# 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:

|  |  |
| --- | --- |
|  | (1) |

using the finite difference method. Investigate into the error associated with this method, and adapt for the boundary conditions .

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 .

### 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:

|  |  |
| --- | --- |
|  | (2) |

where is a function of , and c can be

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

|  |  |
| --- | --- |
|  | (3)  (4) |

to rewrite the linear boundary value system (2):

|  |  |
| --- | --- |
|  | (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 pairs that satisfy the equation above. Thus, we can state that given the first boundary value problem with there are values with corresponding that satisfy the given ODE. Essentially, we are approximating the values of in the range using equally spaced points. If we take as an example,

|  |  |
| --- | --- |
|  | (6) |

where we express as , since they are approximations and not exact values, and . Equation (6) can be manipulated to be as follows:

|  |  |
| --- | --- |
|  | (7) |

Note that this is the form of

|  |  |
| --- | --- |
|  | (8) |

where and are functions of .

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

|  |  |
| --- | --- |
|  | (9) |

This case is special for the boundary values. At , we can substitute the boundary value for , so that:

|  |  |
| --- | --- |
|  | (10) |

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

|  |  |
| --- | --- |
|  | (11) |

where H is a tridiagonal matrix with coefficients :

is the set of s: , and 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 , as prescribed by the coefficients in , give us .

The case is slightly different if we have boundary conditions expressed in the form and . We can no longer simply move the term to the “known” side of the equation in (7). Furthermore, we cannot use the central difference approximation for for , since this depends on knowing the value of . The same restraints apply to trying to satisfy the ODE at . Thus, we deal with boundary conditions in a more general fashion.

Consider the condition . We can express this in terms of and by approximating as , such that, with rearrangement,

|  |  |
| --- | --- |
|  | (12) |

Similarly,

|  |  |
| --- | --- |
|  | (13) |

This allows us to add 2 rows to , and :

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

Tri-diagonal Solver

Given this system of linear equations, we can use row reduction to solve for the column vector . Row reduction involves canceling out every term in the coefficient matrix except the diagonal. This matrix then becomes a scaling matrix relating 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 :

|  |  |
| --- | --- |
|  | (14) |

where positions marked with and have some non-zero entry and s as constants.

In order to row reduce this matrix, we have to get rid of all terms where there are s, and keep the central diagonal of values, so that we end up with five equations where some coefficient times 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:

|  |  |
| --- | --- |
|  | (15) |

Note that each value in the positions marked by and 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, , as indicated by the different indices in equation (15).

Using this tridiagonal matrix solving algorithm, we can calculate the value of 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 . If we use our general way of dealing with boundary conditions with the form and , then 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 and . 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.

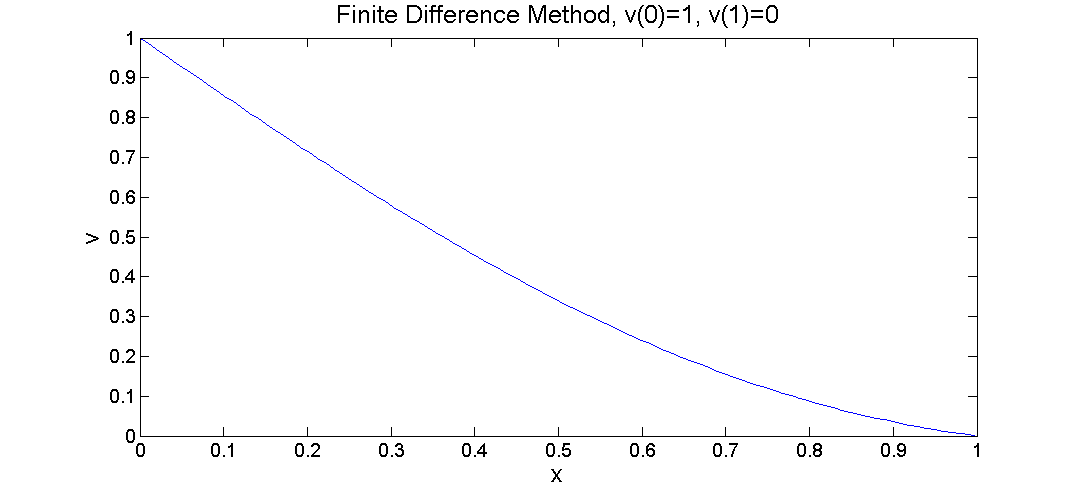
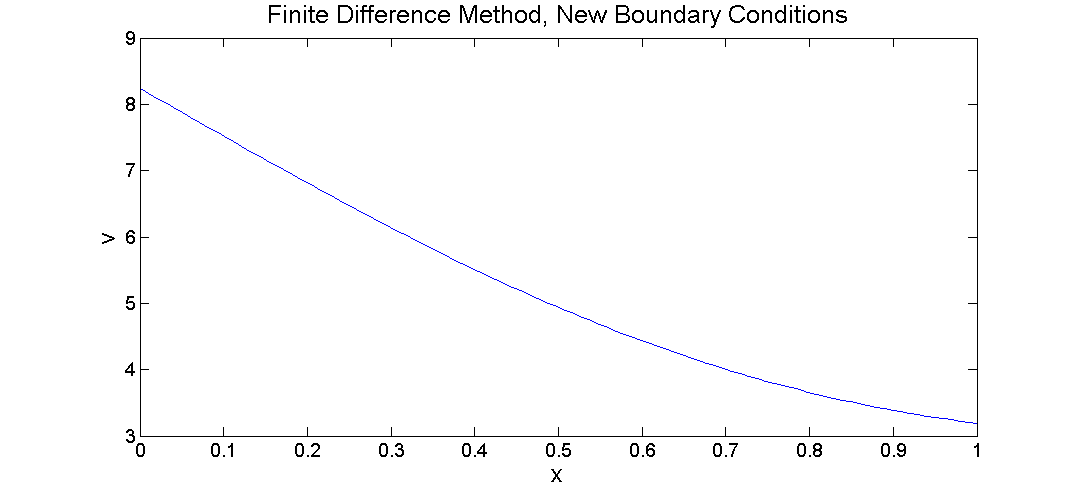


Figure 1.

The same solver was used to find the solution for the ODE with boundary conditions . The solution curve calculated for this BVP is shown in Figure 2.



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. The results of this test can be seen in Figure XXXX.

INSERT FIGURE.

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 . To calculate error, we simply \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. This difference \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. We see that \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_