Peter Cottle *SID 19264824*

ME 280A – 12/08/2011

Homework Assignment #6

Introduction:

For this homework project, the class finally steps away from the comfortable one dimensional world and begins a journey into the two dimensional world. Here we will study the finite element formulation for a 2D thermal diffusion problem; in this scenario, temperature is simply a scalar value and thus there is only one unknown per node.

This journey will involve revisiting the basics of the finite element method (except in 2D) and also the exploration of new concepts. We will use the penalty method to impose boundary conditions, look at variations in thermal properties through the material, and perform adaptive meshing over areas with material discontinuities.

Objectives:

In this homework we have several challenges. The first is to analytically derive the weak form for the thermal diffusion equation in two dimensions, taking care to explicitly write out all the necessary matrices and vectors needed for the final formulation. The second is solve the test case analytically; if done correctly, we will obtain a solution that is independent of the radial position.

The next series of objectives involves the actual implementation of this finite element code. We will write a completely new program from scratch to implement this formulation in 2D. Other sections will go over this in further detail, but this essentially involves a lot of method writing for the phi basis functions, the isoparametric mapping, the arch mesher, and the right-hand-side calculation. After all of this code is complete, we will compare the numerical solution to the analytic one. We will also compare different boundary condition implementations and look at adaptive meshing.

Procedure / Implementation:

This homework involved a fair amount of coding, so an outline of each method (as well as the overall loop) will be detailed below.

Methods:

- ArchMesher
    - Takes in meshing parameters (theta step size, radius step size) and outputs the connectivity map and the global node coordinates.
    - Pseudo Code:

```
Number of points in R direction = r_range / r_step + 1
Number of points in Theta direction = theta_range / theta_step + 1
Number of points = rPoints * thetaPoints;

Initialize the global coordinate matrix

For uniform meshing, do:
      Step through in theta, starting at 0 and incrementing by theta step
towards pi
            Step through in r, starting at r_being and ending at r_end
                  Calculate x by taking r * cos(theta)
                  Y = r * sin(theta)
                  This x and y correspond to the coordinates of this node

For adaptive meshing, do:
      Calculate the small theta step and the large theta step depending on
      the number of elements desired on either side
      Loop through r and theta as before, changing theta step depending on
      which part of the arch is being meshed.

For the connectivity, do:
      Calculate the number of theta slices (one less than theta points) and
      the number of r slices (one less than rPoints)

      Step through each thetaSlice
            Step through each rSlice
                  The lower left node number is 1 + (sliceCounter-1) *
                  (numElementsPerSlice+1);
                  LR is just LL + 1
                  UL is just the LL + numElementsPerSlice + 1
                  UR is just the UL + 1
                  Store these numbers in the connectivity map at this element
      row, plot if necessary

Return the connectivity map and global node coordinates
```

- AssembleK
  - Takes in a local 4x4 K matrix and assembles into the global matrix
  - This method was implemented in earlier homeworks 1-5, consult those for implementation
- assembleLoading
  - Similar to assembleK, but for a 4x1 vector for the right-hand-side. Again, see earlier homeworks for this implementation.
- elementBoundary
  - Take in a given boundary function and calculate the corresponding 4x1 vector for this boundary function over the given element
  - Pseudo Code:

```
First, return a zero vector if this element does not lie on the boundary
where the boundary function is specified.

Next, find the line deformation (Cauchy deformation tensor) for this line
integral. It is either the sqrt( rT * F * F-t * r) or just the ratio of the
lengths

Get the phi functions from the getPhis() function.

Get the isoparametric mapping from the getX1X2 function for this element.

Compile the total function as a 4x1 vector, each entry corresponding to a phi
function * the boundaryFunction at this z1 and z2 value (after isoparametric
mapping) * the ratio of lengths.

Now, do 1D integration with the Gaussian1D function. Important: fix z1 or z2
(usually z2 in our case) to be -1 or +1 to obtain the line integral at the
correct edge of the element.

Return this result.
```

- elementK
    - Take in a current element, the connectivity map, and the global coordinates. Returns the 4x4 local matrix of the stiffness calculation
    - Pseudo Code:

```
Check if this element overlaps the thermal hole by checking the thermal
conductivity at each nod and the midpoint of each edge. If any of these
points are inside the thermal "hole," turn on Gaussian integration with 5
points instead of 2.

Get the f inverse function handle from the fInverse function.

Get the Jacobian function handle from the Jacobian function.

Get the D matrix from the D function.

Get the isoparametric mapping from the getX1X2 function.

Finally, compile the complete function by multiplying (fInv * D)' *
conducitivity * fInv * D * Jacobian.

Integrate this in 2D to get the 4x4
```

- elementLoading
    - Perform the same calculation for the elementBoundary (aka the 4x1 vector of phi functions), but use the loading function instead. Integrate this in 2D.
- Gaussian1D integrate

    - To integrate in 1d on a 2d function, hold either z1 or z2 constant and use the weights and Gaussian points on the other free variable.
- Gaussian2D integrate
    - Instead of having one loop, have two nested for loops that loop over each Gaussian point for each variable. Multiply all of these weights and points together to get a final answer.
- GetD function
    - Return the D matrix, aka the partial derivatives of the phi functions.
        - This is simply a 2x4 matrix corresponding to dPhi_1_dz1, dPhi_2_dz1, etc on the first row and the same partials but with respect to z2 on the second row.
- GetF inverse function
    - Use the jacobian function to obtain the jacobian function and the components of the F matrix. Return the inverse of the F matrix by swapping the diagonals of the 2x2, flipping the sign on the off diagonals, and dividing by the determinant (aka the jacobian).
- Get Jacobian
    - Return the Jacobian function and the F deformation mapping for a given element:
    - Pseudo Code:

```
First, get all the phi functions and phi partial functions from the getPhis
function.

Get the global nodes for this element with the connectivity map.

Compile the x1 and x2 functions with the isoparametric mapping. Aka x1 is a
function of xCoordinateAt1 * phi1 + xCoordinateAt2 * phi2, etc

Compile the deformation matrix F by finding the partial derivatives of this
isoparametric mapping. Aka, dx1/dz1 is phiPartial{1,1} * xCoordinateAt1 +
phiPartial{2,1} * xCoordinateAt2, etc.

Return these function handles for use in other functions.
```

- Get Phis
    - Simply return function handles for all the phi functions and phi partial derivatives in a cell matrix for easy access. These were analytically derived in class and thus I will not go over them here.
- Get x1x2
    - Return the isoparametric mapping functions. These were computed in the getJacobian function and will not be detailed here.
- doPenaltyTerms
    - When using the penalty term to impose the boundary conditions, a few simple operations were performed on the matrix. These are detailed in the pseudo code below:

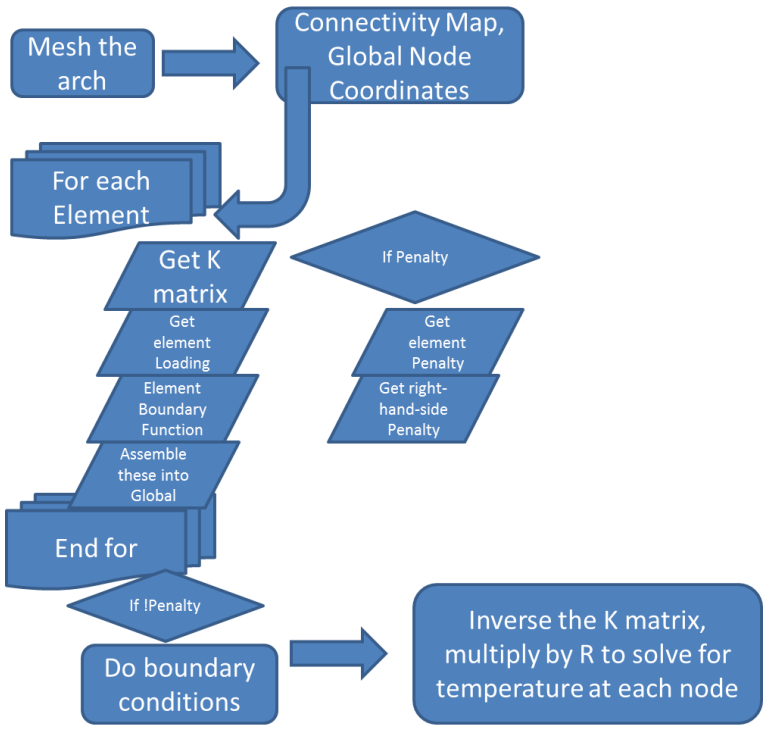
```
For the stiffness matrix, first determine the correct Pstar to use.
To determine this, find the max(kStiffness) and multiply this by the P
scaling factor, usually a number that ranges from 100 to 1000.
```

After obtaining Pstar, modify the stiffness matrix by adding Pstar to each node that is at a fixed boundary temperature. This is performed in a simple loop.

For the right hand side, modify each row that corresponds to each node with a fixed boundary temperature. This is performed by adding the Pstar * T_0 term to the R vector at the indicated node.

Now, skip all the row elimination and simply solve. Depending on the magnitude of Pstar, the resulting temperatures at the fixed nodes will be very close to their true values.

- The main homework 6 function, the flowchart of which is presented below:



Finally, a summary of the main vectors and matrices used in the program:

- gNodeToCoord
  - This matrix maps the node number to actual coordinates in space:

Row	Column1	Column2
Node 1	x1	y1

Node 2	x2	y2
Node 3	x3	y3
…	…	…
Node M	xM	yM

- connectMap
  - This matrix maps the element number to the nodes that it connects

Row	Col1	Col2	Col3	Col4
Element 1	Lower Left Node	Lower Right Node	Upper Right Node	Upper Left Node
Element 2	Lower Left Node	Lower Right Node	Upper Right Node	Upper Left Node
Element 3	Lower Left Node	Lower Right Node	Upper Right Node	Upper Left Node
…	…	…	…	…
Element N	Lower Left Node	Lower Right Node	Upper Right Node	Upper Left Node

- elementK
  - This is the 4x4 matrix that is the result of the 2D Gaussian Integration for the elemental stiffness matrix. This matrix is then divided up and assembled into the global stiffness matrix.
- elementBoundary
  - This is the 4x1 vector that is the result of the 1D Gaussian Integration for the element boundary condition. It is 0 for most elements (for most elements do not lie on the prescribed Von Neumann boundary zone), but it is nevertheless divided up and assembled into the global R vector.
- elementLoading
  - This the 4x1 vector that is the result of the 2D Gaussian Integration for the loading function over the element. It is divided up and assembled into the global R vector almost immediately after being calculated.
- kGlobal
  - The global stiffness matrix, M rows by M columns where M is the number of nodes
- R
  - The global right hand side vector, M rows by 1 column.
- fixedTempNodes
  - A simple column vector that describes which nodes are at the fixed temperature boundary condition. It is calculated from a loop and its length is always equal to the number of radial elements.

Weak Form:

The procedure for finding the weak form of the thermal diffusion function was quite lengthy. First, we must begin with the PDE:

$$\nabla \cdot (k \nabla T) + f = 0$$

Next, hit the entire expression with a kinematically admissible test function:

$$(\nabla \cdot (k \nabla T) + f) v = 0$$

Integrate and split up:

$$\underset{\forall v}{\int_{\Omega}} \nabla \cdot (k \nabla T) v d\Omega + \int_{\Omega} f v d\Omega = 0$$

Next, this is where we would normally use the chain rule "sleight of hand" to weaken the differentiability of the left term. Since we are operating in 2D though, we must use the equivalent of this trick, which is essentially the divergence theorem that states:

$$\nabla \cdot (\sigma v) = (\nabla \sigma) v + \nabla v \sigma$$

Applying this to the above integral:

$$\nabla \cdot (k \nabla T) v = \nabla \cdot (k \nabla T v) - \nabla v^T k \nabla T$$

Substituting into the integral expression:

$$\int_{\Omega} f v d\Omega + \int_{\Omega} \nabla \cdot (k \nabla T v) d\Omega = \int_{\Omega} \nabla v^T k \nabla T d\Omega$$

Then use the divergence theorem to simplify the second term:

$$\int_{\Omega} f v d\Omega + \int_{\Gamma} v k \nabla T \hat{n} d\Gamma = \int_{\Omega} \nabla v^T k \nabla T d\Omega$$

Define this flux condition as the q_0 term:

$$q_o = -k \nabla T \cdot \hat{n}$$

And substitute in, and be careful to note that the traction boundary condition does not overlap the dirichlet temperature boundary condition:

$$\int_\Omega f v d\Omega - \int_{\Gamma_q} v q_o d\Gamma = \int_\Omega \nabla v^T k \nabla T d\Omega$$

$$0 = \Gamma_q \bigcap \Gamma_t$$

We have now obtained the weak form for the thermal diffusion problem. Now substitute in our bilinear basis functions to obtain:

$$\int_\Omega f \phi_j b_j d\Omega - \int_{\Gamma_q} \phi_j b_j q_o d\Gamma = \int_\Omega \nabla (\phi_j b_j)^T k \nabla \phi_i a_i d\Omega$$

And then transform to the zeta world, where J is the jacobian and J_Gamma is the line jacobian

$$\int_\Omega f \hat{\phi}_j b_j J d\Omega - \int_{\Gamma_q} \hat{\phi}_j b_j q_o J_\Gamma d\Gamma =$$
$$\int_\Omega \nabla (\hat{\phi}_j b_j)^T k \nabla \hat{\phi}_i a_i J d\Omega$$

Now we can eliminate the arbitrary b coefficients and express in computational format, where F is the deformation matrix and D is the 2x4 matrix of phi partial derivatives

$$\int_\Omega f \hat{\phi}_j J d\Omega - \int_{\Gamma_q} \hat{\phi}_j q_o J_\Gamma d\Gamma =$$
$$\int_\Omega (F^{-1} D)^T k (F^{-1} D) a_i J d\Omega$$

Analytical Solution:

Now that we have the weak form for the thermal diffusion problem, let us look for an analytical solution to the prescribed test case by going back to the PDE and substituting in the following expressions:

$$T_0 = 100$$
$$q_0(r) = \frac{40}{r}$$
$$f = \frac{80}{r^2} sin(2\theta)$$

Substituting in for the loading function:

$$\nabla \cdot (k \nabla T) + \frac{80}{r^2} sin(2\theta) = 0$$

Taking the partial derivatives in the **polar coordinate system**

$$\nabla \cdot (k \begin{pmatrix} \frac{\partial T}{\partial r} = 0 \\ \frac{1}{r}\frac{\partial T}{\partial \theta} \end{pmatrix}) + \frac{80}{r^2} sin(2\theta) = 0$$

Simplifying:

$$\frac{1}{r^2}\frac{\partial^2 T}{\partial \theta^2} + \frac{80}{r^2} sin(2\theta) = 0$$

Multiplying by R squared and removing the dependence on R:

$$\frac{\partial^2 T}{\partial \theta^2} + 80 sin(2\theta) = 0$$

Integrating once:

$$\frac{\partial T}{\partial \theta} - 40 sin(2\theta) = C_1$$

Integrating once again:

$$T(\theta) = 20 sin(2\theta) + C_1 \theta + C_2$$

Substituting for the boundary condition:

$$\frac{1}{r}\frac{\partial T}{\partial \theta}|_{\theta=0} - \frac{40}{r} cos(0) = \frac{C_1}{r}$$

Obtaining C_1 is zero:

$$\frac{40}{r} - \frac{40}{r} cos(0) = \frac{C_1}{r}$$
$$C_1 = 0$$

Obtaining the second constant of integration:

$$T(\theta = \pi) = 100 = C_2$$

Combining into final expression:

$$T = 20sin(2\theta) + 100$$

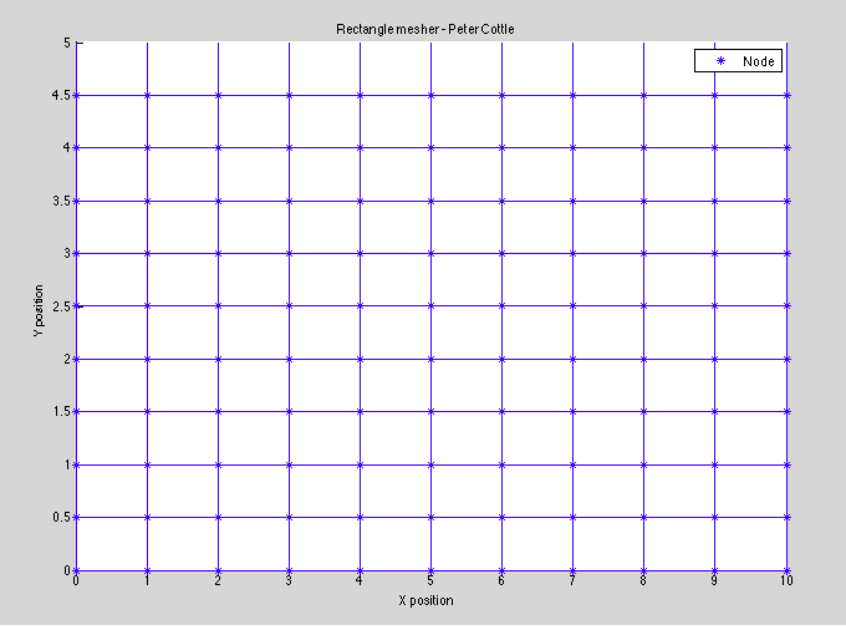
Thus we have finally obtained an analytical solution to the given test case.

Findings

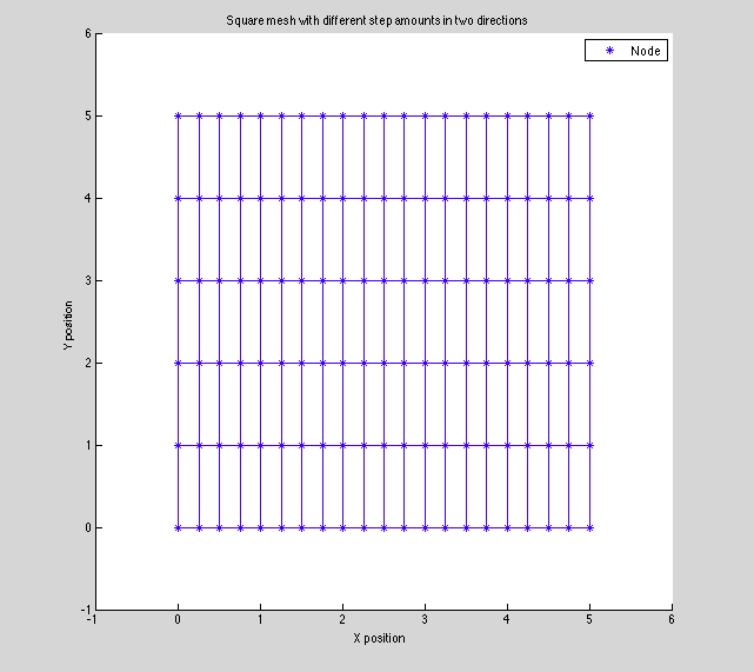
Mesher:

I implemented a very general mesher, as per instructions from the homework. My mesher is capable of rendering arches, circles, circles with holes, and rectangles – all of which can have different stepping in either directions. The rest of the FEM algorithm only depends on the connectivity map and the global node coordinates, so the mesher is quite independent of the rest of the FEM solution. The mesher still has to maintain the convex property of the elements (and the positive jacobian) to have the program work, but this is guaranteed with the current mesher.

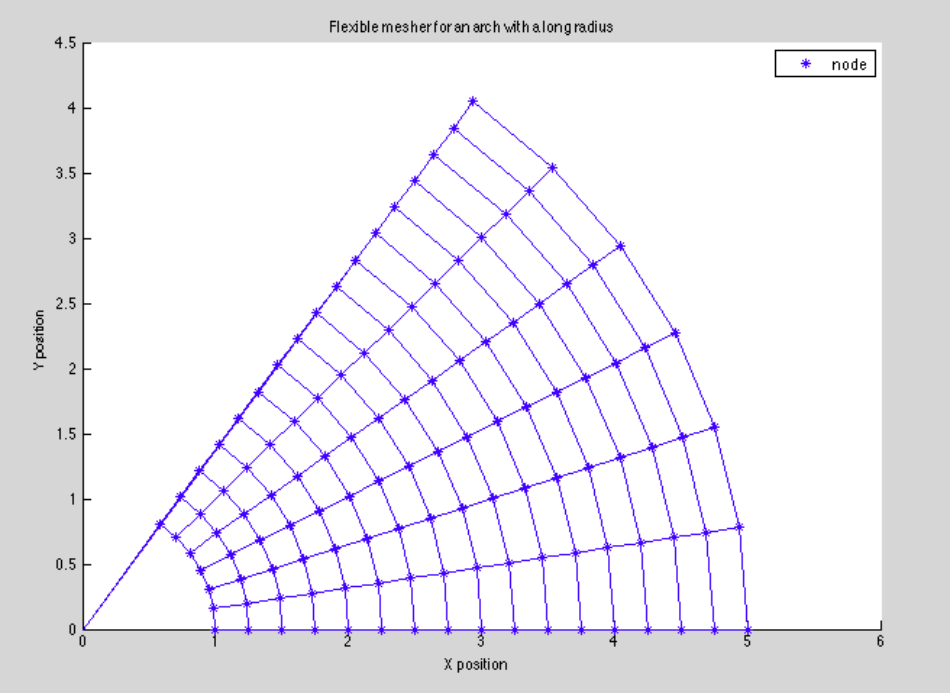
A plot of the mesher making a rectangle:



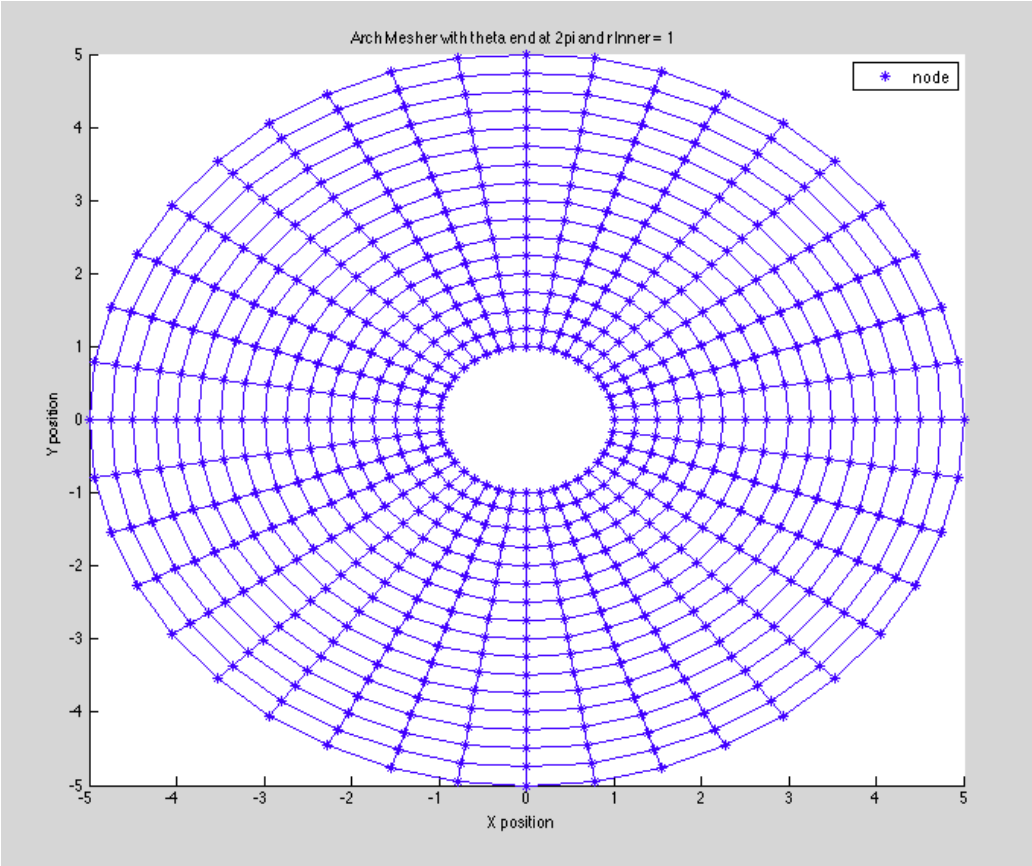
A square with different stepping in either direction:



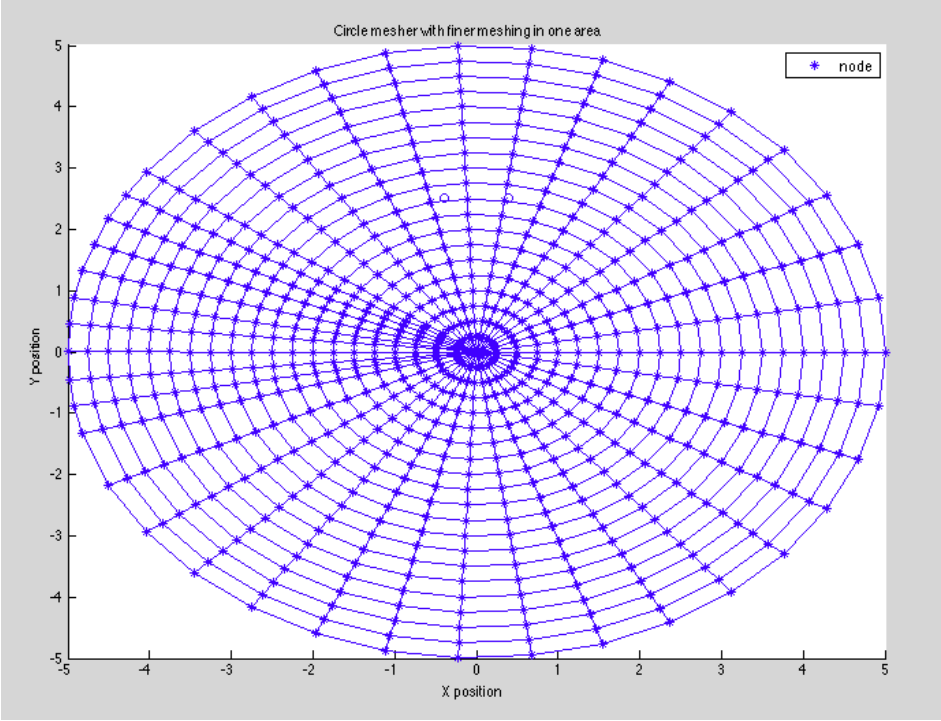
A mesh of an arc:



A mesh of a circle with a hole:



Circle meshing with no hole and finer meshing in the middle:



As you can see, the mesher I built is quite general and works for many different geometrical shapes (as per instructions from the homework).

As a further note, I plot all the node locations with a star and all the element connectivity with blue lines. Hence from these above plots, one can see that both the global coordinates of the nodes and the connectivity of the elements is correct and uniform.

Penalty Term (Pstar) Selection:

When choosing my Pstar variable for the penalty method, I chose the following heuristic: I find the maximum value of the stiffness matrix and multiply this by a Pstar scaling factor to obtain Pstar. This scaling factor can range from 10 to 100,000; the results for different scaling factors are shown below. Pstar is then used to modify the stiffness matrix at the nodes that are at a fixed temperature. It is also multiplied by the boundary condition temperature (T_0) and added to the right hand side to perform the complete penalty method.

In order to determine the relationship between the boundary condition accuracy and the Pstar scaling factor, I compiled a table with the same mesh problem solved with various Pstar scaling amounts:

Penalty Method Accuracy As Function of Pstar Scaling Factor						
Pstar Scalaing	1	10	100	1000	10000	100000
Boundary Node	Temp					
101	99.1146	99.9184	99.9919	99.9992	99.9999	100
102	98.656	99.8569	99.9856	99.9986	99.9999	100
103	98.7397	99.8703	99.987	99.9987	99.9999	100
104	98.8713	99.8825	99.9882	99.9988	99.9999	100
105	99.354	99.9455	99.9947	99.9995	99.9999	100

Incredibly, even a Pstar scaling amount of just one will produce boundary conditions that are quite close to their true values. At a Pstar scaling amount of 100,000, the penalty method is indistinguishable from the fixed temperature method.

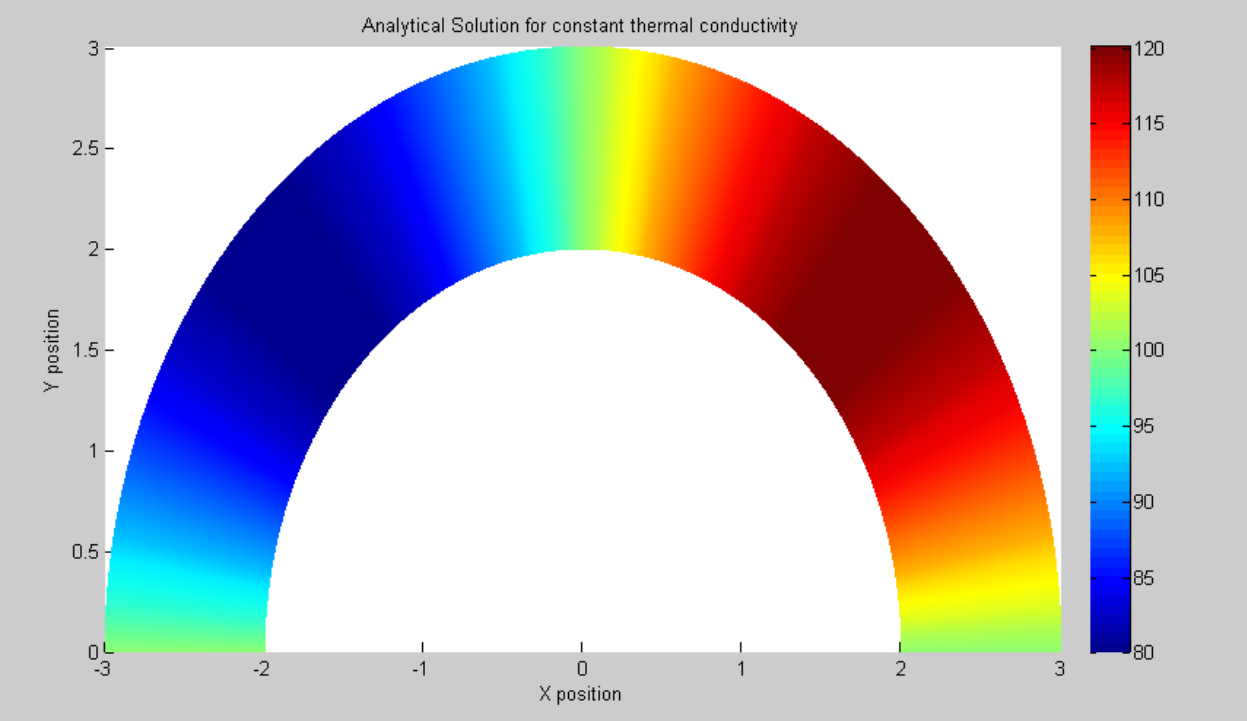
It is important to remember that although the penalty method is quite easy to implement and prevents any changes to our global stiffness matrix, it severely affects the condition number of the matrix and thus the rate of convergence for operations like the conjugate gradient solvers. Because of this, I kept my Pstar scaling factor to be 10,000 for the rest of the calculations which seemed like an optimum balance between condition number and boundary condition accuracy.

Test Case:

After deriving the analytical solution, I obtained this as the equation that describes temperature as a function of theta:

$$T = 20\sin(2\theta) + 100$$

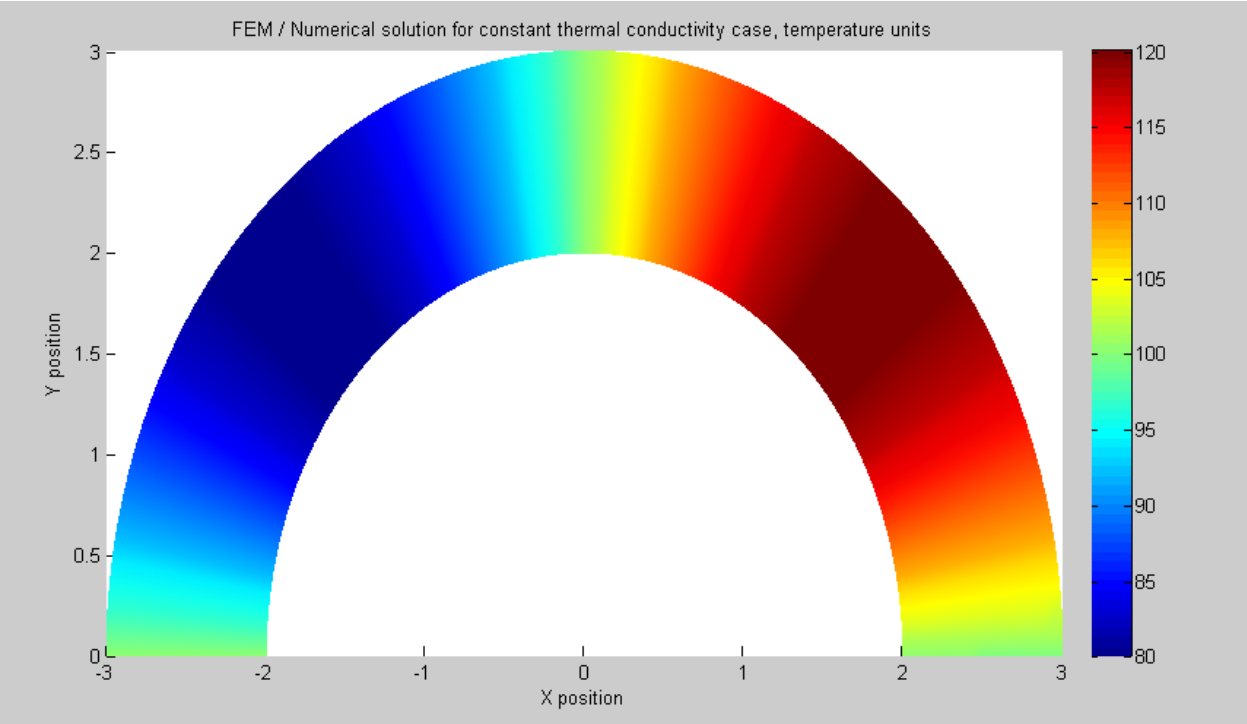
After plugging this into Matlab and solving for the nodal thermal diffusion (assuming that theta is 0 at the right side of the arch), I obtained this plot for temperature across the arch:



As you can see, the resulting plot shows a very smooth gradient. I used 80 elements in the theta direction and ten elements in the r direction for the above mesh. The matlab code for this solution is below:

```
theta = atan(gNodeToCoord(:,2) ./ gNodeToCoord(:,1))
for i = 1:length(theta)
if(theta(i) < 0)
theta(i) = theta(i) + pi;
end
end
answer = 100 + 20*sin(2*theta)
(plot as normal)
```

I then programmed in this test case into my finite-element based heat diffusion solver. After letting Matlab crunch the numbers, I obtained the following plot:

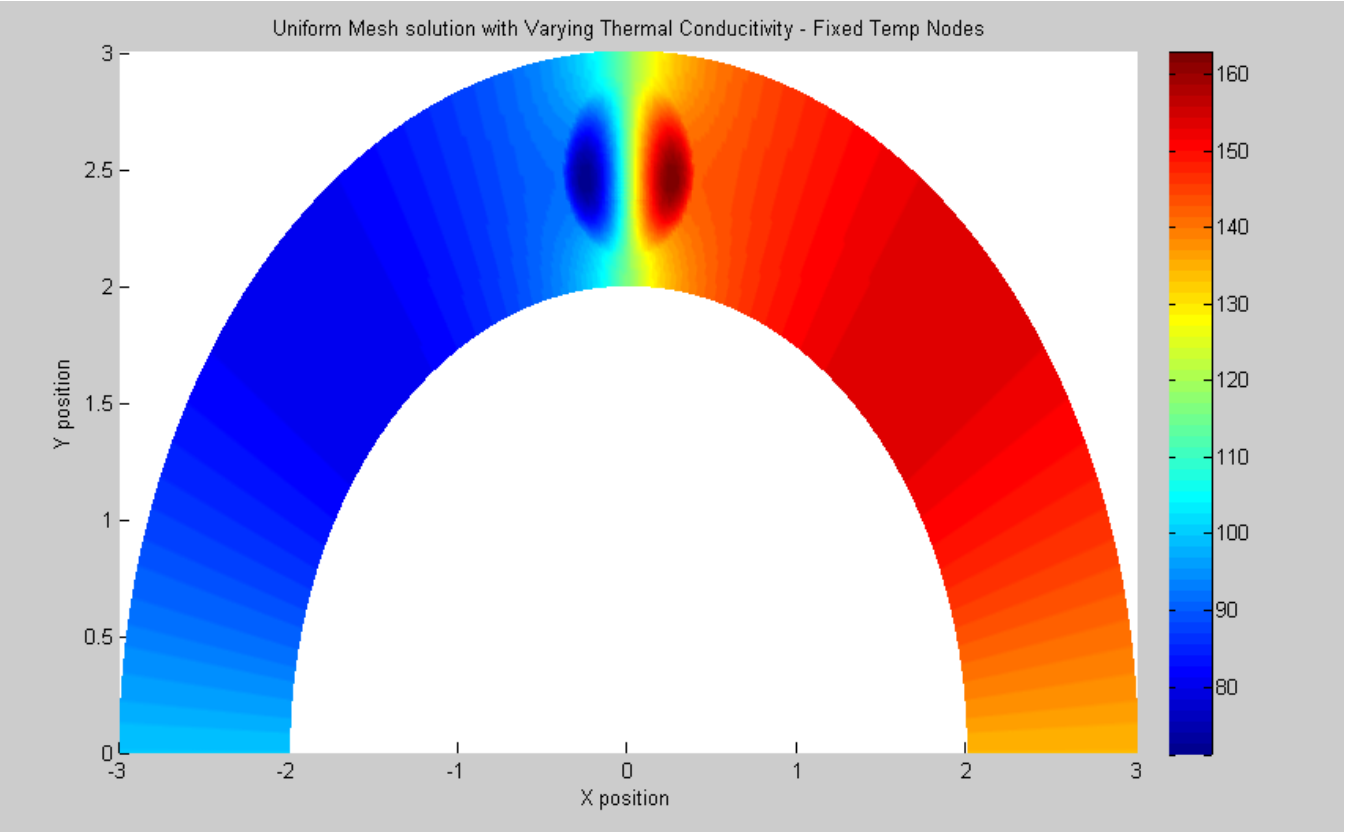


As you can see, these two plots are visually identical in terms of temperature distribution. Furthermore, I manually inspected the numerical answers of the last 10 non-fixed temperature nodes, and the differences in temperature are only one or two hundredths of a degree off which is easily accounted for due to numerical error.

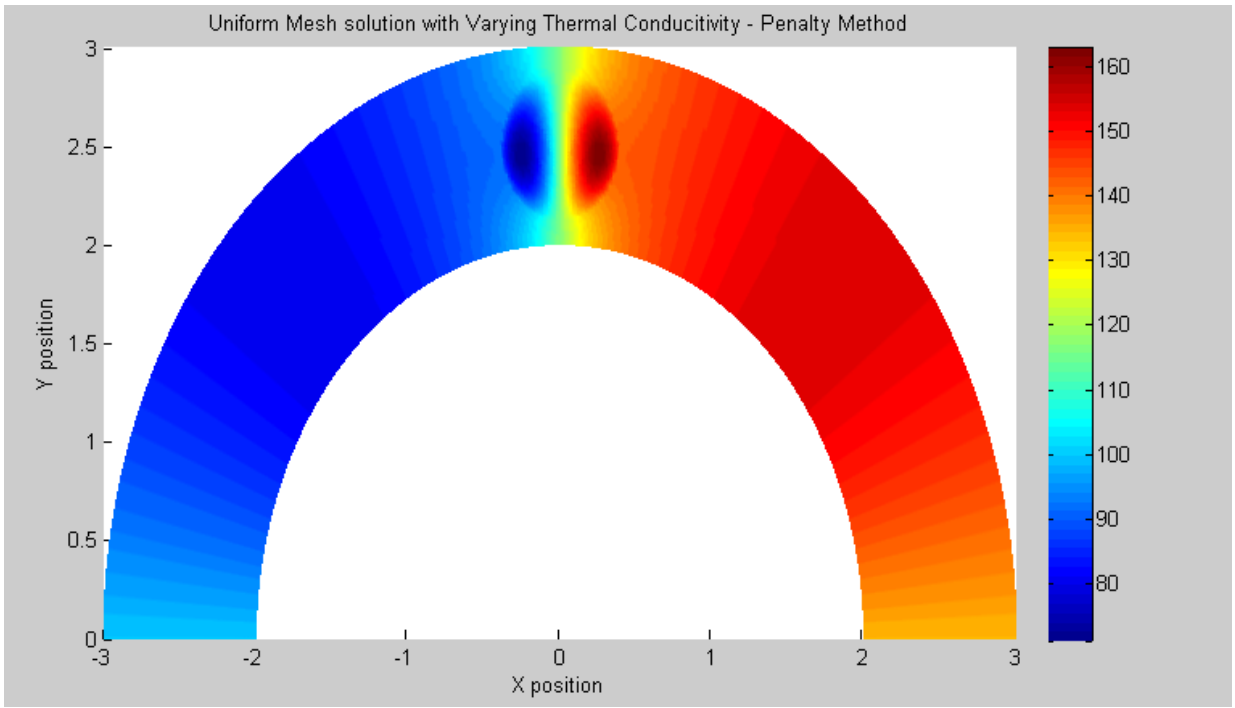
Finite Element Method for 2-Phase structure:

The mesh for the 2-phase structure is considerably finer (50 radial direction elements and 400 theta direction elements), and thus the program took considerably longer to run. For 20,000 elements, the overall stiffness matrix was 20451x20451 which was a considerable memory overhead, even on the fast desktop computer that I built. I had to switch to a conjugate gradient solver here, for the inverse operation on the K matrix would crash Matlab (and my entire computer). A CG solver was considerably faster and did not cause my computer to crash.

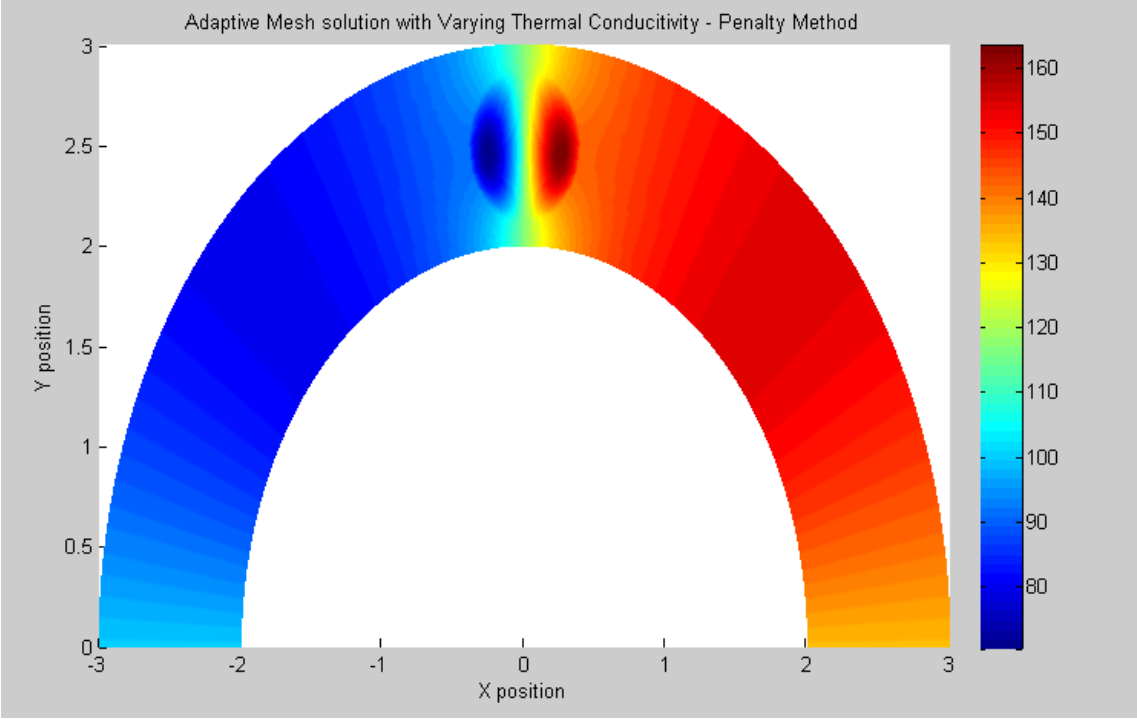
Regardless, after letting the program run in the two-phase structure code, I solved and obtained the following plot for **fixed boundary conditions and uniform meshing**:



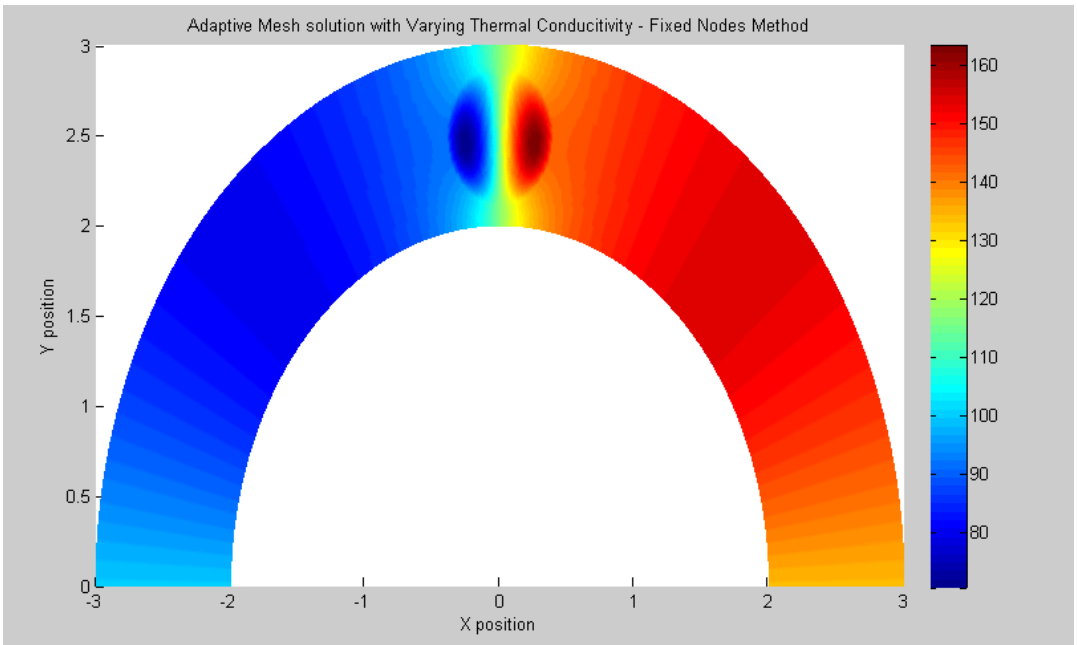
Next, I switched my code to perform the **penalty method** instead of fixed boundary conditions. I used a P_star scaling factor of 10,000 and got the following plot with uniform meshing:



After these two solutions, I went ahead and programmed in the adaptive meshing for a finer mesh in the discontinuity zone. The plots for these meshes were discussed in the mesher section above. After solving here, I obtained the following plot for adaptive meshing and penalty method:



As you can see, there is considerably more detail of the thermal flux around the area with varying thermal conductivity. Finally, I switched back over to the fixed temperature method for the Dirichlet boundary conditions just to make sure it was the same:



As expected, the plot is visually identical.

Discussion

This homework was a great introduction into the finite element method in two dimensions and also served to make some great points about the actual implementation. The penalty method works surprisingly well (even for low Pstar amounts) and is quite easy to implement, so I can see why it is the preferred option for 3D finite element problems.

We also got to see the interplay between the loading function here (which either added or subtracted heat to the arch), the flux boundary condition (which extracted heat from the arch), and the fixed boundary condition (which held the left side at a constant temperature). All of these factors combined together to form a nice sinusoidal temperature gradient throughout the arch, showing the interplay between the flux extracting heat from the entire structure and the loading function adding heat to the arch in certain places.

Furthermore, the arch thermal diffusion problem had *quite* different solutions when the thermal conductivity varied throughout the arch rather than staying constant. For the constant case, we were able to determine an analytical solution so the FEM solution just simply complemented our earlier work.

For the varying thermal conductivity problem, there is no way I could have predicted the solution would take such an interesting distribution throughout the arch. I thought the circular region would just show a slightly denser gradient or some other small factor, but the change in material property completely changed the thermal distribution around the entire arch. This is because the extremely low conductivity traps the heat in this region; all the heat generated on the right side and all the "cold" generated on the left side meet in the middle in this rather bizarre fine temperature gradient. Clearly, it is difficult to interpret the nature of thermal distribution over complex structures, only further solidifying how valuable the FEM method is for industry.

Finally, the derivation of the weak form was great preparation for the final and showed how the divergence theorem must be applied in order to obtain the boundary flux conditions, which is one of the "non essential" boundary conditions as Debanjan mentioned in the final lecture.

In conclusion, this homework was a great summary of the finite element method in two dimensions and another useful part of ME 280a.

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. **Note: When running the code, change the Booleans at the beginning of the mainhw6 file to turn on/off adaptive meshing, the thermal discontinuity, the penalty method, and the plotting of the mesh.**

```
function [ gNodeToCoord, connectMap ] =
archMesher(r_step,theta_step,shouldPlot,doFinerMeshing,bigMult
)
%ok so first we will step in the theta direction
%then we will step in the r direction and generate
%those points. Note that the total
%number of points to generate is
%((r_range/r_step+1)*(theta_range/theta_step+1))
r_end = 3;
r_begin = 2;
r_range = r_end-r_begin;
theta_begin = 0;
theta_end = pi;
theta_range = theta_end-theta_begin;
```

```
disp 'Hi!'
N_r = 4;
N_theta = 20;
% N_r = 10;
% N_theta = 80;
N_r = 50;
N_theta = 400;
r_step = 1/N_r;
theta_step = pi/N_theta;
big_mult = 2;
T_0 = 100;
PstarFactor = 100;
shouldplot = false;
doThermalHole = false;
```

```
numPoints = (theta_range/theta_step + 1) * (r_range/r_step +
1);
numPoints = round(numPoints)
gNodeToCoord = zeros(numPoints,2);
coordNum = 1;
if(shouldPlot)
    close all
    figure(1)
    hold on
end
if(~doFinerMeshing)
    for theta=theta_begin:theta_step:theta_end
        for r=r_begin:r_step:r_end
            %get the x
            x = r*cos(theta);
            y = r*sin(theta);
            gNodeToCoord(coordNum,1) = x;
            gNodeToCoord(coordNum,2) = y;
            coordNum = coordNum + 1;
            if(shouldPlot)
                plot(x,y,'b*');
            end
        end
    end
else
    %get a lot of variables

    numThetaElements = round(pi/theta_step);

    %want one fourth to be small, 3/4 to be big
    numSmall = 1/4 * numThetaElements
    numBig = 3/4 * numThetaElements

    %we know we need it to add up to pi...
    toDivide = bigMult * numBig + numSmall;
    small_step = pi / toDivide;
    big_step = small_step * bigMult;

    %keep track of our elements now
    currentTheta = 0;

    for thetaCount=0:numThetaElements
        if(currentTheta > pi)
            currentTheta = pi;
        end

        for r=r_begin:r_step:r_end
            %get the x
            x = r*cos(currentTheta);
            y = r*sin(currentTheta);
            gNodeToCoord(coordNum,1) = x;
            gNodeToCoord(coordNum,2) = y;
            coordNum = coordNum + 1;
            if(shouldPlot)
                plot(x,y,'b*');
            end
        end

        %decide which to add
        if(thetaCount < numBig/2 - 1)
            currentTheta = currentTheta + big_step;
        elseif (thetaCount < numBig/2 -1 + numSmall)
            currentTheta = currentTheta + small_step;
        else
            currentTheta = currentTheta + big_step;
        end

    end

    plot(0.4,2.5,'bo');
    plot(-0.4,2.5,'bo');

end
%now go through and do connectivity
%we need the number of elements now strangely enough...
%num elements is the same as the number of elements per slice
%times the number of slices. so.
numElementsPerSlice = r_range/r_step
numSlices = round(theta_range/theta_step)
numElements = numElementsPerSlice*numSlices;
numElements = round(numElements)
connectMap = zeros(numElements,4);
%make the connectivity map
%start with sliceCounter and sideCounter
elementCounter = 1;
for sliceCounter=1:numSlices
    for sideCounter=1:numElementsPerSlice
        %get the lower left
        LL = sideCounter + (sliceCounter-
1)*(numElementsPerSlice+1);
        LR = LL + 1;

        UL = sideCounter +
(sliceCounter)*(numElementsPerSlice+1);
        UR = UL + 1;
```

```
doPenaltyMethod = false;
doFinerMeshing = false;
[gNodeToCoord, connectMap] =
archMesher(r_step,theta_step,shouldplot,doFinerMeshing,big_mult);
%last row of connectMap are boundary condition nodes
totalRows = max(size(gNodeToCoord));
numElementsOnEdge = 1/r_step + 1;
nodesThatAreFixed = [];
for i=1:numElementsOnEdge
    nodesThatAreFixed = [nodesThatAreFixed totalRows-(i-1)];
end
fixedTempNodes = nodesThatAreFixed;
thesize = size(connectMap);
numElements = thesize(1);
thesize = size(gNodeToCoord);
numNodes = thesize(1);
kGlobal = spalloc(numNodes,numNodes,5*numNodes);
R = zeros(numNodes,1);
R_b = zeros(numNodes,1);
for currElement = 1:numElements
    if (mod(currElement,50) == 0)
        currElement
    end
    %get the new k matrix?
    miniK = elementK(currElement,gNodeToCoord,connectMap,doThermalHole);

    %get the RHS loading?
    miniRloading = 10*elementLoading(currElement,gNodeToCoord,connectMap);
    %pause
    miniRboundary = elementBoundary(currElement,gNodeToCoord,connectMap);

    R = assembleLoading(R,miniRloading,currElement,connectMap);
    R_b = assembleLoading(R_b,miniRboundary,currElement,connectMap);
    %pause
    %pause(0.05)

    %assemble this
    kGlobal = assembleK(kGlobal,miniK,currElement,connectMap);

    if(doPenaltyMethod)
        %get the penalty k
        %get the penalty RHS
        %add them
        asd = 1;
    end


end
R = R - R_b;
%deal with the boundary conditions
kCopy = kGlobal;
Rcopy = R;
%get a list of the nodes to eliminate in descending order
fixedTempNodes = sort(fixedTempNodes);
fixedTempNodes = fliplr(fixedTempNodes);
if(~doPenaltyMethod)
%loop through these
    for i=1:max(size(fixedTempNodes))
        currNode = fixedTempNodes(i);
        %somehow delete this row, pad this column with the BC in the right
        %spot, and add it to the right hand side
        %delete row
        kGlobal(currNode,:) = [];
        R(currNode,:) = [];
        %get the column
        theCol = kGlobal(:,currNode);
        theCol = theCol * T_0
        R = R - theCol;
        %delete this column too!
        kGlobal(:,currNode) = [];
    end
else
    %do the penalty method!!

    maxK = max(max(kGlobal));
    Pstar = maxK*PstarFactor;

    for i=1:max(size(fixedTempNodes))
        whichToHit = fixedTempNodes(i);

        kGlobal(whichToHit,whichToHit) = kGlobal(whichToHit,whichToHit) +
Pstar;

        R(whichToHit,1) = R(whichToHit,1) + Pstar * T_0;
    end

end
theAns = inv(kGlobal) * R
disp('inserting known ones')
fixedTempNodes = sort(fixedTempNodes);
if(~doPenaltyMethod)
    for i=1:length(fixedTempNodes)
        row = fixedTempNodes(i);
        theAns = [theAns(1:row-1); T_0; theAns(row:end)];
    end
```

```
        %plot this
        if(shouldPlot)
            xs = [gNodeToCoord(LL,1) gNodeToCoord(LR,1) ...
                  gNodeToCoord(LR,1) gNodeToCoord(UR,1) ...
                  gNodeToCoord(UR,1) gNodeToCoord(UL,1) ...
                  gNodeToCoord(UL,1) gNodeToCoord(LL,1) ...
                   ];
            ys = [gNodeToCoord(LL,2) gNodeToCoord(LR,2) ...
                  gNodeToCoord(LR,2) gNodeToCoord(UR,2) ...
                  gNodeToCoord(UR,2) gNodeToCoord(UL,2) ...
                  gNodeToCoord(UL,2) gNodeToCoord(LL,2) ...
                   ];
            line(xs,ys);
            axis([-3 3 0 3]);
            pause(0.05);
        end

        connectMap(elementCounter,:) = [LL LR UR UL];
        elementCounter = elementCounter + 1;

    end

end
end
function [ gNodeToCoord, connectMap ] = rectMesher(
length,height,x_step,y_step,shouldPlot )
%RECTMESHER Summary of this function goes here
%   Detailed explanation goes here
if(shouldPlot)
    figure(1)
    clf
    hold on
    axis([0 length 0 height]);
end
numPoints = (length/x_step + 1)*(height/y_step + 1);
coordNum = 1;
gNodeToCoord = zeros(numPoints,2);
for currX=0:x_step:length
    for currY = 0:y_step:height
        gNodeToCoord(coordNum,1) = currX;
        gNodeToCoord(coordNum,2) = currY;

        coordNum = coordNum + 1;

        if(shouldPlot)
            plot(currX,currY,'b*');
        end

    end
end
numElementsInY = height/y_step;
numElementsInX = length/x_step;
totalElements = numElementsInX * numElementsInY;
connectMap = zeros(totalElements,4);
currElement = 1;
for currSlice=1:numElementsInX
    for currTop=1:numElementsInY

        LL = currTop + (currSlice-1)*(numElementsInY+1);
        UL = LL + 1;

        LR = currTop + (currSlice) * (numElementsInY+1);
        UR = LR + 1;

        connectMap(currElement,:) = [LL LR UR UL];

        currElement = currElement + 1;

        if(shouldPlot)
            xs = [gNodeToCoord(LL,1) gNodeToCoord(LR,1) ...
                  gNodeToCoord(LR,1) gNodeToCoord(UR,1) ...
                  gNodeToCoord(UR,1) gNodeToCoord(UL,1) ...
                  gNodeToCoord(UL,1) gNodeToCoord(LL,1) ...
                   ];
            ys = [gNodeToCoord(LL,2) gNodeToCoord(LR,2) ...
                  gNodeToCoord(LR,2) gNodeToCoord(UR,2) ...
                  gNodeToCoord(UR,2) gNodeToCoord(UL,2) ...
                  gNodeToCoord(UL,2) gNodeToCoord(LL,2) ...
                   ];
            line(xs,ys);
            pause(0.05);

        end

    end
end
end



function [ phis,phipartials] = getPhis()
%GETPHIS Summary of this function goes here
%   Detailed explanation goes here
end
theAns
%plotting this shit, CRAP man!!!
%need a xdata vector like this:
% xNode1el1 xNode1el2
% xNode2el1 xNode2el2... etc
% xNode3el1
% xNode4el1
%same for the y thing, and then just the node value at each!
xdata = zeros(4,numElements);
ydata = zeros(4,numElements);
tempData = zeros(4,numElements);
for currElement=1:numElements
    nodes = connectMap(currElement,:);

    thisX = zeros(4,1);
    thisY = zeros(4,1);
    thisTemp = zeros(4,1);
    for i=1:4
        theNode = nodes(1,i);
        thisX(i,1) = gNodeToCoord(theNode,1);
        thisY(i,1) = gNodeToCoord(theNode,2);
        thisTemp(i,1) = theAns(theNode);
    end

    xdata(:,currElement) = thisX;
    ydata(:,currElement) = thisY;
    tempData(:,currElement) = thisTemp;
end
patch(xdata,ydata,tempData,'EdgeColor','interp');


function [ scalar ] = loadingFunction(combined)
%LOADINGFUNCTION Summary of this function goes here
%   Detailed explanation goes here
%takes in x and y, returns the value of the loading function
x = combined(:,1);
y = combined(:,2);
f = @(r,theta) 8 .* sin(2 .* theta) ./ r.^2;
theta = atan(y./x);
r = sqrt(x.^2 + y.^2);
for i = 1:length(theta)
    if(theta(i) < 0)
        theta(i) = theta(i) + pi;
    end
end
%pause(0.05)
scalar = f(r,theta);
end

function [ x1, x2 ] = getX1X2(currElement,gNodeToCoord,connectMap)
%GETJACOBIAN Summary of this function goes here
%   Detailed explanation goes here
[phis, phipartials] = getPhis();
%ok crap. So we have to build up the mapping matrix essentially
%and then take the determinant of that to get the actual jacobian function
%which may or may not be a constant
%first, get an expression for x1 in terms of phis'
%do this first by getting a list of all the x coordinates for these
%nodes
globalNodes = connectMap(currElement,:);
gXcoords = zeros(1,4);
gYcoords = zeros(1,4);
for i=1:4
    gXcoords(i) = gNodeToCoord(globalNodes(i),1);
    gYcoords(i) = gNodeToCoord(globalNodes(i),2);
end
gCoords = [gXcoords; gYcoords];
%now that we have all of our x coordinates and y coordinates,
%multiply each one of these by the associated phi's to get the mapping
x1 = @(z1,z2)   phis{1}(z1,z2).*gCoords(1,1) +
phis{2}(z1,z2).*gCoords(1,2) + ...
                phis{3}(z1,z2).*gCoords(1,3) +
phis{4}(z1,z2).*gCoords(1,4);
x2 = @(z1,z2)   phis{1}(z1,z2).*gCoords(2,1) +
phis{2}(z1,z2).*gCoords(2,2) + ...
                phis{3}(z1,z2).*gCoords(2,3) +
phis{4}(z1,z2).*gCoords(2,4);
end

function [ answer ] = getThermalConductivity(x,y)
%GETTHERMALCONDUCTIVITY Summary of this function goes here
%   Detailed explanation goes here
%this just returns the thermal conductivity of the arch.
%so basically, the center of this hole is located at (0,2.5) and rc is
%0.4...
centerX = 0;
centerY = 2.5;
r_hole = 0.4;
k_other = 1*10^-3;
k_all = 1;
%get the distance away for each of the results
answer = zeros(max(size(x)),1);
for i = 1:length(answer)
    %get the distance away
```

```
phi1 = @(z1,z2) 0.25 .* (1-z1).*(1-z2);
phi2 = @(z1,z2) 0.25 .* (1+z1).*(1-z2);
phi3 = @(z1,z2) 0.25 .* (1+z1).*(1+z2);
phi4 = @(z1,z2) 0.25 .* (1-z1).*(1+z2);

phis = {phi1 phi2 phi3 phi4};

dphi1dz1 = @(z1,z2) 0.25 .* (-1 + z2);
dphi1dz2 = @(z1,z2) 0.25 .* (-1 + z1);

dphi2dz1 = @(z1,z2) 0.25 .* ( 1  - z2);
dphi2dz2 = @(z1,z2) 0.25 .* (-1  - z1);

dphi3dz1 = @(z1,z2) 0.25 .* (1 + z2);
dphi3dz2 = @(z1,z2) 0.25 .* (1 + z1);

dphi4dz1 = @(z1,z2) 0.25 .* (-1  - z2);
dphi4dz2 = @(z1,z2) 0.25 .* ( 1  - z1);

phipartials = { dphi1dz1 dphi1dz2;
                dphi2dz1 dphi2dz2;
                dphi3dz1 dphi3dz2;
                dphi4dz1 dphi4dz2;};



end

function [ gaussPoints,gaussWeights ] = getGauss(numPoints)
%GETGAUSS Summary of this function goes here
%   Detailed explanation goes here

if(numPoints == 5)

    gaussPoints = zeros(5,1);
    gaussWeights = zeros(5,1);

%     gaussPoints = [0.18343464249564980493947610;-
0.18343464249564980493947610;
%                    0.52553240991632898581773900;-
0.52553240991632898581773900;
%                    0.79666647741362673959155390;-
0.79666647741362673959155390;
%                    0.96028985649753623168356090;-
0.96028985649753623168356090];
%
%     gaussWeights =
[0.36268378337836198296515040;0.36268378337836198296515040;
%
0.31370664587788728733796220;0.31370664587788728733796220;
%
0.22238103445337447054435600;0.22238103445337447054435600;
%
0.10122853629037625915253140;0.10122853629037625915253140];
    gaussPoints = [ -0.90617984593866399279762690;
                    -0.53846931010568309103631440;
                     0.00000000000000000000000000;
                     0.53846931010568309103631440;
                     0.90617984593866399279762690];

    gaussWeights = [0.23692688505618908751426400;
                    0.47862867049936646804129150;
                    0.56888888888888888888888890;
                    0.47862867049936646804129150;
                    0.23692688505618908751426400];

elseif(numPoints == 8)
    gaussPoints = [0.18343464249564980493947610;-
0.18343464249564980493947610;
                   0.52553240991632898581773900;-
0.52553240991632898581773900;
                   0.79666647741362673959155390;-
0.79666647741362673959155390;
                   0.96028985649753623168356090;-
0.96028985649753623168356090];

    gaussWeights =
[0.36268378337836198296515040;0.36268378337836198296515040;

0.31370664587788728733796220;0.31370664587788728733796220;

0.22238103445337447054435600;0.22238103445337447054435600;

0.10122853629037625915253140;0.10122853629037625915253140];
else
    %do 2 point rule
    gaussPoints = zeros(2,1);
    gaussPoints = zeros(2,1);


    gaussPoints = ...
                [ 0.57735026918962576450914880; -
0.57735026918962576450914880];
    thisX = x(i);
    thisY = y(i);

    thisDist = sqrt((thisX-centerX).^2 + (thisY-centerY).^2);

    if(thisDist <= 0.4)
        answer(i) = k_other;
    else
        answer(i) = k_all;
    end

end
end

function [ jacobianFunction, F_UL, F_UR, F_LL, F_LR ] = getJacobian(
currElement, gNodeToCoord, connectMap )
%GETJACOBIAN Summary of this function goes here
%   Detailed explanation goes here
[phis, phipartials] = getPhis();
%ok crap. So we have to build up the mapping matrix essentially
%and then take the determinant of that to get the actual jacobian function
%which may or may not be a constant
%first, get an expression for x1 in terms of phis'
%do this first by getting a list of all the x coordinates for these
%nodes
globalNodes = connectMap(currElement,:);
gXcoords = zeros(1,4);
gYcoords = zeros(1,4);
for i=1:4
    gXcoords(i) = gNodeToCoord(globalNodes(i),1);
    gYcoords(i) = gNodeToCoord(globalNodes(i),2);
end
gCoords = [gXcoords; gYcoords];
%now that we have all of our x coordinates and y coordinates,
%multiply each one of these by the associated phi's to get the mapping
x1 = @(z1,z2)   phis{1}(z1,z2).*gCoords(1,1) +
phis{2}(z1,z2).*gCoords(1,2) + ...
                phis{3}(z1,z2).*gCoords(1,3) +
phis{4}(z1,z2).*gCoords(1,4);
x2 = @(z1,z2)   phis{1}(z1,z2).*gCoords(2,1) +
phis{2}(z1,z2).*gCoords(2,2) + ...
                phis{3}(z1,z2).*gCoords(2,3) +
phis{4}(z1,z2).*gCoords(2,4);
%and the mapping F matrix is like dx1/dz1 etc. so 4x4 cell array
%
% | dx1 / dz1     dx1 / dz2   |
% | dx2 / dz1     dx2 / dz2   |
%
F_UL = @(z1,z2) phipartials{1,1}(z1,z2).*gXcoords(1) +
phipartials{2,1}(z1,z2).*gXcoords(2) + ...
                phipartials{3,1}(z1,z2).*gXcoords(3) +
phipartials{4,1}(z1,z2).*gXcoords(4);
F_LL = @(z1,z2) phipartials{1,2}(z1,z2).*gXcoords(1) +
phipartials{2,2}(z1,z2).*gXcoords(2) + ...
                phipartials{3,2}(z1,z2).*gXcoords(3) +
phipartials{4,2}(z1,z2).*gXcoords(4);
F_UR = @(z1,z2) phipartials{1,1}(z1,z2).*gYcoords(1) +
phipartials{2,1}(z1,z2).*gYcoords(2) + ...
                phipartials{3,1}(z1,z2).*gYcoords(3) +
phipartials{4,1}(z1,z2).*gYcoords(4);
F_LR = @(z1,z2) phipartials{1,2}(z1,z2).*gYcoords(1) +
phipartials{2,2}(z1,z2).*gYcoords(2) + ...
                phipartials{3,2}(z1,z2).*gYcoords(3) +
phipartials{4,2}(z1,z2).*gYcoords(4);
%literally the determinant of the stuff above
jacobianFunction = @(z1,z2) F_UL(z1,z2) .* F_LR(z1,z2) - F_UR(z1,z2) .*
F_LL(z1,z2);
end

function [ fInverse ] = getFinverse(currElement,gNodeToCoord,connectMap)
%GETFINVERSE Summary of this function goes here
%   Detailed explanation goes here

%f inverse is essentially the 2x2 matrix of dz1/dx1 etc etc

%in matrix form:

[ jacobianFunction, F_UL, F_UR, F_LL, F_LR ] =
getJacobian(currElement,gNodeToCoord,connectMap);
%ok, make the matrix thing

f = @(z1,z2)    [F_UL(z1,z2) F_UR(z1,z2);
                 F_LL(z1,z2) F_LR(z1,z2)];

%these are already mixed up
fInverse         = @(z1,z2) (1./jacobianFunction(z1,z2)) .* [F_LR(z1,z2) -
F_UR(z1,z2);
                                                                -F_LL(z1,z2)
F_UL(z1,z2);];



end
```

```
    gaussWeights = ...
                [ 1.0000000000000000000000000;
1.0000000000000000000000000];


end



end
```