Ilia Kheirkhah

ENM 5020

4/28/2023

Homework 4

Galerkin Finite Element Methods for Solving Boundary Value Problem

Contents

Introduction

Up to now, we have worked with Finite Difference Method as our primary form of discretization. This allows us to divide the grid into many evenly spaced grid points. We then used the properties of these grid points to estimate the solution. We estimated derivatives by using the values of adjacent nodes and had one equation for each node. To solve these systems, we generally got nonlinear systems of equations which we then had to solve. However, if the differential equation we were studying was linear, we would end up with a linear system of equations to solve.

Although finite differences is a great method for its simplicity, it has many limitations. First, setting flux boundary conditions costs us accuracy. Since we cannot use a center difference to estimate the first derivative, we lose an entire order of accuracy if we have flux boundary conditions. In addition, finite differences do not deal with irregular geometries well. Since the nodes need to be placed in a regular grid pattern and generally have even spacing, fitting curved boundaries becomes difficult. Although some problems can be dealt with by changing our basis, there will generally be geometries that are not well suited for finite differences.

This is where Finite element methods fill the gap for complex geometries. With finite elements, we tesselate our grid with elements that are lines in 1D, squares or triangles in 2D, and cubes or tetrahedrons in 3D. These elements could all be different sizes and the method would still work. Once we have done this, pick basis functions which we use to approximate the solution to our differential equation. Once we have all this information, we can apply the Finite Element Method to find an approximate solution.

In this paper, we will solve a 1D version of the heat equation using Finite Element Methods and compare the results in terms of graph behavior and method accuracy. To do this, we will first discuss the theory behind FEM. Then we will apply this theory to get the necessary different graphs representing the system. Finally, we will reflect on the method and the results we have seen.

Methods

In this section we will explore the fundamental theory behind Finite Element Method. We will start by understanding the idea behind our Galerkin Finite Elements. We will then move on to see how this influences the form of our Differential Equation and the Weak form it takes when we apply FEM. Finally, we will demonstrate how Quadrature can be used to find the necessary integrals to populate our matrix-vector equation.

Finite Element Method and The Weak Formulation

The differential system we would like to analyze is the classic Heat Equation

$$\frac{d^2T}{dx^2} + \alpha = 0, T(0) = T(1) = 0 \quad \textbf{(1)}$$

If we want to approach solving this problem with a finite element method, we first need to make an approximation and say:

$$T \sim \sum_{i=1}^{N+1} T_i \phi^i(x) \quad \textbf{(2)}$$

Here, we can notice that ϕ^i are our set of basis vectors. Each node will have its own basis vector which are all orthogonal. This means that ϕ^i has a value of 1 at node i and zero at every other node. In between nodes, the function can take any values. If we were to divide our domain up into N elements, we would have $N + 1$ node points and therefore $N + 1$ basis functions and coefficients, T_i. Equation 2 is the basis of

developing the weak form of our ODE. We will now apply the Method of Mean Weighted Residuals. To do these, we will first call our Homogenous differential equation $F(x)$. From here, we can then define weight functions $w_i(x)$. Our method of Mean Weighted Residuals then tells us that we can find the solution to our problem by saying:

$$R_i = \int_0^1 F(x) w_i(x) dx = 0 \tag{3}$$

For our Galerkin Mean Weighted Residuals, our weight function is the same as our basis functions. This means $w_i = \phi^i$. From here, we can plug in everything from our differential equation to the basis functions to get:

$$R_i = \int_0^1 \left(\frac{d^2T}{dx^2} + \alpha \right) \phi^i dx = 0 \tag{4}$$

If we distribute and rearrange, we can then see that this is equivalent to saying:

$$\int_0^1 \phi^i \frac{d^2T}{dx^2} dx = -\int_0^1 \phi^i \alpha dx \tag{5}$$

The right side is well defined and easy to work with. However, the second derivative on the left causes problems for our method. As such, we will want to apply integration by parts to fix this. Recall by chain rule:

$$\frac{d}{dx}\left(\phi^i \frac{dT}{dx} \right) = \phi^i \frac{d^2T}{dx^2} + \frac{d\phi^i}{dx}\frac{dT}{dx} \tag{6}$$

Rearranging this, we can say:

$$\phi^i \frac{d^2T}{dx^2} = \frac{d}{dx}\left(\phi^i \frac{dT}{dx} \right) - \frac{d\phi^i}{dx}\frac{dT}{dx} \tag{7}$$

If we plug this into our integral, we can simplify this problem to:

$$\phi^i \frac{dT}{dx}\Big|_0^1 - \int_0^1 \frac{d\phi^i}{dx}\frac{dT}{dx} dx = -\int_0^1 \phi^i \alpha dx \tag{8}$$

Now is a good time to take a moment to discuss numbering for our problem. Since we are solving a 1D equation, our domain is a line. As such, regardless of the spacing of our nodes we will always number from 1 to $N+1$ increasing from left to right. This means that our boundary nodes are describe T_1 and T_{N+1}. Since these are the locations we want to apply our boundary conditions at, we get two free equations for our system. Those are:

$$T_1 = 0 \tag{9}$$

$$T_{N+1} = 0 \tag{10}$$

These two equations replace R_1 and R_{N+1}. This means that instead of going through i from 1 to $N+1$, we can now go from 2 to N. This is important since, as we discussed before, all basis functions only have nonzero values at their own node. This means that, at 0 and 1 which correspond to nodes 1 and $N+1$, ϕ^i will always be 0 in the set of basis functions we are looking at. This means that we can simply ignore the first term of the above equation and say:

$$\int_0^1 \frac{d\phi^i}{dx}\frac{dT}{dx} dx = \int_0^1 \phi^i \alpha dx \tag{11}$$

From here we can input Equation 2 into Equation 11 to arrive at:

$$\int_0^1 \frac{d\phi^i}{dx}\frac{d}{dx}\left(\sum_{j=1}^{N+1} T_j\phi^j\right)dx = \int_0^1 \phi^i \alpha dx \tag{12}$$

Notice here that there was a minor switch of indexing to j in the sum for clarity. Since the summation is a linear operation and T_j is a constant, we can say this is equivalent to

$$\sum_{j=1}^{N+1} T_j \int_0^1 \frac{d\phi^i}{dx}\frac{d\phi^j}{dx}dx = \int_0^1 \phi^i \alpha dx \tag{13}$$

This is the final form of our ODE in weak form. As can be seen, this is a linear system of equations which we can solve for all T_j. From there, we have a solution to our steady heat transfer problem.

The final step to having a complete description of our problem is to determine our basis functions. Since we are using basis functions, our basis functions can be described using these qualities. First, every basis function ϕ^i is 1 at node i and zero elsewhere. Next, every basis function increases linearly from 0 to 1 from node $i-1$ to i and then decreases from 1 to 0 from node i to node $i+1$. This gives them a triangular shape centered at a given node. Together, these are a good description of a linear basis function.

Within one element, we will see two basis functions. If we consider element i, we will see the decreasing portion of basis i and the increasing portion of basis $i+1$ while all other basis vectors are zero. Within this element, we can then define two local basis functions:

$$\phi^{(1)}(\xi) = \frac{1}{2}(1-\xi) \tag{14}$$

$$\phi^{(2)}(\xi) = \frac{1}{2}(1+\xi) \tag{15}$$

The reason we use the variable ξ here is that we have defined these local basis functions on a unit element. The unit element always goes from -1 to 1 in the ξ axis. This gives it a fixed size. By mapping our real elements to the unit element, we can use these local basis functions as an easy way to get function values without having to derive new equations at each element. We will further look at this mapping in the next section about Quadrature. For now, we will return to our method of mean weighted residuals.

Now that we have a definition for our basis functions, we could theoretically solve this problem completely. We have 2 boundary equations in addition to $N-1$ residual equations for the total of $N+1$ equations we need to solve to for $N+1$ unknown T_i values. However, the difficult part of this process comes in evaluating the integrals. Since the basis functions are defined piecewise and are not uniform, calculating the integrals in the x domain directly is difficult. As such, it would be ideal to have a coordinate transform we could perform to make the integral easier. However, a coordinate transform on the entire line would be difficult to find and work with. As such, the clever step we take is to break our integral up into a sum of integrals along elements. This means that we evaluate the integral at each element then, since our elements fill our entire domain, adding the results together give us the proper answer. Doing this by math, we can say:

$$\sum_{l=1}^{l+1}\sum_{j=1}^{N+1} T_j \int_{x_l}^{x_{l+1}} \frac{d\phi^i}{dx}\frac{d\phi^j}{dx}dx = \sum_{l=1}^{N+1}\int_{x_l}^{x_{l+1}} \phi^i \alpha dx \tag{16}$$

This is the final form of our mean weighted residuals for this problem. We will then show how we can use a mapping to take this integral on a real element and turn it into an integral on the unit element. This will allow us to use the local basis functions we defined to evaluate the integrals using Gaussian Quadrature.

Gaussian Quadrature

Consider we want to solve the general integral:

$$I = \int_{-1}^{1} f(x)dx \tag{17}$$

One of the first methods people learn to estimate an integral such as this one is the trapezoid rule. Using this rule to estimate the integral as follows:

$$I = \big(1-(-1)\big)\cdot\frac{1}{2}\big(f(1)+f(-1)\big) = f(-1)+f(1) \tag{18}$$

Although this can give decent rules, we can use Quadrature to get better results. Instead of being locked into evaluating our function at the endpoints as we do in the trapezoid rule, Quadrature allows us to sample other points in our domain to get a better idea of the behavior of the function. In addition, Gaussian Quadrature has specific weights to use when calculating integrals. If we want to do Quadrature, we can pick the number of points we want to sample. Each additional point adds two orders of accuracy to the method. We can then say:

$$I \approx \sum_{i=1}^{N} w_i f(x_i) \tag{19}$$

Where w_i and x_i are all defined by the properties of quadrature and N is the number of sample points. The number of points essentially determines the order of polynomial we are fitting to determine the weights. In a two point quadrature, we attempt to fit a line of the form:

$$I \approx C_1 f(x_1) + C_2 f(x_2) \tag{20}$$

Where both C and x are unknowns. This gives us four unknowns. Comparing this to other fitting methods, this is like fitting a cubic polynomial which tells us:

$$f(x) \approx a + bx + cx^2 + dx^3 \tag{21}$$

$$I \approx \int_{-1}^{1} a + bx + cx^2 + dx^3 dx \tag{22}$$

Equating these two forms of I and doing algebra and using the fact that coefficients should be equal for the same order terms, we can arrive at the weights C_1, C_2 and the sampling points x_1, x_2 as we have four unknowns and four equations. Although not derived here, these values are often tabulated. For this assignment, we will use three-point quadrature which has weights and sampling points as seen in Table 1.

Index	x_i	w_i
1	$-\sqrt{3/5}$	5/9
2	0	8/9
3	$\sqrt{3/5}$	6/9

Table 1 – Guass Quadrature Properties

However, a major constraint for quadrature is that tabulated values are only for the domain $[-1,1]$ when integrating. However, we can see from equation 16 that our integrals are not in this domain. As such, we need a mapping to take our current element to the unit element which is on the necessary domain. This requires the mapping to take x_l to -1 and x_{l+1} to 1 for a given l. With some convincing, one can see that an appropriate mapping would be:

$$x = \sum_{i=1}^{2} x_{(i)}\phi^{(i)}(\xi) \tag{23}$$

Notice here that we are using local numberings here. This means that $x_l = x_{(1)}$ and $x_{l+1} = x_{(2)}$. This mapping allows us to transform our integrals into the necessary form to apply quadrature. We want to change all our derivatives and bounds from the x domain to the ξ domain of the unit element. To do this, let us take the derivative of Equation 23 with respect to ξ. This allows us to say:

$$\frac{dx}{d\xi} = \sum_{i=1}^{2} x_{(i)} \frac{d\phi^{(i)}}{d\xi} \tag{24}$$

Since our local basis functions were defined on the unit element, finding their derivative is easy when looking at Equations 14 and 15. Now, if we apply chain rule we can see the following truths:

$$\frac{d\phi^i}{dx} = \frac{d\phi^i}{d\xi}\frac{d\xi}{dx} = \frac{d\phi^i}{d\xi}\left(\frac{dx}{d\xi}\right)^{-1} \tag{25}$$

$$dx = \frac{dx}{d\xi} d\xi \tag{26}$$

Notice that equation 25 also holds for ϕ^j as we have made no assumptions about i or j when applying the chain rule. Finally, notice that when going from the real domain to the unit element, we go to the respective local basis function. This means our integral becomes:

$$\sum_{l=1}^{l+1}\sum_{j=1}^{N+1} T_j \int_{-1}^{1} \frac{d\phi^{(i)}}{d\xi}\frac{d\phi^{(j)}}{d\xi}\left(\frac{dx}{d\xi}\right)^{-1} d\xi = \sum_{l=1}^{N+1} \int_{-1}^{1} \phi^i \alpha \frac{dx}{d\xi} d\xi \tag{27}$$

Notice that we are still iterating through l although there is no explicit inclusion of l in the equation. This is because the index is implied through the mapping when determining $\frac{dx}{d\xi}$. Finally, we can simply apply quadrature to find the values of these integrals and populate our matrix-vector problem. Let us now apply this theory and look at the results we find.

Results

First, let us compare the influence of the heat generation term and the number of elements we use.

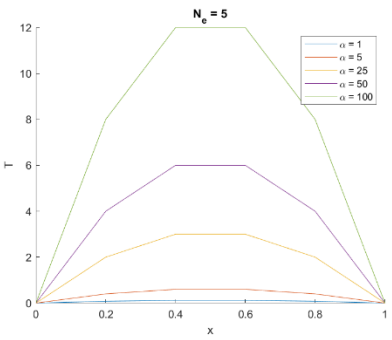

Figure 1a – 5 Elements, Vary Heat Generation

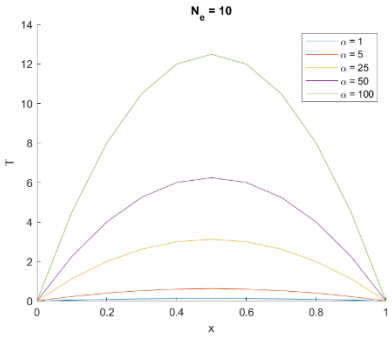

Figure 1b – 10 Elements, Vary Heat Generation

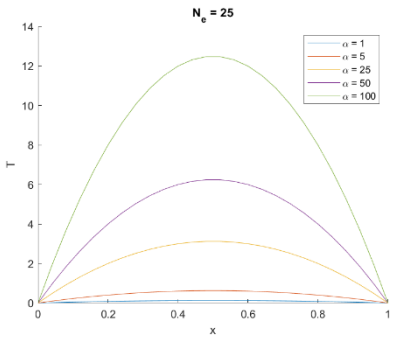

Figure 1c – 25 Elements, Vary Heat Generation

As can be seen in Figures 1a and 1b, the graphs with less elements are visibly inaccurate. We know from analysis of our differential equation that we expect to see a downward facing parabola. However, when we have 5 or 10 elements we can clearly see the linear interpolation of our basis functions. When we get to 25 elements, the jaggedness is less noticeable. At higher element numbers, starting above 25, the graphs look almost perfectly smooth.

For example, if we compare figures 1d and 1e, the visual smoothness difference is unnoticeable. Although this is not a very rigorous method of noting the quality of a method, it is a good start to getting an intuition of the result. Graphs that are dominated by the interpolation do not necessarily give a good insight to the physical system.

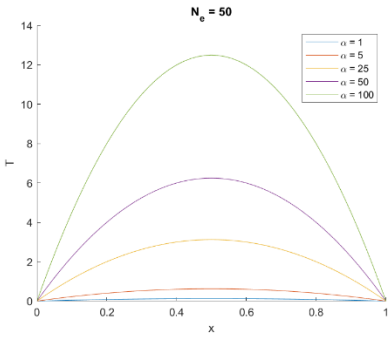

Figure 1d - 50 Elements, Vary Heat Generation

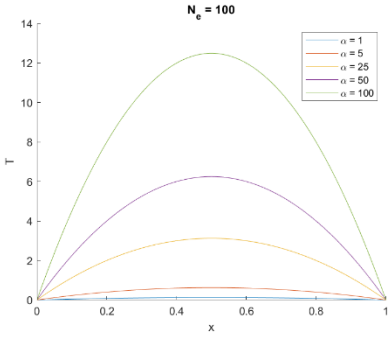

Figure 1e -100 Elements, Vary Heat Generation

Now that we have commented on the effect of elements on the look of the graphs, let us look at the influence of α on the graphs. Looking at figure 1a-e, we can see that α serves to simply scale the graph. The smaller values of α simply reach a lower peak than the higher values.

Now, let us take a more rigorous look at the error in this method. If we first take a more generous look, we can notice from Figure 2 that it seems our finite elements are giving values that are exactly equal to each other and to the theory curve at the node points.

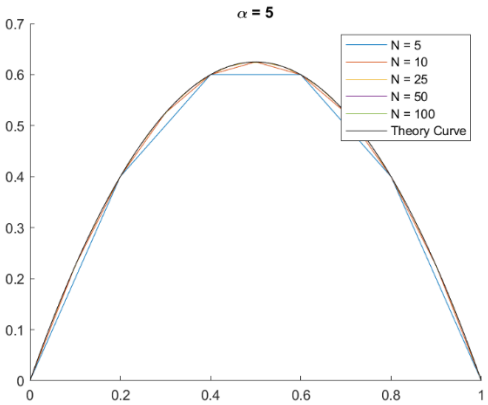

Figure 2 – Comparing Element Variation with Theory Curve

If we look at Figure 3, we can actually see the visuals are not lying to us. To no computer accuracy, our results are exactly the same. This means that we can get arbitrary close to the true result by increasing the number of elements we use. Although I am not perfectly confident as to why this is occurring, I believe it has to do with the simple, parabolic nature of the equation we are solving relative to the linear elements we are using. Intuition tells us if were to use quadratic elements, we could get a perfect fit to the graph at all node points and all intermediate points. Therefore, it makes intuitive sense as to why linear elements would result in us losing the intermediate accuracy but keeping the node point accuracy.

Although the intuition-based argument may work for some people, it is not a rigorous argument that makes me too confident. However, due to time constraints I was unable to think through and fully understand a good argument as to why we have zero error at the node points. This would be a good place of further investigation. However, we have thoroughly explored 1D Galerkin Finite Element Methods.

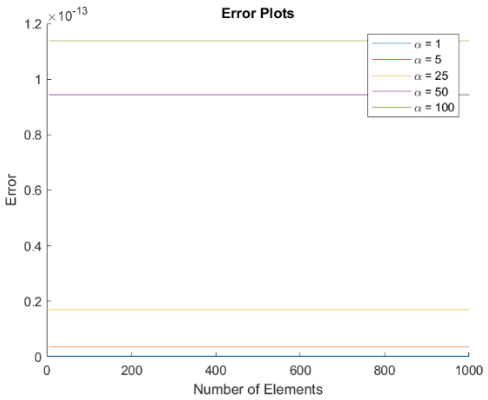

Figure 3 – Error Graphs based on Theory Curve and L2 Norms

Conclusion

In this assignment, we have explored the theory behind and application of Galerkin Finite Element Methods. First we took a detailed look at the theory behind this method starting with the Method of Mean Weighted Residuals including the Finite Element approximation. We then showed the importance of Galerkin methods in relation to our basis functions. Once we used MWR to find the Weak Formulation of our Differential equation, we showed that it was a linear system of equations that can be solved as a normal $Ax = b$ system. However, to set up our matrix- vector system, we had to evaluate integrals using quadrature. To enable us to do this simply, we investigated a mapping from any general element to a unit element which was in the Gaussian domain $[-1,1]$ which is necessary when applying previous finding of quadrature approximations. Finally, we compared the results both visually and in terms of error where we found interesting results in terms of finding our problem solution resulted in zero error.

We have done a fairly elementary exploration of Finite Element methods. The most obvious point of further investigation would be expanding our code to 2 or 3 dimensions. This would allow us to explore different types of elements as well such as square, triangular, or potentially other less standard element shapes. In addition to this, we could explore higher order elements such as quadratic or even cubic elements. Although cubic elements are rare, quadratic elements are not uncommon and give good results in many applications.

Another path to take is into understanding why this problem is providing zero error. Although the quadratic-linear relationship may be a good start, having a rigorous understanding of this phenomenon would be necessary for a deeper understanding of Finite Elements, its limitations, and its powers.

Finally, I think the most interesting detail to look at is the basis and weight functions and their relationships. In this assignment, we used a Galerkin FEM which means our weight functions are the same as our basis functions. However, this is not generally required and one could choose any basis function to go with any weight function. However, finding pairs that have meaning, provide good answers, and are easy to work with is an art that requires exploring.

Beyond all this, we have a good start into looking at the powerful tool known as Finite Element Methods.

Appendix

Main

```
clear; clc; close all;
% Compare Parameters
alphaSet = [1,5,25,50,100];
Nset = [5,10,25,50,100];


for i=1:length(Nset)
    figure;
    hold on;
    N = Nset(i);
    xset = linspace(0,1,N+1);
    errors = zeros(N,1);
    title("N_e = "+num2str(N))
    for j = 1:length(alphaSet)
        alpha = alphaSet(j);
        [Ti, plotY] = heatFEM(N,alpha,xset);
        plot(xset,plotY,'-','DisplayName',"\alpha = "+num2str(alpha))
    end
    legend;
```

```
    xlabel("x");
    ylabel("T");
    exportgraphics(gcf,"N = "+num2str(N)+" Vary Alpha.png")
end

figure;
hold on;
alpha = 5;

title("\alpha = "+num2str(alpha))
for j = 1:length(Nset)
    N = Nset(j);
    xset = linspace(0,1,N+1);
    errors = zeros(N,1);
    [Ti, plotY] = heatFEM(N,alpha,xset);
    plot(xset,plotY,'-','DisplayName',"N = "+num2str(N))
end
legend;
exportgraphics(gcf,"alpha = 5, Vary N for Error.png");

% Error
figure;
hold on;
alpha = 5;

title("\alpha = "+num2str(alpha))
for j = 1:length(Nset)
    N = Nset(j);
    xset = linspace(0,1,N+1);
    errors = zeros(N,1);
    [Ti, plotY] = heatFEM(N,alpha,xset);
    plot(xset,plotY,'-','DisplayName',"N = "+num2str(N))
end
legend;
plotset = linspace(0,1,1000);
theoryY = -alpha*plotset.^2/2+alpha*plotset/2;
plot(plotset,theoryY,'k-','DisplayName',"Theory Curve");
exportgraphics(gcf,"alpha = 5, Vary N for Error with theory.png");

% Error
alphaSet = [1,5,25,50,100];
Nset = 5:5:1000;
errors = zeros(length(alphaSet),length(Nset));
figure;
hold on;
for i = 1:length(alphaSet)
    alpha = alphaSet(i);
    theoryFunc = @(x) (-alpha*x.^2./2+alpha.*x./2);
    for j = 1:length(Nset)
        N = Nset(i);
        xset = linspace(0,1,N+1);
        [Ti,plotY] = heatFEM(N,alpha,xset);
        errVec = plotY-theoryFunc(xset);
        errors(i,j) = sqrt(sum(errVec.^2));
    end
    plot(Nset,errors(i,:),'-',"DisplayName","\alpha = "+num2str(alpha));
end
title("Error Plots");
xlabel("Number of Elements");
ylabel("Error");
legend;
exportgraphics(gcf,"Error.png")
```

HeatFEM – The Solver Function

```
function [Ti,plotY] = heatFEM(Nel,alpha,plotset)
% Inputs
N = Nel;

% Other Parameters
Nnodes = N+1; %Number of Points

xset = linspace(0,1,Nnodes); % Evenly Spaced Points

gaussPoints = [-sqrt(3/5) 0 sqrt(3/5)];
gaussWeights = [5/9 8/9 5/9];

% Allocate Space
A = zeros(Nnodes);
b = zeros(Nnodes,1);

% Center Elements
for k = 1:N % Loop Through Elements
    Aloc = zeros(2); % Local 2x2 A matrix
    bloc = zeros(2,1); % Local b vector
    x1 = xset(k); % Left Point
    x2 = xset(k+1); % Right Point

    for i = 1:2 % Loop through First Local Basis Functions
        for j = 1:2 % Loop Through Second Local Basis Function
            % Populate Variables
            [phii, dphiidchi] = basisLocal(i,gaussPoints); % dphii/dchi
            [~, dphijdchi] = basisLocal(j,gaussPoints); % dphij/dchi

            [~, dxdc] = mapping(x1,x2,gaussPoints); % Mapping Jacobian

            % Generate Local A matrix
            for g = 1:3 %sum
                Aloc(i,j) = Aloc(i,j) + gaussWeights(g)*dphiidchi(g)...
                    *dphijdchi(g)/dxdc(g);
            end

            % Generate Local b vector
            bloc(i) = sum(gaussWeights.*phii.*alpha.*dxdc);
        end
    end
    % Place local information into global matrix and vector
    A(k:k+1,k:k+1) =  A(k:k+1,k:k+1) + Aloc;
    b(k:k+1) = b(k:k+1) + bloc;
end

% Set Boundary Conditions
% Override first row
A(1,:) = 0;
A(1,1) = 1;
% Override last row
A(Nnodes,:) = 0;
A(Nnodes,Nnodes) = 1;

% Fix Vector
b(1) = 0;
b(Nnodes) = 0;

% Find Solution
Ti = A\b;
```

```
% Now that solution is found, apply approximation to say u = sum(Ti*phi(i))
% This last section is just to make plotting and processing easier
plotY = zeros(size(plotset));
for i = 1:length(plotset)
    xCurr = plotset(i);
    val = 0;

    for j = 1:length(xset)
        Tcurr = Ti(j);
        val = val + Tcurr*basisGlobal(j,xset,xCurr);
    end
    plotY(i) = val;
end

end
```

Global Basis Function

```
% This function is returns the value of the global basis function at a
% given point
% The i variable selects which basis function we want. We then go through
% xset and determine the bounds on the elemnt we are in to determine the
% mapping from the true element to the unit element. We then map our x
% location to the proper location in the unit element and find the value.
function [val,x0,x1,x2,chi,type]= basisGlobal(i,xset,x)
    if(i <= length(xset) && i >= 1) % Determine Bounds
        x1 = xset(i);
        if(i ~= 1)
            x0 = xset(i-1);
        else
            x0 = xset(1);
        end
        if(i~=length(xset))
            x2 = xset(i+1);
        else
            x2 = xset(end);
        end
    else % Out of bound error catch
        val = -1;
        x0 = -1;
        x1 = -1;
        x2 = -1;
        chi = -1;
        type =-1;
        return;
    end

    if(x <= x0 || x >= x2) % Outside of this range returns 0
        val =0;
        chi = -2;
        type = -2;
    else
        if(x < x1) % Positive slope case
            chi = x - ((x0+x1)/2);
            chi = chi/((x1-x0)/2);
            chi = round(chi*100000)/100000; % Fix floating point jazz
            val = basisLocal(2,chi);
            type = 2;
        else % Negative Slope Case
            chi = x - ((x1+x2)/2);
            chi = chi/((x2-x1)/2);
            chi = round(chi*100000)/100000;
            val = basisLocal(1,chi);
```

```
            type = 1;
        end
    end
end
```

Local Basis Function

```
% This function evaluates the local basis function on the unit element
% The type is the local type of the linear basis
% 1 represents the downwards sloped basis
% 2 represents the upwards sloped basis
% The value of chi determines the value. If it is from -1 to 1, the output
% is determined using the type of basis function we select. If it is out of
% that range, we return 0.
function [val, dval] = basisLocal(type,chi)
    val = zeros(size(chi));
    dval = zeros(size(chi));
    for i = 1:length(chi)
        if(type == 1)
            val(i) = .5*(1-chi(i));
            dval(i) = -.5;
        else
            val(i) = .5*(1+chi(i));
            dval(i) = .5;
        end
        % disp(abs(chi(i)))
        if (abs(chi(i)) > 1)
            val(i) = 0;
            dval(i) = 0;
        end
    end
end
```

Mapping Function

```
% This is a simple mapping from the unit domain to the element domain.
function [x, dxdc] = mapping(x1,x2,chi)

    [val1, dval1] = basisLocal(1,chi);
    [val2,dval2] = basisLocal(2,chi);
    x = x1*val1 + x2*val2;
    dxdc = x1*dval1 + x2*dval2;
end
```