AEP4380 HW3
Root Finding and Special Functions

Dan Girshovich

Feb 14, 2013

1 Overview

This document details the results of a C++ program which implements two root finding methods on an expression involving Bessel functions. Specifically, the root finding methods used were Bisection and False Position. The code for the Bessel functions is given in section 6.5 of Numerical Recipes [1] and the root finding methods are given in sections 9.1 and 9.2 of the same text.

2 Bessel Functions

The code plots the Bessel functions $J_0(x), J_1(x), Y_0(x)$, and $Y_1(x)$ for x between 0 and 20. Specifically, the `gen_plot_data` function created this data using 100 equally spaced points in this range for each function. The plots are shown below.

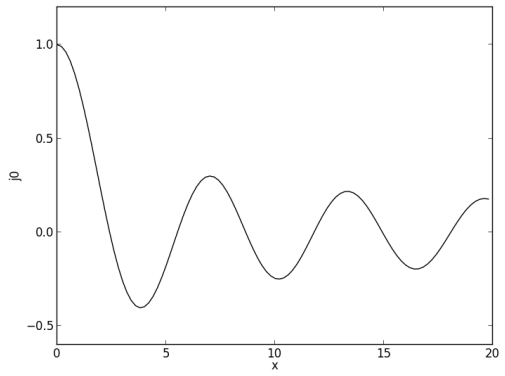

Figure 1: Bessel function J_0

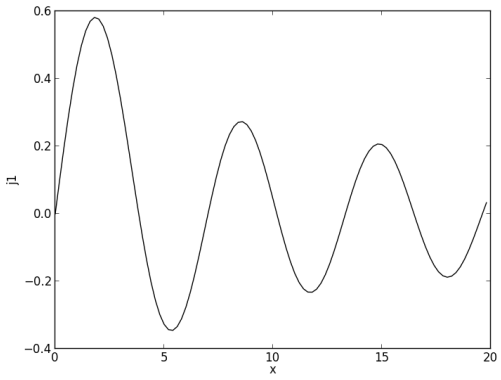

Figure 2: Bessel function J_1

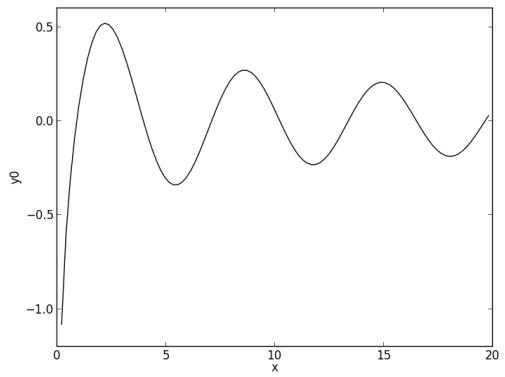

Figure 3: Bessel function Y_0

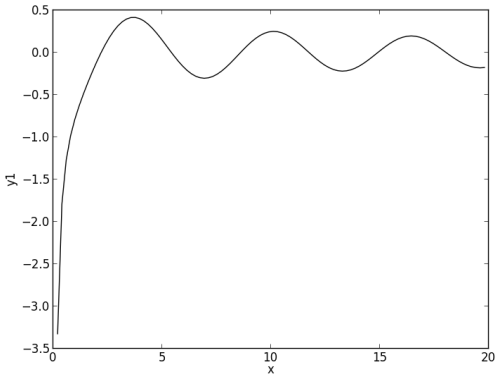

Figure 4: Bessel function Y_1

3 Root Finding

3.1 Equality to Solve

Numerical root finding methods were used to find the first five positive values of x that (approximately) solve the following equality:

$$J_0(x)Y_0(x) = x^2 J_1(x)Y_1(x)$$

I approached this by defining a new function:

$$f(x) = x^2 J_1(x)Y_1(x) - J_0(x)Y_0(x)$$

whose roots are values of x that satisfy the equality. This function is defined in the code as `bessel_difference` and is plotted below.

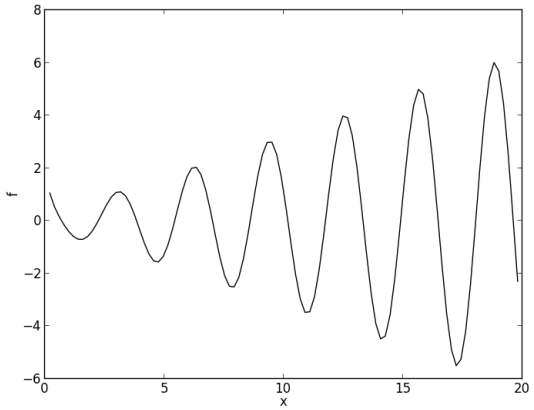

Figure 5: New function f(x) whose roots solve the given equality.

3.2 Bracketing Roots

Both root finding methods require value pairs that bracket a root as input. One approach to finding these brackets is to perform many equally spaced function evaluations in the desired range and search for changes of sign. This process is tricky to generalize, as roots could be arbitrarily close together. Luckily, f(x) has only a few roots in the range, and they are roughly evenly spaced. The function `find_brackets` is my implementation of this simple algorithm.

3.3 Bisection

The Bisection method is a kind of binary search for a root, given two bracketing values. This is implemented in the function `bisection` using tail-recursion. The function `gen_root_data` uses `find_brackets`, `bisection`, and a tolerance of $\epsilon = 10^{-7}$ to produce the data shown below. Importantly, this was generated with brackets of size $\epsilon_0 = 0.0146$.

x	iterations
0.69363733	16
2.23128571	17
3.83963641	18
5.43262872	18
7.01698486	17

The bisection algorithm is expected to double in accuracy with each iteration. For the given tolerance, the expected number of iterations is then given by :

$$log_2 \frac{\epsilon_0}{\epsilon} \approx 17.2 \qquad \text{(Equation 9.1.2 [1])}$$

This prediction nicely matches the actual results.

3.4 False Position

The False Position method uses the approximate slope at points near the root to quickly steer the approximation towards the root. This relies on the fact that the function is locally linear near the root, which f(x) fortunately is. In the code, the function `gen_root_data` uses `find_brackets`, `false_position`, and a tolerance of $\epsilon = 10^{-7}$ to produce the data shown below. This was generated using the same brackets used for Bisection ($\epsilon_0 = 0.0146$).

x	iterations
0.69363728	3
2.23128573	3
3.83963641	3
5.43262873	3
7.01698486	3

Comparing the root values to those found with bisection, it is clear that both methods have found the correct values with the desired accuracy. As expected, the linear nature of f(x) allowed the False Position algorithm to use the slope of points near the root to rapidly converge.

4 Source Code

```
/*
AEP 4380 HW #3
Dan Girshovich
2/14/13
Generates plot data and root info related to Bessel functions.
Compile with: g++ -std=c++0x -O2 -o gen_data hw3.cpp
Tested on OSX 10.8.2 with a Intel Core 2 Duo
*/

#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <vector>
#include "nr3.h"
#include "bessel.h"

using namespace std;

// func/type defs
typedef double (*math_func)(double);
typedef double (*root_finder)(math_func, double, double, double, int &);
double bessel_difference(double);
double bisection(math_func, double, double, double, int &);
double false_pos(math_func, double, double, double, int &);
void find_brackets(math_func, double, double, int, vector<double> &,
                   vector<double> &);

void gen_plot_data(const char *, math_func, double, double, int);
void gen_root_data(const char *, root_finder, math_func, double, double, int,
                   double);
void output(FILE *, int, double, double);
FILE *open_file_w(const char *);


int main()
{
    // generate data for plotting different Bessel functions
    gen_plot_data("j0_points.dat", j0, 0.0, 20.0, 100);
    gen_plot_data("j1_points.dat", j1, 0.0, 20.0, 100);
    gen_plot_data("y0_points.dat", y0, 0.2, 20.0, 100);
    gen_plot_data("y1_points.dat", y1, 0.2, 20.0, 100);
    gen_plot_data("difference_points.dat", bessel_difference, 0.2, 20, 100);
    // generate data about 2 root finding methods: bisection and false position
    gen_root_data("bisect_results.dat", bisection, bessel_difference,
                  0.2, 7.5, 5, 1e-7);
    gen_root_data("false_pos_results.dat", false_pos, bessel_difference,
                  0.2, 7.5, 5, 1e-7);
}
```

```
// generates data about the first n roots between x1 and x2 with given tolerance
void gen_root_data(const char *fname, root_finder get_root, math_func f,
                   double x1, double x2, int n, double tol)
{
    FILE *fp = open_file_w(fname);
    int num_intervals = n * 100; // increase this if roots get missed
    vector<double> right_b, left_b;
    find_brackets(f, x1, x2, num_intervals, left_b, right_b);
    int num_found = left_b.size();
    if (num_found != n)
    {
        printf("Looked for %d roots, but bracketed %d. Finding first %d.\n",
               n, num_found, n);
    }
    for (int i = 0; i < num_found; i++)
    {
        int num_its = 0;
        double root = get_root(f, left_b[i], right_b[i], tol, num_its);
        fprintf(fp, "%16.8f %d \n", root, num_its);
    }
    fclose(fp);
}

// writes n points of the function f from x1 to x2 to file fname
void gen_plot_data(const char *fname, math_func f, double x1, double x2, int n)
{
    FILE *fp = open_file_w(fname);
    double x;
    double dx = (x2 - x1) / n;
    for (int i = 0; i < n; i++)
    {
        x = x1 + i * dx;
        fprintf(fp, "%16.8f %16.8f \n", x, f(x));
    }
    fclose(fp);
}

// tail-recursive bisection. GCC optimizes by clearing stack frame on each call
// num_its will get set on completion with the number of bisections
double bisection(math_func f, double x1, double x2, double e, int &num_its)
{
    double x3 = (x1 + x2) / 2;
    double f1 = f(x1), f2 = f(x2), f3 = f(x3);
    num_its++;
    if (num_its > 30) throw ("Error: Too many bisections performed.\n");
    if (f1 * f2 >= 0.0) throw ("Error: Root must be bracketed in bisection.\n");
    if (abs(f3) < e)  // root found
    {
        return x3;
    }
    else if (f1 * f3 >= 0.0)
    {
        return bisection(f, x3, x2, e, num_its);
```

```
    }
    else if (f2 * f3 >= 0.0)
    {
        return bisection(f, x1, x3, e, num_its);
    }
    else
    {
        throw ("Error: Uncaught case in bisection!");
    }
}

// slightly modified rtflsp function (Numerical Recipes, p. 451)
double false_pos(math_func f, double x1, double x2, double e, int &num_its)
{
    int MAXIT = 30;
    double xl, xh, del;
    double fl = f(x1);
    double fh = f(x2);
    if (fl * fh > 0.0) throw("Root must be bracketed in false_pos.");
    if (fl < 0.0)
    {
        xl = x1;
        xh = x2;
    }
    else
    {
        xl = x2;
        xh = x1;
        SWAP(fl, fh);
    }
    double dx = xh - xl;
    for (int j = 0; j < MAXIT; j++)
    {
        num_its++;
        double rtf = xl + dx * fl / (fl - fh);
        double fnew = f(rtf);
        if (fnew < 0.0)
        {
            del = xl - rtf;
            xl = rtf;
            fl = fnew;
        }
        else
        {
            del = xh - rtf;
            xh = rtf;
            fh = fnew;
        }
        dx = xh - xl;
        if (abs(del) < e || fnew == 0.0) return rtf;
    }
    throw ("Maximum number of iterations exceeded in false_pos.");
}
```

```
// cleaned up version of zbrak (Numerical Recipes, p 447)
// left_b and right_b will contain the bracket values upon completion
void find_brackets(math_func f, double x1, double x2, int n,
                   vector<double> &left_b, vector<double> &right_b)
{
    double x;
    double dx = (x2 - x1) / n;
    for (int i = 0; i < n; i++)
    {
        x = x1 + i * dx;
        if (f(x) * f(x + dx) < 0.0) // now enclosing at least one root
        {
            left_b.push_back(x);
            right_b.push_back(x + dx);
        }
    }
}

// evaluates the difference of the two Bessel expressions we're equating
double bessel_difference(double x)
{
    return (pow(x, 2) * j1(x) * y1(x) - j0(x) * y0(x));
}

// opens a new file for output
FILE *open_file_w(const char *fname)
{
    FILE *fp;
    fp = fopen(fname, "w+");
    if (NULL == fp)
    {
        printf("cannot open file\n");
        return (EXIT_SUCCESS);
    }
    return fp;
}
```

References

[1] W.H. Press, S.A Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipies, The Art of Scientific Computing*. Camb. Univ. Press, 3rd Edition, 2007.