Peter Cottle SID 19264824

ME 280A – 10/20/2011

Homework Assignment #3

Introduction:

Now that the class has been introduced to the finite element method and examined the choice of different basis elements, Homework #3 will examine computationally efficient solution techniques and the behavior of the 'potential' across a wide range of nodes. Modern FEM codes in mechanical industry never use full-matrix storage or Gaussian elimination because using those techniques on large node networks (in the millions) would require ridiculous amounts of memory and CPU cycles to solve. This is even more paramount in graphics simulation, where FEM problems need to be solved during every frame render which can be up to 60 times a second. Professor O'Brien in the CS department has done some great work in this area on improving over typical conjugate gradient solvers in graphics applications.

Nevertheless, this homework will examine efficient methods of storage and solution computation. It will also introduce the idea of interface conditions – situations where there is a large jump in element stiffness (a change of material) yet the displacement and stress must remain continuous across the boundary. This is more of an issue in determining the exact solution rather than numerical one.

Objectives:

The objectives of this homework build upon those from Homework #1 (solving an FEM problem with linear basis elements) but the inner code that computes the solution is completely ripped out. Instead of storing the entire stiffness matrix (in a sparse matrix) and using Gaussian elimination, we aim to create our own element-by-element stiffness matrix storage and use a pre-conditioned conjugate gradient method to achieve a solution. The exact details of this process are outlined below:

- Store only the diagonals and one off-diagonal term from each element in a custom stiffness matrix data structure. Use this data structure instead of the sparse matrix for all following computations.
- Use this custom data structure to precondition the conjugate gradient method. This procedure is not entirely trivial, for we must first determine the correct L matrix to use (by examining the stiffness matrix as a whole) and then we must multiply on the left and on the right by this diagonal matrix. We use heuristics here to solve this coding challenge.
- Write a custom matrix and vector multiplication scheme that takes in our custom data structure and outputs the result of a matrix and vector product. This again is not trivial, especially for edge cases.
- Integrate the above methods into a conjugate gradient solver and potential calculator to solve the overall FEM problem.

Lastly, we will also program our A1(x) values (the stiffness of the material) to change with position on the bar. This mainly means using an x to zeta transformation on a simple function. This will also be reflected in the exact solution.

Procedure:

This homework examined the given problem from both an analytical and numerical approach. The analytical approach built upon the previous work in Homework #1 and Homework #2 but instead involved creating a 20x20 system of equations. These equations reflect the boundary conditions and interface conditions; after creating this system, we then solve for each of these 20 constants. This is outlined in the findings section.

The numerical side of this problem involved coding all of the features outlined in the objectives section (which I will not repeat here). We build upon the weak form that was derived in Homework #2 and instead work with a different solution technique and variable element stiffness. Although the different solution technique required extra coding work and introduces more room for error, it performs considerably faster than a traditional Gaussian elimination solver on a full (non-dense) matrix.

Findings

Re-solving with new Interface Conditions

We can pick up from the end of HW#1 after the first integration and replace A1 with A1(x) and partially substitute 16 for k:

$$A_1(x)\frac{du}{dx} = \frac{16}{\pi}sin(\frac{\pi k x}{L}) - \frac{5L(1-k^2)}{2\pi k}cos(\frac{2\pi k x}{L}) + C_1$$

Now we note that since A1(x) is dependent on x across 10 different domains, we will have 10 different C1's and 10 different C2's in the resulting system of equations. Replace C1 and A1 with their respective domain values and domain-specific constants, which results in 10 different equations:

$$A_0\frac{du}{dx} = \frac{16}{\pi}sin(\frac{\pi k x}{L}) - \frac{5L(1-k^2)}{2\pi k}cos(\frac{2\pi k x}{L}) + C_{00}$$
$$A_1\frac{du}{dx} = \frac{16}{\pi}sin(\frac{\pi k x}{L}) - \frac{5L(1-k^2)}{2\pi k}cos(\frac{2\pi k x}{L}) + C_{02}$$
$$A_2\frac{du}{dx} = \frac{16}{\pi}sin(\frac{\pi k x}{L}) - \frac{5L(1-k^2)}{2\pi k}cos(\frac{2\pi k x}{L}) + C_{04}$$
$$A_3\frac{du}{dx} = \frac{16}{\pi}sin(\frac{\pi k x}{L}) - \frac{5L(1-k^2)}{2\pi k}cos(\frac{2\pi k x}{L}) + C_{06}$$

etc, up until:

$$A_9\frac{du}{dx} = \frac{16}{\pi}sin(\frac{\pi kx}{L}) - \frac{5L(1-k^2)}{2\pi k}cos(\frac{2\pi kx}{L}) + C_{18}$$

These 10 different equations for the stress in the bar are then solved for (du/dx):
(only the last equation is shown here)

$$\frac{du}{dx} = \frac{16}{A_9\pi}sin(\frac{\pi kx}{L}) - \frac{5L(1-k^2)}{A_9 2\pi k}cos(\frac{2\pi kx}{L}) + \frac{C_{18}}{A_9}$$

And then integrated once to yield 20 different equations with a new constant:

$$u(x) = -\frac{L^2}{A_9\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_9 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_{18}}{A_9}x + C_{19}$$

There is one equation for each domain:

$$u(x) = -\frac{L^2}{A_8\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_8 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_{16}}{A_8}x + C_{17}$$

$$u(x) = -\frac{L^2}{A_7\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_7 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_{14}}{A_7}x + C_{15}$$

$$u(x) = -\frac{L^2}{A_6\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_6 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_{12}}{A_6}x + C_{13}$$

$$u(x) = -\frac{L^2}{A_5\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_5 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_{10}}{A_5}x + C_{11}$$

$$u(x) = -\frac{L^2}{A_4\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_4 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_8}{A_4}x + C_9$$

$$u(x) = -\frac{L^2}{A_3\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_3 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_6}{A_3}x + C_7$$

$$u(x) = -\frac{L^2}{A_2\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_2 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_4}{A_2}x + C_5$$

$$u(x) = -\frac{L^2}{A_1\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_1 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_2}{A_1}x + C_3$$

$$u(x) = -\frac{L^2}{A_0\pi^2}cos(\frac{\pi kx}{L}) - \frac{5L^2(1-k^2)}{A_0 4\pi^2k^2}sin(\frac{2\pi kx}{L}) + \frac{C_0}{A_0}x + C_1$$

Thus, there are now 10 equations total for the displacement (listed above) and 10 equations total for the stress in the bar (the 10 equations partially listed earlier). There are also 9 interface conditions in both stress and displacement between the domains, meaning that there are 18 interface conditions total. These 18 conditions are then supplemented by 2 boundary displacement conditions:

$$u(0) = 0$$
$$u(1) = 1$$

To yield, in total, 20 equations and 20 unknowns (constants C_00 through C_19). The interface conditions require that the displacement and the stress in the bar are continuous when crossing domains. This essentially requires that the values of the displacement and stress in the bar, when evaluated at the boundary for each of the two domains, are the same. This can be expressed in equation form for the displacement:

$$u_{domain1}(0.1) = -\frac{L^2}{A_0\pi^2}cos(\frac{\pi k0.1}{L}) - \frac{5L^2(1-k^2)}{A_0 4\pi^2 k^2}sin(\frac{2\pi k0.1}{L}) + \frac{C_0}{A_0}0.1 + C_1$$

$$u_{domain2}(0.1) = -\frac{L^2}{A_1\pi^2}cos(\frac{\pi k0.1}{L}) - \frac{5L^2(1-k^2)}{A_1 4\pi^2 k^2}sin(\frac{2\pi k0.1}{L}) + \frac{C_2}{A_1}0.1 + C_3$$

$$u_{domain1}(0.1) - u_{domain2}(0.1) = 0$$

And for the stress:

$$A_0 u'_{domain1}(0.1) = \frac{k}{\pi}sin(\frac{\pi k0.1}{L}) - \frac{5L(1-k^2)}{2\pi k}cos(\frac{2\pi k0.1}{L}) + C_{00}$$

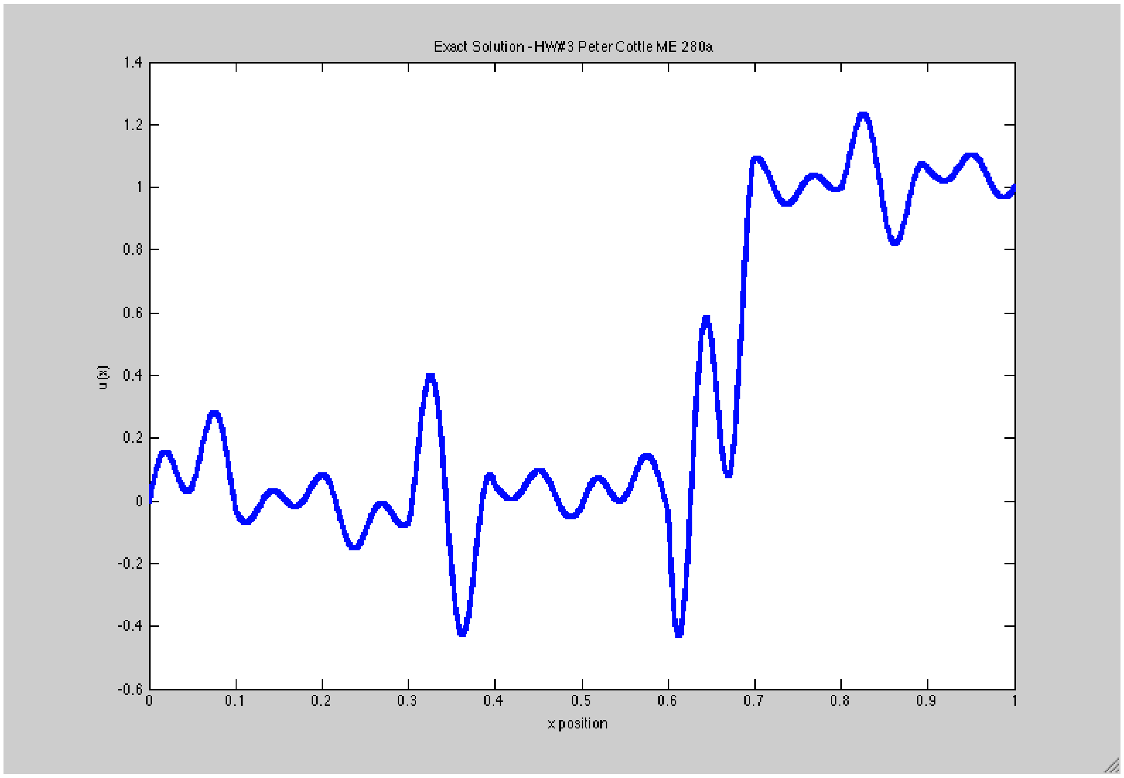
$$A_1 u'_{domain2}(0.1) = \frac{k}{\pi} sin(\frac{\pi k 0.1}{L}) -$$
$$\frac{5L(1-k^2)}{2\pi k} cos(\frac{2\pi k 0.1}{L}) + C_{02}$$

$$A_1 u'_{domain2}(0.1) - A_0 u'_{domain1}(0.1) = 0$$

These equations are constructed at each of the interface conditions. Finally, a system of 20 equations with 20 unknowns is produced; 18 are from interface conditions and 2 are from the boundary conditions. I then wrote a MATLAB scrip that utilized the symbolics package to solve this system of equations for each of the constants. As I had expected during office hours, all the odd C constants turned out to be the same value because the stress interface conditions are just functions of x, not A1(x). Regardless, the constants are listed below:

Stress constant	Value	Displacement Constant	Value
C00	-0.248809919162060	C01	0.1013211836423377
C02	-0.248809919162060	C03	0.023114762618553
C04	-0.248809919162060	C05	-0.002975084032984
C06	-0.248809919162060	C07	0.157960077111768
C08	-0.248809919162060	C09	0.065209579236402
C10	-0.248809919162060	C11	0.112113712893205
C12	-0.248809919162060	C13	0.985538837306934
C14	-0.248809919162060	C15	1.086578155681257
C16	-0.248809919162060	C17	1.237434247197302
C18	-0.248809919162060	C19	1.116710367601466

These constants produce the following plot when evaluated along with the trigonometric terms:



Thus, the analytical solution to the problem was obtained. The MATLAB code to produce this system of equations and constants is provided in the Appendix.

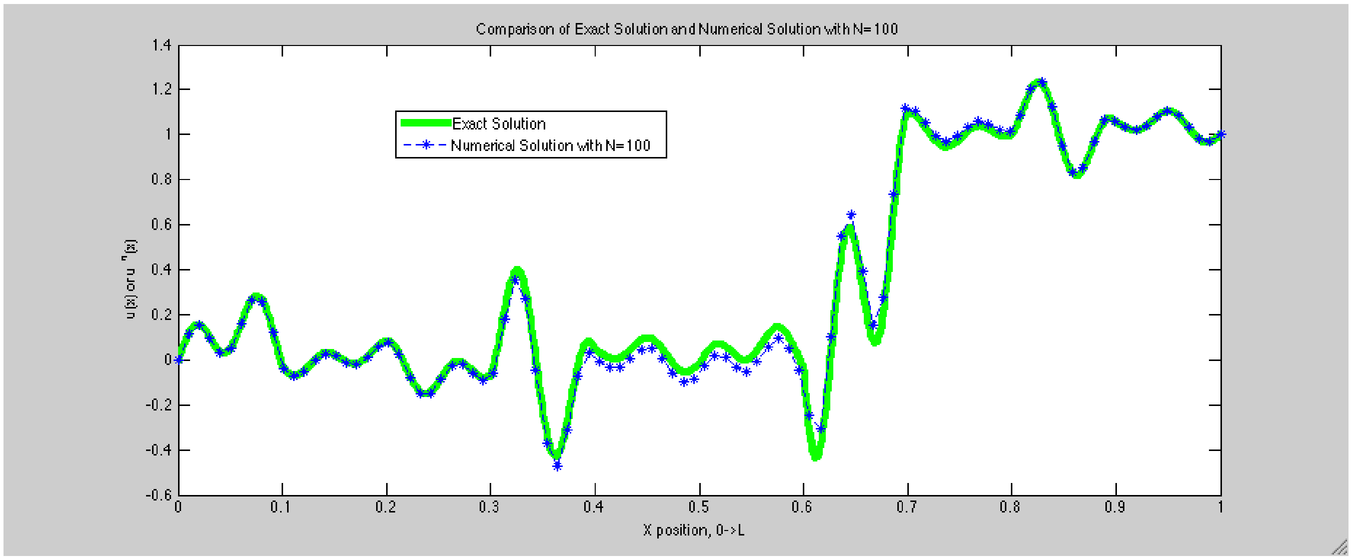
Creating the Conjugate Gradient Solver

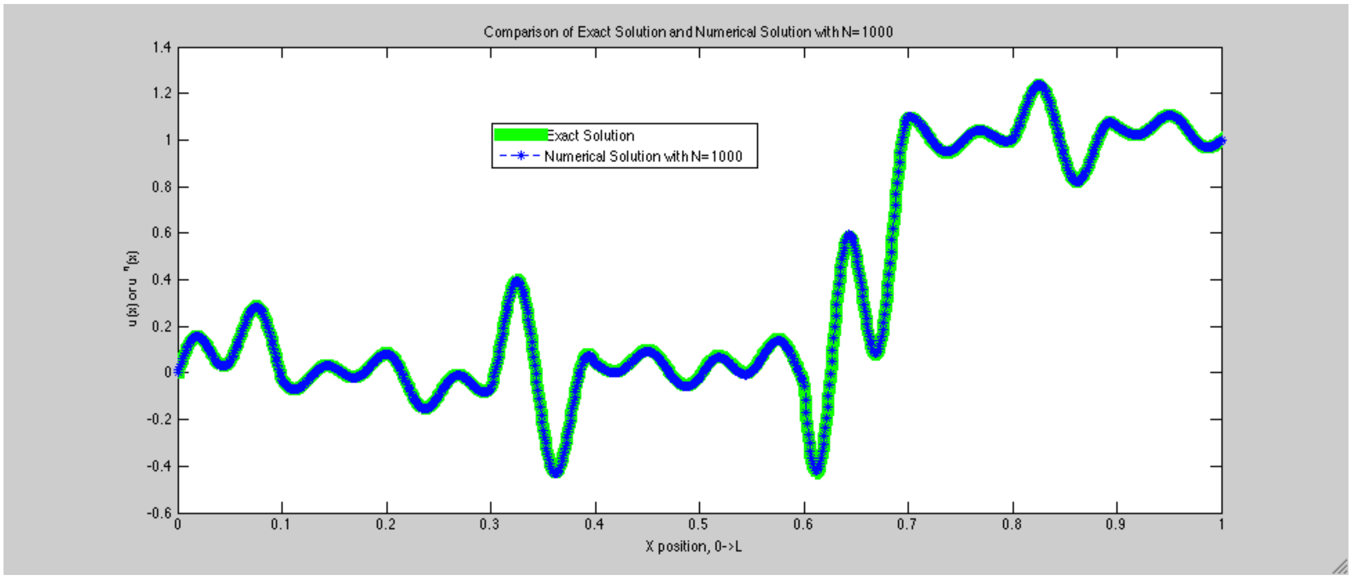
The process of creating the conjugate gradient solver and sparse element matrix was time-intensive. The code for these algorithms can be viewed in the Appendix.

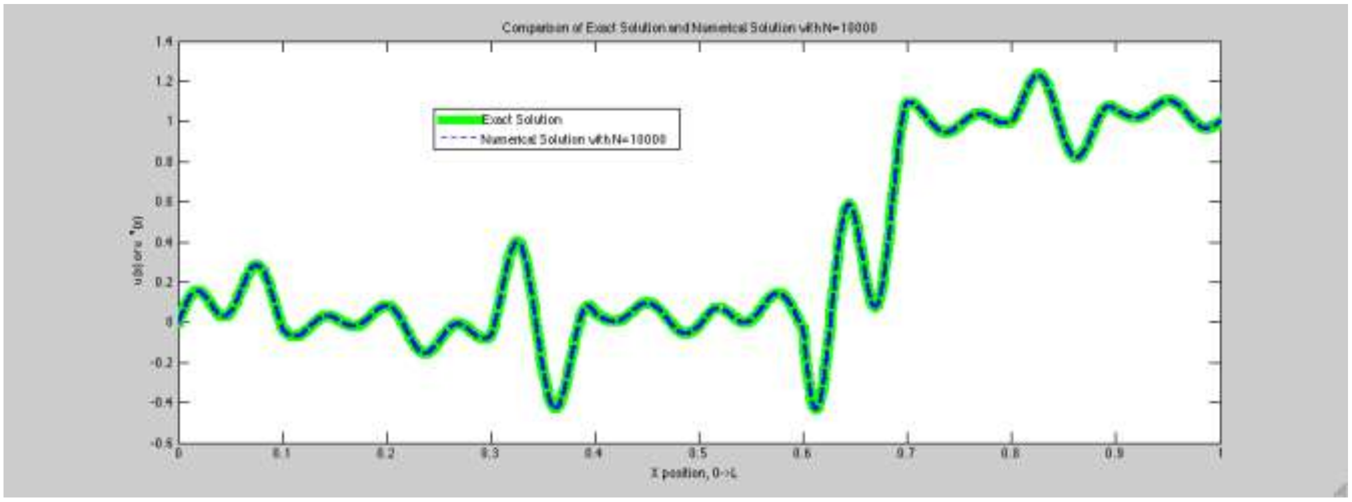
Solving the given boundary value problem with various values of N

Plotting the numerical solution for several values of N:

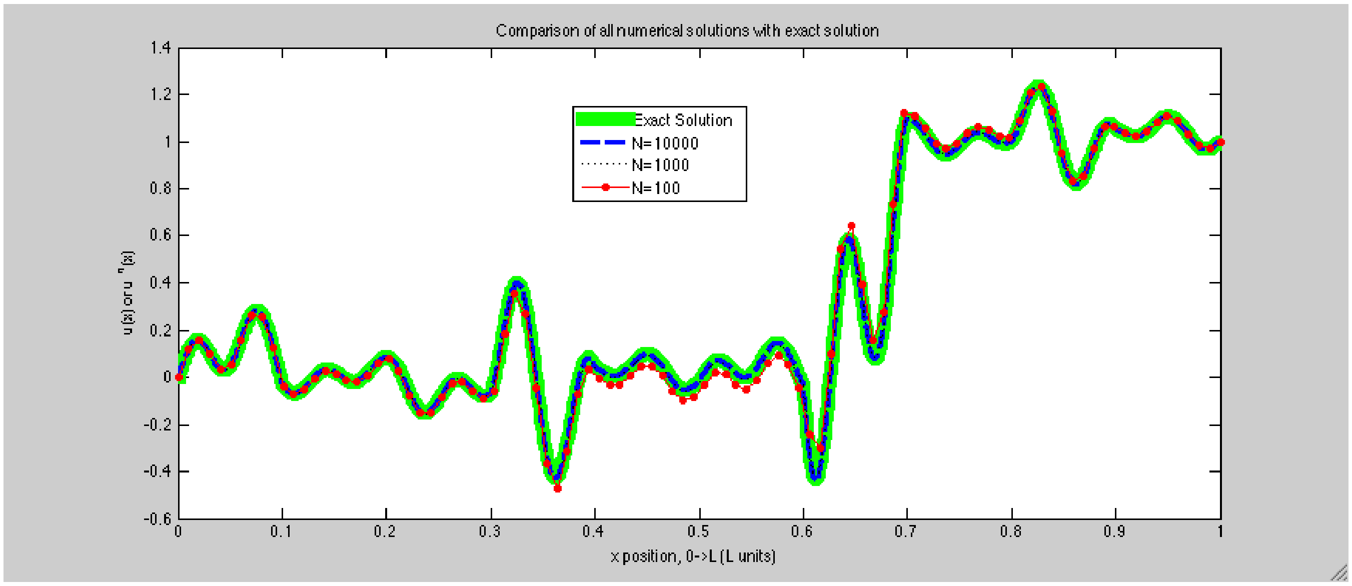
N=100:


Comparison of Exact Solution and Numerical Solution with N=100
Exact Solution
Numerical Solution with N=100
u(x) or u^n(x)
X position, 0->L

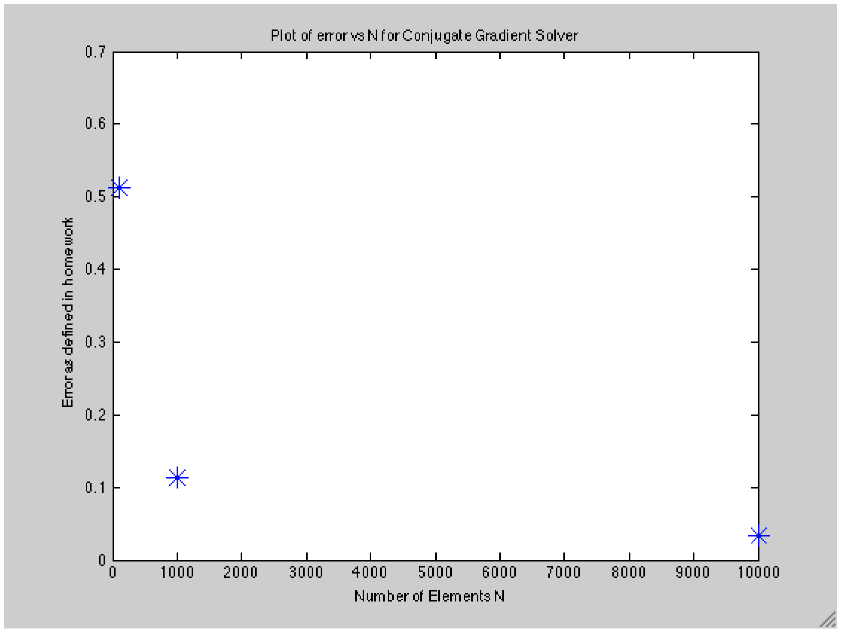

Comparison of Exact Solution and Numerical Solution with N=1000
Exact Solution
Numerical Solution with N=1000
u(x) or u^n(x)
X position, 0->L


Comparison of Exact Solution and Numerical Solution with N=10000
Exact Solution
Numerical Solution with N=10000
u(x) or u^n(x)
X position, 0->L

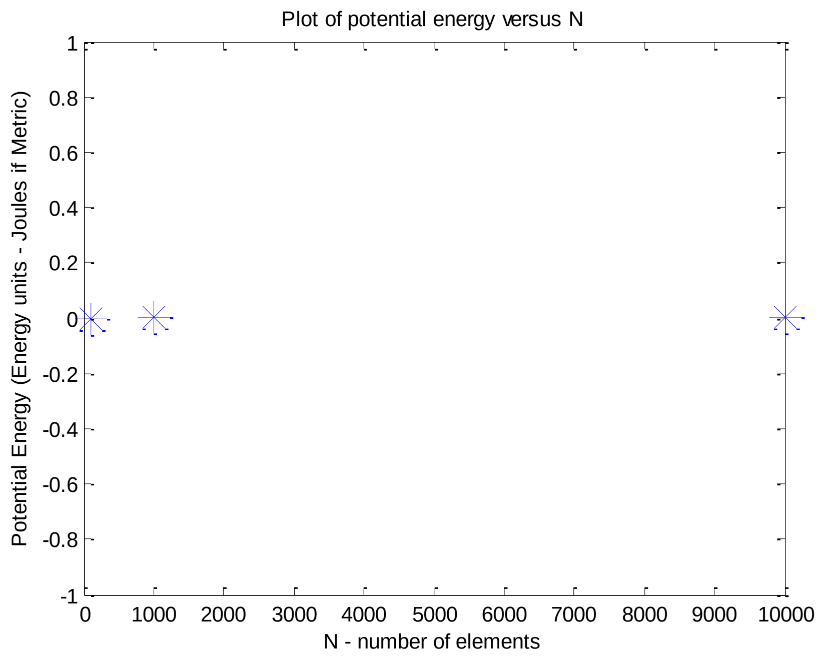
Plotting all together:



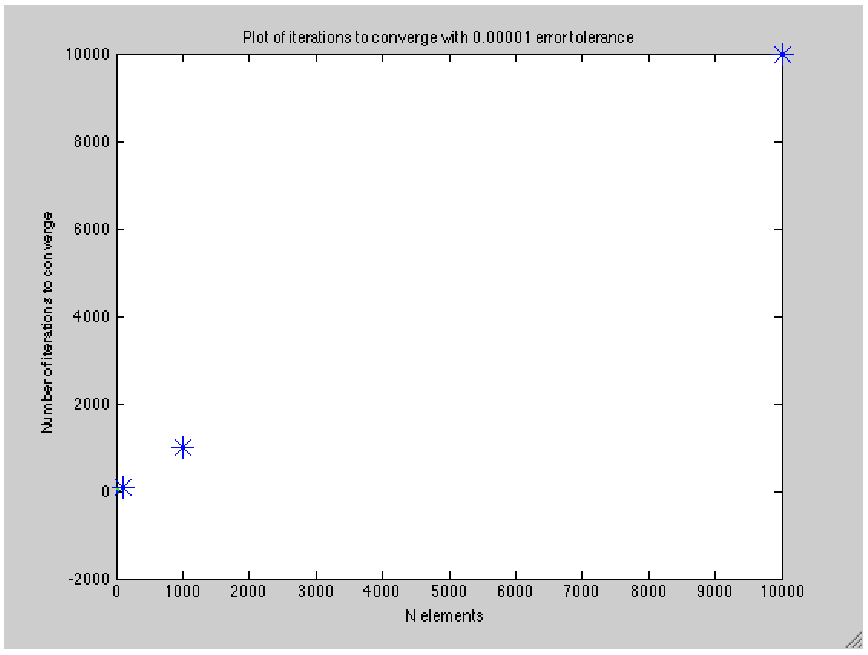
Plotting error vs N:



Plotting POTENTIAL ENERGY for each N:

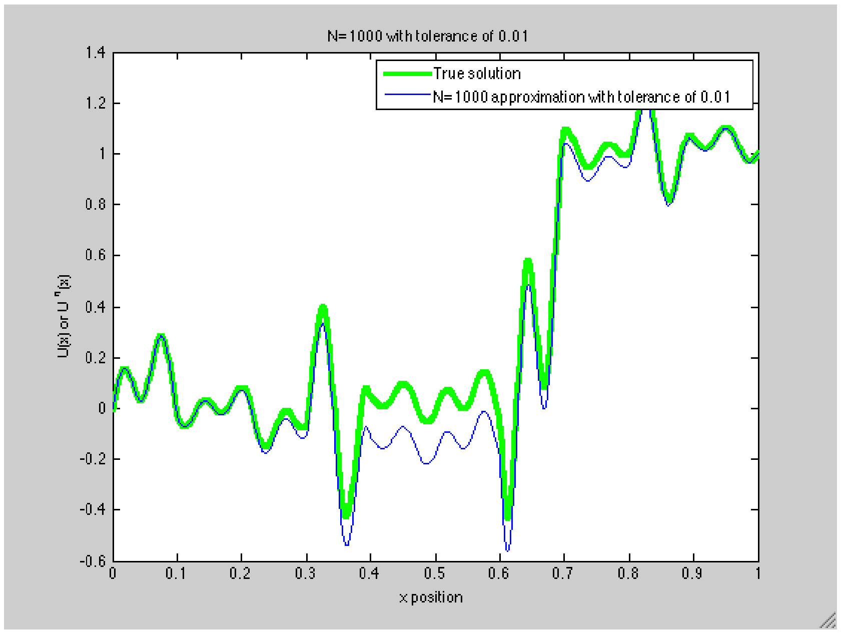


Plot of Number of Iterations to Converge versus N:

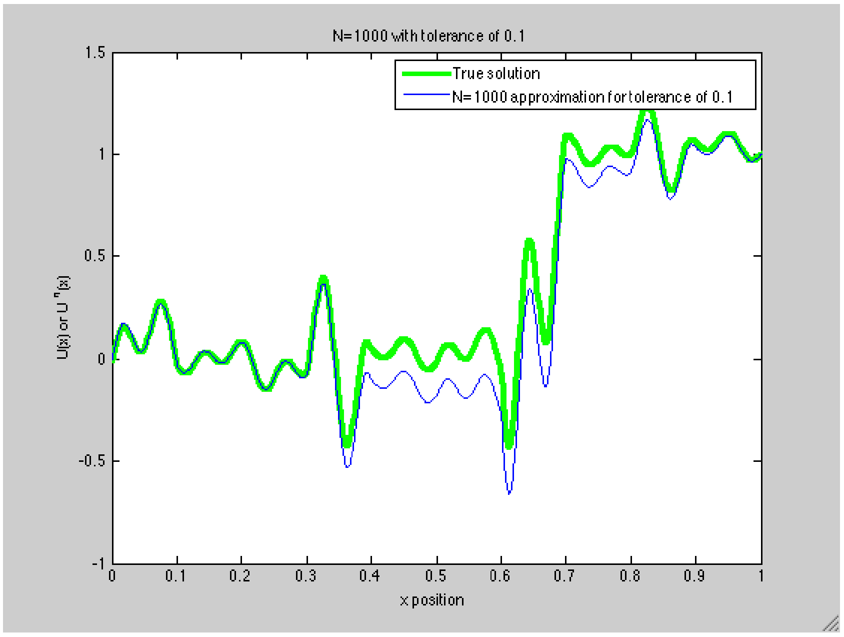


Note: I used the Conjugate Gradient method that was outlined in the class notes, but I used the outline of the code from the Wikipedia which uses the norm of the residual vector as the error. This is why the number of iterations is so closely related to the number of elements; since there are N-1 degrees of freedom, the conjugate gradient method essentially moves in one of these degrees of freedom each iteration. Thus after N-1 iterations, the conjugate gradient method has moved in all possible directions and thus now has reached the final solution.

When I alter the error tolerance, no noticeable artifacts occur until around 0.01 for the tolerance. At this point, sometimes the middle area of the solution diverges considerably but the edges of the solution are still very close to the true solution. A plot of this is below:



And even with a tolerance of 0.1:



The middle and ends of the solution are the only areas that diverge. There are two aspects to consider here:

- I am using the Wikipedia conjugate gradient solver, so this might perform better with low error tolerance because it examines the r (residual) vector.
- Also, because the middle of the solution is furthest from either of the boundary conditions, it may "absorb" most of the error since the solution must match 0 and 1 at the boundary. Another aspect to consider is that the areas with lower stiffness (A1 value) both displace more and have higher errors. This is to be expected however when considering the material stiffness analogy.

Implementation Details

The prior section of this homework was printed out on a color printer before the implementation section was complete; consequently, I have decided to add the implementation details section here rather than sacrifice the color-coding of the graphs. Below is a concise and brief summary of how to implement this procedure (as per the homework guidelines):

- Storing the stiffness matrix element-by-element
    - Instead of using Matlab's built-in 'sparse' matrix, the students in the class were required to create a new data structure that eliminated repetition of storage. This meant that each element was stored individually; only the two diagonal elements and one off-diagonal element were stored for each element. The actual storage of these elements is easy; performing operations on this "matrix" was the hard part. Luckily, only one method had to be implemented: matrix multiplication with a vector. This involves programming a loop to loop through and multiply each element by the correct element in the vector and sum the results into a result vector.
- Preconditioning the matrix
    - This was also a slight challenge, for the preconditioning matrix L is defined by the diagonal of the full stiffness matrix, a piece of data that we do not have available. Consequently, we had to create an algorithm that looped through and determined the global diagonal elements (the sum of the overlapping elemental diagonal entries) and then calculated the necessary L value. After this, we had to multiply this matrix into the sparse matrix twice. This was fortunately easy, for L is a diagonal matrix and a simple heuristic can be used to produce the correct solution.
- Conjugate Gradient
    - This section simply involved the pre-conditioning mentioned earlier and then performing the conjugate gradient method on our linear algebra problem. Several implementations of this algorithm are available online and thus the details will not be discussed here. The main points are just to obtain an initial guess, the residual vector, the "p" vector, and the necessary constants to update the initial guess over successive iterations. Once the residual or the error is low enough, the loop quits.

Discussion

There were many interesting things to note about this homework:

Firstly, it appears that the finite element method essentially minimizes the potential energy of the system. As can be seen from our plot of potential energy vs N, all the final solutions achieved very low potential energies. This makes sense logically, for the system will equilibrate at the minimum potential energy configuration; in fact, the undergraduate FEM course at UCSD approaches the problem from this angle rather than the weak form. Understanding the potential will be a useful tool when going into 3 dimensions.

Secondly, the conjugate gradient method took the exact number of iterations to converge as the number of degrees of freedom in the problem. This may have been a unique feature of my implementation of the conjugate gradient, but as I mentioned in the findings section, it makes sense from a theoretical standpoint. This essentially means that the number of iterations to converge is simply a **linear function** of the number of nodes, where the Gaussian elimination solution is easily seen as O(n^3). This is an incredible improvement in performance, and it shows how some finite element problems could not be solved with normal Gaussian elimination due to the time and space complexities of cubic Big-O growth. Also, let it be noted that the homework assignment did not specify whether to use residual or delta x as the stopping point for the conjugate gradient method, so I stand by my implementation.

Thirdly, the number of iterations to converge with a larger error tolerance was not affected until the error tolerance was severely large (in the 0.01 range). This is because the conjugate gradient method still moves in each direction once; only once the residual vectors are within the 0.01 magnitude range (for the final iterations) does the algorithm quit early. I still obtained N-1 iterations to converge with 0.001 as my error tolerance.

Lastly, this entire homework has shown me the importance of iterative methods for solving large linear algebra problems and the theoretical factors that influence convergence behavior with iterative methods.

APPENDIX:

Like in homework #1, the raw data for the above figures is not provided here but can be provided upon request. It is also quite easily generated from the included matlab code below. Also, the file 'interfaceSolver.m' must be used to solve the system of equations. The code for this is shown at the top of the right column below.

```
% Peter Cottle ME280a hw1!!
%clear all
clc
%close all

% Start / endpoints of the bar
domainStart = 0;
domainEnd = 1;

k = 16;
L = domainEnd - domainStart;

showextraGraphs = false;

order = 1;

loadingInZ = @(x) -1 * (k^2 * cos((pi * k * x)/L) + 5 * (1-
```

```
disp1  = '-1*(1/(pi^2*A0))*cos(pi*16*0.1) +
(1275/(A0*4*pi^2*16^2))*sin(2*pi*16*0.1) + C00*0.1/A0 + C01 = -
1*(1/(pi^2*A1))*cos(pi*16*0.1) + (1275/(A1*4*pi^2*16^2))*sin(2*pi*16*0.1)
+ C02*0.1/A1 + C03'
disp2  = '-1*(1/(pi^2*A1))*cos(pi*16*0.2) +
(1275/(A1*4*pi^2*16^2))*sin(2*pi*16*0.2) + C02*0.2/A1 + C03 = -
1*(1/(pi^2*A2))*cos(pi*16*0.2) + (1275/(A2*4*pi^2*16^2))*sin(2*pi*16*0.2)
+ C04*0.2/A2 + C05'
disp3  = '-1*(1/(pi^2*A2))*cos(pi*16*0.3) +
(1275/(A2*4*pi^2*16^2))*sin(2*pi*16*0.3) + C04*0.3/A2 + C05 = -
1*(1/(pi^2*A3))*cos(pi*16*0.3) + (1275/(A3*4*pi^2*16^2))*sin(2*pi*16*0.3)
+ C06*0.3/A3 + C07'
disp4  = '-1*(1/(pi^2*A3))*cos(pi*16*0.4) +
(1275/(A3*4*pi^2*16^2))*sin(2*pi*16*0.4) + C06*0.4/A3 + C07 = -
1*(1/(pi^2*A4))*cos(pi*16*0.4) + (1275/(A4*4*pi^2*16^2))*sin(2*pi*16*0.4)
+ C08*0.4/A4 + C09'
```

```
k^2) * sin((2*pi*k*x)/L));
A1inZ = @(x) A1inX(x);
BC_left = 0;
BC_right = 1;

exactSol = @(x) NaN;
% get error norm to divide by
range = [0:0.0001:L];
eVals = 0.2 *
exactDudxFunction(range).*exactDudxFunction(range);
eInt = trapz(range,abs(eVals));
eNorm = sqrt(eInt);

if(order == 1)
    xInTermsOfZ = @(xi,xip1,xip2,z) 1/2*((xip1-
xi).*z+xip1+xi);
elseif (order == 2)
    xInTermsOfZ = @(xi,xip1,xip2,z) xi.*(z-1)*0.5.*z + -
xip1*(z+1).*(z-1) + xip2*(z+1).*z*0.5;
    %xInTermsOfZ = @(xi,xip1,xip2,z) z.^2*(0.5*xi + 0.5*xip2 -
xip1) + 0.5*z*(xip2 + xi) + xip1;
end

% DONT FORGET ABOUT the -1 term (because it goes to the other
side

% Initialize nodes and error
error = 10;
tol = 0.00001;

% 2 for linear, 3 for quadratic, etc
N = 1000;
figure(1)
clf
plot([domainStart:0.01:domainEnd],exactSolFunction([domainStar
t:0.01:domainEnd]),'g', 'LineWidth',6);
hold on
if(showextraGraphs)
    figure(4)
    clf

plot([domainStart:0.01:domainEnd],exactSolFunction([domainStar
t:0.01:domainEnd]),'g', 'LineWidth',6);
    hold on
    errorForPlot = zeros(32,1);
end

xPointsForError = [0];
yPointsForError = [1];

% loop while error is below tolerance
while error > tol

    R = zeros(N,1);

    % Obtain the mesh
    myMesh = mesh(domainStart,domainEnd,N);

    % Loop over elements
    numElements = (length(myMesh) - 1) / order;

    mySparseK = zeros(numElements,3);

    for currElement=1:numElements

        % Get the K matrix
        %currElement
        kElement = miniK(myMesh,currElement,A1inZ,order);
        mySparseK(currElement,:) = [kElement(1,1)
kElement(2,2) kElement(1,2)];

        % Compute the loading factor...
        R =
computeLoading2(currElement,myMesh,R,xInTermsOfZ,loadingInZ,or
der,A1inZ);

    end

    % do the initial condition switchover, but the a's are 0
at the edges
    % so don't bother for now

    % Impose the initial conditions by deleting 1st and last
rows,
    % and deleting first and last columns

    %before deleting the columns, make sure to multiply them
to dirichlet
    %BC's
    R_adderLeft = -1* mySparseK(1,2) * BC_left;
    R_adderRight = -1 * mySparseK(end,2) * BC_right;
    R(2) = R(2) - R_adderLeft;
    R(end-1) = R(end-1) - R_adderRight;
    %delete these columns
```

```
disp5  = '-1*(1/(pi^2*A4))*cos(pi*16*0.5) +
(1275/(A4*4*pi^2*16^2))*sin(2*pi*16*0.5) + C08*0.5/A4 + C09 = -
1*(1/(pi^2*A5))*cos(pi*16*0.5) + (1275/(A5*4*pi^2*16^2))*sin(2*pi*16*0.5)
+ C10*0.5/A5 + C11'
disp6  = '-1*(1/(pi^2*A5))*cos(pi*16*0.6) +
(1275/(A5*4*pi^2*16^2))*sin(2*pi*16*0.6) + C10*0.6/A5 + C11 = -
1*(1/(pi^2*A6))*cos(pi*16*0.6) + (1275/(A6*4*pi^2*16^2))*sin(2*pi*16*0.6)
+ C12*0.6/A6 + C13'
disp7  = '-1*(1/(pi^2*A6))*cos(pi*16*0.7) +
(1275/(A6*4*pi^2*16^2))*sin(2*pi*16*0.7) + C12*0.7/A6 + C13 = -
1*(1/(pi^2*A7))*cos(pi*16*0.7) + (1275/(A7*4*pi^2*16^2))*sin(2*pi*16*0.7)
+ C14*0.7/A7 + C15'
disp8  = '-1*(1/(pi^2*A7))*cos(pi*16*0.8) +
(1275/(A7*4*pi^2*16^2))*sin(2*pi*16*0.8) + C14*0.8/A7 + C15 = -
1*(1/(pi^2*A8))*cos(pi*16*0.8) + (1275/(A8*4*pi^2*16^2))*sin(2*pi*16*0.8)
+ C16*0.8/A8 + C17'
disp9  = '-1*(1/(pi^2*A8))*cos(pi*16*0.9) +
(1275/(A8*4*pi^2*16^2))*sin(2*pi*16*0.9) + C16*0.9/A8 + C17 = -
1*(1/(pi^2*A9))*cos(pi*16*0.9) + (1275/(A9*4*pi^2*16^2))*sin(2*pi*16*0.9)
+ C18*0.9/A9 + C19'


stress1 = '(16/pi)*sin(pi*16*0.1) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.1) + C00 = (16/pi)*sin(pi*16*0.1) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.1) + C02'
stress2 = '(16/pi)*sin(pi*16*0.2) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.2) + C02 = (16/pi)*sin(pi*16*0.2) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.2) + C04'
stress3 = '(16/pi)*sin(pi*16*0.3) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.3) + C04 = (16/pi)*sin(pi*16*0.3) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.3) + C06'
stress4 = '(16/pi)*sin(pi*16*0.4) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.4) + C06 = (16/pi)*sin(pi*16*0.4) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.4) + C08'
stress5 = '(16/pi)*sin(pi*16*0.5) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.5) + C08 = (16/pi)*sin(pi*16*0.5) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.5) + C10'
stress6 = '(16/pi)*sin(pi*16*0.6) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.6) + C10 = (16/pi)*sin(pi*16*0.6) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.6) + C12'
stress7 = '(16/pi)*sin(pi*16*0.7) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.7) + C12 = (16/pi)*sin(pi*16*0.7) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.7) + C14'
stress8 = '(16/pi)*sin(pi*16*0.8) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.8) + C14 = (16/pi)*sin(pi*16*0.8) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.8) + C16'
stress9 = '(16/pi)*sin(pi*16*0.9) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.9) + C16 = (16/pi)*sin(pi*16*0.9) - (5*(1-
16^2)/(2*pi*16))*cos(2*pi*16*0.9) + C18'

firstBoundary = '-1/(pi^2) + C01 = 0'
secondBoundary = '-1*(1/(pi^2*A9))*cos(pi*16*1) +
(1275/(A9*4*pi^2*16^2))*sin(2*pi*16*1) + C18*1/A9 + C19 = 1'



C00 = -0.248809919162060;
C01 = 0.10132118364233777144387946320973;
C02 = -0.248809919162060;
C03 =  0.023114762618553;
C04 = -0.248809919162060;
C05 = -0.002975084032984;
C07 = 0.157960077111768;
C09 =    0.065209579236402;
C11 = 0.112113712893205;
C13 =    0.985538837306934;
C15 =    1.086578155681257;
C17 =    1.237434247197302;
C19 =    1.116710367601466;



C06 = -0.248809919162060;
C08 = -0.248809919162060;
C10 = -0.248809919162060;
C12 = -0.248809919162060;
C14 = -0.248809919162060;
C16 = -0.248809919162060;
C18 = -0.248809919162060;



function [ error ] = calculateError(
myMesh,A,dudx,banana,eNorm,order,exactSol )
%CALCULATEERROR calc error

% order here is linear, so loop through elements. aka just increment one

function [ outputVector ] = exactSolFunction(xVector)
%EXACTSOLCONSTANT Summary of this function goes here
%   Detailed explanation goes here
k = 16;
L = 1;
```

```
    % Do the same for R
    R(N,:) = [];
    R(1,:) = [];

    % Solve for A's based on this
    A = solveViaConjugateGradient(0,R,tol,mySparseK);

    % Pad the a's for the initial conditions
    A = A;
    A = [0; A; 1];

    % Calculate error, aka integral of abs value of difference

    % display our N
    N

    plotAnswer(myMesh,A,order,exactSol);
    error =
calculateError(myMesh,A,exactSol,A1inZ,eNorm,order,exactSol)

    figure(14)
    hold on
    plot(N,error,'b*');
    xPointsForError(end+1) = N;
    yPointsForError(end+1) = error;
    if(order == 3)
        N = N + order*2;
    elseif(order == 2)
        N = N + order*4;
    else
        N = N + 10*min(50,ceil(N/50));
        %N = N + 1;
    end

    pause(0.05)
    pause
    if(showextraGraphs)
        %pause
        figure(4)
        clf
        figure(8)
        clf
        figure(9)
        clf
    end


end


function [ outputVector ] = exactDudxFunction(xVector)
%EXACTSOLCONSTANT Summary of this function goes here
%   Detailed explanation goes here
k = 16;
L = 1;


outputVector = zeros(size(xVector));

C00 = -0.248809919162060;
C01 = 0.10132118364233777144387946320973;
C02 = -0.248809919162060;
C03 =  0.023114762618553;
C04 = -0.248809919162060;
C05 = -0.002975084032984;
C07 = 0.157960077111768;
C09 =    0.065209579236402;
C11 = 0.112113712893205;
C13 =    0.985538837306934;
C15 =    1.086578155681257;
C17 =    1.237434247197302;
C19 =    1.116710367601466;



C06 = -0.248809919162060;
C08 = -0.248809919162060;
C10 = -0.248809919162060;
C12 = -0.248809919162060;
C14 = -0.248809919162060;
C16 = -0.248809919162060;
C18 = -0.248809919162060;


A0 = 1.0;
A1 = 2.5;
A2 = 1.75;
A3 = 0.5;
A4 = 2.75;
A5 = 1.75;
A6 = 0.25;
A7 = 2.75;
A8 = 1.0;
A9 = 3.0;
```

```
outputVector = zeros(size(xVector));

C00 = -0.248809919162060;
C01 = 0.10132118364233777144387946320973;
C02 = -0.248809919162060;
C03 =  0.023114762618553;
C04 = -0.248809919162060;
C05 = -0.002975084032984;
C07 = 0.157960077111768;
C09 =    0.065209579236402;
C11 = 0.112113712893205;
C13 =    0.985538837306934;
C15 =    1.086578155681257;
C17 =    1.237434247197302;
C19 =    1.116710367601466;



C06 = -0.248809919162060;
C08 = -0.248809919162060;
C10 = -0.248809919162060;
C12 = -0.248809919162060;
C14 = -0.248809919162060;
C16 = -0.248809919162060;
C18 = -0.248809919162060;


A0 = 1.0;
A1 = 2.5;
A2 = 1.75;
A3 = 0.5;
A4 = 2.75;
A5 = 1.75;
A6 = 0.25;
A7 = 2.75;
A8 = 1.0;
A9 = 3.0;


for(i = 1:length(outputVector))
    xPos = xVector(i);




    if(xPos <= 0.1)
        thisVal = -1*((L^2)/(pi^2*A0))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A0*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C00*xPos)/A0 + C01;
    elseif(xPos <= 0.2)
        thisVal = -1*((L^2)/(pi^2*A1))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A1*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C02*xPos)/A1 + C03;
    elseif(xPos <= 0.3)
        thisVal = -1*((L^2)/(pi^2*A2))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A2*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C04*xPos)/A2 + C05;
    elseif(xPos <= 0.4)
        thisVal = -1*((L^2)/(pi^2*A3))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A3*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C06*xPos)/A3 + C07;
    elseif(xPos <= 0.5)
        thisVal = -1*((L^2)/(pi^2*A4))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A4*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C08*xPos)/A4 + C09;
    elseif(xPos <= 0.6)
        thisVal = -1*((L^2)/(pi^2*A5))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A5*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C10*xPos)/A5 + C11;
    elseif(xPos <= 0.7)
        thisVal = -1*((L^2)/(pi^2*A6))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A6*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C12*xPos)/A6 + C13;
    elseif(xPos <= 0.8)
        thisVal = -1*((L^2)/(pi^2*A7))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A7*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C14*xPos)/A7 + C15;
    elseif(xPos <= 0.9)
        thisVal = -1*((L^2)/(pi^2*A8))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A8*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C16*xPos)/A8 + C17;
    else
        thisVal = -1*((L^2)/(pi^2*A9))*cos((pi * k * xPos)/(L)) ...
                  - ((5*L^2*(1-k^2))/(A9*4*pi^2*k^2))*sin(2*pi*k*xPos/L)
...
                  + (C18*xPos)/A9 + C19;
```

```
for(i = 1:length(outputVector))
    xPos = xVector(i);

    if(xPos <= 0.1)
        thisVal = 1*((16)/(pi*A0))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A0*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C00)/A0;
    elseif(xPos <= 0.2)
        thisVal = 1*((16)/(pi*A1))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A1*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C02)/A1;
    elseif(xPos <= 0.3)
        thisVal = 1*((16)/(pi*A2))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A2*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C04)/A2;
    elseif(xPos <= 0.4)
        thisVal = 1*((16)/(pi*A3))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A3*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C06)/A3;
    elseif(xPos <= 0.5)
        thisVal = 1*((16)/(pi*A4))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A4*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C08)/A4;
    elseif(xPos <= 0.6)
        thisVal = 1*((16)/(pi*A5))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A5*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C10)/A5;
    elseif(xPos <= 0.7)
        thisVal = 1*((16)/(pi*A6))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A6*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C12)/A6;
    elseif(xPos <= 0.8)
        thisVal = 1*((16)/(pi*A7))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A7*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C14)/A7;
    elseif(xPos <= 0.9)
        thisVal = 1*((16)/(pi*A8))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A8*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C16)/A8;
    else
        thisVal = 1*((16)/(pi*A9))*sin((pi * k * xPos)/(L))
...
                  - ((5*L*(1-
k^2))/(A9*2*pi*k))*cos(2*pi*k*xPos/L) ...
                  + (C18)/A9;
    end

    outputVector(i) = thisVal;
end

end

function [ R ] =
computeLoading2(currElement,myMesh,R,xInTermsOfZ,loadingInZ,or
der,exactSol)
%computeLoading computes and adds the R elements for a given
element
% It takes in the phi functions in zeta domain (as function of
x) and
% numerically integrates them with trapz

showextraGraphs = false;

x_i = myMesh(1 + (currElement-1)*order);
x_ip1 = myMesh(2 + (currElement-1)*order);

if(order > 1)
    x_ip2 = myMesh(3 + (currElement-1)*order);
else
    x_ip2 = 0;
end

if(order > 2)
    x_ip3 = myMesh(4 + (currElement-1)*order);
else
```

```
    end

    outputVector(i) = thisVal;
end

end

showextraGraphs = false;
error = 0;

if(order==1)
    for leftIndex=1:length(myMesh)-1
        rightIndex = leftIndex + 1;

        % get the x coordinate at this point in real world
        xLeft = myMesh(leftIndex);
        xRight = myMesh(rightIndex);

        % get the a's
        aLeft = A(leftIndex);
        aRight = A(rightIndex);

        % get the range over this, 100 is probably good
        xRange = linspace(xLeft,xRight,100);

        % get exactDudxFunction over this
        dudxOnRange = exactDudxFunction(xRange);

        % get the derivative of our approx
        % for order one is this easy
        slope = (aRight - aLeft) / (xRight - xLeft);

        dApproxOnRange = (xRange).*0 + slope;

        %calculate the integral thing. A1ofX here is just 0.2
        product = (dudxOnRange - dApproxOnRange).^2 .* A1inX(xRange);
        integral = trapz(xRange,product);
        error = integral + error;

    end
end


if(order==2)
    numElements = (length(A) - 1)/2;
    for(i=1:numElements)
        currElement = i;
        x_i = myMesh(1 + (currElement-1)*order);
        x_ip1 = myMesh(2 + (currElement-1)*order);
        x_ip2 = myMesh(3 + (currElement-1)*order);

        ypoints = [A(1 + (currElement-1)*order);...
                   A(2 + (currElement-1)*order);...
                   A(3 + (currElement-1)*order)];

        coefficients = polyfit([x_i;x_ip1;x_ip2;],ypoints,2);

        miniRange = [x_i:0.001:x_ip2];

        %take derivative...
        dApproxOnRange = coefficients(1) * miniRange * 2 +
coefficients(2);

        dExactOnRange = exactDudxFunction(miniRange);

        product = (dExactOnRange - dApproxOnRange).^2 * A1inX(miniRange);
        integral = trapz(miniRange,product);
        error = integral + error;

        if(showextraGraphs)
            figure(12)
            plot(miniRange,yvals);
            figure(13)
            hold on
            plot(miniRange(2:end),dApproxOnRange,'g');
            plot(miniRange(2:end),dExactOnRange,'r');
            pause(0.05)
        end
    end
end

if(order==3)
    numElements = (length(A) - 1)/3;
    for(i=1:numElements)
        currElement = i;
        x_i = myMesh(1 + (currElement-1)*order);
        x_ip1 = myMesh(2 + (currElement-1)*order);
        x_ip2 = myMesh(3 + (currElement-1)*order);
        x_ip3 = myMesh(4 + (currElement-1)*order);


        ypoints = [A(1 + (currElement-1)*order);...
                   A(2 + (currElement-1)*order);...
                  A(3 + (currElement-1)*order);...
                   A(4 + (currElement-1)*order)];
    x_ip3 = 0;
end


%construct the phi's in the x domain.
%for linear, this means subbing in (x - x_i) for z
%and having (xip1 - xi) for lengths... i think

if(order == 1)
    phi1 = @(x,xi,xip1) 1 + ((-1)/(xip1-xi)).*(x-xi);
    phi2 = @(x,xi,xip1) 1 + ((1)/(xip1-xi)).*(x-xip1);
elseif(order == 2)
    %use polyfit to make the equations, LOL!
    phi1points = [1; 0; 0];
    phi2points = [0; 1; 0];
    phi3points = [0; 0; 1];
    xPoints = [x_i;x_ip1;x_ip2];

    phi1coef = polyfit(xPoints,phi1points,2);
    phi2coef = polyfit(xPoints,phi2points,2);
    phi3coef = polyfit(xPoints,phi3points,2);

    phi1 = @(x,xi,xip1) 1 * (phi1coef(1)*x.^2 + phi1coef(2).*x
+ phi1coef(3));
    phi2 = @(x,xi,xip1) 1 * (phi2coef(1)*x.^2 + phi2coef(2).*x
+ phi2coef(3));
    phi3 = @(x,xi,xip1) 1 * (phi3coef(1)*x.^2 + phi3coef(2).*x
+ phi3coef(3));

elseif(order == 3) %TODO

    xPoints = [x_i;x_ip1;x_ip2;x_ip3];

    coeffs = getCubicCoeff(false,xPoints);

    phi1 = @(x,xi,xip1) coeffs(1,1)*x.^3 +  coeffs(1,2)*x.^2 +
...
                        coeffs(1,3)*x +     coeffs(1,4);

    phi2 = @(x,xi,xip1) coeffs(2,1)*x.^3 +  coeffs(2,2)*x.^2 +
...
                        coeffs(2,3)*x +     coeffs(2,4);

    phi3 = @(x,xi,xip1) coeffs(3,1)*x.^3 +  coeffs(3,2)*x.^2 +
...
                        coeffs(3,3)*x +     coeffs(3,4);

    phi4 = @(x,xi,xip1) coeffs(4,1)*x.^3 +  coeffs(4,2)*x.^2 +
...
                        coeffs(4,3)*x +     coeffs(4,4);

end
if(order == 1)
    xrange = linspace(x_i,x_ip1,100);
elseif(order == 2)
    xrange = linspace(x_i,x_ip2,200);
elseif(order == 3)
    xrange = linspace(x_i,x_ip3,300);
end


if(showextraGraphs)
    figure(4)

plot(linspace(0,1,200),exactSol(linspace(0,1,200)),'g','LineWi
dth',3);
    hold on

%plot(linspace(0,1,200),loadingInZ(linspace(0,1,200)),'k','Lin
eWidth',2);
    plot(xrange,phi1(xrange,x_i,x_ip1))
    plot(xrange,phi2(xrange,x_i,x_ip1))
        plot(xrange,phi3(xrange,x_i,x_ip1))

        plot(xrange,phi4(xrange,x_i,x_ip1),'r','LineWidth',2)

end



product1 = phi1(xrange,x_i,x_ip1) .* loadingInZ(xrange);
integral1 = trapz(xrange,product1);

product2 = phi2(xrange,x_i,x_ip1) .* loadingInZ(xrange);
integral2 = trapz(xrange,product2);

if(order == 3)
    product4 = phi4(xrange,x_i,x_ip1) .* loadingInZ(xrange);
    integral4 = trapz(xrange,product4);
end

if(order > 1)
    product3 = phi3(xrange,x_i,x_ip1) .* loadingInZ(xrange);
    integral3 = trapz(xrange,product3);
    if(showextraGraphs)
```

```
        coefficients = polyfit([x_i;x_ip1;x_ip2;x_ip3;],ypoints,3);

        miniRange = [x_i:0.001:x_ip3];

        %take derivative...
        dApproxOnRange = coefficients(1) * miniRange.^2 * 3 + ...
                         coefficients(2) * miniRange * 2 + ...
                         coefficients(3);

        dExactOnRange = exactDudxFunction(miniRange);

        product = (dExactOnRange - dApproxOnRange).^2 * A1inX(miniRange);
        integral = trapz(miniRange,product);
        error = integral + error;

        if(showextraGraphs)
            figure(12)
            plot(miniRange,ypoints);
            figure(13)
            hold on
            plot(miniRange,dApproxOnRange,'g');
            plot(miniRange,dExactOnRange,'r');
            pause(0.05)
        end
    end
end





error = sqrt(error) / eNorm;

end


function [ result ] = trickyMultiply2(mySparseK,a)
%TRICKYMULTIPLY Summary of this function goes here
%   Detailed explanation goes here

%lets artificially pad a and then delete those rows at the end
a = [0; a; 0];
result = zeros(length(a),1);

%now just loop over every element basically
numElements = length(a) - 1;

%dont do the first element or the last one
for i=2:numElements-1

    % all the local k's
    k11 = mySparseK(i,1);
    k22 = mySparseK(i,2);
    k12 = mySparseK(i,3);
    k21 = k12;


    result(i) = result(i) + k11 * a(i) + k21 * a(i+1);
    result(i+1) = result(i+1) + k12*a(i) + k22*a(i+1);
end

% do the first edge of the element
k22 = mySparseK(1,2);
result(2) = result(2) + k22 * a(2);
k11 = mySparseK(end,1);
result(end-1) = result(end-1) + k11 * a(end-1);

%delete pads on result, they should be zero anyways though
result(end) = [];
result(1) = [];


end

%Apcurrent = trickyMultiply(A,p_current)

    Apcurrent = trickyMultiply2(mySparseK,p_current);

    alpha = (r_current'*r_current)/(p_current'*Apcurrent);

    x = x + alpha*p_current;
    iterations = iterations + 1;

    r_next = r_current - alpha*Apcurrent;
    if(norm(r_next) < tolerance)
        %disp('Done!!!')
        %uncondition
        outputSolution = L * x;
        iterations
        break
    end

    beta = (r_next'*r_next)/(r_current'*r_current);
```

```
        figure(8)
        hold on
        plot(xrange,product1,'g')
        plot(xrange,product2,'b')
        plot(xrange,product3,'r')
        plot(xrange,product4,'k')
        figure(9)
        hold on
        plot(xrange,phi1(xrange,x_i,x_ip1),'g','LineWidth',3)
        plot(xrange,phi2(xrange,x_i,x_ip1),'b','LineWidth',3)
        plot(xrange,phi3(xrange,x_i,x_ip1),'r','LineWidth',3)
        plot(xrange,phi4(xrange,x_i,x_ip1),'k','LineWidth',3)
    %integral2 = 0.5*(integral1 + integral3);
    %pause
    end
end



if(order == 1)
    R(currElement) = R(currElement) + integral1;
    R(currElement+1) = R(currElement+1) + integral2;
elseif(order == 2)
    R(currElement*2 - 1) = R(currElement*2 - 1) + integral1;
    R(currElement*2 - 1 + 1) = R(currElement*2 - 1 + 1) +
integral2;
    R(currElement*2 - 1 + 2) = R(currElement*2 - 1 + 2) +
integral3;
elseif(order == 3)
    R(currElement*3 - 2) = R(currElement*3 - 2) + integral1;
    R(currElement*3 - 2 + 1) = R(currElement*3 - 2 + 1) +
integral2;
    R(currElement*3 - 2 + 2) = R(currElement*3 - 2 + 2) +
integral3;
    R(currElement*3 - 2 + 3) = R(currElement*3 - 2 + 3) +
integral4;
end


function [ outputSolution ] =
solveViaConjugateGradient(A,b,tolerance,mySparseK)
%SOLVEVIACONJUGATEGRADIENT Summary of this function goes here
%   Detailed explanation goes here

% K for the left hand side, R for the right hand side.
% we want to basically do K \ R to get the A's...

% guess the first a?

clear A

%first, precondition the matrix!
%construct L...

%sizeA = max(size(A))
sizeA = max(length(mySparseK(:,1)))-1;
L = zeros(sizeA);

% for i=1:sizeA
%     L(i,i) = 1 / sqrt(A(i,i));
% end
% L
% L = zeros(sizeA);

L(1,1) = 1 / sqrt(mySparseK(1,2) + mySparseK(2,1));
for i=2:length(mySparseK(:,1))-1
    L(i,i) = 1 / sqrt(mySparseK(i,2) + mySparseK(i+1,1));
end

% A
% L
% mySparseK
% pause

%precondition the matrix
%A = L *  A * L;
%do the same for mySparseK -> this is actually complicated

%the format is that K11 gets A^2, K12 and K21 get AB,
%and K22 gets B^2. but need that on both
%get first
mySparseK(1,2) = mySparseK(1,2) * L(1,1) * L(1,1);
for i=1:sizeA
    currentL = L(i,i);
    if(i<=sizeA-1)
        nextL = L(i+1,i+1);
    else
        nextL = 1;
    end
    %modify k11, but lead by 1 element
    mySparseK(i+1,1) = mySparseK(i+1,1) * currentL * currentL;
    %modify K12, both
    mySparseK(i+1,3) = mySparseK(i+1,3) * currentL * nextL;
    %finally, modify tail end
    mySparseK(i+1,2) = mySparseK(i+1,2) * nextL * nextL;
```

```
    p_next = r_next + beta*p_current;

    p_current = p_next;
    r_current = r_next;

%     disp('another iteration')
%     x
%     disp('real')
%     inv(A) * b
%
%     pause

end



end
```

```
    %mySparseK(i,2) + mySparseK(i+1,1)
    %pause
end

% mySparseK
% A
% pause
b =  L * b;

x_1 = zeros(length(b),1) + 1;
x = x_1;
% x = [4; 2];
% x = rand(2,1);


p_0 = b - trickyMultiply2(mySparseK,x);
r_0 = p_0;

p_current = p_0;
r_current = r_0;
iterations = 1;


while(1)
    %A * p_current
```