Computer Project I

Jacob Triebwasser

October 25, 2019

In this project, we constructed code initially for the computation of the Bessel function of order zero at various points and later for the calculation of roots of that function using several algorithms covered in class. We discuss rates of convergence for these algorithms, backed by the data provided by our programs.

Calculating Bessel Functions

We start with two formulae for calculating the Bessel function of order 0 at a given point. The Bessel functions of order n provide solutions to the differential equations

$$y''(x)x^2 + y'(x)x + y(x)(x^2 - n^2) = 0$$

This particular equation arises in the study of Mark Kac's famous question "can one hear the shape of a drum?", specifically in the context of circular drum heads.
Two methods of calculation for these solutions exist. The explicit formula for $J_n(x)$ is given as

$$J_n(x) = \sum_{p=0}^{\infty} \frac{(-1)^p}{p!(n+p)!} \left(\frac{x}{2}\right)^{n+2p}$$

One method of effectively calculating this is to truncate the infinite series at a fixed number of terms M. We implemented this for J_0 with the following section of code, in Python:

```
factorialList = [1]

def factorial(n):
    """ This is just a relatively fast, dynamic way of calculating factorials.
     Since I expect to call this function fairly often, saving the
     intermediate values cuts down computation time slightly at the
     expense of needing more memory to hold the array. """
    cur = len(factorialList)
    if cur <= n :
        for i in range(cur, n+1):
            factorialList.append(factorialList[i-1] * i)
    return factorialList[n]
```

```
def truncatedBessel(x,M):
    """ Calculates the Bessel function of order zero at the point x,
     using the truncated infinite series to obtain a reasonable
     approximation. """
    value = 0
    for p in range(M+1):
        innerCoeff = ((-1)**p)/(factorial(p)**2)
        variable = (x/2)**(2*p)
        value += innerCoeff * variable
    return value
```

An alternative method of calculating these values is by using a known recurrence relation for the Bessel functions:

$$J_{n-1}(x) = \frac{2n}{x} J_n(x) - J_{n+1}(x)$$

From the series representation, we know that as $n \to \infty$ we have $J_n(x) \to 0$. So, by choosing N high enough, we can assume $J_N(x) \approx 0$. For the algorithm, we set this value equal to zero, then assume $J_{N-1} = 1$, then use the recurrence relation to get to $J_0(x)$. However, we have to introduce a normalizing factor to account for our assumption that $J_{N-1} = 1$. This factor is given as

$$\lambda = J_0(x) + 2 \sum_{p=1}^{\lfloor N/2 \rfloor} J_{2p}(x)$$

Dividing our values by this normalizing factor yields a proper approximation. We have implemented this as follows:

```
def dynamicBessel(x,N):
    """ Calculates the Bessel function of order zero at the point x,
     using the recursion formula and taking the appropriate corrective
     factor. Occasionally requires slightly more terms than truncatedBessel to
         achieve the same relative error bound. """
    J = [0]*(N+1)
    # It's important to pick N large enough so that the expression
    # $$sum_(p=0)^(infinity) [(((-1)**p)/(p!(n+p)!))(x/2)**(n+2p)]$$
    # becomes negligible for n >= N.
    J[N-1] = 1
    for i in range(1,N)[::-1]:
        intermediate = (2*(i)/x)*J[i] - J[i+1]
        J[i-1] = intermediate
    lambdaArray = [J[0]] + [2*x for x in J[2::2]]
    correctiveFactor = sum(lambdaArray)
    return J[0] / correctiveFactor
```

The relative error for each of these methods versus the number of terms taken is plotted in Figure 1, where the red line represents the recursive strategy and the blue line represents

the truncation strategy. The relative error was calculated using the value for $J_0(x)$ given by the scipy library.

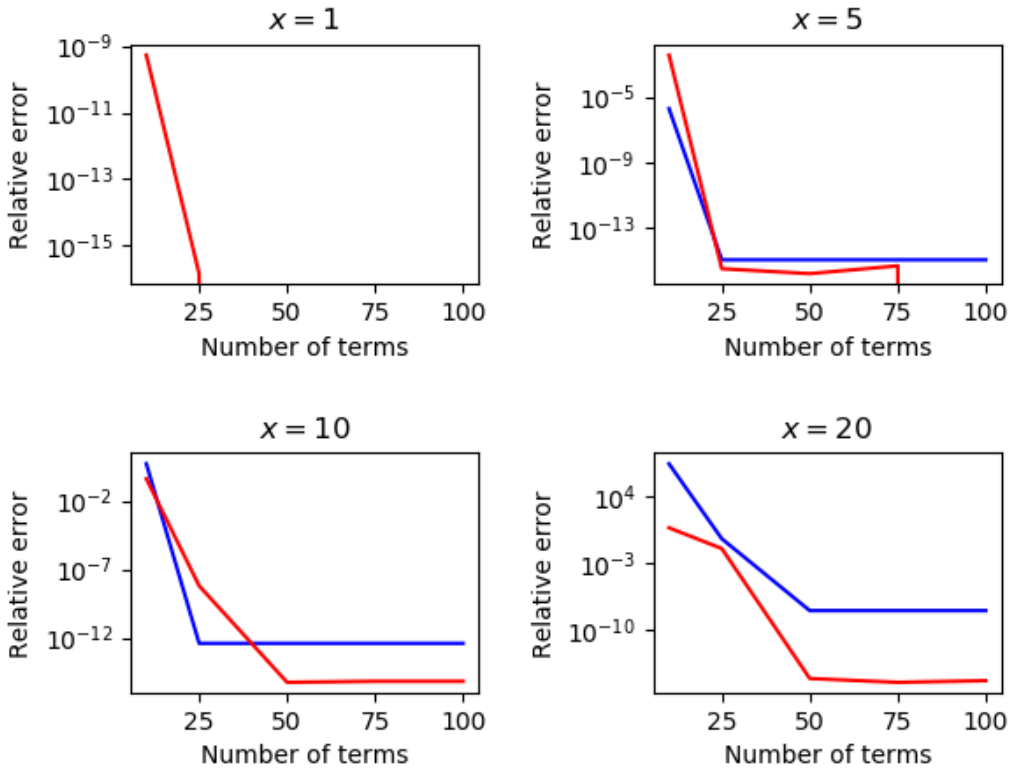

Figure 1: Relative error for the two calculations of $J_0(x)$.

In the plot of $x = 1$, the blue line is not visible at all, since the truncation strategy led to a relative error smaller than computer precision for any of the chosen numbers of terms. However, as our value of x moves further from 1 we see that the recursive strategy results in smaller relative errors on average. In fact, for low numbers of terms that are very far from $x = 1$, the relative error for the truncation strategy increases virtually exponentially. Tables of these errors, as output by the program, are given in Figure 2.

Table of relative errors for the truncated Bessel function:

x \ N	10	25	50	75	100
1	0.0	0.0	0.0	0.0	0.0
5	1.9247235135039960e-06	1.0939896478496191e-15	1.0939896478496191e-15	1.0939896478496191e-15	1.0939896478496191e-15
10	5.166297006757969	3.980466826966472e-13	3.980466826966472e-13	3.980466826966472e-13	3.980466826966472e-13
20	21537115.6073679	0.32336948794381026	1.0135855563038338e-08	1.0135855563038338e-08	1.0135855563038338e-08

Table of relative errors for the recursive Bessel function:

x \ N	10	25	50	75	100
1	5.289770222925645e-10	1.4508969958066554e-16	0.0	0.0	0.0
5	0.0036186807552131156	3.1256847081417693e-16	1.5628423540708846e-16	4.688527062212654e-16	0.0
10	0.41688917630118416	6.331018823904651e-09	5.642850619459132e-16	6.771420743350959e-16	6.771420743350959e-16
20	4.505215402547121	0.030507313823770935	8.308825443597952e-16	3.323530177439181e-16	4.985295266158772e-16

Figure 2: Relative errors for the calculation of J_0

As the data shows, for $N = 10$ and $x = 20$ the truncation leads to a relative error in excess of 10^7, which is undesirable. Hence, for values of x near 1, using the truncation is preferable to the recurrence algorithm. Far from 1, however, using the recurrence is preferable, with performance of each improving significantly with the introduction of more terms.

Calculating roots of $J_0(x)$

For the purpose of calculating the roots of $J_0(x)$, we will use the bisection method, the secant method, and Newton's method.

Bisection Method

The bisection method for finding roots of a function proceeds in a fashion similar to binary search.
We begin with an interval $[a, b]$ such that for our function f we have $sign(f(a)) \neq sign(f(b))$. Assuming that f is continuous on this interval, by the intermediate value theorem the interval must contain at least one root. We then divide the interval in half and check the sign of the function at the midpoint, c. If that sign is different from the sign of $f(a)$, we must have our root in the interval $[a, c]$. Otherwise, the root is inside $[c, b]$. Continuing this process of taking subdivisions until the width of our interval is below some constant tolerance, δ, or our value of $|f(c)|$ is below some constant tolerance ϵ, gives an approximation of our root. This algorithm was implemented as follows:

```
def bisectionMethod(function, interval, maxIterations, epsilon, delta):
    """ Calculates and returns a root of the given function (if one exists)
        within the specified interval. The algorithm continues until either:
        1. maxIterations has been reached.
        2. The function applied to the current guess has modulus < epsilon.
        3. The associated error bound is less than delta.
    """
```

```
    a = float(interval[0])
    b = float(interval[1])
    u = function(a)
    v = function(b)
    sign = lambda x: 1 if x >= 0 else -1
    assert(sign(u) != sign(v)), "Bisection method inapplicable on this interval.
        Find an interval whose endpoints yield different signs."
    error_bound = b-a
    for k in range(1,maxIterations+1):
        # Bisect our interval
        error_bound = error_bound / 2.0
        # Get value at the bisection
        c = a + error_bound
        w = function(c)
        # If we are within our stated acceptable error, we are done
        if abs(error_bound) < delta or abs(w) < epsilon:
            return (c, w, k)
        # Otherwise, check if the root is between a and c or c and b
        if sign(w) != sign(u):
            # Root is between a and c, so check interval [a,c]
            b = c
            v = w
        else:
            # Root is between c and b, so check interval [c,b]
            a = c
            u = w
    # This code should be unreachable:
    return (0,0,-1)
```

The bisection method of finding roots is known to converge linearly, with

$$|e_n| \leq \frac{1}{2}|e_{n-1}|$$

Secant Method

The secant method is an approximate form of Newton's method, which will be discussed in the following section. Newton's method requires knowing the derivative of the function whose roots are being sought, whereas the secant method replaces this derivative with an approximation via a difference quotient. For the secant method, we have the following definition for the sequence given a function f:

$$x_{n+1} = x_n - f(x_n)\left[\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\right]$$

For the algorithm's implementation, we set a maximum number of iterations as well as maximum error bounds (ϵ and δ). Once either the maximum iterations are met, the value of

$|f(x_n)| < \epsilon$, or $|x_n - x_{n-1}| < \delta$ we stop the algorithm and output x_n, $f(x_n)$, and the number of iterations. The implementation is as follows:

```
def secantMethod(function, interval, maxIterations, epsilon, delta):
    """ Calculates a root of the given function within the given interval,
        using the secant method. """
    a = interval[0]
    b = interval[1]
    fa = function(a)
    fb = function(b)
    for k in range(2, maxIterations+1):
        if abs(fa) > abs(fb):
            # If this is the case, we need to swap a with b and fa with fb.
            tmp = a
            a = b
            b = tmp
            tmp = fa
            fa = fb
            fb = tmp
        s = (b - a)/(fb - fa)
        b = a
        fb = fa
        a = a - fa * s
        fa = function(a)
        if abs(fa) < epsilon or abs(b-a) < delta:
            return (a, fa, k)
    # This should be unreachable:
    return (0,0,-1)
```

The secant method is known to converge superlinearly, with

$$|e_{n+1}| \leq A|e_n|^{(1+\sqrt{5})/2}$$

Newton's Method

Newton's method for finding the roots of a function is given by the following sequence definition:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

which starts with an estimated solution x_0. This algorithm arises from the Taylor expansion of our function f, whose first terms give the following linearization at the point c:

$$f(x) \approx f(c) + f'(c)(x - c)$$

Taking our $c = x_{n-1}$ and seeking $f(x_n) = 0$, we get

$$\begin{aligned} f(x_n) &= f(x_{n-1}) + f'(x_{n-1})(x_n - x_{n-1}) \\ 0 &= f(x_{n-1}) + f'(x_{n-1})x_n - f'(x_{n-1})x_{n-1} \\ -x_n &= f(x_{n-1})/f'(x_{n-1}) - x_{n-1} \\ x_n &= x_{n-1} - f(x_{n-1})/f'(x_{n-1}) \end{aligned}$$

We implement this method in the following code:

```
def newtonsMethod(function, derivative, guess, maxIterations, epsilon, delta):
    """ Calculates a root of the given function with the given derivative, using
        Newton's method. """
    xNew = 0
    xOld = guess
    v = function(xOld)
    if abs(v) < epsilon:
        return (guess, v, 0)
    for k in range(1, maxIterations+1):
        xNew = xOld - (v / derivative(xOld))
        v = function(xNew)
        if abs(xNew - xOld) < delta or abs(v) < epsilon:
            return (xNew, v, k)
        xOld = xNew
    # This code should be unreachable:
    return (0,0,-1)
```

This requires subprocesses for both the function and its derivative. It also requires a maximum number of iterations and error bounds for the value of $|f(x_n)|$ and $|x_n - x_{n-1}|$. Newton's method is known to converge quadratically, with

$$|e_{n+1}| \leq C|e_n|^2$$

Comparing Performance

Here we compare the convergence and overall performance of each of the preceding algorithms at finding roots of J_0. Using the fact that $J_0'(x) = -J_1(x)$ allows us to use Newton's method. We first consider the raw data output by the program, in Figure 3.

```
Bisection method results for J_0:
                    eps.|         1e-06          |         1e-07          |         1e-08          |         1e-09          |         1e-10
          [a,b]         |                        |                        |                        |                        |
=============================================================================================================================================
          [1,3]          | 2.404827117792, 19 iters. |2.404825687408, 22 iters. |2.404825568199, 24 iters. |2.404825557023, 29 iters. |2.404825557722, 33 iters.
          [4,6]          |5.520080566406, 15 iters. | 5.52007818222, 22 iters. |5.520078122616, 25 iters. |5.520078107715, 27 iters. |5.520078110509, 31 iters.
          [8,10]         | 8.65372467041, 18 iters. | 8.65372800827, 22 iters. |8.653727889061, 24 iters. |8.653727911413, 28 iters. |8.653727913275, 30 iters.


Secant method results for J_0:
                    eps.|         1e-06          |         1e-07          |         1e-08          |         1e-09          |         1e-10
          [a,b]         |                        |                        |                        |                        |
=============================================================================================================================================
          [1,3]          |2.404826775528, 5 iters. | 2.40482555761, 6 iters. | 2.40482555761, 6 iters. | 2.40482555761, 6 iters. | 2.40482555761, 6 iters.
          [4,6]          |5.520078095074, 5 iters. |5.520078095074, 5 iters. |5.520078095074, 5 iters. |5.520078110286, 6 iters. |5.520078110286, 6 iters.
          [8,10]         |8.653727918253, 5 iters. |8.653727918253, 5 iters. |8.653727918253, 5 iters. |8.653727912911, 6 iters. |8.653727912911, 6 iters.


Newton's method results for J_0:
                    eps.|         1e-06          |         1e-07          |         1e-08          |         1e-09          |         1e-10
          Guess         |                        |                        |                        |                        |
=============================================================================================================================================
            3           |2.404825557691, 4 iters. |2.404825557691, 4 iters. |2.404825557691, 4 iters. |2.404825557691, 4 iters. |2.404825557691, 4 iters.
            6           |5.520078102671, 3 iters. |5.520078102671, 3 iters. |5.520078102671, 3 iters. |5.520078110286, 4 iters. |5.520078110286, 4 iters.
           10           |5.520078044927, 5 iters. |5.520078044927, 5 iters. |5.520078110286, 6 iters. |5.520078110286, 6 iters. |5.520078110286, 6 iters.
```

Figure 3: Program Output

Clearly, the bisection method converges much more slowly than the other two. Since the bisection method and secant method share the same parameters, we compare their performance directly in Figure 4 (bisection is blue and secant is red). Newton's method, while significantly faster than the bisection method, can be seen to miss one of the roots found by the secant method. This is a common failing of Newton's method in certain scenarios. However, the convergence rate is still significant, as seen in Figure 5.

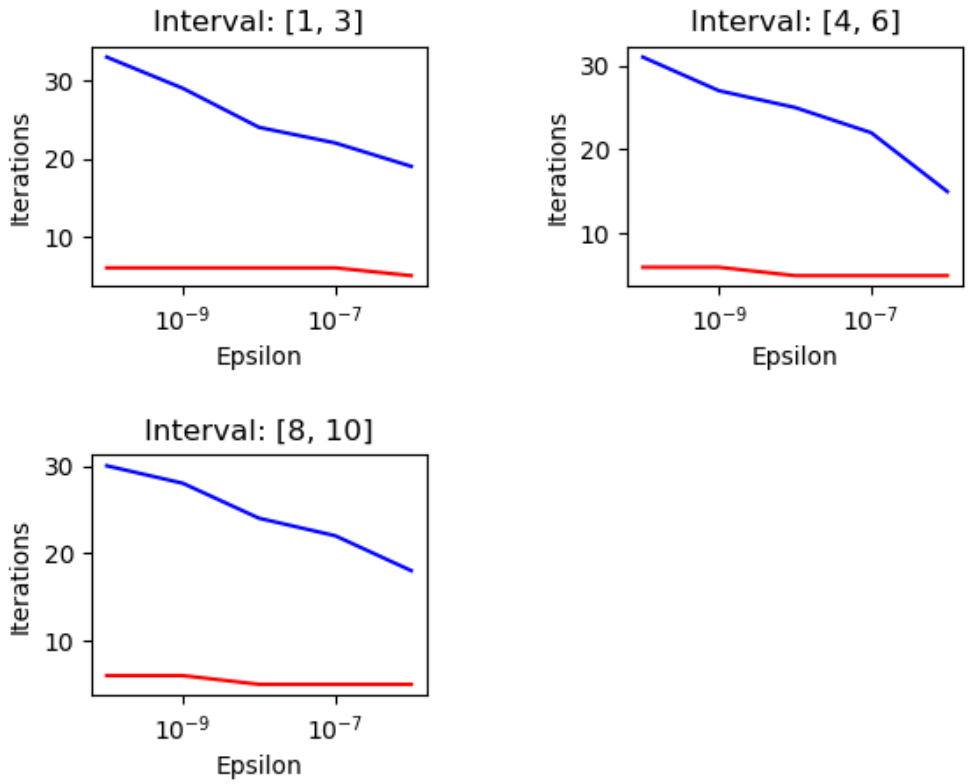

Figure 4: Bisection Method vs. Secant Method

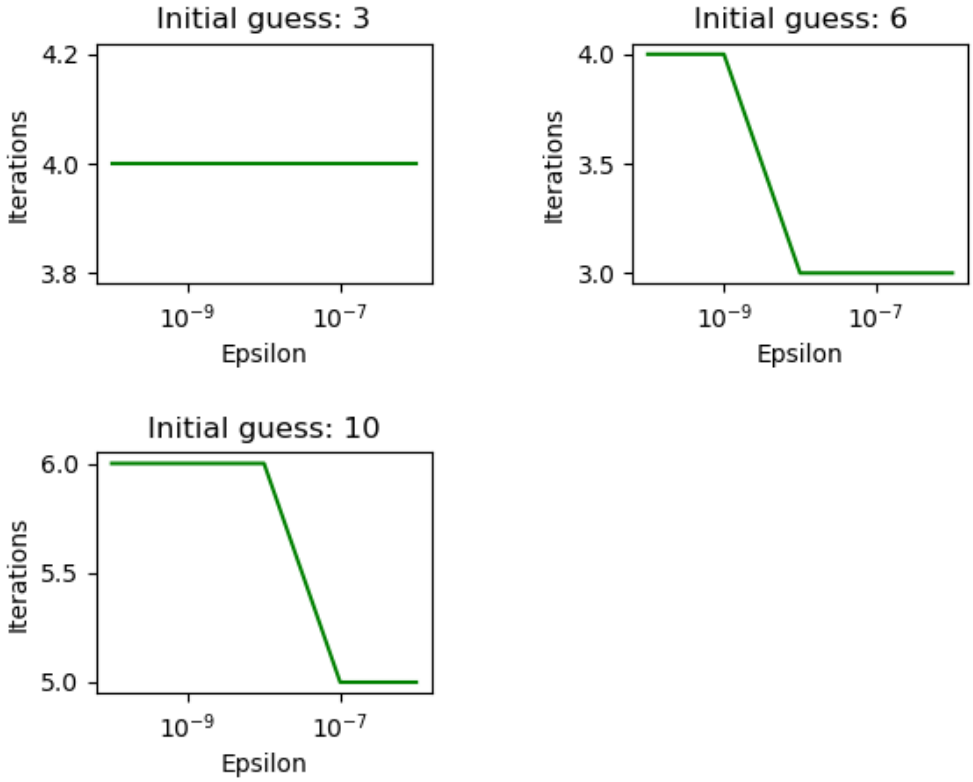

Figure 5: Newton's Method

Conclusions

It may be unsurprising that each method of calculation of $J_0(x)$ improves with more terms, but it is interesting that there is a clearly superior choice of method given the value of x that one is interested in. From the data, it is very clear that the closer our value is to 1, the more we should prioritize using the truncation over the recurrence relation. This does seem reasonable, however, upon consideration of the series itself. For values of x near 1, the value of the summands decreases much more rapidly over each term than it would for larger values of x. Meanwhile, the value generated by the recurrence relation has no such obvious correlation to the size of x.

For the methods of finding roots, we can immediately see the relationship between the rates of convergence of the methods. The bisection method converges linearly, and is thus the slowest method. The secant method and Newton's method perform similarly, despite being superlinear and quadratic, respectively. The discrepancy in convergence rates would be more apparent if we had initially started further from the roots, thus requiring more iterations. However, the speed at which Newton's method converges led to the algorithm entirely missing a root close to the starting point for our initial guess of $x = 10$. It is apparent that one must choose the appropriate method of finding roots for whichever problem is at hand.