Root Finding Using Bisection and False Position

Kevin Juan (kj89)

September 19^{th} 2018

The Numerical Methods and Algorithms

The objective of this assignment was to implement two different algorithms for numerically finding roots of equations using the bisection and false position method. Both methods depend on a root to be bracketed within a range $[x_1, x_2]$ for the root to be determined. For the bisection method, the function is evaluated at the two ends of the initial bracket to determine the sign of $f(x)$. It is required that $f(x_1)$ and $f(x_2)$ have opposite signs for the method to converge. A third point x_3 is picked such that x_3 is halfway between x_1 and x_2 then evaluated. If $f(x_3)$ has the same sign as $f(x_1)$, then the next bracket is changed such that $x_1 \leftarrow x_3$ and $f(x_1) \leftarrow f(x_3)$. If $f(x_3)$ and $f(x_2)$ have the sign, then $x_2 \leftarrow x_3$ and $f(x_2) \leftarrow f(x_3)$. This method of halving the bracket is repeated until a specific level of tolerance is achieved. Typically, bisection is a slow method compared to other root finding algorithms. More on bisection can be found in §9.1 in Press et al[1].

One such method that generally converges faster than bisection is the false position, or *regula falsi*, method. This method is a combination of the bisection and secant method (§9.2 in Press et al[1]). Staring with the bracket $[x_1, x_2]$ within which a root is known to exist, $f(x_1)$ and $f(x_2)$ are evaluated and checked for opposite signs. A line is drawn between the two calculated points. The point at which the line between the two points crosses the x-axis is assigned to x_3. This point is located at $x_1 - f(x_1)\frac{x_2 - x_1}{f(x_2) - f(x_1)}$. Once this point is found, $f(x_3)$ is evaluated. Similar to the bisection method, if $f(x_3)$ has the same sign as $f(x_1)$, then $x_1 \leftarrow x_3$ and $f(x_1) \leftarrow f(x_3)$. If $f(x_3)$ and $f(x_2)$ have the sign, then $x_2 \leftarrow x_3$ and $f(x_2) \leftarrow f(x_3)$. This is repeated until a specified tolerance is achieved. Additional info on false position can be found in §9.2 in Press et al[1].

Implementation and Results

The two methods were implemented on Bessel functions of integer order. The Bessel function of the first order $J_v(x)$ is given as shown:

$$J_v(x) = \left(\frac{1}{2}x\right)^v \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!\Gamma(v+k+1)} \tag{1}$$

The Bessel function of the second order $Y_v(x)$ is shown below.

$$Y_v(x) = \frac{J_v(x)cos(v\pi) - J_{-v}(x)}{sin(v\pi)} \tag{2}$$

Both equations are solutions to Bessel's differential equation as shown below. However, unlike $J_v(x)$, $Y_v(x)$ contains a singularity at $x = 0$.

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - v^2)y = 0 \tag{3}$$

The plot below shows the Bessel functions $J_0(x)$, $J_1(x)$, $Y_0(x)$, and $Y_1(x)$ over the domain $0 \leq x \leq 20$ for $J_v(x)$ and the domain $0.75 \leq x \leq 20$ for $Y_v(x)$.

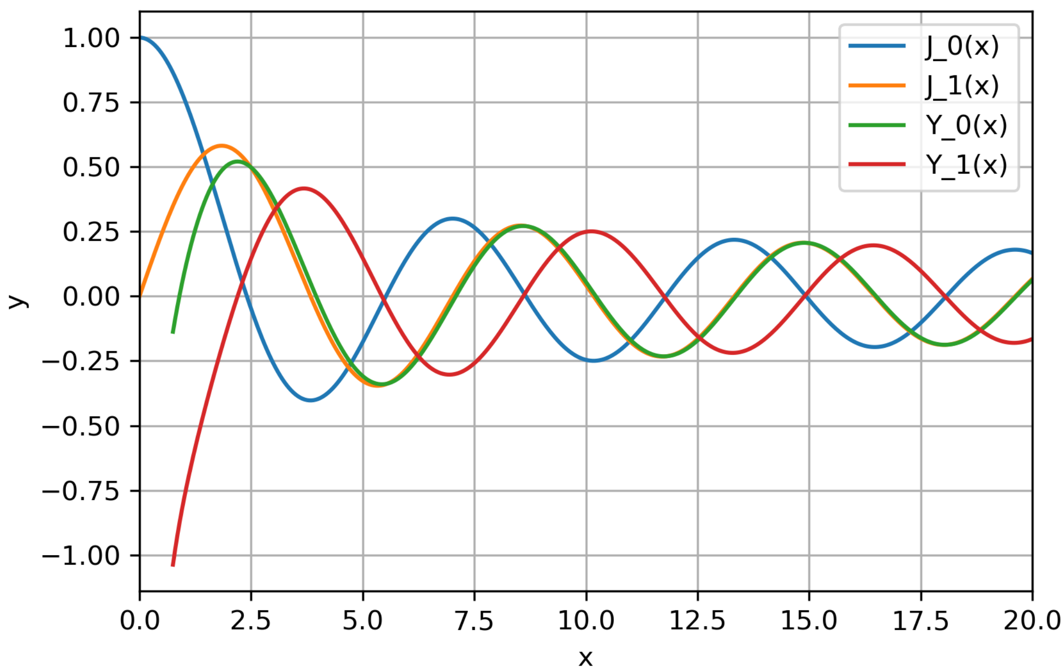

Figure 1: $J_0(x)$, $J_1(x)$, $Y_0(x)$, and $Y_1(x)$ plotted over $0 \leq x \leq 20$ and $0.75 \leq x \leq 20$

The goal of this assignment was to find the first five smallest positive values of x for $x > 0$ that satisfy:

$$J_0(x)J_1(x) = x^2 Y_0(x) Y_1^2(x) \tag{4}$$

While one could write a program that finds the intersection of the two different functions in Equation 4, one could also rearrange the equation as such:

$$f(x) = x^2 Y_0(x) Y_1^2(x) - J_0(x)J_1(x) \tag{5}$$

This effectively turns the problem into a root solving problem for $f(x) = 0$ that can be achieved with bisection or false position. To identify brackets to solve for the roots, Equation 5 was plotted.

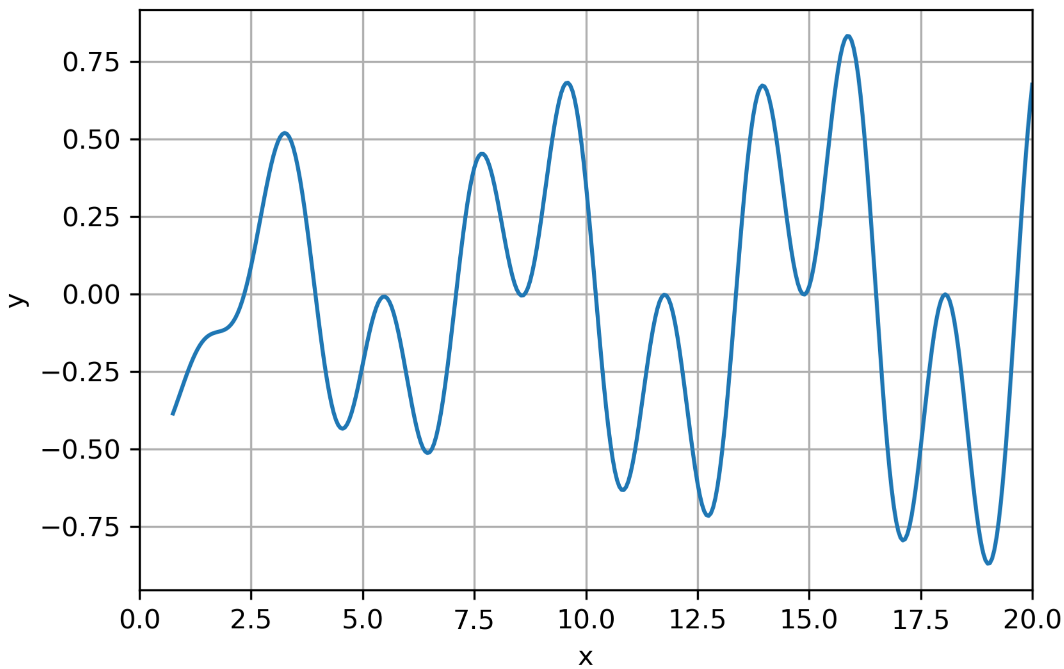

Figure 2: Plot of $f(x) = x^2 Y_0(x) Y_1^2(x) - J_0(x)J_1(x)$ over the domain $0.75 \leq x \leq 20$

For the purpose of this assignment, the files `nr3.h` and `bessel.h` provided by Press et al[1] were used to get an instance of `Bessjy()` for the 4 different Bessel functions used.

The following pseudo-code was used to write a template for the bisection method to a user defined tolerance:

Evaluate $f_1 = f(x_1)$, and $f_1 = f(x_2)$
Check that $f(x_1)$ and $f(x_2)$ have opposite signs (Exit if not)
Repeat

Evaluate $f_3 = f(x_3)$ at $x_3 = \frac{1}{2}(x_1 + x_2)$
If $f(x_3)$ has the same sign as $f(x_1)$ then $(x_1, f_1) \leftarrow (x_3, f_3)$
Else if f_3 has the same sign as f_2 then $(x_2, f_2) \leftarrow (x_3, f_3)$

Until $|f_3| < \epsilon =$ specified tolerance
x_3 is the root

A similar pseudo-code was used to write a template for the false position method to a specified level of tolerance:

Evaluate $f_1 = f(x_1)$, and $f_1 = f(x_2)$
Check that $f(x_1)$ and $f(x_2)$ have opposite signs (Exit if not)
Repeat

Evaluate $f_3 = f(x_3)$ at $x_3 = x_1 - f_1 \frac{x_2 - x_1}{f_2 - f_1}$
If $f(x_3)$ has the same sign as $f(x_1)$ then $(x_1, f_1) \leftarrow (x_3, f_3)$
Else if f_3 has the same sign as f_2 then $(x_2, f_2) \leftarrow (x_3, f_3)$

Until $|f_3| < \epsilon =$ specified tolerance
x_3 is the root

The data shown in Table 1 displays the results for root finding of Equation 5 using bisection. The tolerance was set to 10^{-7}. Because Figure 2 has a low resolution, points that seemed to touch the x-axis were checked to see whether they truly crossed. This was done by evaluating $f(x)$ for small values of x to obtain high resolution points for $f(x)$ and scanning through the areas in question to determine whether $f(x)$ actually crossed the x-axis.

Bracket	x	$f(x)$	Number of Iterations Needed
$[2, 3]$	2.34710550	$1.37037411 \times 10^{-8}$	21
$[3, 4]$	3.94076693	$-2.53081092 \times 10^{-8}$	23
$[6, 7.4]$	7.09101371	$4.68885913 \times 10^{-9}$	24
$[8, 8.6]$	8.51026688	$1.61286816 \times 10^{-8}$	18
$[8.6, 9]$	8.63023834	$7.15456535 \times 10^{-8}$	18

Table 1: The first five zeros found for $f(x)$ using the bisection method.

The data shown in Table 2 displays the results for root finding of Equation 5 using false position. The tolerance was set to 10^{-7} identical to bisection, and the same brackets were used.

Bracket	x	$f(x)$	Number of Iterations Needed
$[2, 3]$	2.34710539	$-4.3571571 \times 10^{-8}$	12
$[3, 4]$	3.94076691	$-1.4494999 \times 10^{-9}$	4
$[6, 7.4]$	7.09101370	$2.54846911 \times 10^{-9}$	5
$[8, 8.6]$	8.51026746	$-8.37187723 \times 10^{-8}$	42
$[8.6, 9]$	8.63023739	$-9.50513338 \times 10^{-8}$	36

Table 2: The first five zeros found for $f(x)$ using the false position method.

As expected, false position typically converged to a value below the set level of tolerance quicker than bisection. False position was able to identify these zeros using between one half to one sixth the number of iterations needed for bisection to find the same zero. For roots where false position out-performed bisection, the values of of x typically varied little. Often, the deviation did not occur until the 8^{th} or 9^{th} trailing digit.

One interesting observation was that false position did poorly in determining the 4^{th} and 5^{th} roots. Both of these roots took between two to two and a half times the number of iterations that bisection needed to accomplish the same tolerance level. This was likely a result of the function shape at that area due to the closeness of the two roots and the bracket size. For small brackets, halving the interval causes fairly quick convergence. In this case, the brackets were extremely unequal about the root. That is, the distance from the leftmost value of the bracket to the zero was much larger than the distance from the rightmost value of the bracket to the zero and vice versa. For false position, the adjustments to x_1 as shown in the pseudo-code can be fairly small when considering the concavity of the function in the bracket. The concavity makes it such that f_1 and f_3 will not have the same sign, thus x_1 does not inherit the value of x_3 and the adjustments to x_1 remain small, resulting in more iterations for the second to last root. Likewise, the concavity of the function for the bracket used for the last root caused small adjustments to x_2. In this particular instance, bisection caused a quicker convergence since it was able to trim large portions of the bracket quicker. Using smaller brackets like [8.4,8.6] and [8.6,8.7] caused false position to converge as fast, if not slightly faster, than bisection for the same brackets.

Source Code

```
/* AEP 4380 Homework #3

Test root finding
Bisection and False Position methods are tested

Run on a core i7 using clang 902.0.39.2
```

```
Kevin Juan 19 September 2018
*/

#include <cstdlib> // plain C
#include <cmath>  // use math package
#include "nr3.h"  // use nr3 file
#include "bessel.h" // use bessel function file

#include <iostream> // stream IO
#include <fstream>  // stream file IO
#include <iomanip>  // to format the output

using namespace std;

Bessjy bessFunc; // create instance of the Bessel function

/* Employs the bisection method for a function f within bracket
[xa, xb] to a tolerance level of tol
*/
template<class T, T (*f)(T)>
T bisect(T xa, T xb, T tol) {
  double f3, x1 = xa, x2 = xb, x3 = 0.5 * (x1 + x2);
  double f1 = f(x1), f2 = f(x2);
  int iter = 0;
  if ((f1 > 0 && f2 < 0) || (f1 < 0 && f2 > 0)) { // check function signs
    do { // begin bisection algorithm
      x3 = 0.5 * (x1 + x2);
      f3 = f(x3);
      if ((f3 > 0 && f1 > 0) || (f3 < 0 && f1 < 0)) {
        x1 = x3;
        f1 = f3;
      }
      else {
        x2 = x3;
        f2 = f3;
      }
      iter++;
    } while (abs(f3) >= tol);
    cout << "The root is x = " << x3 << " found in " << iter << " iterations
    at f(x) = " << f3 << endl;
  }
  else { // function signs are not opposite
    cout << "f(x1) and f(x2) must have opposite signs" << endl;
  }
  return x3;
```

```
} // end bisect()

/* Employs the false position method for a function f within bracket
[xa, xb] to a tolerance level of tol
*/
template<class T, T (*f)(T)>
T falsePosition(T xa, T xb, T tol) {
  double x1 = xa, x2 = xb, f1 = f(x1), f2 = f(x2);
  double x3 = x1 - f1 * (x2 - x1) / (f2 - f1), f3;
  int iter = 0;
  if ((f1 > 0 && f2 < 0) || (f1 < 0 && f2 > 0)) { // check function signs
    do { // begin false position algorithm
      x3 = x1 - f1 * (x2 - x1) / (f2 - f1);
      f3 = f(x3);
      if ((f3 > 0 && f1 > 0) || (f3 < 0 && f1 < 0)) {
        x1 = x3;
        f1 = f3;
      }
      else {
        x2 = x3;
        f2 = f3;
      }
      iter++;
    } while (abs(f3) >= tol);
    cout << "The root is x = " << x3 << " found in " << iter << " iterations
    at f(x) = "<< f3 << endl;
  }
  else { // function signs are not opposite
    cout << "f(x1) and f(x2) must have opposite signs" << endl;
  }
  return x3;
} // end falsePosition()

double testBessel(double x) { // Bessel function to be tested
  return bessFunc.y0(x) * bessFunc.y1(x) * bessFunc.y1(x) * x * x -
  bessFunc.j0(x) * bessFunc.j1(x);
}

int main() {
  double x, tol = 0.0000001;
  int i;
  cout.precision(9);

  ofstream fp1;              // output file using streams
  fp1.open("besselj_func.dat"); // open new file for output
```

```
fp1.precision(9);          // select 9 digits

if (fp1.fail()) {
  // or fp.bad()
  cout << "cannot open file" << endl;
  return (EXIT_SUCCESS);
}

for (i = 0; i <= 2000; i++) { // generate points for Bessel function plot
  x = i * 0.01;
  fp1 << setw(20) << x << setw(20) << bessFunc.j0(x) << setw(20) <<
  bessFunc.j1(x) << setw(20) << endl;
}
fp1.close();

ofstream fp2;              // output file using streams
fp2.open("bessely_func.dat"); // open new file for output
fp2.precision(9);          // select 9 digits

if (fp2.fail()) {
  // or fp.bad()
  cout << "cannot open file" << endl;
  return (EXIT_SUCCESS);
}

for (i = 75; i <= 2000; i++) { // generate points for Bessel function plot
  x = i * 0.01;
  fp2 << setw(20) << x << setw(20) << bessFunc.y0(x) << setw(20) <<
  bessFunc.y1(x) << setw(20) << testBessel(x) << endl;
}
fp2.close();

/* Use root finding methods for brackets obtained from plot to obtain the
first five roots
*/
bisect<double, testBessel>(2, 3, tol);
bisect<double, testBessel>(3, 4, tol);
bisect<double, testBessel>(6, 7.4, tol);
bisect<double, testBessel>(8, 8.6, tol);
bisect<double, testBessel>(8.6, 9, tol);
falsePosition<double, testBessel>(2, 3, tol);
falsePosition<double, testBessel>(3, 4, tol);
falsePosition<double, testBessel>(6, 7.4, tol);
falsePosition<double, testBessel>(8, 8.6, tol);
falsePosition<double, testBessel>(8.6, 9, tol);
```

```
  return(EXIT_SUCCESS);
}
```

References

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, editors. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK ; New York, 3rd ed edition, 2007. OCLC: ocn123285342.