ASSIGNMENT 4: BOUNDARY VALUE PROBLEMS

October, 2010 Guanqing Ou and Arash Ushani

PROBLEM STATEMENT

Implement an algorithm to solve a tri-diagonal system of linear equations. Then use this solver to solve the following linear boundary value system:

$$v'' + 2xv' - x^2v = x^2, v(0) = 1, v(1) = 0 \quad (1)$$

using the finite difference method. Investigate into the error associated with this method, and adapt for the boundary conditions $v(0) + v'(0) = 1, v'(1) + 0.5v(1) = 0$.

Establish the dependence of these methods on the tolerances associated. For example, with the finite difference method, show that the algorithm is second-order accurate with respect to Δx.

COMPUTATIONAL APPROACH

Finite Difference Method

In order to think about this problem generally, we define a standard form of a second order boundary value system:

$$v'' + av' + bv + c = 0 \quad (2)$$

where v is a function of x, a, b, and c can be

In solving the above boundary value problem, we can use the central difference approximation to $v''$and v':

$$v'' = \frac{v_{i+1} - 2v_i + v_{i-1}}{\Delta x^2} \quad (3)$$

$$v' = \frac{v_{i+1} - v_{i-1}}{2\Delta x} \quad (4)$$

to rewrite the linear boundary value system (2):

$$\frac{v_{i+1} - 2v_i + v_{i-1}}{\Delta x^2} - a\frac{v_{i+1} - v_{i-1}}{2\Delta x} + bv + c = 0. \quad (5)$$

To find a numerical solution to this ODE, we must find a sequence of points that satisfy the ODE, as well as the boundary conditions. This means that our solution curve would consist

of $(x_i, v(x_i))$ pairs that satisfy the equation above. Thus, we can state that given the first boundary value problem with $v(0) = 1, v(1) = 0$, there are n values $(x_1, x_2, \dots, x_{n-1}, x_n)$ with corresponding $(v(x_1), v(x_2), \dots v(x_{n-1}), v(x_n))$ that satisfy the given ODE. Essentially, we are approximating the values of v in the range $x = [0,1]$ using n equally spaced points. If we take $\big(x_1, v(x_1)\big)$ as an example,

$$\frac{v_2 - 2v_1 + v_0}{\Delta x^2} - a\frac{v_2 - v_0}{2\Delta x} + bv_1 + c = 0. \tag{6}$$

where we express $v(x_i)$ as v_i, since they are approximations and not exact values, and $\Delta x = x_{i+1} - x_i$. Equation (6) can be manipulated to be as follows:

$$\left(1 - \frac{a\Delta x}{2}\right)v_0 + (-2 + b\Delta x^2)v_1 + \left(1 + \frac{a\Delta x}{2}\right)v_2 = -c\Delta x^2 \tag{7}$$

Note that this is the form of

$$Av_0 + Bv_1 + Cv_2 = K \tag{8}$$

where A, B, C, and K are functions of x.

Given that we know what v_0 is from the boundary values, we can further rearrange (7):

$$(-2 + b\Delta x^2)v_1 + \left(1 + \frac{a\Delta x}{2}\right)v_2 = \left(1 - \frac{a\Delta x}{2}\right)v(0) - c\Delta x^2 \tag{9}$$

This case is special for the boundary values. At v_n, we can substitute the boundary value $v(1)$ for v_{n+1}, so that:

$$\left(1 - \frac{a\Delta x}{2}\right)v_{n-1} + (-2 + b\Delta x^2)v_n = -c\Delta x^2 + \left(1 + \frac{a\Delta x}{2}\right)v(1) \tag{10}$$

Besides these boundary cases, we can adapt equation (7) for all intermediate values of v. When viewed in aggregate, these equations exhibit a pattern that allows us to express the relationships between xs and vs in a condensed manner:

$$Hv = D \tag{11}$$

where H is a tridiagonal matrix with coefficients :

$$H = \begin{bmatrix} -2 + b\Delta x^2 & 1 + \frac{a\Delta x}{2} & 0 & \dots & 0 \\ 1 - \frac{a\Delta x}{2} & -2 + b\Delta x^2 & 1 + \frac{a\Delta x}{2} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 - \frac{a\Delta x}{2} & -2 + b\Delta x^2 & 1 + \frac{a\Delta x}{2} \\ 0 & \dots & \dots & 1 - \frac{a\Delta x}{2} & -2 + b\Delta x^2 \end{bmatrix}$$

v is the set of vs: $\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix}$, and D is the matrix of known values on the right side of all equations with the same form as (7). Essentially, a linear combination of the column vector $\vec{v}$, as prescribed by the coefficients in H, give us D.

The case is slightly different if we have boundary conditions expressed in the form $mv(0) + nv'(0) = p$ and $qv(1) + rv'(1) = s$. We can no longer simply move the v_0 term to the "known" side of the equation in (7). Furthermore, we cannot use the central difference approximation for v' for v_1, since this depends on knowing the value of v_0. The same restraints apply to trying to satisfy the ODE at v_n. Thus, we deal with boundary conditions in a more general fashion.

Consider the condition $mv(0) + nv'(0) = p$. We can express this in terms of v_0 and v_1 by approximating $v'(0)$ as $\frac{(v_1 - v_0)}{2}$, such that, with rearrangement,

$$v_0(m\Delta x - n) + nv_1 = p\Delta x \tag{12}$$

Similarly,

$$rv_n + v_{n+1}(q\Delta x - r) = s\Delta x \tag{13}$$

This allows us to add 2 rows to H, v, and D:

$$H = \begin{bmatrix} m\Delta x - n & n & 0 & \dots & \dots & 0 \\ 1 - \frac{a\Delta x}{2} & -2 + b\Delta x^2 & 1 + \frac{a\Delta x}{2} & 0 & \dots & \vdots \\ 0 & 1 - \frac{a\Delta x}{2} & -2 + b\Delta x^2 & 1 + \frac{a\Delta x}{2} & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & 1 - \frac{a\Delta x}{2} & -2 + b\Delta x^2 & 1 + \frac{a\Delta x}{2} \\ 0 & \dots & \dots & \dots & r & q\Delta x - r \end{bmatrix}$$

$$v = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_n \\ v_{n+1} \end{bmatrix}, D = \begin{bmatrix} p\Delta x \\ -c\Delta x^2 \\ \vdots \\ -c\Delta x^2 \\ s\Delta x \end{bmatrix}.$$

This method of handling boundary conditions allows us to deal more generally with BVPs, including the case where boundary conditions are $v(0) = c_1$, $v(1) = c_2$, where c_1 and c_2 are constants. In this case, the coefficients n and r in equations (12) and (13) would be zero, $m = p = q = 1$, and r= 0.

Tri-diagonal Solver

Given this system of linear equations, we can use row reduction to solve for the column vector $\vec{v}$. Row reduction involves canceling out every term in the coefficient matrix except the diagonal. This matrix then becomes a scaling matrix relating v and D. While a generalized row reduction algorithm would need to be able to handle input matrices with terms in every position in the matrix, the row reduction we employ here is particular to the tridiagonal matrix, which significantly simplifies the problem.

Consider a typical 5x5 tridiagonal coefficient matrix for a generic column vector $\vec{z}$:

$$\begin{bmatrix} x & y & 0 & 0 & 0 \\ y & x & y & 0 & 0 \\ 0 & y & x & y & 0 \\ 0 & 0 & y & x & y \\ 0 & 0 & 0 & y & x \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} \quad (14)$$

where positions marked with x and y have some non-zero entry and cs as constants.

In order to row reduce this matrix, we have to get rid of all terms where there are ys, and keep the central diagonal of x values, so that we end up with five equations where some coefficient times z_n is equal to a constant. To do this, we first get rid of the all the non-zero terms to the left of the main diagonal, by subtracting the right multiple of the column row above the row we are operating on so that the term to the left of the diagonal cancels out. This gives us the following:

$$\begin{bmatrix} x & z & 0 & 0 & 0 \\ 0 & x & z & 0 & 0 \\ 0 & 0 & x & z & 0 \\ 0 & 0 & 0 & x & z \\ 0 & 0 & 0 & 0 & x \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} c_6 \\ c_7 \\ c_8 \\ c_9 \\ c_{10} \end{bmatrix} \quad (15)$$

Note that each value in the positions marked by x and z is different. This is simply meant to indicate the three different diagonals. Once we have eliminated the lower diagonal, we can then move from the bottom row up to cancel out the top diagonal. This can be done by subtracting the appropriate multiple of the row below the one being operated upon. As we perform these operations, we update the values in the "answer" vector, $\vec{c}$, as indicated by the different indices in equation (15).

Using this tridiagonal matrix solving algorithm, we can calculate the value of $\vec{v}$ in equation (11).

IMPLEMENTATION AND EVALUATION

TRI-DIAGONAL SOLVER

The tridiagonal matrix solving algorithm described above was implemented and tested with randomly generated diagonals in MATLAB as follows:

```
tridiagonal (rand(1, N-1), rand(1,N), rand(1,N-1), rand(1,N))
```

where the function tridiagonal is:

```
function res = tridiagonal (bottom, main, top, known)
      if nargin < 3
          disp ( 'No Diagonal Values');
      end

      for i = 1: (length (main)-1);
          factor = bottom(i)/ main (i);
          bottom(i) = bottom (i) - factor * main(i);
          main (i+1) = main(i+1) - factor* top(i);
          known(i+1) = known(i+1) - factor* known(i);
```

```
        end

        solution = zeros( length(known), 1);

        for i = length (top):-1:1;
            factor = top(i)/ main(i+1);
            known(i)  = known (i) -factor * known(i+1);
            solution (i) = known(i)/ main(i);

        end
        solution(end) = known(end)/ main(end);

        res=solution;
end
```

The accuracy of this solver was verified by comparing the solution with the existing linear system solver in MATLAB, "linsolve", as well as other students.

FINITE DIFFERENCE METHOD

With the validity of our tridiagional solver established, we can proceed to using it to solve the given ODE. Recall that $v'' + 2xv' - x^2v = x^2, v(0) = 1, v(1) = 0$. If we use our general way of dealing with boundary conditions with the form $mv(0) + nv'(0) = p$ and $qv(1) + rv'(1) = s$, then $m = 1, n = 0, p = 1, q = 1, r = 0$. To calculate the values of the tridiagonal coefficient matrix, we implement the following:

```
function res = diagmatrixbound(boundinit, boundend,n, range, x0, A,B,C)
    if nargin < 8
        C= @(x)-x.^2;
    end
    if nargin < 7
        B= @(x)-x.^2;
    end
    if nargin < 6
        A = @(x)2*x;
    end

    dx = range/(n+1);
    x = x0+dx: dx: x0+range-dx;
    low = [1-A(x)*dx/2, -boundend(2)];
    main =[(boundinit(1)*dx-boundinit(2)), -2+B(x)*(dx)^2,
(dx*boundend(1)+boundend(2))];
    up = [boundinit(2), 1+A(x)*dx/2];
    known = [boundinit(3)*dx, -C(x)*(dx)^2,boundend(3)*dx];
    res = [up 0; main; 0 low; known];
end
```

The boundary values are stored in "boundinit" and "boundend", which are 1x3 matrices with the coefficients m, n, p and q, r, s. We store the lower, main, and upper diagonal values in 3 vectors, which are used to give us a tridiagonal matrix that can then be input into the tridiagonal solver described above.

To do this, we implement the following code, and plot the solution calculated by the tridiagonal solver to visualize the solution:

```
function res= tut4(range,n,x0)
      if nargin < 3
          x0 = 0;
      end
      if nargin < 2
          n = 100;
      end
      if nargin < 3
          range = 1;
      end

      dx = range/(n+1);
      matrices =diagmatrixbound ([1 0 1], [1 0 0], n);
      bottom = matrices (3,:);
      main = matrices(2,:);
      up = matrices (1,:);
      known = matrices (4,:);
      vs=tridiagonal (bottom(2:end), main, up(1:end-1), known);
      xs= [x0:range/(n+1):x0+range];
      plot (xs,vs);
end
```

For the first set of initial conditions, the solution can be seen in Figure 1.

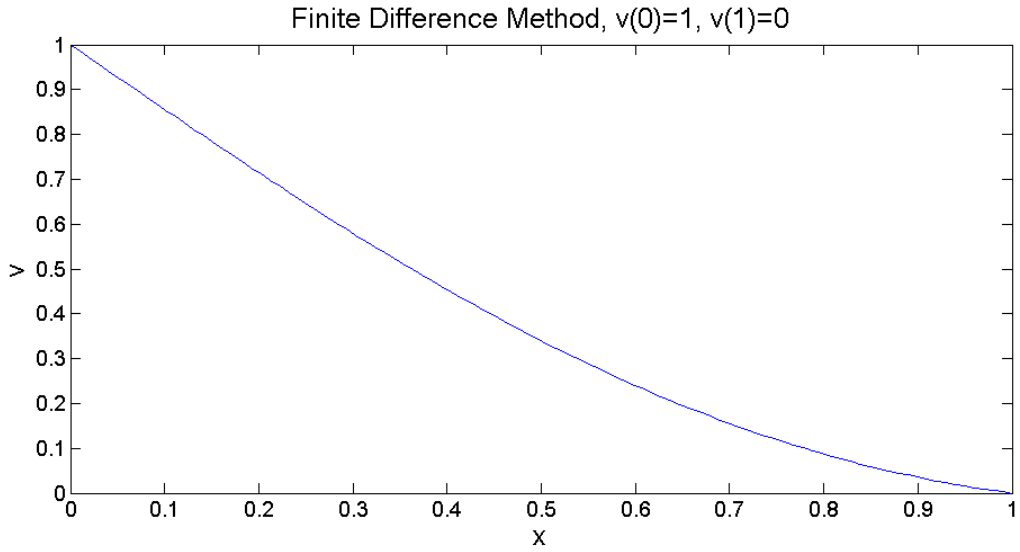

Figure 1. Solution curve found using the finite difference method for boundary conditions as shown.

The same solver was used to find the solution for the ODE with boundary conditions $m = 1, n = 1, p = 1, q = 0.5, r = 1, s = 0$. The solution curve calculated for this BVP is shown in Figure 2.

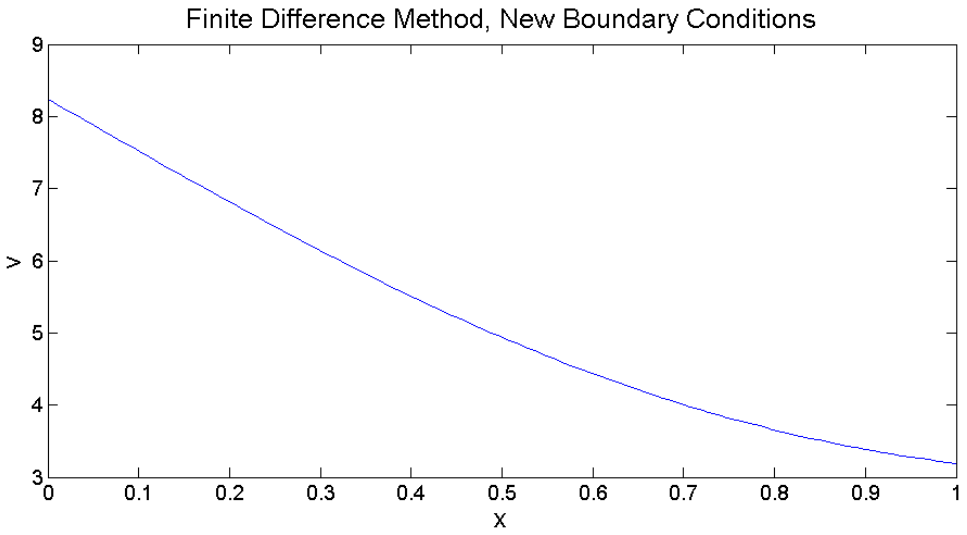

Figure 2. Solution curve found using the finite difference method for boundary conditions $v'(0) + v(0) = 1, v'(1) + .5v(1) = 0$.

The accuracy of this solution curve was verified by comparing it to the result of the previous assignment, which involved solving the same boundary value problem using the shooting method.

EFFICIENCY AND ACCURACY

The number of computations made by the tridiagonal solver is proportional to the size of the tridiagonal. Thus, we would expect the computation time to increase proportionally with the size of the matrix. We test this hypothesis by inputting a range of matrix sizes, and keeping track of the computation time associated with each matrix calculation. This test was implemented using code as follows:

```
function tridiagtime (N)
    time = zeros (N, 1);
    n = zeros (N,1);
    for i = 10:N
        n(i)= i;
        tic
        solution=tridiagonal (rand (1,N-1), rand(1,N), rand (1,N-1),
rand(1,N));
        time (i) = toc;
    end
    loglog (n, time);
```

and the results of this test can be seen in Figure 3.

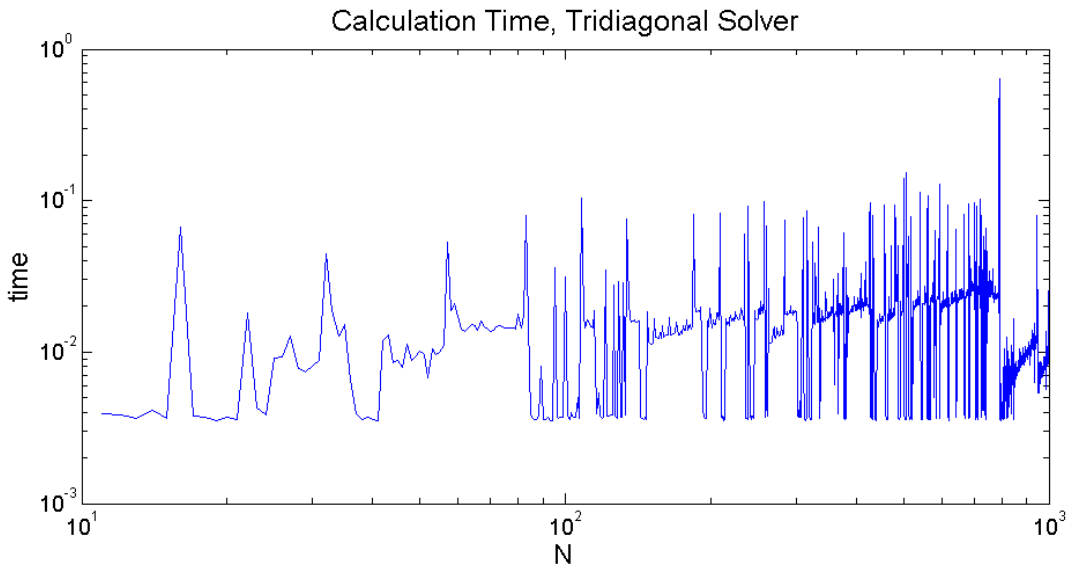

Figure 3. Plot of calculation time as found using "tic toc" in MATLAB as a function of N, the size of the square matrix being solved by the tridiagonal solver.

The trend shown is not quite the result we expect. Since the number of calculations varies linearly with N, we should expect a fairly linear dependence between the size of the matrix being solved and calculation time. However, our results may be confounded by the time it takes to generate a series of random test matrices, so that this time dominates compared to the time it actually takes to solve. This hypothesis was tested by moving the generation of the diagonals before counting starts. However, the resulting time graph does not look different. It is possible that the calculations take so little time that fluctuations in CPU and the speed of the computer overshadow actual variations in calculations time. It is difficult to conceive of other factors that may result in the trend we see.

Evaluation of the finite different method centers on accuracy more than efficiency. As a method of approximation, we must be able to characterize the errors associated with this method. It is standard to look at the dependency of error on tolerance, which in this case is the size of Δx. Based on the Taylor series approximation we used to arrive at this method, we know that the error associated with this method should vary depending on the square of Δx. To approximate the exact solution, we first attempted to use the finite difference method with many points (> 10^6). However, this is limited by the fact that there is always some level of error associated with this method. And especially when comparing this "exact" solution with implementation of the finite difference method with low tolerance (small values of Δx), the difference between exact and approximate values become dominated by the differences in tolerance instead of the actual error

associated with the method. As such, a new boundary value problem with a known solution was used to evaluate the accuracy of this method:

$$U'' = -e^{4x}, -1 < x < 1, U(-1) = U(1) = 0 \tag{16}$$

with solution

$$U(x) = \frac{e^{4x} - x\sinh(4) - \cosh(4)}{16}. \tag{17}$$

Error between this exact solution and the approximation found with the finite difference method was determined by identifying corresponding terms between the two solution curves. That is, a vector of "exact" values was calculated by defining a set of n x-values between -1 and 1. Then, a vector of approximate values with some n/k values, where k is an integer is produced. Each ith value in the approximate vector is compared to the $k * i$th value in the exact vector. The difference between the two values is squared, the mean of all these values is found, the square root of this mean is taken, and this RMS (root mean square) value is plotted as function of the Δx used to produce the approximation vector. This curve should have a slope of 2 to reflect the square dependency of error on Δx. The code to implement this was as follows, where RMS was actually calculated by taking the norm of the difference then dividing by the square root of the length of the vectors:

```
function res=errortutrl4 (functn, range, tolrange, x0, highest, coeff1,
coeff2, A,B,C, functnexact)
    y = linspace(-1,1,10^highest);
    exact = functnexact(y);

        for i = 1:1:tolrange;
            numdiv(i) = 10^i;
            factor = 10^highest/numdiv(i);
            x = 2*factor: factor:10^highest-factor;
            temp1= functn(range,numdiv(i),x0,coeff1,coeff2,A,B,C);
            vs = temp1(2:length(temp1)-1);
            real = exact(x);
            dx(i)=range/(numdiv(i)-1);
            diff(i) = norm (real'-vs) /sqrt(length(vs));
        end

     loglog(dx, diff);
     a = log(dx);
     b = log(diff);
    slope = (b(length(dx)-1)-b(1))/(a(length(dx)-1)-a(1));
    res =[slope (dx); 0 (diff)];

end
```

In this function, "functn" outputs the approximate solution found using the finite difference method. "Range" is the range of x-values for which the boundary value problem is defined, "tolrange" is the range of tolerances the function should sweep, "x0" is the lower bound of x, "highest" defines the number of terms used to generate the exact solution, the vectors "coeff1" and "coeff2" define the coefficients in the general boundary definition cases $av'(x_0) + bv(x_0) = c$ and $av'(x_{end}) + bv(x_{end}) = c$. A,B, and C refer to coefficients in the definition of the second derivative $v'' + A(x)v' + B(x)v + C = 0$, and "functnexact" is the exact solution to the ODE.

The results of this analysis are shown in Figure 4.

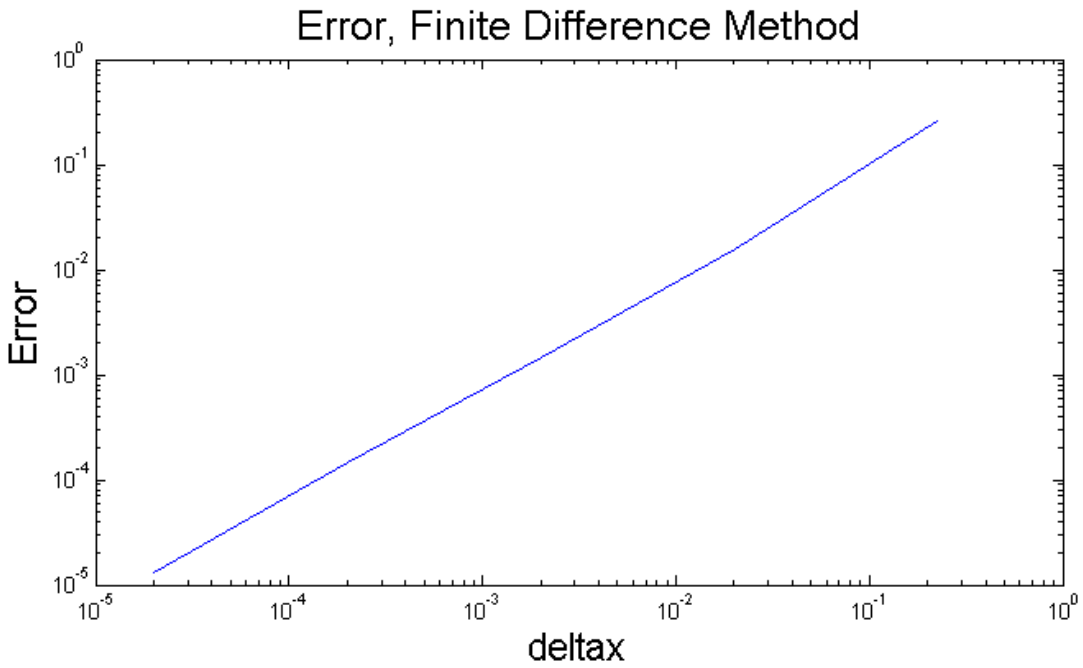

Figure 4. Error associated with the finite difference method as a function of the Δx, plotted on a loglog scale. The slope of this line is 1, while the expected slope is 2.

It is unclear why this discrepancy occurred. Attempts were made to find the source of this error. The solution curve produced by the finite difference method was compared to the exact solution, and they were comparable. This indicates that the error must have arisen in defining error. Instead of RMS, the midpoint of each solution curve was used to calculate error, and the same line with slope 1 was found. In the end, this discrepancy between predictions and actual results could not be reconciled.

CONCLUSION

In this assignment, we implemented a tridiagonal solver, which was then used as part of the finite difference method to approximate solutions to boundary value problems. Certain problems arose when attempting to analyze the efficiency and accuracy of our methods, and discrepancies between predictions based on theory and actual results could not be explained satisfactorily. Our efforts to do this continues, and will be reported if successful.