ME 280a: HW 6

April Novak

December 7, 2016

1 Introduction and Objectives

The purpose of this study is to construct the Finite Element (FE) problem statement for a 2-D heat conduction problem, and then to solve said problem on a relatively complex domain.

2 Procedure

This section details the problem statement and mathematical method used for solving the problem. The particular boundary conditions and problem parameters will be discussed following the discussion of the weak form and finite element implementation.

2.1 The Weak Form

The weak form is obtained by multiplying the heat conduction equation, Eq. (32), through by a test function ψ and integrating over the body:

$$\int_\Omega \nabla \cdot (k\nabla T)\psi d\Omega = -\int_\Omega f\psi d\Omega \tag{1}$$

Applying the product rule in Eq. (1):

$$\begin{aligned} -\int_\Omega k\nabla T\nabla\psi d\Omega + \int_{\partial\Omega} k\nabla T\psi\cdot\hat{n}dA &= -\int_\Omega f\psi d\Omega \\ \int_\Omega k\nabla T\nabla\psi d\Omega &= \int_\Omega f\psi d\Omega + \int_{\partial\Omega} k\nabla T\psi\cdot\hat{n}dA \end{aligned} \tag{2}$$

Because the shape functions are defined to be zero on Dirichlet boundaries, the above area integral can be specified as the boundaries over which there are flux boundary conditions:

$$\int_\Omega k\nabla T\nabla\psi d\Omega = \int_\Omega f\psi d\Omega + \int_{\Gamma_q} \mathbf{q}^*\psi dA \tag{3}$$

where Γ_q is the boundary over which the heat flux is specified, and $\mathbf{q}^* = k\nabla T\cdot\hat{n}$ is the known heat flux on the boundary. On boundaries for which there are Dirichlet conditions, denoted as Γ_d, those nodes are either subject to a penalty term to enforce the boundary, or those nodes are removed from the final matrix system. Both of these methods are investigated in this assignment - the method of removing nodes from the global stiffness matrix and load vector will be referred to as "static condensation." Hence, the weak form can be stated as:

$$\begin{aligned} &\text{Find } T \in \mathbf{H}^T(\Omega) \subset \mathbf{H}^1(\Omega) \text{ so that } T|_{\Gamma_d} = T^* \text{ and so that } \forall\ \psi \in \mathbf{H}^\psi(\Omega) \subset \mathbf{H}^1(\Omega), \psi|_{\Gamma_d} = \mathbf{0}, \\ &\qquad\qquad \text{and for } \mathbf{q} \in \mathbf{L}^2(\Gamma_q), \mathbf{q} = \mathbf{q}^*|_{\Gamma_q} \text{ and } f \in \mathbf{L}^2(\Omega) \\ &\qquad\qquad \int_\Omega k\nabla T\nabla\psi d\Omega = \int_\Omega f\psi d\Omega + \int_{\Gamma_q} \mathbf{q}^*\psi dA \end{aligned} \tag{4}$$

where this weak form applies if the method of static condensation is to be used to apply the Dirichlet boundary conditions. If the penalty method is to be used:

$$\begin{gathered}\text{Find } T \in \mathbf{H}^T(\Omega) \subset \mathbf{H}^1(\Omega) \text{ so that } T|_{\Gamma_d} = T^* \text{ and so that } \forall\ \psi \in \mathbf{H}^\psi(\Omega) \subset \mathbf{H}^1(\Omega) \\ \text{and for } \mathbf{q} \in \mathbf{L}^2(\Gamma_q), \mathbf{q} = \mathbf{q}^*|_{\Gamma_q} \text{ and } f \in \mathbf{L}^2(\Omega) \\ \int_\Omega k\nabla T \nabla\psi d\Omega + P^* \int_{\Gamma_d} T\psi dA = \int_\Omega f\psi d\Omega + \int_{\Gamma_q} \mathbf{q}^*\psi dA + P^* \int_{\Gamma_d} T^*\psi dA\end{gathered} \tag{5}$$

where P^* is the penalty coefficient that is a large, positive number that essentially applies a traction that forces the temperature on the Dirichlet boundaries to equal the specified temperature T^*.

2.1.1 The Finite Element Weak Form

This section covers the details regarding finite element implementation of Eq. (5) (the non-penalty method simply sets $P^* = 0$. To implement this weak form, first the unknown, the temperature, must be expanded in a series of shape functions ϕ:

$$T = \sum_{i=1}^{n_{en}} C_j \phi_j = \mathbf{NT} \tag{6}$$

where n_{en} are the number of nodes per element and the following vectors have been defined (for a linear, 2-D element):

$$\mathbf{N} \equiv \begin{bmatrix}\phi_1(x,y) & \phi_2(x,y) & \phi_3(x,y) & \phi_4(x,y)\end{bmatrix} \tag{7}$$

$$\mathbf{T} \equiv \begin{bmatrix}C_1 & C_2 & C_3 & C_4\end{bmatrix}^T \tag{8}$$

Likewise, the weight function is also expanded in a series of these shape functions. The gradient of temperature is required in the weak form:

$$\nabla T = \frac{\partial T}{\partial x}\hat{x} + \frac{\partial T}{\partial y}\hat{y} = \mathbf{BT} \tag{9}$$

where the following matrix is defined:

$$\mathbf{B} \equiv \begin{bmatrix}\frac{\partial\phi_1}{\partial x} & \frac{\partial\phi_2}{\partial x} & \frac{\partial\phi_3}{\partial x} & \frac{\partial\phi_4}{\partial x} \\ \frac{\partial\phi_1}{\partial y} & \frac{\partial\phi_2}{\partial y} & \frac{\partial\phi_3}{\partial y} & \frac{\partial\phi_4}{\partial y}\end{bmatrix} \tag{10}$$

The weak form can now be written using these definitions:

$$\int_\Omega k\mathbf{BT}\cdot(\mathbf{B}\Psi)d\Omega + P^* \int_{\Gamma_d} \mathbf{NT}\cdot(\mathbf{N}\Psi)dA = \int_\Omega f\mathbf{N}\Psi d\Omega + \int_{\Gamma_q} \mathbf{q}^*(\mathbf{N}\Psi)dA + P^* \int_{\Gamma_d} T^*(\mathbf{N}\Psi)dA \tag{11}$$

Then, noting that the dot product of two vectors can be written as $\mathbf{a}\cdot\mathbf{b} = \mathbf{b}^T\mathbf{a}$, and noting the equivalence between $\mathbf{N}\Psi$ and $(\mathbf{N}\Psi)^T$:

$$\int_\Omega (\mathbf{B}\Psi)^T k\mathbf{BT}d\Omega + P^* \int_{\Gamma_d} (\mathbf{N}\Psi)^T\mathbf{NT}dA = \int_\Omega f(\mathbf{N}\Psi)^T d\Omega + \int_{\Gamma_q} \mathbf{q}^*(\mathbf{N}\Psi)^T dA + P^* \int_{\Gamma_d} T^*(\mathbf{N}\Psi)^T dA \tag{12}$$

The transpose of a product is:

$$(AB)^T = A^T B^T \tag{13}$$

Using this identity, Ψ^T cancels from every term (to be more exact, the above could be rearranged such that Ψ multiplies every term within an integrand, and that term equals zero, in which case the term multiplied by Ψ must also be zero).

$$\int_\Omega \mathbf{B}^T k \mathbf{B}\mathbf{T} d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T \mathbf{N}\mathbf{T} dA = \int_\Omega f \mathbf{N}^T d\Omega + \int_{\Gamma_q} \mathbf{q}^* \mathbf{N}^T dA + P^* \int_{\Gamma_d} T^* \mathbf{N}^T dA \tag{14}$$

Introducing the definitions of some convenient matrices and vectors:

$$\begin{aligned} \mathbf{K} &\equiv \int_\Omega \mathbf{B}^T k \mathbf{B}\mathbf{T} d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T \mathbf{N}\mathbf{T} dA \\ \mathbf{R} &\equiv \int_\Omega f \mathbf{N}^T d\Omega + \int_{\Gamma_q} \mathbf{q}^* \mathbf{N}^T dA + P^* \int_{\Gamma_d} T^* \mathbf{N}^T dA \end{aligned} \tag{15}$$

Then, the matrix system to be solved reduces to $\mathbf{K}\mathbf{T} = \mathbf{R}$. However, the actual system is solved element-by-element to take advantage of Gaussian quadrature that is defined over a master element. Hence, all the integrals above, which are over the physical domain, must be transformed to integrals over the master domain, defined as $-1 \leq \xi_1 \leq 1$ and $-1 \leq \xi_2 \leq 1$. The coordinates are mapped according to:

$$\begin{aligned} x(\xi_1, \xi_2) &= \sum_{j=1}^{n_{en}} X_j \phi_j(\xi_1, \xi_2) \\ y(\xi_1, \xi_2) &= \sum_{j=1}^{n_{en}} Y_j \phi_j(\xi_1, \xi_2) \end{aligned} \tag{16}$$

where X and Y are the physical coordinates. To transform the integrals, the initial volume integrals over $d\bar{x}$ are transformed to integrals over $d\bar{\xi}$ using:

$$d\bar{x} = \mathbf{F} d\bar{\xi} \tag{17}$$

where $\mathbf{F}$ is the deformation gradient of the transformation, defined as:

$$\mathbf{F} \equiv \begin{bmatrix} \frac{\partial x}{\partial \xi_1} & \frac{\partial x}{\partial \xi_2} \\ \frac{\partial y}{\partial \xi_1} & \frac{\partial y}{\partial \xi_2} \end{bmatrix} \tag{18}$$

This allows the differentials in the integrals to be transformed, but the matrix $\mathbf{B}$, which contains derivatives with respect to the physical coordinates, must also be transformed to a version applying to the master domain:

$$\begin{bmatrix} \frac{\partial \xi_1}{\partial x} & \frac{\partial \xi_2}{\partial x} \\ \frac{\partial \xi_1}{\partial y} & \frac{\partial \xi_2}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial \phi_1}{\partial \xi_1} & \frac{\partial \phi_2}{\partial \xi_1} & \frac{\partial \phi_3}{\partial \xi_1} & \frac{\partial \phi_4}{\partial \xi_1} \\ \frac{\partial \phi_1}{\partial \xi_2} & \frac{\partial \phi_2}{\partial \xi_2} & \frac{\partial \phi_3}{\partial \xi_2} & \frac{\partial \phi_4}{\partial \xi_2} \end{bmatrix} = \begin{bmatrix} \frac{\partial \phi_1}{\partial x} & \frac{\partial \phi_2}{\partial x} & \frac{\partial \phi_3}{\partial x} & \frac{\partial \phi_4}{\partial x} \\ \frac{\partial \phi_1}{\partial y} & \frac{\partial \phi_2}{\partial y} & \frac{\partial \phi_3}{\partial y} & \frac{\partial \phi_4}{\partial y} \end{bmatrix} \tag{19}$$

The first term on the LHS is simply the inverse of the deformation gradient $\mathbf{F}$. The LHS will be denoted as $\mathbf{F}^{-1}\mathcal{B}$. The Jacobian is defined as:

$$\mathcal{J} \equiv \det \mathbf{F} \tag{20}$$

The area integrals must also be transformed using the Jacobian of the area transformation, which does not necessarily equal that of the volume transformation.

$$dA_e = (\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e \tag{21}$$

where $\hat{N}$ and $\hat{n}$ are the unit normals for the surfaces in the master and physical domains. With these definitions, the integrals over the entire domain given in Eq. (15) are transformed to integrals only over each individual element, in the master domain:

$$\begin{aligned}\mathbf{K}^e &\equiv \int_{\Omega_e} (\mathbf{F}^{-1}\mathcal{B})^T k(\mathbf{F}^{-1}\mathcal{B})\mathbf{T}\mathcal{J}d\xi_1 d\xi_2 + P^* \int_{\Gamma_{d,e}} \mathbf{N}^T\mathbf{N}\mathbf{T}(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n}d\xi_v \\ \mathbf{R}^e &\equiv \int_{\Omega_e} f\mathbf{N}^T\mathcal{J}d\xi_1 d\xi_2 + \int_{\Gamma_{q,e}} \mathbf{q}^*\mathbf{N}^T(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n}d\xi_v + P^*\int_{\Gamma_{d,e}} T^*\mathbf{N}^T(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n}d\xi_v\end{aligned} \tag{22}$$

where $d\xi_v$ is one of $d\xi_1$ or $d\xi_2$, depending on the orientation of the surface ("v" stands for "variable"). So, all the integrals in $\mathbf{K}^e$ and $\mathbf{R}^e$ are performed over a single element by transforming the integrals to the master domain. All the integrals above are performed using quadrature. To integrate in higher than a single dimension using a Gaussian quadrature rule, simply apply the rule in each direction. So, in 1-D, where a single loop sums over all the quadrature points, three loops are needed to sum over the ξ_1, ξ_2, ξ_3 directions. For instance, the integrals above, in quadrature form, are:

$$\mathbf{K}^e = \sum_{r=1}^{g}\sum_{s=1}^{g} w_g w_s \left\{(\mathbf{F}^{-1}\mathcal{B})^T k(\mathbf{F}^{-1}\mathcal{B})\mathbf{T}\mathcal{J}\right\} + \underbrace{\sum_{s=1}^{g} w_s \left\{P^*\mathbf{N}^T\mathbf{N}\mathbf{T}(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n}\right\}}_{\text{Dirichlet boundaries only}} \tag{23}$$

$$\mathbf{R}^e \equiv \sum_{r=1}^{g}\sum_{s=1}^{g} w_g w_s f\mathbf{N}^T\mathcal{J} + \sum_{s=1}^{g} w_s \left\{\underbrace{\mathbf{q}^*\mathbf{N}^T(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n}}_{\text{Flux boundaries only}} + \underbrace{P^*T^*\mathbf{N}^T(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n}}_{\text{Dirichlet boundaries only}}\right\}$$

where g is the number of quadrature points, where the same quadrature rule has been assumed to be applied in each spatial dimension. Care must be taken for determining if an element is on a Dirichlet boundary. If it is, then penalty terms must be added, and if not, the penalty terms are absent. So, whether or not the penalty method is used, there is some amount of bookkeeping required to record which nodes correspond to Dirichlet boundaries.

2.1.2 Global-Local Transformation

After all computations over the elements are complete, all the local stiffness matrices and load vectors must be organized into the global stiffness matrix and load vector. The placement of local matrices into the global matrix is performed using a connectivity matrix that relates the local node numbers to the global node numbers. A connectivity matrix (referred to in this document as a location matrix **LM**) is usually organized so that each row corresponds to a single element. Then, each column in that row refers to each local node to that element, and the information held in the connectivity matrix are the global node numbers relating to the local node numbers. For example, for a 2-D domain with four elements, and global node numbering beginning in the bottom left corner and moving up to the top right corner, the location matrix has the following form:

$$\mathbf{LM} = \begin{bmatrix} 1 & 2 & 5 & 4 \\ 2 & 3 & 6 & 5 \\ 4 & 5 & 8 & 7 \\ 5 & 6 & 9 & 8 \end{bmatrix} \tag{24}$$

where the local nodes are numbered in a counterclockwise manner beginning from the bottom left node. Then, for example, the second row in the global stiffness matrix would be assembled as:

$$\mathbf{K}(2,:) = \begin{bmatrix} k_{2,1}^{e=1}, & k_{2,2}^{e=1} + k_{1,1}^{e=2}, & k_{1,2}^{e=2}, & k_{2,4}^{e=1}, & k_{1,4}^{e=2} + k_{2,3}^{e=1}, & k_{1,3}^{e=2}, & 0, & 0, & 0 \end{bmatrix} \tag{25}$$

Then, after the global system is solved, the solution must be postprocessed to give the solution over each element. Over each element, the solution is of the form:

$$T(x,y) = C_1 + C_2 x + C_3 y + C_4 xy \tag{26}$$

where x, y are the physical coordinates in the real domain. Solving the FE problem gives four constants per element that can be used to determine the form of $T(x, y)$ over each element:

$$\begin{bmatrix} T(\text{local node 1}) \\ T(\text{local node 2}) \\ T(\text{local node 3}) \\ T(\text{local node 4}) \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 & x_1 y_1 \\ 1 & x_2 & y_2 & x_2 y_2 \\ 1 & x_3 & y_3 & x_3 y_3 \\ 1 & x_4 & y_4 & x_4 y_4 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix} \tag{27}$$

where the subscripts indicate the local node numberings. This linear system is solved over each element to find the coefficients that determine the solution over that element.

2.2 Shape Functions

The shape functions for 2-D finite elements are a natural extension of the shape functions used in lower dimensions. This choice of shape functions represents a nodal basis, since the shape functions go to zero at all nodes except for the node for which they are defined. These shape functions are defined beginning with the bottom left node and moving in a counterclockwise manner around the master element domain. The bilinear shape functions are:

$$\begin{aligned} \phi_1(\xi_1, \xi_2) &= \frac{1}{4}(1 - \xi_1)(1 - \xi_2) \\ \phi_2(\xi_1, \xi_2) &= \frac{1}{4}(1 + \xi_1)(1 - \xi_2) \\ \phi_3(\xi_1, \xi_2) &= \frac{1}{4}(1 + \xi_1)(1 + \xi_2) \\ \phi_4(\xi_1, \xi_2) &= \frac{1}{4}(1 - \xi_1)(1 + \xi_2) \end{aligned} \tag{28}$$

This gives 4 total unknowns per element, since temperature is a scalar.

3 Mesh Generator

This section discusses the mesh generator used to mesh the "S" structure given in the assignment. The meshing begins in each θ-chunk. For each plane (each plane intersects (0,0) if $r_i = 0$), the x and y coordinates are given by:

$$\begin{aligned} x &= R\cos(\theta) \\ y &= R\sin(\theta) \end{aligned} \tag{29}$$

where R is the radius of the current node. For instance, $R = r_i$ for the very first node, and then in each slice is incremented by $(r_o - r_i)/Nr$ until reaching the outside of the structure. Then, for the next slice, R is reset to r_i. This is repeated, beginning with $\theta = \pi$, and incrementing θ by π/N_θ until completing the structure. The mesh for $N_r = 3, N_\theta = 12$ is shown below. The global node numbering is shown next to each coordinate.

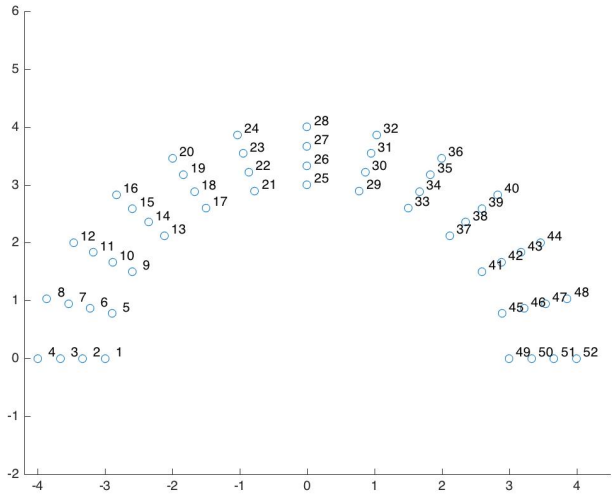

Figure 1. Mesh for $N_r = 3, N_\theta = 12$, with global node numbers given next to each node.

3.1 The Connectivity Matrix

In order for this mesh to be useful for finite element implementation, a connectivity function must be defined to relate the local node numbering to the global node numbering. The connectivity matrix is an $N \times 4$ matrix, where N is the total number of elements and 4 is the number of local nodes per element (bilinear elements are assumed). The local node numbering is performed according to a counterclockwise fashion, beginning with the "bottom left" node of each element, as shown below:

$$\begin{array}{c} 4 - -3 \\ 1 - -2 \end{array} \tag{30}$$

The location matrix for the mesh shown in Fig. 2 is shown below in order to illustrate the node numbering scheme and to be as explicit as possible about the meshing technique.

$$\mathbf{LM} = \begin{bmatrix} 1 & 5 & 6 & 2 \\ 2 & 6 & 7 & 3 \\ 3 & 7 & 8 & 4 \\ 5 & 9 & 10 & 6 \\ 6 & 10 & 11 & 7 \\ 7 & 11 & 12 & 8 \\ 9 & 13 & 14 & 10 \\ 10 & 14 & 15 & 11 \\ 11 & 15 & 16 & 12 \\ 13 & 17 & 18 & 14 \\ 14 & 18 & 19 & 15 \\ 15 & 19 & 20 & 16 \\ 17 & 21 & 22 & 18 \\ 18 & 22 & 23 & 19 \\ 19 & 23 & 24 & 20 \\ 21 & 25 & 26 & 22 \\ 22 & 26 & 27 & 23 \\ 23 & 27 & 28 & 24 \\ 25 & 29 & 30 & 26 \\ 26 & 30 & 31 & 27 \\ 27 & 31 & 32 & 28 \\ 29 & 33 & 34 & 30 \\ 30 & 34 & 35 & 31 \\ 31 & 35 & 36 & 32 \\ 33 & 37 & 38 & 34 \\ 34 & 38 & 39 & 35 \\ 35 & 39 & 40 & 36 \\ 37 & 41 & 42 & 38 \\ 38 & 42 & 43 & 39 \\ 39 & 43 & 44 & 40 \\ 41 & 45 & 46 & 42 \\ 42 & 46 & 47 & 43 \\ 43 & 47 & 48 & 44 \\ 45 & 49 & 50 & 46 \\ 46 & 50 & 51 & 47 \\ 47 & 51 & 52 & 48 \end{bmatrix} \tag{31}$$

3.2 The Analytical Solution

This problem solves the heat conduction problem, with a governing equation given by:

$$\nabla \cdot (k \nabla T) + f = 0 \tag{32}$$

where k is the thermal conductivity, T the temperature (scalar), and f a volumetric heat source/sink. This problem is to be solved with two different sets of specifications. For one of these sets, there is an analytical solution to offer a comparison to ensure that the code functions correctly, and for the second, there is no analytical solution. Both specifications have the following boundary conditions:

$$\begin{aligned} T(\theta = \pi) &= T_0 \\ -k \nabla T \cdot \hat{n}|_{\theta=0} &= q_o(r) \\ -k \nabla T \cdot \hat{n}|_{\theta \neq 0 \cup \theta \neq \pi} &= 0 \end{aligned} \tag{33}$$

For the analytical solution, these specifications are:

$$\begin{aligned} T_o &= 110 \\ q_o(r) &= \frac{20}{r} \\ f(r,\theta) &= \frac{40}{r^2}\sin(2\theta) \end{aligned} \tag{34}$$

To obtain the analytical solution, the governing equation is written explicitly in polar coordinates:

$$\frac{1}{r}\frac{\partial}{\partial r}\left(kr\frac{\partial T}{\partial r}\right) + \frac{1}{r^2}\frac{\partial}{\partial \theta}\left(k\frac{\partial T}{\partial \theta}\right) + \frac{40}{r^2}\sin(2\theta) = 0 \tag{35}$$

where $f(r,\theta)$ has been inserted from Eq. (34). Because k is constant with these specifications, it can be moved outside the derivatives. Because boundary conditions are only asymmetric in the θ direction (i.e. boundary conditions are insulated on the r-sides of the tube), the solution is not a function of r.

$$k\frac{\partial}{\partial \theta}\left(\frac{\partial T}{\partial \theta}\right) + 40\sin(2\theta) = 0 \tag{36}$$

Rearranging, and integrating once in θ:

$$d\left(\frac{dT}{d\theta}\right) = -\frac{40}{k}\sin(2\theta)d\theta \quad \rightarrow \quad \frac{dT}{d\theta} = \frac{20}{k}\cos(2\theta) + C_o \tag{37}$$

Integrating once more:

$$T(\theta) = \frac{10}{k}\sin(2\theta) + C_o\theta + C_1 \tag{38}$$

where C_o and C_1 are constants of integration. These constants are determined by applying the boundary conditions:

$$T(\pi) = T_o = C_o\pi + C_1 \tag{39}$$

$$-k\frac{1}{r}\frac{dT(\theta=0)}{d\theta} = -k\frac{1}{r}\left(\frac{20}{k} + C_o\right) = \frac{20}{r} \tag{40}$$

These equations give:

$$\begin{aligned} C_1 &= T_o - \frac{40}{k}\pi \\ C_o &= 0 \end{aligned} \tag{41}$$

3.3 The FE Solution

This section presents the solution obtained for the test specifications in Section 3.2 for $N_r = 10, N_\theta = 80$. The FE solution is shown with the analytical solution below, where the analytical solution is plotted on the right. As can be seen, there is very good agreement between the results, in particular because a relatively fine mesh was used.

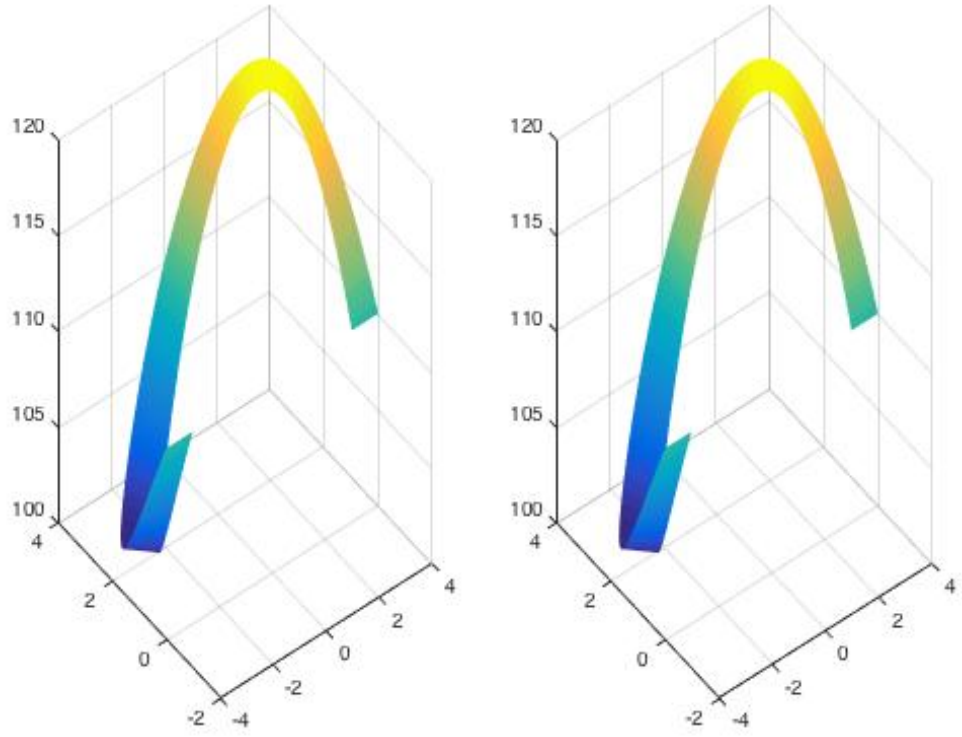

Figure 2. (left) FE solution, and (right) analytical solution for the specifications given in Section 3.2 for $N_r = 40, N_\theta = 100$.

3.4 Impact of the Penalty Term

I did not complete the penalty term investigation, but it would be expected that a very large value of P^* be required to obtain the correct implementation of Dirichlet boundary conditions. Too small a value would only somewhat enforce the Dirichlet boundary conditions, but too large a parameter can also cause difficulties. Because P^* appears in the stiffness matrix and load vectors, if one term is significantly larger than all other terms, or if there is otherwise large discrepancies between the smallest and largest values in a matrix, then Gaussian elimination, or iterative methods as well, can cause numerical errors related to roundoff error, overflow, or underflow.

3.5 The FE Solution, Discontinuous Material Properties

I did not complete the solution for the discontinuous material properties, but all that would be required is to implement another loop within the assembly process for the element stiffness matrix and load vectors. If the global coordinates of the quadrature point (which would be determined using the master-to-physical mapping) are within the circular region, then a different value of k is used. For elements that contain any part within the circular region, a higher-order Gaussian integration scheme is used, but otherwise a 2-point scheme would be used. I plan on having this assignment be my lowest grade, so I'm not completing it so that I can study for my other finals.

4 Appendix

This section contains the complete code used in this assignment.

4.1 BCnodes.m

This function applies boundary conditions.

```
% Script to return the node numbers associated with different types of
% boundary conditions

function [dirichlet_nodes, neumann_nodes, a_k] = BCnodes(left, right,
    ↪ left_value, right_value, num_nodes, Nr, No)

% arrays that hold the nodes in the first row and the values in each column
dirichlet_nodes = [];
neumann_nodes = [];

% assign the nodes to either dirichlet or neumann BCs
i = 1;
switch left
    case 'Dirichlet'
        dirichlet_nodes(1, :) = 1:1:(Nr + 1);
        dirichlet_nodes(2, :) = left_value .* ones(1, Nr + 1);
    case 'Neumann'
        neumann_nodes(1, :) = 1:1:(Nr + 1);
        neumann_nodes(2,:) = left_value .* ones(1, Nr + 1);
    otherwise
        disp('You entered an incorrect field for the type of BC on the
            ↪ boundaries.');
end

i = i + 1;
switch right
    case 'Dirichlet'
        dirichlet_nodes(1, :) = (num_nodes - Nr):1:num_nodes;
        dirichlet_nodes(2, :) = right_value .* ones(1, Nr + 1);
    case 'Neumann'
        neumann_nodes(1, :) = (num_nodes - Nr):1:num_nodes;
        neumann_nodes(2,:) = right_value .* ones(1, Nr + 1);
    otherwise
        disp('You entered an incorrect field for the type of BC on the right
            ↪ boundary.');
end

a_k = [];

if isempty(dirichlet_nodes)
    disp('no dirichlet nodes')
else
    a_k = dirichlet_nodes(2,:)';
end
```

4.2 condensation.m

This program performs static condensation in order to apply Dirichlet nodes.

```
% Performs static condensation and removes Dirichlet nodes from the global
% matrix solve K * a = F
```

```
% To illustrate the process here, assume that the values at the first and
% last nodes (1 and 5) are specified. The other nodes (2, 3, and 4) are
% unknown. For a 5x5 node system, the following matrices are defined:

% K =           K(1,1)  K(1,2)  K(1,3)  K(1,4)  K(1,5)
%               K(2,1)  K(2,2)  K(2,3)  K(2,4)  K(2,5)
%               K(3,1)  K(3,2)  K(3,3)  K(3,4)  K(3,5)
%               K(4,1)  K(4,2)  K(4,3)  K(4,4)  K(4,5)
%               K(5,1)  K(5,2)  K(5,3)  K(5,4)  K(5,5)


% K_uu_rows =   K(2,1)  K(2,2)  K(2,3)  K(2,4)  K(2,5)
%               K(3,1)  K(3,2)  K(3,3)  K(3,4)  K(3,5)
%               K(4,1)  K(4,2)  K(4,3)  K(4,4)  K(4,5)



% K_uu =                K(2,2)  K(2,3)  K(2,4)
%                       K(3,2)  K(3,3)  K(3,4)
%                       K(4,2)  K(4,3)  K(4,4)



% K_uk =        K(2,1)                          K(2,5)
%               K(3,1)                          K(3,5)
%               K(4,1)                          K(4,5)



% K_ku =                K(1,2)  K(1,3)  K(1,4)
%
%
%
%                       K(5,2)  K(5,3)  K(5,4)



% K_kk =        K(1,1)                          K(1,5)
%
%
%
%               K(5,1)                          K(5,5)

function [K_uu, K_uk, F_u, F_k] = condensation(K, F, num_nodes,
    ↪ dirichlet_nodes)

K_uu_rows = zeros(num_nodes - length(dirichlet_nodes(1, :)), num_nodes);
K_uk = zeros(num_nodes - length(dirichlet_nodes(1,:)), length(dirichlet_nodes
    ↪ (1,:)));
F_u = zeros(num_nodes - length(dirichlet_nodes(1, :)), 1);
F_k = zeros(length(dirichlet_nodes(1,:)), 1);

K_row = 1;
```

```
i = 1;          % index for dirichlet_nodes
j = 1;          % index for condensed row
l = 1;          % index for unknown condensed row
m = 1;          % index for known condensed row

for K_row = 1:num_nodes
    if (find(dirichlet_nodes(1, :) == K_row))
        F_k(m) = F(K_row);
        m = m + 1;
        i = i + 1;
    else
        K_uu_rows(j,:) = K(K_row,:);
        F_u(l) = F(K_row);
        j = j + 1;
        l = l + 1;

    end
end

% perform static condensation to remove Dirichlet node columns from solve
K_uu = zeros(num_nodes - length(dirichlet_nodes(1, :)), num_nodes - length(
    ↪ dirichlet_nodes(1, :)));

K_column = 1;
i = 1;              % index for dirichlet nodes
j = 1;              % index for condensed column
m = 1;              % index for K_uk column

for K_column = 1:num_nodes
    if (find(dirichlet_nodes(1, :) == K_column))
        K_uk(:,m) = K_uu_rows(:, K_column);
        m = m + 1;
        i = i + 1;
    else
        K_uu(:,j) = K_uu_rows(:, K_column);
        j = j + 1;
    end
end
```

4.3 FEProgram.m

This program is the master function to perform the FE analysis.

```
clear all

L = pi;                         % problem domain (theta)
k_th = 1;                       % thermal conductivity
shape_order = 2;                % number of nodes per element
E = 0.1;                        % elastic modulus
To = 110;                       % temperature at theta = pi
left = 'Dirichlet';             % boundary condition at theta = pi
left_value = To;                %
right = 'Neumann';              % boundary condition at theta = 0
right_value = 1.0;              % dummy
```

```
fontsize = 16;                          % fontsize for plots
Nr = 1;                                 % number of radial layers
No = 16;                                % number of theta layers
N_elem = Nr * No;                       % number of elements
num_nodes = (Nr + 1) * (No + 1);% number of nodes
num_nodes_per_elem = 4;                 % linear elements
ri = 3;                                 % inner radius of arch
ro = 4;                                 % outer radius of arch
dt = (ro - ri)/Nr;                      % thickness of each radial layer

% form the permutation matrix for assembling the global matrices
[permutation] = permutation(num_nodes_per_elem);

for num_elem = N_elem

    % --- ANALYTICAL SOLUTION --- %
    parent_domain = -1:0.01:1;
    physical_domain = linspace(0, L, num_elem * length(parent_domain) - (
        ↪ num_elem - 1));
    C_o = 0;
    C_1 = To - C_o * pi;

    % for a 2-D mesh polar mesh
    [coordinates, LM] = polar_mesh(No, Nr, dt, num_nodes, ri, ro, num_elem);
    %[plot] = mesh_plots(coordinates, num_nodes, ro, LM);

    % specify the boundary conditions
    [dirichlet_nodes, neumann_nodes, a_k] = BCnodes(left, right, left_value,
        ↪ right_value, num_nodes, Nr, No);

    % define the quadrature rule
    [wt, qp] = quadrature(shape_order);

    % assemble the elemental k and elemental f
    K = zeros(num_nodes);
    F = zeros(num_nodes, 1);

    for elem = 1:num_elem
        k = 0;
        f = 0;

        for ll = 1:length(qp) % eta loop
            for l = 1:length(qp) % xe loop
                [N, dN_dxe, dN_deta, x_xe_eta, y_xe_eta, dx_dxe, dx_deta,
                    ↪  dy_dxe, dy_deta, B] = shapefunctions(qp(l), qp(ll)
                    ↪ , num_nodes_per_elem, coordinates, LM, elem);
                F_mat = transpose([dx_dxe, dx_deta; dy_dxe, dy_deta]);
                J = det(F_mat);
                r = sqrt(x_xe_eta^2 + y_xe_eta^2);
                theta = acos(x_xe_eta / r);

                % assemble the (elemental) forcing vector
                f = f + wt(ll) * wt(l) * (40 * sin(2 * theta) / (r^2)) *
                    ↪ transpose(N) * J;
```

```
                % assemble the (elemental) stiffness matrix - correct
                k = k + wt(ll) * wt(l) * transpose(inv(F_mat) * B) * k_th
                    ↪  * inv(F_mat) * B * J;
            end

            % apply flux boundary conditions (xe is constant, so this is
            % outside of the xe loop). Only the last Nr elements have BCs.
            if (num_elem - Nr) <= elem
                q_flux = 20 / r;

                % in the physical domain
                N_hat = [0, -1];
                % in the master domain
                n_hat = [1, 0]';

                % using both quadrature points (incorrect?) gives good
                % results...
                f = f + wt(ll) * q_flux * transpose(N) * J * (N_hat *
                    ↪ transpose(inv(F_mat)) * n_hat);
            end
        end

        % place the elemental k matrix into the global K matrix
        for m = 1:length(permutation(:,1))
            i = permutation(m,1);
            j = permutation(m,2);
            K(LM(elem, i), LM(elem, j)) = K(LM(elem, i), LM(elem, j)) + k(i,j)
                ↪ ;
        end

        % place the elemental f matrix into the global F matrix
        for i = 1:length(f)
            F(LM(elem, i)) = F((LM(elem, i))) + f(i);
        end
    end

% perform static condensation to remove known Dirichlet nodes from solve
[K_uu, K_uk, F_u, F_k] = condensation(K, F, num_nodes, dirichlet_nodes);

% perform the solve
a_u_condensed = K_uu \ (F_u - K_uk * dirichlet_nodes(2,:)');

% expand a_condensed to include the Dirichlet nodes
a = zeros(num_nodes, 1);

a_row = 1;
i = 1;      % index for dirichlet_nodes
j = 1;      % index for expanded row

for a_row = 1:num_nodes
    if (find(dirichlet_nodes(1, :) == a_row))
        a(a_row) = dirichlet_nodes(2,i);
        i = i + 1;
```

```
    else
        a(a_row) = a_u_condensed(j);
        j = j + 1;
    end
end

% assemble the solution in the physical domain
[mat] = postprocess(num_elem, parent_domain, a, LM, num_nodes_per_elem,
    ↪ shape_order, coordinates, physical_domain);

% subplot(1,2,1)
% start_theta = pi - pi/No;
% end_theta = pi;
% for i = 1:length(mat(:,1))
%     [r, theta] = meshgrid(ri:0.1:ro, linspace(start_theta, end_theta, length
    ↪ (ri:0.1:ro)));
%     x = r .* cos(theta);
%     y = r .* sin(theta);
%     z = mat(i,1) + mat(i,2).*x + mat(i,3).*y + mat(i,4).*x.*y;
%     surf(x,y,z, 'EdgeColor', 'none')
%     hold on
%
%     if (mod(i, Nr) == 0)
%         end_theta = start_theta;
%         start_theta = start_theta - pi/No;
%     end
% end

% analytical solution
subplot(1,2,1)
ylabel('FE_Solution')
start_theta = pi - pi/No;
end_theta = pi;
for i = 1:length(mat(:,1))
    [r, theta] = meshgrid(ri:0.1:ro, linspace(start_theta, end_theta, length(
        ↪ ri:0.1:ro)));
    x = r .* cos(theta);
    y = r .* sin(theta);
    z = 10 * sin(2*theta)/k_th + C_o*theta + C_1;
    surf(x,y,z, 'EdgeColor', 'none')
    hold on

    if (mod(i, Nr) == 0)
        end_theta = start_theta;
        start_theta = start_theta - pi/No;
    end
end

subplot(1,2,2)
ylabel('FE_Solution')
start_theta = pi - pi/No;
end_theta = pi;
for i = 1:length(mat(:,1))
    [r, theta] = meshgrid(ri:0.1:ro, linspace(start_theta, end_theta, length(
```

```
        ↪ ri:0.1:ro)));
    x = r .* cos(theta);
    y = r .* sin(theta);
    z = 10 * sin(2*theta)/k_th + C_o*theta + C_1;
    surf(x,y,z, 'EdgeColor', 'none')
    hold on

    if (mod(i, Nr) == 0)
        end_theta = start_theta;
        start_theta = start_theta - pi/No;
    end
end


end
```

4.4 mesh_plots.m

This program plots the mesh generated.

```
function [plot] = mesh_plots(coordinates, num_nodes, ro, LM)

% mesh of the semi-circle, with node numbering
x = coordinates(:,1);
y = coordinates(:,2);
scatter(x,y)
a = [1:num_nodes]';
b = num2str(a);
c = cellstr(b);
dx = 0.1; dy = 0.1; % displacement so the text does not overlay the data
    ↪ points
text(x+dx, y+dy, c);
xlim([-ro-2*dx, ro+5*dx])
ylim([-ro/2, ro + ro/2])
%saveas(gcf, 'Mesh', 'jpeg')

% output the LM for report
for e = 1:length(LM)
    fprintf('%i & %i & %i & %i \\\\\n', LM(e, 1), LM(e,2), LM(e,3), LM(e,4))
end

plot = 1;
```

4.5 permutation.m

This program generates the permutation matrix for cycling through each element local node.

```
function [permutation] = permutation(num_nodes_per_element)

permutation = zeros(num_nodes_per_element ^ 2, 2);

r = 1;
c = 1;
for i = 1:num_nodes_per_element^2
```

```
    permutation(i,:) = [r, c];
    if c == num_nodes_per_element
        c = 1;
        r = r + 1;
    else
        c = c + 1;
    end
end
```

4.6 polar_mesh.m

This program generates the polar mesh.

```
function [coordinates, LM] = polar_mesh(No, Nr, dt, num_nodes, ri, ro,
    ↪ num_elem)

coordinates = zeros(num_nodes, 2);
theta = pi;
c = 1;  % index for coordinate number

for O = 1:(No + 1)
    % mesh in each radial slice
    for r = 1:(Nr + 1)
        coordinates(c,1) = (ri + dt * (r-1)) * cos(theta);
        coordinates(c,2) = -(ri + dt * (r-1)) * sin(theta);
        c = c + 1;
    end

    % increment theta for next slice
    theta = theta + pi / No;
end

LM = zeros(num_elem, 4);
i = 1;
j = i + (Nr + 1);
l = 1; % index for the layer
for o = 1:No
    for r = l:(l + Nr - 1)
        LM(r,:) = [i, j, j + 1, i + 1];
        i = i + 1;
        j = i + (Nr + 1);
    end
    i = i + 1;
    j = i + (Nr + 1);
    l = l + Nr;
end

end
```

4.7 postprocess.m

This program postprocesses the FE results for plotting.

```
function [coefficients_mat] = postprocess(num_elem, parent_domain, a, LM,
    ↪ num_nodes_per_elem, shape_order, coordinates, physical_domain)

b = zeros(1, num_nodes_per_elem);
A = zeros(num_nodes_per_elem);
coefficients_mat = zeros(num_elem, 4);
%m = length(parent_domain) + 1;
p = 1;

%u_sampled_solution_matrix = zeros(num_elem, length(parent_domain));
%u_sampled_solution_derivative_matrix = zeros(num_elem, length(parent_domain))
    ↪ ;

for elem = 1:num_elem
    % over each element, figure out the polynomial by solving a linear
    % system, Ax = b, where A depends on the order of the shape functions
    for i = 1:num_nodes_per_elem
        b(i) = a(LM(elem, i));
    end

    for j = 1:num_nodes_per_elem % loop over the rows of A
        x_coordinate = coordinates(LM(elem, j), 1);
        y_coordinate = coordinates(LM(elem, j), 2);
        A(j,:) = [1, x_coordinate, y_coordinate, x_coordinate * y_coordinate];
    end

    % solve for the coefficients on the actual polynomial
    coefficients = A\(b');

    % save those coefficients into a matrix (each row is an element)
    coefficients_mat(p,:) = coefficients;
    p = p + 1;

    % determine the solution over the element
%     solution_over_element = zeros(1, length(parent_domain));
%     element_domain = linspace(coordinates(LM(elem, 1)), coordinates(LM(elem,
    ↪  num_nodes_per_elem)), length(parent_domain));
%     for i = 1:num_nodes_per_elem
%         solution_over_element = solution_over_element + coefficients(i) .* (
    ↪ element_domain .^ (i - 1));
%     end

    % determine the derivative over the element
%     derivative_over_element = zeros(1, length(parent_domain));
%     for i = 2:num_nodes_per_elem % the derivative of the constant is zero
%         derivative_over_element = derivative_over_element + coefficients(i)
    ↪ .* (i - 1) .* (element_domain .^ (i - 2));
%     end

    % put into a matrix
%     u_sampled_solution_matrix(p,:) = solution_over_element;
%     u_sampled_solution_derivative_matrix(p,:) = derivative_over_element;
%     p = p + 1;
end
```

```
% assemble solution and derivative into a single vector
% solution_FE = zeros(1, length(physical_domain));
% solution_derivative_FE = zeros(1, length(physical_domain));
% for i = 1:length(u_sampled_solution_matrix(:,1))
%     if i == 1
%         solution_FE(1:length(u_sampled_solution_matrix(i,:))) =
    ↪ u_sampled_solution_matrix(i,:);
%         solution_derivative_FE(1:length(u_sampled_solution_derivative_matrix
    ↪ (i,:))) = u_sampled_solution_derivative_matrix(i,:);
%     else
%         solution_FE(m:(m + length(u_sampled_solution_matrix(1,:)) - 2)) =
    ↪ u_sampled_solution_matrix(i,2:end);
%         solution_derivative_FE(m:(m + length(
    ↪ u_sampled_solution_derivative_matrix(1,:)) - 2)) =
    ↪ u_sampled_solution_derivative_matrix(i,2:end);
%         m = m + length(u_sampled_solution_matrix(1,:)) - 1;
%     end
% end
```

4.8 quadrature.m

This program selects the quadrature rule.

```
function [wt, qp] = quadrature(shape_order)

switch shape_order
    case 2
        wt = [1.0, 1.0];
        qp = [-sqrt(1/3), sqrt(1/3)];
    case 3
        wt = [5/9, 8/9, 5/9];
        qp = [-sqrt(3/5), 0, sqrt(3/5)];
    otherwise
        disp('You entered an unsupported shape function order for the
            ↪ quadrature rule.');
end

wt = [(322-13*sqrt(70))/900, (322+13*sqrt(70))/900, 128/225, (322+13*sqrt(70))
    ↪ /900, (322-13*sqrt(70))/900];
qp = [-(1/3)*sqrt(5+2*sqrt(10/7)), -(1/3)*sqrt(5-2*sqrt(10/7)), 0.0, (1/3)*
    ↪ sqrt(5-2*sqrt(10/7)), (1/3)*sqrt(5+2*sqrt(10/7))];

%wt = [1];
%qp = [0];
```

4.9 shapefunctions.m

This program returns the shape functions, derivatives of shape functions, the Jacobian, and other physical-to-isoparametric quantities needed for computing the stiffness matrix and load vector.

```
function [N, dN_dxe, dN_deta, x_xe_eta, y_xe_eta, dx_dxe, dx_deta, dy_dxe,
    ↪ dy_deta, B] = shapefunctions(xe, eta, num_nodes_per_elem, coordinates,
    ↪ LM, elem)
```

```
N = zeros(num_nodes_per_elem, 1);
dN = zeros(num_nodes_per_elem, 1);

switch num_nodes_per_elem
    case 4
        N(1) = 0.25 * (1 - xe) * (1 - eta);
        N(2) = 0.25 * (1 + xe) * (1 - eta);
        N(3) = 0.25 * (1 + xe) * (1 + eta);
        N(4) = 0.25 * (1 - xe) * (1 + eta);
        dN_dxe(1) = 0.25 * -(1 - eta);
        dN_dxe(2) = 0.25 * (1 - eta);
        dN_dxe(3) = 0.25 * (1 + eta);
        dN_dxe(4) = 0.25 * -(1 + eta);
        dN_deta(1) = 0.25 * (1 - xe) * -1;
        dN_deta(2) = 0.25 * (1 + xe) * -1;
        dN_deta(3) = 0.25 * (1 + xe) * 1;
        dN_deta(4) = 0.25 * (1 - xe) * 1;
        B = [dN_dxe(1), dN_dxe(2), dN_dxe(3), dN_dxe(4); dN_deta(1), dN_deta
            ↪ (2), dN_deta(3), dN_deta(4)];
    otherwise
        disp('You entered an unsupported number of nodes per element.');
end

% x(xe, eta) and y(xe, eta) transformation to the parametric domain
x_xe_eta = 0.0;
y_xe_eta = 0.0;
dx_dxe = 0.0;
dy_dxe = 0.0;
dx_deta = 0.0;
dy_deta = 0.0;

for i = 1:num_nodes_per_elem
    x_xe_eta = x_xe_eta + coordinates(LM(elem, i), 1) * N(i);
    y_xe_eta = y_xe_eta + coordinates(LM(elem, i), 2) * N(i);

    dx_dxe   = dx_dxe   + coordinates(LM(elem, i), 1) * dN_dxe(i);
    dy_dxe   = dy_dxe   + coordinates(LM(elem, i), 2) * dN_dxe(i);

    dx_deta = dx_deta   + coordinates(LM(elem, i), 1) * dN_deta(i);
    dy_deta = dy_deta   + coordinates(LM(elem, i), 2) * dN_deta(i);
end
```