); %plots graph

hLine2.LineStyle = '--'; %makes dotted line

ylabel(hAx(1),'rCrit') %left axis

ylabel(hAx(2),'Approximate Error') %right axis

xlabel('z')

**MATLAB function:**

The purpose of this function was to model, using the Krogh cylinder, the amount of solute beyond a capillary. To do this, we have to pass an equation (for the Krogh cylinder) from a script over to the bisect function in order to solve for the roots. Using the roots, we can then solve for r\_crit—the critical distance past the tissue where no solute is available. To do this, we first generate a vector for one of the variables, z, for theoretical values. We can then pass the equation as an anonymous function and solve.

z = .001:.01:.1; %creates vector

This first line of code generates z values that we can plug into the equation

DT = 9e-6; %given parameters

V = .006;

rc = .0005;

tm = 5e-5;

K0 = 5.75e-5;

C0 = 6;

R0 = .01;

These 7 lines of code were the given parameters for the Krogh cylinder that model the parameters of the human body. These values will be used later on when solving for the roots.

for i = 1:length(z) %loop for values of z

This line of code creates a for loop for the z vector. In doing so, we can work element wise with each number in the vector and pass each value individually to the bisect function.

f=@(y,z) ((y .^2 \* log(y .^2)) - (y .^2) + 1 - (4\*DT\*C0/(R0\*((rc+tm)^2))) + ((4\*DT)\*(y .^2 - 1)\*z)/ (V\*(rc^2)) + (((2\*DT)\*(y .^2 - 1))/ (rc \* K0))); %function

This line of code is the equation of the Krogh cylinder, written as an anonymous function so that it can be passed to the bisect function.

[root,fx,ea] = bisect(f,4.9,10.5,.0001,50,z(i)); %outputs and parameters for bisect

This line of code calls the bisect function and passes all of the variables back to the main function.

rcrit = root .\* (rc + tm); %calculates rcrit

rCrit(i) = rcrit; %creates vector

These 2 lines of code calculate rCrit using the y value solved by the bisect function and pass it into the rCrit vector, taking advantage of the for loop used to pass values of z to the function.

error(i) = ea; %creates vector

This line of code also takes advantage of the for loop, passing the approximate errors of each successive iteration to the error vector.

end

This line of code is very straightforward, closing the for loop.

[hAx,hLine1,hLine2] = plotyy(z,rCrit,z,error); %plots graph

This line of code uses the yy plot function to plot rCrit vs z and the error vs z using different axis scales.

hLine2.LineStyle = '--'; %makes dotted line

This line of code denotes that the second line (error vs z) should be a dashed line.

ylabel(hAx(1),'rCrit') %left axis

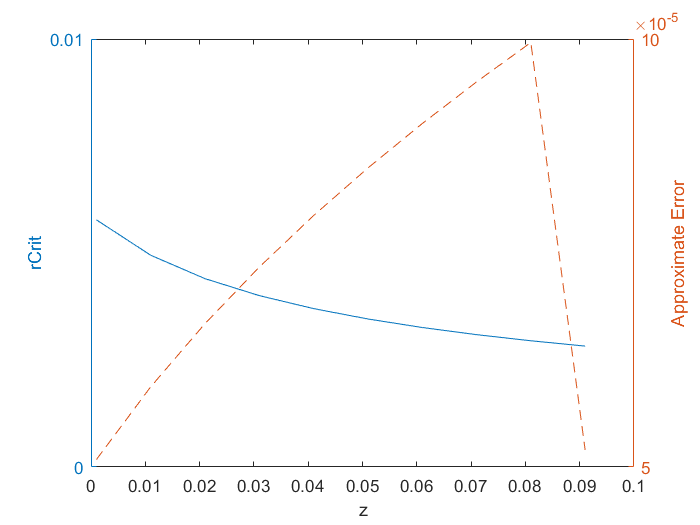
ylabel(hAx(2),'Approximate Error') %right axis

These last 2 lines of code label the left and right hand values for the y axis according to the rCrit and approximate error plots respectively.

xlabel('z')

This last line of code labels the x axis.

**Results:**



**Discussion:**

As shown by the results of the graph, the bisect function was able to successfully calculate the rcrit value that satisfied the approximate error under a specified value. Despite the value of approximate error gradually increasing before finally coming down, because of the way the bisect function works, the root (and subsequently rCrit) continues to gradually approach the value satisfying the desired error. To determine the upper and lower guesses for bisect, we could plug in the highest and lowest values of z to find the highest and lowest guesses, respectively. When solved for, we found that the lowest and highest values that y could be were around +/- 4.95765 and +/- 10.4981. Because of the nature of the problem, we could disregard the negative portion of the roots, caring only about the positive portion.

From this problem, we learned how to use equation modeling real life scenarios to solve for parameters in certain conditions. We also learned how to pass equations as anonymous functions in order to use root solving methods to solve for variables. Furthermore, we reviewed how to use for loops in order to work with elements of a vector. Lastly, we learned how to use yyplot in order to plot graphs of 2 different scales on the same plot.

**Problem 2**

**<false position method>** Magnetic resonance imaging (MRI) is a rapidly advancing area of biomedical imaging. MRI interrogates the nuclear environment of hydrogen ions in the soft tissue of the body, providing contrast imaging as well as functional imaging of tissues. A common device used for generating magnetic field in MRI experiments is the solenoid. A solenoid coil can be used to couple energy into a sample and to detect the time-varying magnetic flux density during the acquisition of signals. Assume the solenoid has length and radius . The long axis of the solenoid is along the y-axis. The solenoid produces a flux density (*B*) at any observation point y, which depends on the total number of turns in the coil , the amount of current, and , the permeability of free space, which is typically H/m. On the other hand, the inductance of a solenoid (*L*) can be approximately expressed as:

Compute, using the false position method in Matlab, the radius of the solenoid coil (cm), to provide inductance of 2.6 μH, assuming the following parameters: turns, cm.

Complete the code in solver *falsepos.m*, then use *falsepos.m* to compute (cm). Use default approximate relative error 0.0001%, lower guess 0 cm and upper guess 100 cm. How many iterations are needed in order to get ? Report the final . To achieve , do you expect the bisection method to require more or fewer iterations? Please explain why. Then run the solver *bisect.m* to verify.

Paste the code that you insert in *falsepos.m* here and briefly comment them:

[1] xr = xu - (func(xu,varargin{:}) \* (xl - xu) / (func(xl,varargin{:}) - func(xu,varargin{:})));

This first line of code is just the equation for the false position method—it vis used to calculate the midpoint between 2 arbitrary points.

[2] ea = abs((xr - xrold)/xr) \* 100;

This line of code just calculates the approximate error when the root is not found.

[3] if test < 0

We know that this line of code works because basically we are trying to find the root of the function. If the test is less than 0 then we can move our arbitrary upper bound to our midpoint (as we know the root must be below).

[4] elseif test > 0

Same as the line above except with the lower bound.

In line [2], why do we need a if-statement (*if* *xr~=0)* before calculating *ea*?

The statement xr~=0 is true if xr is not equal to 0. We need this statement because we only need to calculate ea when xr isn’t 0. If xr is 0, then we know that falsepos has found the root and doesn’t need to calculate anymore (error would be 0).

**MATLAB code:**

N = 10; %given parameters

l = .2;

L = 2.6e-6;

mu0 = 1.25663706e-6;

f =@(a,L) mu0\*N^2\*pi()\*a^2/l^2\*((l^2+a^2)^(1/2)-a)-L; %equation

[root,fx,ea,iter] = falsepos(f,0,1,.0001,50,L); %calls falsepos function

%[root,fx,ea,iter] = bisect(f,0,1,.0001,50,L); %calls bisect function

a = root \* 100 %outputs a in cm

n = iter %outputs number iterations

finale = ea %outputs final error

**MATLAB function:**

The purpose of this function was to compare the efficiency of 2 root solving methods in order to solve for a value. To do this, we pass a function to both the falsepos and bisect functions and compare the number of iterations to reach the value desired.

N = 10; %given parameters

l = .2;

L = 2.6e-6;

mu0 = 1.25663706e-6;

These first 4 lines of code outline the parameters of the solenoid for which we are calculating inductance.

f =@(a,L) mu0\*N^2\*pi()\*a^2/l^2\*((l^2+a^2)^(1/2)-a)-L; %equation

This line of code outlines the equation of a solenoid that we are solving for. L is subtracted from both sides in order to set the equation to 0 (so that we are solving for the roots). It is written as an anonymous function so that we can pass it to our falsepos function to be solved for.

[root,fx,ea,iter] = falsepos(f,0,1,.0001,50,L); %calls falsepos function

%[root,fx,ea,iter] = bisect(f,0,1,.0001,50,L); %calls bisect function

These 2 lines of code call the falsepos and bisect functions, respectively. The bisect function is currently commented out because the question is more interested in the falsepos results rather than the bisect results. Both functions were allowed the same upper and lower bounds, as well as the same number of maximum iterations in order to compare which function is more efficient

a = root \* 100 %outputs a in cm

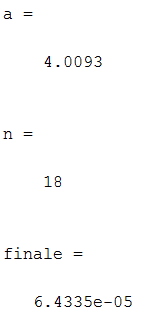
This line of code calculates the a values (in centimeters) and outputs it so that the user can see.

n = iter %outputs number iterations

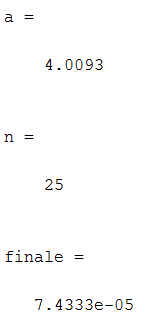
finale = ea %outputs final error

These last 2 lines of code output the last number or iteration and final error and output them to the user. Using this with both the falsepos and bisect functions, we can compare efficiency.

**Results:**

****

(For the falsepos function)



(For the bisect function)

**Discussion:**

As shown by the results of running both the falsepos function and the bisect function, the false position method (in this situation) is more efficient than the bisection method. The falsepos function only took 18 iterations to calculate the value of A while the bisect function took 25 iterations to reach the same value of A. The falsepos function was also marginally more accurate below the minimum error that we set, as shown by the final error values of both functions, although more iterations could have been used in the bisect function to reach a similar error. These results point to the efficiency of the falsepos function over the bisect method, at least in the context of this problem; however, that may not always be the case so the usefulness of the bisect method is not negated.

From this problem, we reviewed how to pass equations over to functions. We also reviewed how to work with outputs from the function, in the main script. Furthermore, we learned how the falsepos and bisect functions worked, as we were asked to fill in lines for the falsepos function and could compare it to the bisect function code. Lastly we learned about the efficiency of both functions for certain cases.

**Problem 3**

**< bisection method >** The mean squared displacement for cell migration in two-dimensional media is given by Dunn equation:

where stands for mean squared displacement

is the root mean squared cell speed (m/s)

is the directional persistence time (s)

is the time interval (s)

White blood cells stimulated with chemoattractant factors migrated at 20 μm/min (i.e. *S*) on an expanded polytetrafluoroethylene, used as a vascular prosthetic biomaterial. What is the persistence time , necessary for a population of white blood cells to achieve a mean squared displacement of 4.3x10-3 cm2 in 3 hours? Plot versus (minutes) using function *fplot* to help you determine a reasonable lower guess and upper guess of (minutes) before calling bisect.m. Use default value of desired approximate relative error and maximum number of iterations. Your initial guess of lower bound and upper bound of the root should make the solver converge in no more than 23 iterations.

**MATLAB code:**

t = 3\*60; %parameters, units fixed

ds = 4.3e-3/100^2;

S = 20e-6;

f =@(P) (2 .\* S.^2 \* (P .\* t - P.^2 .\* (1 - exp(-t./P)))-ds); %equation

fplot(f,[0 5]) %graph to find root

[root,fx,ea,iter] = bisect(f,2,4,.0001,23); %calls function

P = root %outputs

n = iter

**MATLAB function:**

The purpose of this problem was very similar to that of the first problem’s. Basically, we were supposed to pass an equation to the bisect function again, in order to solve for a value. In this case, we were supposed to first plot our function in order to graphically make an estimate of where our root was. In doing so, we could reduce the number of iterations that bisect needed in order to calculate the root within the approximate error that we set.

t = 3\*60; %parameters, units fixed

ds = 4.3e-3/100^2;

S = 20e-6;

These 3 lines of code are the given parameters for the problems as well as the result that we were supposed to be solving for. Special care needed to be used to ensure that the numbers would work—specifically in the units for each of the values. Since we wanted to solve for P in minutes, the rest of the values had to be left in minutes. For length, all units had to be converted to meters.

f =@(P) (2 .\* S.^2 \* (P .\* t - P.^2 .\* (1 - exp(-t./P)))-ds); %equation

This line of code, like the previous 2 problems, creates an equation as an anonymous function that is then passed to the bisect function. We had to set the equation given equal to 0 by subtracting ds from both sides because the bisect function only solves for roots.

fplot(f,[0 5]) %graph to find root

This line of code plots the equation that we outlined in the previous line, so that we could use graphical measure in order to determine about where the root was. When no window was set, the graph was all over the place, so by setting a window from 0 to 5 we could better gauge numbers. Based on the graph, you could tell that the root was somewhere around 3, but to be certain I made sure to incorporate all values in the area in the bisect function.

[root,fx,ea,iter] = bisect(f,2,4,.0001,23); %calls function

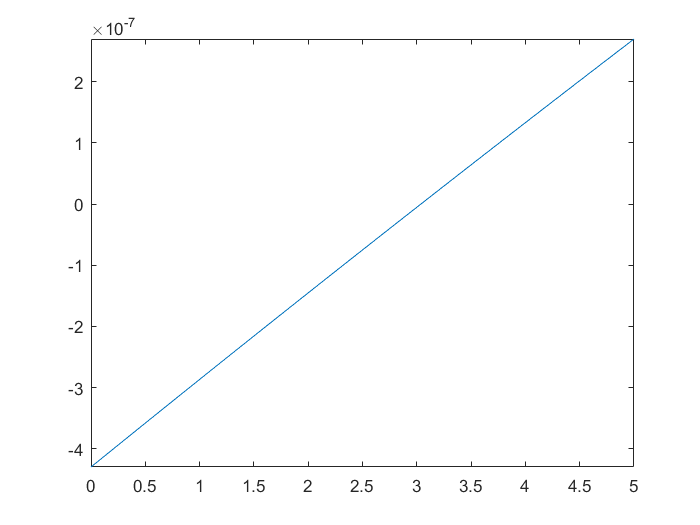
This line of code calls the bisect function and passes all of the variables back to the main function.

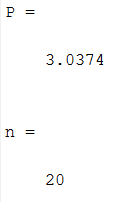
P = root %outputs

n = iter

These 2 lines of code display the P value that we solved for using bisect, as well as the number of iterations used to reach that value.

**Results:**





**Discussion:**

As shown by the results, we could use graphical methods in order to reduce the number of iterations needed for root finding functions. In this problem, we experimented with setting the number of iterations and relative error to a constant value while playing with the upper and lower bounds. In doing so we could see how changes to the upper and lower guesses would affect the number of iterations. From this we could reach the conclusion that even slight changes to the upper and lower guess can reduce the number of iterations (for example: going from 2-4 instead of 0-4 reduces the iterations from 21 to 20).

From this problem reviewed how to pass equations to functions in order to solve for roots (done as an anonymous function). We also reviewed how to plot functions using the fplot function in MATLAB. Lastly, we reviewed how to