AEP 4380: Homework 3

Gregory Kaiser

September 25th, 2019

1 Problem Background and Solution Overview

In this assignment, I found the roots of a function via two methods of root finding: the bisection method, and Regula Falsi, aka the False Position method. Both methods require the assumption that a single root exists within the range given to execute properly. If more than one root lies between the range of values being searched, either method will fail to return all of the desired values.

I was asked to find all values of x that satisfy the following:

$$J_0(x)Y_0(x) = J_2(x)Y_2(x) \tag{1}$$

Which is the same as asking where:

$$J_0(x)Y_0(x) - J_2(x)Y_2(x) = 0 \tag{2}$$

In other words, find the roots of the expression on the left hand side of equation 2.

After verifying that my function calls were operating properly, I found the first 5 solutions to equation 2 using both bisection and Regula falsi. I then compared the number of evaluations that each algorithm took to generate a root to within the same tolerance value.

2 Solution Description

The first part of my main function simply uses the downloaded header file Bessel.h in order to generate the first three Bessel Functions of the First Kind (J_0), and the first three Bessel functions of the Second Kind (Y_0). I plotted them to confirm that I was accessing the correct functions.

I also used these functions in order to test my bisect and Regula Falsi functions, since their roots are well-known and easily found online.

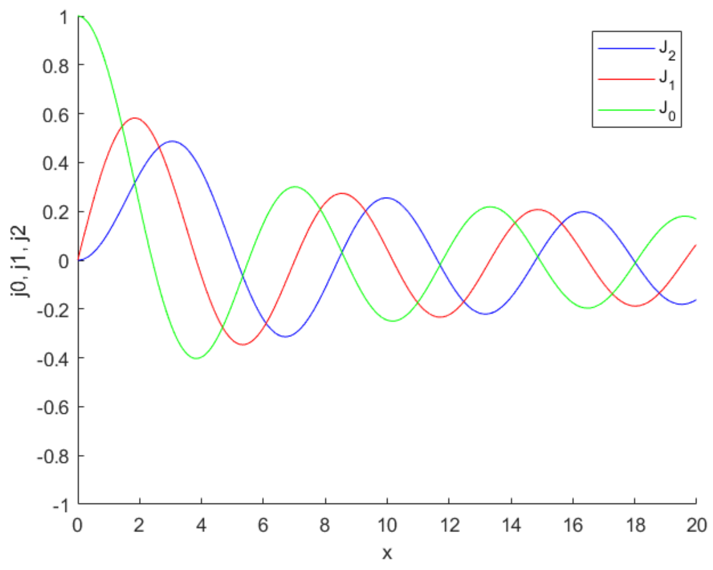


Figure 1: Bessel functions of the First Kind

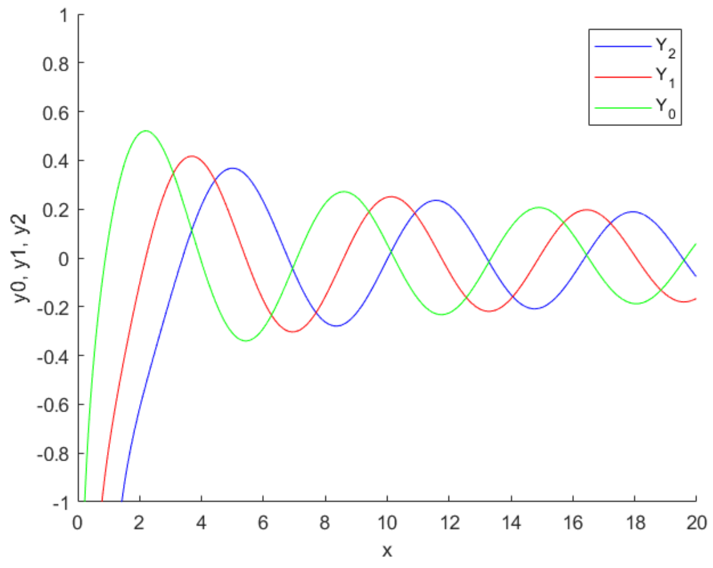


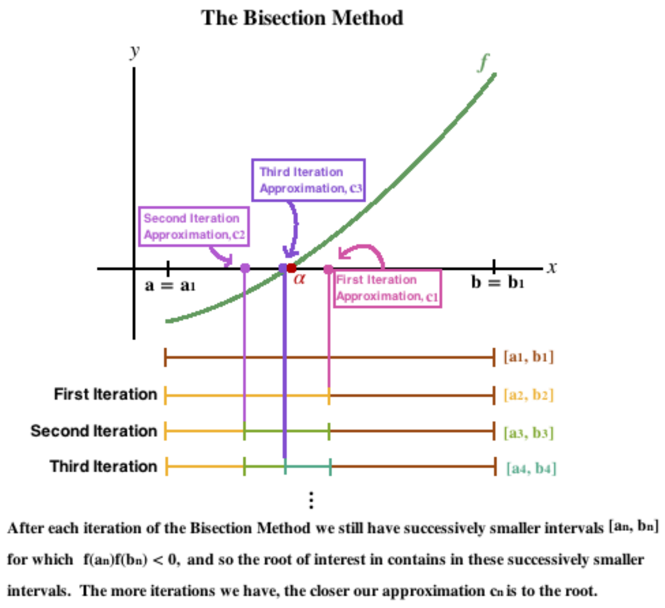
Figure 2: Bessel functions of the Second Kind

2.1 Bisect

I then implemented bisect by using a method header that took an arbitrary function as its argument. So as not to worry about the types Doub and Int in the Bessel function library, I nested those function calls within my own functions above main(). The bisect method also took bounds and a tolerance value.

The bisection method resembles a binary search, where a range of possible positions is found by narrowing the current range by about 50% repeatedly. Between two values on the domain (x_1 and

x_2), bisect finds the value halfway between those (x_3), and checks the sign of the function at that point (f_3) against the sign of the function evaluated at the original bounds (f_1 and f_2). If it has the same sign as, say, f_1, then f_3 becomes the new lower bound for the next iteration of the function. The root is closed in on rapidly in this way, until f_3 is within the tolerance value of zero. The x_3 value at which this loop terminates is the root of the function between those bounds.



After each iteration of the Bisection Method we still have successively smaller intervals [a_n, b_n] for which $f(a_n)f(b_n) < 0$, and so the root of interest in contains in these successively smaller intervals. The more iterations we have, the closer our approximation c_n is to the root.

Figure 3: A diagram of the first few iterations of the bisect method. [2]

If the bounds given to the function were the same, then the assumption that they bound a single root is broken, and the function returns -1 to indicate that this is the case. Otherwise, it will start the process of bisection as described.

I first tested this on a Bessel function of the first kind before moving forward.

2.2 Regula Falsi

I implemented Regula Falsi similarly, by using an arbitrary function as a parameter which is fed to the algorithm along with bounds and a tolerance value.

Regula Falsi uses the slope evaluated using the points that bound the root to estimate where the zero would be, as if it were a linear function. The estimated slope is quite rough when the bounds are far apart, but closes in on the actual value quickly for a well-behaved function.

Using the simple point slope formula, the zero point can be determined by the equation below:

$$(x_3 - x_1)\frac{(f_2 - f_1)}{(x_2 - x_1)} = -f_1 \tag{3}$$

Therefore,

$$x_3 = x_1 - f_1\frac{(x_2 - x_1)}{(f_2 - f_1)} \tag{4}$$

Which simply means that if you move along a line for distance x_3-x_1 with slope determined by f_1 and f_2, you should get back to zero from f_1.

The function is then evaluated at this estimated x_3 point, and its sign is checked just like in bisect. If it matches the sign of either f_1 or f_2, it becomes that side of the bounding values, but this time much closer to the zero of the function, and the cycle is repeated.

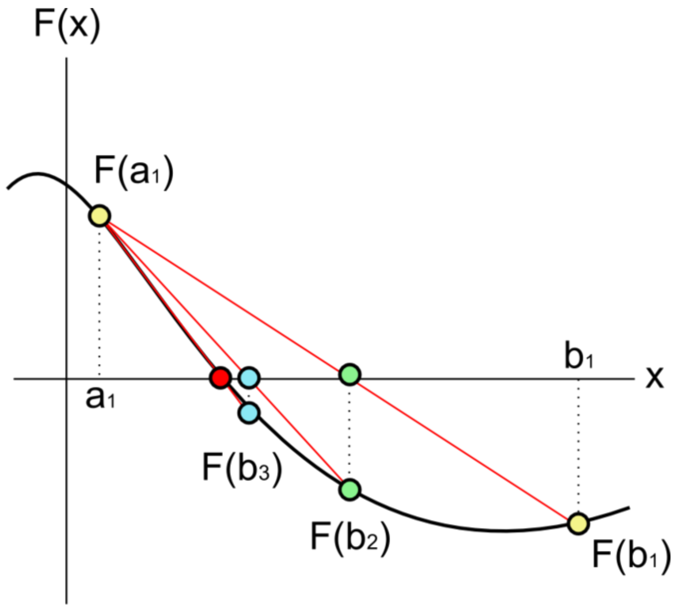


Figure 4: A diagram of the first few iterations of the Regula Falsi method. [2]

If the original bounds are of the same sign, my Regula falsi function returns -1 just like bisect to indicate the lack of roots.

I first tested this on a Bessel function of the first kind, before moving forward.

2.3 Finding the First Five Roots

In order to find the first five roots of equation 2, I needed to feed my bisect and Regula Falsi methods the correct domains on which those roots lie. First, I plotted the function from equation 2 to take a look at what I was dealing with.

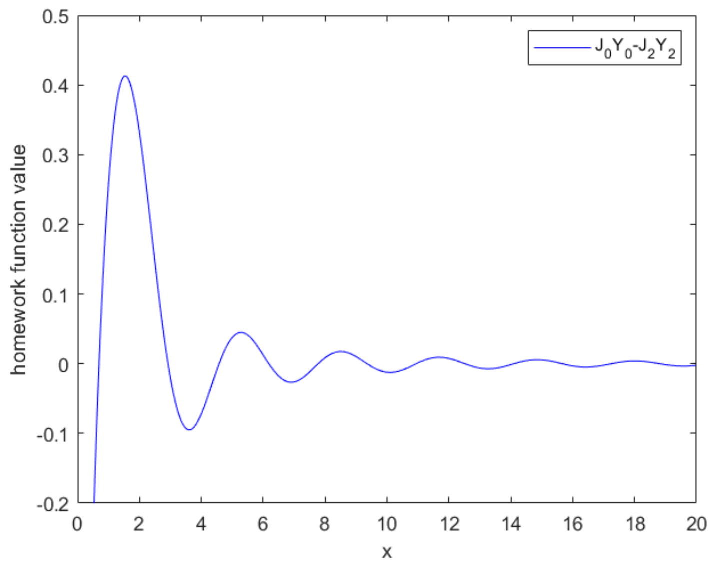


Figure 5: The Homework Function

On an interval of about 0.5, I can be reasonably certain that a Bessel function of this kind won't oscillate across zero more than once. I therefore implemented a loop which steps over intervals of this size and checks them for a root in succession until five roots are found. To be safe, I set the interval to 0.25 for my final function. As this interval gets smaller, my algorithm becomes less efficient, and it becomes less and less useful as the algorithm turns into simply evaluating a function at every point until you cross zero. As it gets larger, however, it becomes more likely that you will catch two roots inside the bounds, return -1 as if a root wasn't found, and miss both of them. Therefore the interval must be tuned slightly based on the function being analyzed.

3 Results and Interpretation

After running my root finding algorithm, I found the first five roots with both methods accurately. I also outputted the number of function evaluation steps up to a given point in the program, to compare the two. It is clear that Regula Falsi worked faster than Bisect for this given function, since it evaluated the five roots in far fewer steps. I believe that this is due to the fact that equation 2, when plotted, looks like it is very nearly linear near its roots. This means that when Regula Falsi is used, it will likely get very close to the root within a single iteration, given its inherent linearity. Perhaps with a function less well behaved, Bisect could work faster than Regula Falsi under those circumstances.

Root found and efficiency of algorithm			
Root number	Root Value	Total Bisect Evaluations	Total Regula Falsi Evaluations
1	0.694391	20	7
2	2.92456	38	12
3	4.57483	54	15
4	6.18169	67	18
5	7.77338	78	20

These values can be confirmed at least to some certainty with a closer look at Figure 3.

In the future it could be beneficial to use a mixture of these methodologies to close in on a root quickly. Using bisect to get to a narrower, more linear section, and then implementing Regula Falsi, could perform even better than either of the methods alone.

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes, The Art of Scientific Computing.* (3rd edit.), Cambridge Univ. Press, 2007, (ISBN 978-0-521-88068- 8, QA297 .N866 2007)

[2] Diagram of the bisection method. *The Bisection Method for Approximating Roots.* http://mathonline.wdfiles.com/local–files/the-bisection-method-for-approximating-roots/Screen%20Shot%202015-01-20%20at%207.54.07%20PM.png

[3] Diagram of the regula falsi method. *Regula Falsi on Wikimedia Commons* https://upload.wikimedia.org/wikipedia/commons/thumb/5/57/Regula_falsi_method.png/668px-Regula_falsi_method.png

Source Code

```
/*  AEP 4380 Assignment #3
    Root Finding

    Run on core i7 with gcc version 8.2.0 (MinGW.org GCC-8.2.0-3)

    Gregory Kaiser Sept 25 2019

*/
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include "../Library/nr3.h"
#include "../Library/bessel.h"

Bessjy Bessel; //an instance of the bessel function

using namespace std; //makes writing code easier

//typedef double Doub;
//typedef int Int;

double pi = 4.0*atan(1.0);

void bisect_Test();
void reguli_falsi_Test();
void bisect_hw();

double j_0(double);
double j_2(double);
double j_0(double);
double y_2(double);
double hw_func(double);


double xmin, xmax;
int n;
int function_evals_b=0;//the number of times that the bessel functions are evaluated
int function_evals_rf=0;
//finds the root of this function between x1 and x2 assuming
//that there is a root between x1 and x2.
double bisect( double(*f)(double), double x1, double x2, double tol){
    //get starting values for f1 and f2
    double f1 = f(x1);
    //function_evals_b++;
    double f2 = f(x2);
    //function_evals_b++;
    double x3 = (x1+x2)/2;//the x value between the first two points
    double f3 = f(x3);
    //function_evals_b++;
    //while the points have opposite sign and also the point
    //between them has a value above tolerance, continue the search
    if((f2>0&&f1>0)||(f2<0&&f1<0)){
        //if the initial values of f1 and f2 are of the same sign
        //then we should assume that there is no root between them
```

```
            //and retry this function with a smaller interval to catch the
            //even number of roots between these values
            return -1; //should never be negative unless this condition is met
        }
        else{
            while(abs(f(x3))>tol){
                if((f3<0&&f1<0)||(f3>0&&f1>0)){
                    //f3 and f1 have the same sign
                    x1 = x3;
                    f1 = f3;
                }
                else{
                    //f3 and f2 have the same sign
                    x2 = x3;
                    f2 = f3;
                }
                x3 = (x1+x2)/2;
                f3 = f(x3);
                function_evals_b++;
            }
        return x3;
        }
    }//end bisect

    //finds the root of this function between x1 and x2 assuming
    //that there is a root between x1 and x2.
    double reguli_falsi( double(*f)(double), double x1, double x2, double tol){
        //get starting values for f1 and f2
        double f1 = f(x1);
        //function_evals_rf++;
        double f2 = f(x2);
        //function_evals_rf++;
        double x3 = x1-f1*(x2-x1)/(f2-f1);//starting x3 value

        double f3 = f(x3);//starting f3
        //function_evals_rf++;
        //while the points have opposite sign and also the point
        //between them has a value above tolerance, continue the search
        if((f2>0&&f1>0)||(f2<0&&f1<0)){
            //if the initial values of f1 and f2 are of the same sign
            //then we should assume that there is no root between them
            //and retry this function with a smaller interval to catch the
            //even number of roots between these values
            return -1; //should never be negative unless this condition is met
        }
        else{
            while(abs(f(x3))>tol){
                if((f3<0&&f1<0)||(f3>0&&f1>0)){
                    //f3 and f1 have the same sign
                    x1 = x3;
                    f1 = f3;
                }
                else{
                    //f3 and f2 have the same sign
                    x2 = x3;
                    f2 = f3;
                }
                x3 = x1-(f1*(x2-x1)/(f2-f1));//the x value between the first two points
                f3 = f(x3);
                function_evals_rf++;

            }
        }
        return x3;

    }//end Reguli Falsi

    double j_0(double x){return Bessel.j0(x);}
    double j_1(double x){return Bessel.j1(x);}
    double j_2(double x){return Bessel.jn(2,x);}
    double y_0(double x){return Bessel.y0(x);}
    double y_1(double x){return Bessel.y1(x);}
    double y_2(double x){return Bessel.yn(2,x);}
    double hw_func(double x){return (j_0(x)*y_0(x))-(j_2(x)*y_2(x));}

    int main(){
        //Testing that the Bessel functions work
        ofstream fp; //output file for j's
        ofstream fp2; //output file for ys
        ofstream fp3; //output file for the homework function
        fp.precision(9);
        fp2.precision(9);
        fp3.precision(9);

        fp.open("hw3_1.dat");
        if(fp.fail()){
            cout<<"cannotopenfile"<<endl;
            return(EXIT_SUCCESS); //defined by standard library
        }
        fp2.open("hw3_2.dat");
        if(fp2.fail()){
            cout<<"cannotopenfile"<<endl;
            return(EXIT_SUCCESS); //defined by standard library
        }
        fp3.open("hw3_3.dat");
        if(fp3.fail()){
            cout<<"cannotopenfile"<<endl;
            return(EXIT_SUCCESS); //defined by standard library
        }

        //generate the set of bessel functions for plotting (part 1)
        //simply confirms that my function calls are correct
        //J's first
        xmax = 20;
        xmin = 0;
        n = 1000;
        double dx = (xmax-xmin)/(n);
        double j0[n];
        double j1[n];
        double j2[n];
        double x = 0;
        for (int i=0; i<n; i++){
            x = i*dx;
            j0[i] = j_0(x);
            j1[i] = j_1(x);
            j2[i] = j_2(x);
            //write data into a file for matlab to parse. needs to be in separate columns
            fp << setw(18) << j0[i] << setw(18) << j1[i] << setw(18)<<j2[i]<<setw(18) << x << endl;
        }

        //now the Y's
        xmax = 20;
        xmin = 0.75;
        n = 1000;
        //dx = (xmax-xmin)/(n);
        double y0[n];
        double y1[n];
        double y2[n];
        x = 0;
        for (int i=0; i<n; i++){
            x = i*dx;
            y0[i] = y_0(x);
            y1[i] = y_1(x);
            y2[i] = y_2(x);
            //write data into a file for matlab to parse. needs to be in separate columns
            fp2 << setw(18) << y0[i] << setw(18) << y1[i] << setw(18)<<y2[i]<<setw(18) << x << endl;
        }

        //and now the hw_function in order to observe its behavior

        xmax = 20;
        xmin = 0.75;
        n = 1000;
        //dx = (xmax-xmin)/(n);
        double output[n];
        x = 0;
        for (int i=0; i<n; i++){
            x = i*dx;
            output[i] = hw_func(x);
            //write data into a file for matlab to parse. needs to be in separate columns
            fp3 << setw(18) << output[i] << setw(18)<< x << endl;
        }

        //now evaluate the roots of the real homework function
        bisect_hw();
        cout<<"bisect evals: "<<setw(15)<<function_evals_b<<endl;
        cout<<"reguli falsi evals: "<<setw(15)<<function_evals_rf<<endl;

        //bisectTest();
        fp.close();
        fp2.close();
        fp3.close();
        return(EXIT_SUCCESS);
    }//end main()

    //hardcoded a bessel function to test my bisect function on
    //a function with well known roots
    void bisect_Test(){
        double root = bisect(j_0,0,4,.001);
        //root should return the first root of J0, which is approx: 2.4048
        cout << root << endl;
        //it gave 2.40625 so im happy with that
    }//end bisect test

    //hardcoded a bessel function to test my reguli falsi function on
    //a function with well known roots
    void reguli_falsi_Test(){
        double root = reguli_falsi(j_0,0,4,.001);
        //root should return the first root of J0, which is approx: 2.4048
        cout << root << endl;
        //it gave 2.40487 so im happy with that
    }//end bisect test

    //Uses the bisect method to find the first five roots of the hw function
    //by moving an interval that is much larger than the slowly moving oscillations
    //of the function itself
    void bisect_hw(){
        /*using the bisect function*/
        double root; //to store the root when it is found
        double root_rf; //to store the root from reguli falsi separately
        int num_roots=0; //the number of roots found so far

        //The following assumes that the function does not cross zero
        //more than once between intervals of int_tol, the interval tolerance
        double int_tol = .25;

        double int_xmin = .2; //the minimum of the interval being checked
        //int_xmin is shifted slightly to the right to start since
        //the special function goes to negative infinity at zero
        double int_xmax = int_tol;
        double int_num = 1; //the number of intervals checked

        while(num_roots<5){
            //find the next root
            root = bisect(hw_func,int_xmin,int_xmax,.0000001);
            root_rf = reguli_falsi(hw_func,int_xmin,int_xmax,.0000001);
            if(root<=0){
                //no root was found, move to next interval
                int_xmin = int_num*int_tol; //keeps the interval to one multiplication error from value
                int_xmax = int_xmin+int_tol; //should have multiplication and addition error
                int_num++;
            }
            else{
                //a root was found! record it and move to next interval
                num_roots++;
                cout << num_roots << setw(15)<< "b root: "<<root<<setw(15)<<"iterations: "<< function_evals_b<<se
                int_xmin = int_num*int_tol; //keeps the interval to one multiplication error from value
                int_xmax = int_xmin+int_tol; //should have multiplication and addition error
                int_num++;
            }
        }
    }//end bisect_hw()

    %
    % MATLAB script to decode and plot AEP4380 data
    % Gregory Kaiser 2019
    %
    load 'hw3_1.dat'
    load 'hw3_2.dat'
    load 'hw3_3.dat'

    x = hw3_1(:,4);
    y1 = hw3_1(:,3);
    y2 = hw3_1(:,2);
    y3 = hw3_1(:,1);
    figure(1); hold on;
```

```
a = plot(x,y1,'b-');
b = plot(x,y2,'r-');
c = plot(x,y3,'g-');
axis([0 20 -1 1])
legend([a ;b; c],['J_2'; 'J_1'; 'J_0'])
xlabel('x')
ylabel('j0, j1, j2')

x = hw3_2(:,4);
y1 = hw3_2(:,3);
y2 = hw3_2(:,2);
y3 = hw3_2(:,1);
figure(2);hold on;
a = plot(x,y1,'b-');
b = plot(x,y2,'r-');
c = plot(x,y3,'g-');
axis([0 20 -1 1])
legend([a ;b; c],['Y_2'; 'Y_1'; 'Y_0'])
axis([0 20 -1 1])
xlabel('x')
ylabel('y0, y1, y2')

x = hw3_3(:,2);
y = hw3_3(:,1);
figure(3);
a = plot(x,y,'b-');
axis([0 20 -1 1])
legend([a],['J_0Y_0-J_2Y_2'])
axis([0 20 -.2 .5])
xlabel('x')
ylabel('homework function value')
print -deps hw2_2.eps
```