# ME 280a: HW 7

April Novak

November 30, 2016

## 1 Introduction and Objectives

The purpose of this study is to describe the process for solving a 3-D diffusion-reaction equation with no time dependence, and then to describe the differences in the solution method once the problem is allowed to have time dependence. Then, a simple 1-D diffusion-reaction equation is solved as a transient case with Backward Euler (BE) time stepping.

## 2 Procedure

This section details the problem statement and mathematical method used for solving the problem.

### 2.1 Theoretical Problem Statement

Before deriving the governing equation, some preliminaries are necessary. The deformation gradient tensor $\mathbf{F}$ is used to map between two different coordinate systems. One of these coordinates frames is defined by the coordinates $x_i$ (the present coordinates), and the other by $X_i$ (the reference coordinates):

$$\bar{dx} = \mathbf{F}\bar{d\xi} \tag{1}$$

The Jacobian $\mathcal{J}$ is defined as the determinant of the deformation gradient tensor, and is required to transform integrals over the physical domain to the master element for application of quadrature rules:

$$\mathcal{J} \equiv \det \mathbf{F} \tag{2}$$

Finally, to have a physically meaningful transformation between two coordinate frames, the Jacobian must be positive. The diffusion-reaction equation to be solved in this assignment represents a balance between the time rate of change of the concentration in a control volume with the diffusion of mass into the control volume, a reaction that either produces or removes mass from the volume, and any sources of mass:

$$\dot{c} = \nabla \cdot (D\nabla c) - \tau c + f \tag{3}$$

where $c$ is the concentration, an overhead dot represents differentiation in time, $D$ the diffusion coefficient, $\tau$ the reaction rate frequency, and $f$ the concentration source per unit time. This problem is to be solved to determine the concentration. In general, the diffusion coefficient can be an $n$-th order tensor, where $n$ is the number of spatial dimensions, but for this assignment, is assumed to be a scalar, such that the medium is isotropic.

### 2.2 The Weak Form

The weak form of Eq. (3) is obtained by multiplying through by a test function $v$. Then, integrating over the body:

$$\int_\Omega \dot{c} v d\Omega = \int_\Omega \nabla \cdot (D\nabla c) v d\Omega - \int_\Omega \tau c v d\Omega + \int_\Omega f v d\Omega \tag{4}$$

where $\Omega$ is the entire domain. Applying the product rule to the diffusion term:

$$\int_\Omega \dot{c}v d\Omega = -\int_\Omega (\nabla v) D(\nabla c) d\Omega + \int_\Gamma D\nabla c \cdot \hat{n} v d\Gamma - \int_\Omega \tau c v d\Omega + \int_\Omega f v d\Omega \tag{5}$$

where $\Gamma$ is the boundary area of the domain $\Omega$ with unit normal vector $\hat{n}$. The boundary integral applies over the entire domain $\Gamma = \Gamma_q \cup \Gamma_d$, where $\Gamma_q$ is the boundary on which the mass flux is specified (Neumann boundary condition) and $\Gamma_d$ is the boundary on which the concentration is specified (Dirichlet boundary condition). Because the value of the mass flux is unknown on the Dirichlet boundaries, specifying the third term above on the boundaries would be difficult, and hence for simplicity, the shape functions are assumed to equal zero on the Dirichlet boundaries. With this simplification:

$$\int_\Omega \dot{c}v d\Omega = -\int_\Omega (\nabla v) D(\nabla c) d\Omega - \int_{\Gamma_q} q v d\Gamma - \int_\Omega \tau c v d\Omega + \int_\Omega f v d\Omega \tag{6}$$

where $q \equiv -D(\nabla c) \cdot \hat{n}$. Hence, the weak form can be stated as:

$$\begin{aligned} &\text{Find } c \in H^c(\Omega) \subset H^1(\Omega) \text{ so that } c|_{\Gamma_d} = \bar{c} \text{ and so that } \forall\ v \in H^v(\Omega) \subset H^1(\Omega), v|_{\Gamma_d} = 0, \\ &\qquad\qquad \text{and for } q \in L^2(\Gamma_q) \text{ and } f \in L^2(\Omega) \\ &\int_\Omega \dot{c}v d\Omega = -\int_\Omega (\nabla v) D(\nabla c) d\Omega - \int_{\Gamma_q} q v d\Gamma - \int_\Omega \tau c v d\Omega + \int_\Omega f v d\Omega \end{aligned} \tag{7}$$

where $\bar{c}$ is the vector of known concentrations on the displacement boundary. This weak form is more general than the strong form because it does not assume twice-differentiability of the concentration. The particular weighted residual method to be applied is the Bubnov-Galerkin method, where both the solution and the weight function are expanded in the same basis functions. Hence, both the displacement and the weight function are in $H^1(\Omega)$, the space necessary to ensure finite integrals in the weak form above.

The space $H^1(\Omega)$ is a Hilbert-space norm, where the 1 superscript indicates that it contains all functions whose highest finite derivative is the first derivative. This is the space from which the shape and weight functions must come because at most a first derivative is required in the weak form. For other applications, where for example, the highest derivative present in the weak form is a second derivative, then the weight and shape functions would need to be in $H^2(\Omega)$ in order for all integrals to remain finite. In other words, $c$ is in $H^1(\Omega)$ if the following statement is true:

$$\|c\|^2_{H^1(\Omega)} = \int_\Omega \left(\frac{\partial c}{\partial x}\right)^2 d\Omega + \int_\Omega u^2 d\Omega < \infty \tag{8}$$

Because the flux and source also appear in the integrals in Eq. (7), there is a requirement on the space of functions from which they can inhabit. From the weak form, no differentiation of these functions is required, so they must be within $H^0(\Omega)$, sometimes referred to as $L^2(\Omega)$. The weak form in Eq. (7) is equivalent to the strong form provided that the solution is sufficiently differentiable that the higher derivatives required in the strong form are defined. Next, the specifics of the finite element implementation are given in order to specify the above to the finite element method.

### 2.2.1 The Finite Element Weak Form

This section covers the details regarding finite element implementation of Eq. (7). To implement this weak form, first the solution for the concentration $c$ and the weight function $v$ are expanded in a series of shape functions:

$$\begin{aligned} c^h &= \sum_{j=1}^{n_{en}} a_j \phi_j = \quad \mathbf{N}\Phi \\ v^h &= \sum_{i=1}^{n_{en}} b_i \phi_i = \quad \mathbf{N}\Psi \end{aligned} \tag{9}$$

where $a$ are the expansion coefficients to be solved for, $\phi$ are the expansion functions defined in the physical domain, and the $h$ superscript indicates that this approximation occurs over each element, with $n_{en}$ nodes per element. Because $c$ is a scalar, there are $n_{en}$ unknowns per element. The number and order of these shape functions determines the order of the finite element approximation. For convenience, the shape functions and unknowns are grouped into vectors:

$$\mathbf{N} \equiv \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 & \phi_5 & \phi_6 & \phi_7 & \phi_8 \end{bmatrix} \tag{10}$$

$$\begin{aligned} \Phi &\equiv \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \end{bmatrix}^T \\ \Psi &\equiv \begin{bmatrix} b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 \end{bmatrix}^T \end{aligned} \tag{11}$$

where a trilinear approximation has been assumed, such that in 3-D there are 8 nodes per element, with each node having one associated unknown. The gradient of the solution and the weight function are required in the weak form. This can be expressed conveniently as:

$$\begin{aligned} \nabla c &= \frac{\partial c}{\partial x_1}\hat{x_1} + \frac{\partial c}{\partial x_2}\hat{x_2} + \frac{\partial c}{\partial x_3}\hat{x_3} \\ &= \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \frac{\partial \phi_2}{\partial x_1} & \frac{\partial \phi_3}{\partial x_1} & \frac{\partial \phi_4}{\partial x_1} & \frac{\partial \phi_5}{\partial x_1} & \frac{\partial \phi_6}{\partial x_1} & \frac{\partial \phi_7}{\partial x_1} & \frac{\partial \phi_8}{\partial x_1} \\ \frac{\partial \phi_1}{\partial x_2} & \frac{\partial \phi_2}{\partial x_2} & \frac{\partial \phi_3}{\partial x_2} & \frac{\partial \phi_4}{\partial x_2} & \frac{\partial \phi_5}{\partial x_2} & \frac{\partial \phi_6}{\partial x_2} & \frac{\partial \phi_7}{\partial x_2} & \frac{\partial \phi_8}{\partial x_2} \\ \frac{\partial \phi_1}{\partial x_3} & \frac{\partial \phi_2}{\partial x_3} & \frac{\partial \phi_3}{\partial x_3} & \frac{\partial \phi_4}{\partial x_3} & \frac{\partial \phi_5}{\partial x_3} & \frac{\partial \phi_6}{\partial x_3} & \frac{\partial \phi_7}{\partial x_3} & \frac{\partial \phi_8}{\partial x_3} \end{bmatrix} \Phi \\ &= \mathbf{B}\Phi \end{aligned} \tag{12}$$

The gradient of the shape function is defined similarly. Inserting these approximations into the weak form:

$$\int_\Omega \mathbf{N}\dot{\Phi} \cdot (\mathbf{N}\Psi) d\Omega = -\int_\Omega D(\mathbf{B}\Psi) \cdot (\mathbf{B}\Phi) d\Omega - \int_{\Gamma_q} q(\mathbf{N}\Psi) d\Gamma - \int_\Omega \tau(\mathbf{N}\Phi) \cdot (\mathbf{N}\Psi) d\Omega + \int_\Omega f(\mathbf{N}\Psi) d\Omega \tag{13}$$

where it has been assumed that the shape functions are not functions of time, such that the time derivative acts only on the vector of unknowns $\Phi$. The dot product is defined such that $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b}$, and noting the equivalency between $\mathbf{N}\Psi$ and $(\mathbf{N}\Psi)^T$

$$\int_\Omega (\mathbf{N}\Psi)^T \mathbf{N}\dot{\Phi} d\Omega = -\int_\Omega (\mathbf{B}\Psi)^T D(\mathbf{B}\Phi) d\Omega - \int_{\Gamma_q} q(\mathbf{N}\Psi)^T d\Gamma - \int_\Omega \tau(\mathbf{N}\Psi)^T(\mathbf{N}\Phi) d\Omega + \int_\Omega f(\mathbf{N}\Psi)^T d\Omega \tag{14}$$

Then, because $(\mathbf{ab})^T = \mathbf{b}^T\mathbf{a}^T$:

$$\int_\Omega \Psi^T \mathbf{N}^T \mathbf{N}\dot{\Phi} d\Omega = -\int_\Omega \Psi^T \mathbf{B}^T D(\mathbf{B}\Phi) d\Omega - \int_{\Gamma_q} q\Psi^T \mathbf{N}^T d\Gamma - \int_\Omega \tau\Psi^T \mathbf{N}^T(\mathbf{N}\Phi) d\Omega + \int_\Omega f\Psi^T \mathbf{N}^T d\Omega \tag{15}$$

Then, because $\Psi^T$ appears in every term, it can effectively be cancelled (the above could be rearranged so that $\Psi^T$ multiplies a large term, and then if that entire integral must be zero, then the integrand must also be zero).

$$\int_\Omega \mathbf{N}^T \mathbf{N}\dot{\Phi} d\Omega + \int_\Omega \mathbf{B}^T D\mathbf{B}\Phi d\Omega + \int_\Omega \tau\mathbf{N}^T\mathbf{N}\Phi d\Omega = -\int_{\Gamma_q} q\mathbf{N}^T d\Gamma + \int_\Omega f\mathbf{N}^T d\Omega \tag{16}$$

At this point, it is assumed that the system is in steady state. Then, the above reduces to the following:

$$\int_\Omega \mathbf{B}^T D\mathbf{B}\Phi d\Omega + \int_\Omega \tau\mathbf{N}^T\mathbf{N}\Phi d\Omega = -\int_{\Gamma_q} q\mathbf{N}^T d\Gamma + \int_\Omega f\mathbf{N}^T d\Omega \tag{17}$$

For simplicity, the above terms can be defined as matrices:

$$\begin{aligned} \mathbf{K} &\equiv \int_\Omega \mathbf{B}^T D\mathbf{B}\Phi d\Omega + \int_\Omega \tau \mathbf{N}^T\mathbf{N}\Phi d\Omega \\ \mathbf{R} &\equiv -\int_{\Gamma_q} q\mathbf{N}^T d\Gamma + \int_\Omega f\mathbf{N}^T d\Omega \end{aligned} \tag{18}$$

to give the matrix system:

$$\mathbf{K}\Phi = \mathbf{R} \tag{19}$$

where $\Phi$ represents what is to be solved for over the domain. So, the weak form for the time-independent FE method (with Dirichlet boundary conditions applied using static condensation by removing them from the matrix system) is:

Find $\Phi \in \mathbf{H}^\phi(\Omega) \subset \mathbf{H}^1(\Omega)$ so that $\Phi|_{\Gamma_d} = \bar{\Phi}$ and so that $\forall\ \Psi \in \mathbf{H}^\psi(\Omega) \subset \mathbf{H}^1(\Omega), \Psi|_{\Gamma_d} = \mathbf{0}$, and for $q \in L^2(\Gamma_q)$ and $f \in L^2(\Omega)$

$$\int_\Omega \mathbf{B}^T D\mathbf{B}\Phi d\Omega + \int_\Omega \tau \mathbf{N}^T\mathbf{N}\Phi d\Omega = -\int_{\Gamma_q} q\mathbf{N}^T d\Gamma + \int_\Omega f\mathbf{N}^T d\Omega \tag{20}$$

The definitions for all the terms that appear above have been given previously. Section 2.2.2 will show the weak form with the penalty method, while Section 2.3 will show how the above is applied element-by-element.

### 2.2.2 The Finite Element Weak Form - Penalty Method

The penalty method is a means by which to apply Dirichlet boundary conditions without the tedious need to separate rows and columns from the stiffness and loading vectors. From Eq. (20), the weak form for a finite element implementation requires that the weight functions are zero on essential boundaries. The penalty method relaxes this requirement, and adds a term to the weak form to account for violation of a Dirichlet boundary condition on a Dirichlet boundary. This method is widely-used, but is not strictly required to apply Dirichlet boundary conditions - the alternative of separating rows and columns containing known quantities, and subtracting from the load vector, can always be performed. The penalty method adds a term to the weak form of the form $P^* \int_{\Gamma_d} (\mathbf{N}\Psi) \cdot (\bar{c} - \mathbf{N}\Phi) d\Gamma$ into Eq. (20):

Find $\Phi \in \mathbf{H}^\phi(\Omega) \subset \mathbf{H}^1(\Omega)$ so that $\Phi|_{\Gamma_d} = \bar{\Phi}$ and so that $\forall\ \Psi \in \mathbf{H}^\psi(\Omega) \subset \mathbf{H}^1(\Omega)$ and for $q \in L^2(\Gamma_q)$ and $f \in L^2(\Omega)$

$$\int_\Omega \mathbf{B}^T D\mathbf{B}\Phi d\Omega + \int_\Omega \tau \mathbf{N}^T\mathbf{N}\Phi d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T\mathbf{N}\Phi d\Gamma = -\int_{\Gamma_q} q\mathbf{N}^T d\Gamma + \int_\Omega f\mathbf{N}^T d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T \bar{c} d\Gamma \tag{21}$$

where $\bar{c}$ is a vector of known concentrations on the Dirichlet boundary and $P^*$ represents something like a spring constant, and is a large, positive number. A high value of this artificial spring constant will apply a traction to "force" the displacement boundary to be satisfied on $\Gamma_d$, the displacement boundary. In other words, the penalty term represents a traction that enforces the Dirichlet boundary condition. This term, however, would never be applied if we still required $\Psi|_{\Gamma_d} = \mathbf{0}$, and so the kinematic restrictions on the weight functions are dropped, and they do not need to be zero on the Dirichlet boundaries. With this weak form, the matrix equation $\mathbf{K}\Phi = \mathbf{R}$ remains the same, but the contents of $\mathbf{K}$ and $\mathbf{R}$ change to:

$$\begin{aligned} \mathbf{K} &\equiv \int_\Omega \mathbf{B}^T D\mathbf{B} d\Omega + \int_\Omega \tau \mathbf{N}^T\mathbf{N} d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T\mathbf{N} d\Gamma \\ \mathbf{R} &\equiv -\int_{\Gamma_q} q\mathbf{N}^T d\Gamma + \int_\Omega f\mathbf{N}^T d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T \bar{c} d\Gamma \end{aligned} \tag{22}$$

These additional penalty terms are only added for elements that are on Dirichlet boundaries, which is evident from the $\Gamma_d$ appearing in the integral bounds.

## 2.3 Finite Element Implementation

### 2.3.1 Element-by-Element Matrices and Vectors

This section provides in explicit detail the forms of the element stiffness matrices and load vectors using the penalty method (the penalty terms could simply be dropped if needed, and hence the penalty method is discussed here to be as complete as possible). While Eq. (56) holds over the entire domain, the strength of the finite element method is that the integrals in Eq. (56) can be performed over each element, since the shape functions are a nodal basis such that they are only nonzero at a single node. Once assembling into the global stiffness matrix, this gives a sparse system. So, for an element $e$, the element stiffness matrix and load vector are:

$$\begin{aligned} \mathbf{K}^e &\equiv \int_{\Omega_e} \mathbf{B}^T D \mathbf{B} d\Omega + \int_{\Omega_e} \tau \mathbf{N}^T \mathbf{N} d\Omega + P^* \int_{\Gamma_{d,e}} \mathbf{N}^T \mathbf{N} d\Gamma \\ \mathbf{R}^e &\equiv -\int_{\Gamma_{q,e}} q \mathbf{N}^T d\Gamma + \int_{\Omega_e} f \mathbf{N}^T d\Omega + P^* \int_{\Gamma_{d,e}} \mathbf{N}^T \bar{c} d\Gamma \end{aligned} \tag{23}$$

where $\Gamma_{d,e}$ is the intersection of the boundary of element $e$ with the Dirichlet boundary and $\Gamma_{q,e}$ is the intersection of the boundary of element $e$ with the Neumann boundary. All of these integrals are performed in the master domain using quadrature rules. This master domain is a cube defined over $-1 \leq \xi_1 \leq 1, -1 \leq \xi_2 \leq 1, -1 \leq \xi_3 \leq 1$. In order to transform between the physical and master domain, a transformation rule is needed to map between $x, y, z$ and $\xi_1, \xi_2, \xi_3$. This transformation rule can take many forms, but a convenient one is to simply use the shape function expansion:

$$x_i = \sum_{j=1}^{n_{en}} X_{i,j} \phi_j(\xi_1, \xi_2, \xi_3) \tag{24}$$

where the shape functions from here forward are implied to be defined over the master element, $X_{i,j}$ are the physical (real) coordinates, and $i$ refers to the fact that the above expansion is assumed to apply equally for $x_1$, $x_2$ and $x_3$. This mapping, which uses the shape functions as the basis for the mapping, is called a parametric map. 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. The integrals in the physical domain are with respect to $d\bar{x}$, and to transform them to the master domain, the deformation gradient tensor $\mathbf{F}$ defined by Eq. (1) is used. Using the chain rule reveals the form of $\mathbf{F}$:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{\partial x_1}{\partial \xi_1} & \frac{\partial x_1}{\partial \xi_2} & \frac{\partial x_1}{\partial \xi_3} \\ \frac{\partial x_2}{\partial \xi_1} & \frac{\partial x_2}{\partial \xi_2} & \frac{\partial x_2}{\partial \xi_3} \\ \frac{\partial x_3}{\partial \xi_1} & \frac{\partial x_3}{\partial \xi_2} & \frac{\partial x_3}{\partial \xi_3} \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix} \tag{25}$$

Or, in shorthand notation:

$$dx_i = F_{ij} \xi_j \tag{26}$$

where suffix notation is implied. The inverse relationship is:

$$d\xi_j = F_{ji} dx_i = F_{ij}^{-1} dx_i \tag{27}$$

The above transformation rule holds for the volume integral Jacobian - Nanson's formula must be used to transform the area integrals appearing in the element stiffness matrix and load vector to area integrals in the master domain. There is no reason that the Jacobian for the volume transformation be the same as that for the surface transformation, which is why more care must be taken here to use the correct transformation. In this case, area integrals are transformed by Nanson's rule:

$$dA_e = (\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e \tag{28}$$

where $\hat{n}$ is a unit normal to the master element surface (must be determined for each element on the surface), $\hat{N}$ the normal to the physical element surface (must be determined), $\mathcal{J}$ is the Jacobian defined in Eq. (2), and $\mathbf{F}$ is the deformation gradient tensor defined in Eq. (25). Then, all the integrals over areas must set one of $\xi_i$ to $\pm 1$ to be consistent with the fact that on a surface, one of the master coordinates is held constant. So, in order to perform computations over each element, Eq. (29) becomes:

$$\begin{aligned}
\mathbf{K}^e \equiv & \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \mathbf{B}^T D\mathbf{B}|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \tau \mathbf{N}^T\mathbf{N}|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + P^* \int_{\Gamma_{d,e}} \mathbf{N}^T\mathbf{N}(\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e \\
\mathbf{R}^e \equiv & -\int_{\Gamma_{q,e}} q\mathbf{N}^T(\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f\mathbf{N}^T|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + P^* \int_{\Gamma_{d,e}} \mathbf{N}^T \bar{c}(\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e
\end{aligned} \tag{29}$$

where it is implied in the area integrals that one of the spatial coordinates is set to $\pm 1$ according to the orientation of the surface. To accurately perform the integration in the master element, everything within the integrand must also be converted to the master coordinate frame. Hence, $\mathbf{B}$ becomes:

$$\begin{bmatrix} \frac{\partial \xi_1}{\partial x} & \frac{\partial \xi_2}{\partial x} & \frac{\partial \xi_3}{\partial x} \\ \frac{\partial \xi_1}{\partial y} & \frac{\partial \xi_2}{\partial y} & \frac{\partial \xi_3}{\partial y} \\ \frac{\partial \xi_1}{\partial z} & \frac{\partial \xi_2}{\partial z} & \frac{\partial \xi_3}{\partial z} \end{bmatrix} \begin{bmatrix} \frac{\partial \phi_1}{\partial \xi_1} & \frac{\partial \phi_2}{\partial \xi_1} & \frac{\partial \phi_3}{\partial \xi_1} & \frac{\partial \phi_4}{\partial \xi_1} & \frac{\partial \phi_5}{\partial \xi_1} & \frac{\partial \phi_6}{\partial \xi_1} & \frac{\partial \phi_7}{\partial \xi_1} & \frac{\partial \phi_8}{\partial \xi_1} \\ \frac{\partial \phi_1}{\partial \xi_2} & \frac{\partial \phi_2}{\partial \xi_2} & \frac{\partial \phi_3}{\partial \xi_2} & \frac{\partial \phi_4}{\partial \xi_2} & \frac{\partial \phi_5}{\partial \xi_2} & \frac{\partial \phi_6}{\partial \xi_2} & \frac{\partial \phi_7}{\partial \xi_2} & \frac{\partial \phi_8}{\partial \xi_2} \\ \frac{\partial \phi_1}{\partial \xi_3} & \frac{\partial \phi_2}{\partial \xi_3} & \frac{\partial \phi_3}{\partial \xi_3} & \frac{\partial \phi_4}{\partial \xi_3} & \frac{\partial \phi_5}{\partial \xi_3} & \frac{\partial \phi_6}{\partial \xi_3} & \frac{\partial \phi_7}{\partial \xi_3} & \frac{\partial \phi_8}{\partial \xi_3} \end{bmatrix} = \mathbf{B} \tag{30}$$

As mentioned previously, it is implied that all shape functions appearing in this section are defined over the master domain. So, $\mathbf{B}$ is replaced by $\mathbf{F}^{-1}\mathcal{B}$, where $\mathcal{B}$ is the second matrix on the LHS above. This leads to the following element-wise stiffness matrix and load vector:

$$\begin{aligned}
\mathbf{K}^e \equiv & \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} (\mathbf{F}^{-1}\mathcal{B})^T D(\mathbf{F}^{-1}\mathcal{B})|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \tau \mathbf{N}^T\mathbf{N}|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \\
& P^* \int_{\Gamma_{d,e}} \mathbf{N}^T\mathbf{N}(\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e \\
\mathbf{R}^e \equiv & -\int_{\Gamma_{q,e}} q\mathbf{N}^T(\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f\mathbf{N}^T|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \\
& P^* \int_{\Gamma_{d,e}} \mathbf{N}^T \bar{c}(\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e
\end{aligned} \tag{31}$$

The trilinear brick shape functions (in the master domain) are defined in Eq. (39). The unit normals are of length 3, and are oriented such that the correct dimensions are obtained in the matrix and load integrals above. 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 $\xi_1, \xi_2, \xi_3$ directions. For instance, the integrals above, in quadrature form, are:

$$\mathbf{K}^e = \sum_{q=1}^{g}\sum_{r=1}^{g}\sum_{s=1}^{g} w_q w_g w_s \left\{ (\mathbf{F}^{-1}\mathcal{B})^T D(\mathbf{F}^{-1}\mathcal{B})|\mathbf{F}| + \tau \mathbf{N}^T\mathbf{N}|\mathbf{F}| \right\} + \sum_{r=1}^{g}\sum_{s=1}^{g} w_g w_s \left\{ P^* \mathbf{N}^T\mathbf{N}(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n} \right\}$$

$$\mathbf{R}^e \equiv \sum_{q=1}^{g}\sum_{r=1}^{g}\sum_{s=1}^{g} w_q w_g w_s f\mathbf{N}^T|\mathbf{F}| + \sum_{r=1}^{g}\sum_{s=1}^{g} w_g w_s \left\{ -q\mathbf{N}^T(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n} + P^*\mathbf{N}^T\bar{c}(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n} \right\} \tag{32}$$

where $g$ is the number of quadrature points, where the same quadrature rule has been assumed to be applied in each spatial dimension. It is implied that in the area integrals, $d\hat{A}_e$ refers to one of $d\xi_1 d\xi_2$, $d\xi_2 d\xi_3, d\xi_3 d\xi_1$, depending on the particular surface. To be as explicit as possible, there are essentially four types of elements. The element stiffness matrix and load vector for each of these possible combinations is shown below for completeness.

1. an element on the interior (no displacement or traction boundaries)

$$\begin{aligned} \mathbf{K}^e &\equiv \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} (\mathbf{F}^{-1}\mathcal{B})^T D(\mathbf{F}^{-1}\mathcal{B})|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \tau \mathbf{N}^T\mathbf{N}|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 \\ \mathbf{R}^e &\equiv \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f\mathbf{N}^T|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 \end{aligned} \tag{33}$$

2. an element on the boundary with only displacement boundaries

$$\begin{aligned} \mathbf{K}^e &\equiv \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} (\mathbf{F}^{-1}\mathcal{B})^T D(\mathbf{F}^{-1}\mathcal{B})|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \tau \mathbf{N}^T\mathbf{N}|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \\ &\quad P^* \int_{\Gamma_{d,e}} \mathbf{N}^T\mathbf{N}(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n} d\hat{A}_e \\ \mathbf{R}^e &\equiv \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f\mathbf{N}^T|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + P^* \int_{\Gamma_{d,e}} \mathbf{N}^T\bar{c}(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n} d\hat{A}_e \end{aligned} \tag{34}$$

3. an element on the boundary with only traction boundaries

$$\begin{aligned} \mathbf{K}^e &\equiv \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} (\mathbf{F}^{-1}\mathcal{B})^T D(\mathbf{F}^{-1}\mathcal{B})|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \tau \mathbf{N}^T\mathbf{N}|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 \\ \mathbf{R}^e &\equiv -\int_{\Gamma_{q,e}} q\mathbf{N}^T(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n} d\hat{A}_e + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f\mathbf{N}^T|\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 \end{aligned} \tag{35}$$

4. an element on the boundary with both displacement and traction boundaries

$$
\begin{aligned}
\mathbf{K}^e \equiv & \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1}(\mathbf{F}^{-1}\mathcal{B})^T D(\mathbf{F}^{-1}\mathcal{B})|\mathbf{F}|d\xi_1 d\xi_2 d\xi_3 + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1}\tau\mathbf{N}^T\mathbf{N}|\mathbf{F}|d\xi_1 d\xi_2 d\xi_3 + \\
& P^* \int_{\Gamma_{d,e}} \mathbf{N}^T\mathbf{N}(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n}d\hat{A}_e \\
\mathbf{R}^e \equiv & -\int_{\Gamma_{q,e}} q\mathbf{N}^T(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n}d\hat{A}_e + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f\mathbf{N}^T|\mathbf{F}|d\xi_1 d\xi_2 d\xi_3 + \\
& P^* \int_{\Gamma_{d,e}} \mathbf{N}^T\bar{c}(\mathcal{J}\ \mathbf{F}^{-T}\cdot\hat{N})\cdot\hat{n}d\hat{A}_e
\end{aligned}
\tag{36}
$$

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.

### 2.3.2 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 is usually organized so that each row corresponds to a single element. Then, each column in that row refers to each local node in 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 connectivity matrix (also called the location matrix **LM** in this document) has the following form:

$$
\mathbf{LM} = \begin{bmatrix} 1 & 2 & 5 & 4 \\ 2 & 3 & 6 & 5 \\ 4 & 5 & 8 & 7 \\ 5 & 6 & 9 & 8 \end{bmatrix} \tag{37}
$$

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:

$$
\mathbf{K}(2,:) = \begin{bmatrix} k_{2,1}^{e=1}, & k_{2,2}^{e=1} + k_{1,1}^{e=2}, & k_{1,2}^{e=2}, & k_{2,4}^{e=1}, & k_{1,4}^{e=2} + k_{2,3}^{e=1}, & k_{1,3}^{e=2}, & 0, & 0, & 0 \end{bmatrix} \tag{38}
$$

## 2.4 Shape Functions

The shape functions for 3-D finite elements are a natural extension of the shape functions used in lower dimensions. Trilinear elements, or 3-D linear elements, are used for the remainder of this assignment. 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. The trilinear shape functions are:

$$
\begin{aligned}
\phi_1(\xi_1,\xi_2,\xi_3) &= \frac{1}{8}(1-\xi_1)(1-\xi_2)(1-\xi_3) \\
\phi_2(\xi_1,\xi_2,\xi_3) &= \frac{1}{8}(1+\xi_1)(1-\xi_2)(1-\xi_3) \\
\phi_3(\xi_1,\xi_2,\xi_3) &= \frac{1}{8}(1+\xi_1)(1+\xi_2)(1-\xi_3) \\
\phi_4(\xi_1,\xi_2,\xi_3) &= \frac{1}{8}(1-\xi_1)(1+\xi_2)(1-\xi_3) \\
\phi_5(\xi_1,\xi_2,\xi_3) &= \frac{1}{8}(1-\xi_1)(1-\xi_2)(1+\xi_3) \\
\phi_6(\xi_1,\xi_2,\xi_3) &= \frac{1}{8}(1+\xi_1)(1-\xi_2)(1+\xi_3) \\
\phi_7(\xi_1,\xi_2,\xi_3) &= \frac{1}{8}(1+\xi_1)(1+\xi_2)(1+\xi_3) \\
\phi_8(\xi_1,\xi_2,\xi_3) &= \frac{1}{8}(1-\xi_1)(1+\xi_2)(1+\xi_3)
\end{aligned}
\tag{39}
$$

# 3 Mesh Generator

This section discusses the mesh generator used to mesh the tubular "S" structure given in the assignment, and is largely a repeat of the mesh used for assignment 5, except that it has been modified to work for an unknown that is a scalar quantity, rather than a vector quantity. While the mesh is the same as that for assignment 5, the location matrix now does not need to have an associated matrix that relates the three unknowns for each node to their position in the global matrix (here, there is only one unknown per node). The meshing begins in each $\theta$-chunk. For the circular cross-section structure, the $x$ and $y$ coordinates are related to each other by:

$$x^2 + y^2 = r \tag{40}$$

where $r$ is the inner radius of the tubular structure. Each piece in the circumferential direction is defined according to $\theta$, where $0 \leq \theta \leq 2\pi$ defines the "slice" parallel to $\vec{e}_r$. The generation of the mesh is based on determining the coordinates of each node. The first node is assigned to the first "slice" for $\theta = 0$. In the discussion of the mesh generator, "slice" refers to each plane for which there exists a hollowed annulus (that is meshed according to the number of layers and circumferential points). $\Theta$ is the angle referring to the circumferential angle, while $\theta$ refers to the angle in each slice. The overall algorithm for generating the coordinates for the mesh is as follows:

1. Begin with slice for $\Theta = 0$. Beginning then for $\theta = 0$, move counterclockwise around the first layer (inner surface of the tube). Increment $\theta$ in units of $2\pi/N_c$, and for each $\theta$, assign the $x$ and $y$ coordinates according to Eq. (40).

2. After finishing the inner layer, increase $r$ by $dt$, where $dt$ is the thickness of each layer, to move to the next layer, then repeat step 1.

3. Repeat steps 1 and 2 until all layers in each slice have been meshed, where for each layer, $dt$ is added.

4. Now that a slice has been meshed, repeat for all slices. This requires determining the $x, y, z$ centers for each new slice. This is performed by sweeping through $\Theta$. The $y$ coordinate is $y = 0$ for all points, while the $x$ coordinate continually increases moving from slice to slice, and the $z$ coordinate is positive for the left half of the tube, and negative for the right half of the tube.

5. Now that all slices have been meshed, the mesh looks like as follows for $N_c = 8, N_\theta = 8, N_t = 3$.

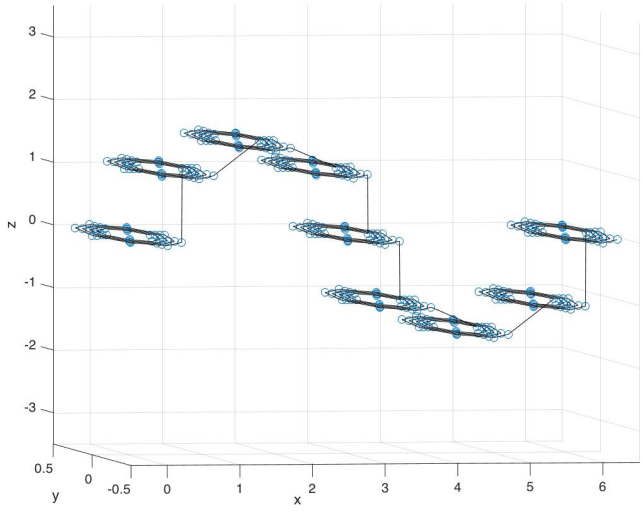


**Figure 1.** Mesh for $N_c = 8, N_\theta = 8, N_t = 3$ with no tilt to the slices. Lines connect each coordinate for better visuality.

The next step is to rotate the slices appropriately through the angle $\Theta$ for each slice so that a tubular structure is formed. Only the $x$ and $z$ coordinates must be modified. To perform the tilt, the following quantities are computed using trigonometry:

$$\begin{aligned} w &= \sin(\pi/2 - \Theta)(r + dt)\cos(\theta)/\sin(pi/2) \\ h &= w\sin(\Theta) \\ p &= w\cos(\Theta) \end{aligned} \tag{41}$$

To tilt the z-coordinate, for points in the first and fourth quadrant of each slice:

$$z_{new} = z - (-1)^{tube} h \tag{42}$$

And for points in the second and third quadrant of each slice:

$$z_{new} = z + (-1)^{tube} h \tag{43}$$

where *tube* is a variable indicating whether or not the slice is in the left or right half of the tube. For the left half of the tube, $tube = 1$, and in the right half, $tube = 2$.

6. Then, tilt the x-coordinates. For points in the first and fourth quadrants of each slice:

$$x_{new} = x + p \tag{44}$$

And for points in the second and third quadrants of each slice:

$$x_{new} = x - p \tag{45}$$

7. Finally, tilt the slices that exactly align with the peak and valley of the tube (for odd numbers of $N_\theta$, this would not be performed). For the slices that align with the peaks and for nodes in the first and fourth quadrants, adjust the $z$ coordinates according to:

$$z_{new} = z - (r + dt)\cos(\theta) \tag{46}$$

And for points in the second and third quadrants:

$$z_{new} = z + (r + dt)\cos(\theta) \tag{47}$$

The $x$-coordinates are adjusted by simply setting all of them to the centroid coordinate for that slice.

This process is fairly complicated, and reveals why meshing software is so valuable. The process here is left fairly general that it can apply for any values of $N_\theta, N_c, N_t$, but any slight change in the geometry completely invalidates the program. The final mesh for $N_c = 8, N_\theta = 8, N_t = 3$ is shown below. This mesh is shown because the requested mesh with only $N_c = 4$ is relatively difficult to perceive in a 3-D plot in Matlab, so the following plot better reveals the mesh. Lines are drawn between each coordinate.

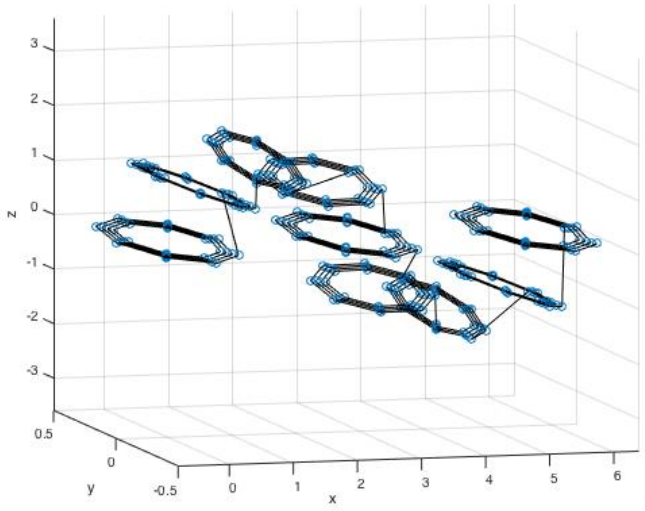


**Figure 2.** Mesh for $N_c = 8, N_\theta = 8, N_t = 3$. Lines connect each coordinate for better visuality.

The coarser mesh, for $N_c = 4, N_\theta = 8, N_t = 3$ is shown below, again with lines connecting each coordinate for better visibility.

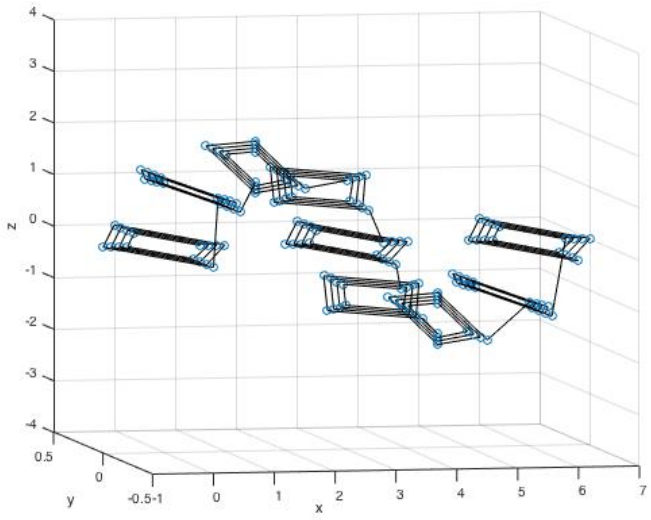


**Figure 3.** Mesh for $N_c = 4, N_\theta = 8, N_t = 3$. Lines connect each coordinate for better visuality.

The mesh in Fig. 3 is shown below without the lines connecting each coordinate.

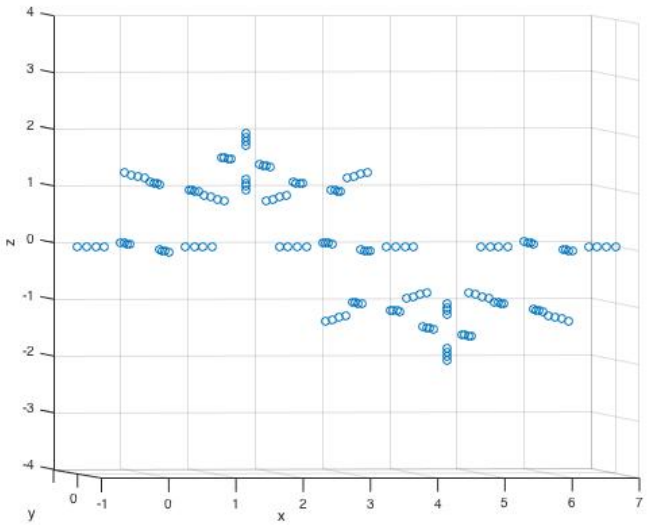


**Figure 4.** Mesh for $N_c = 4, N_\theta = 8, N_t = 3$.

Note that this assignment did not provide the dimensions of the tube, so I assumed that the inner radius of each arch was 1, the radius of the inner hole of the tube 0.3, and the thickness of the tube 0.2.

## 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 mesh generated numbers the global nodes according to the order in which they were generated. For instance, the first 8 nodes are in the inner layer of the first slice, the next 8 are in the second layer of the first slice, and so on for the first slice. Then, moving to the next $\Theta$ slice, the node numbering again begins on the inside of the tube and moves counterclockwise in layers until reaching the outside of the tube. This is shown schematically in the figures above by the black lines connecting the coordinates *in the order in which the coordinates are generated.*

The connectivity matrix is an $N \times 8$ matrix, where $N$ is the total number of elements and 8 is the number of local nodes per element (linear elements are assumed). The local node numbering is performed according to a clockwise fashion. The following schematic shows the node numbering, where the left portion shows the front face, and the right portion shows the back face, all while looking at the front face (i.e. the back face is not written with the perspective of looking at the outward-facing portion of the back face).

$$\begin{array}{cc} 4 - -3 & 8 - -7 \\ 2 - -1 & 6 - -5 \end{array} \tag{48}$$

So, for each slice, the nodes on the face of each element can be determined using a numbering scheme that follows the order in which the nodes were defined. Beginning with $\theta = 0$, and moving counterclockwise, the local nodes are numbered, moving progressively outwards in the layers until reaching the last node for a particular slice.

There are $N_t \cdot N_c \cdot N_\theta$ total elements. For each slice, the nodes are numbered moving counterclockwise, beginning at the same node that is meshed first. After each layer is complete, the numbering moves to the next layer in the same fashion. Once an entire slice is complete, the next slice is also meshed. This defines only the *frontal* node numberings shown in the above equation. For example, for $N_\theta = 2, N_t = 3, N_c = 4$, the connectivity matrix **LM** looks like the following *before* the nodes on the backs of the first 12 elements are related to the nodes on the fronts of the next 12 elements.

$$\mathbf{LM} = \begin{bmatrix}
1 & 2 & 5 & 6 & 0 & 0 & 0 & 0 \\
2 & 3 & 6 & 7 & 0 & 0 & 0 & 0 \\
3 & 4 & 7 & 8 & 0 & 0 & 0 & 0 \\
4 & 1 & 8 & 5 & 0 & 0 & 0 & 0 \\
5 & 6 & 9 & 10 & 0 & 0 & 0 & 0 \\
6 & 7 & 10 & 11 & 0 & 0 & 0 & 0 \\
7 & 8 & 11 & 12 & 0 & 0 & 0 & 0 \\
8 & 5 & 12 & 9 & 0 & 0 & 0 & 0 \\
9 & 10 & 13 & 14 & 0 & 0 & 0 & 0 \\
10 & 11 & 14 & 15 & 0 & 0 & 0 & 0 \\
11 & 12 & 15 & 16 & 0 & 0 & 0 & 0 \\
12 & 9 & 16 & 13 & 0 & 0 & 0 & 0 \\
17 & 18 & 21 & 22 & 0 & 0 & 0 & 0 \\
18 & 19 & 22 & 23 & 0 & 0 & 0 & 0 \\
19 & 20 & 23 & 24 & 0 & 0 & 0 & 0 \\
20 & 17 & 24 & 21 & 0 & 0 & 0 & 0 \\
21 & 22 & 25 & 26 & 0 & 0 & 0 & 0 \\
22 & 23 & 26 & 27 & 0 & 0 & 0 & 0 \\
23 & 24 & 27 & 28 & 0 & 0 & 0 & 0 \\
24 & 21 & 28 & 25 & 0 & 0 & 0 & 0 \\
25 & 26 & 29 & 30 & 0 & 0 & 0 & 0 \\
26 & 27 & 30 & 31 & 0 & 0 & 0 & 0 \\
27 & 28 & 31 & 32 & 0 & 0 & 0 & 0 \\
28 & 25 & 32 & 29 & 0 & 0 & 0 & 0 \\
33 & 34 & 37 & 38 & 0 & 0 & 0 & 0 \\
34 & 35 & 38 & 39 & 0 & 0 & 0 & 0 \\
35 & 36 & 39 & 40 & 0 & 0 & 0 & 0 \\
36 & 33 & 40 & 37 & 0 & 0 & 0 & 0 \\
37 & 38 & 41 & 42 & 0 & 0 & 0 & 0 \\
38 & 39 & 42 & 43 & 0 & 0 & 0 & 0 \\
39 & 40 & 43 & 44 & 0 & 0 & 0 & 0 \\
40 & 37 & 44 & 41 & 0 & 0 & 0 & 0 \\
41 & 42 & 45 & 46 & 0 & 0 & 0 & 0 \\
42 & 43 & 46 & 47 & 0 & 0 & 0 & 0 \\
43 & 44 & 47 & 48 & 0 & 0 & 0 & 0 \\
44 & 41 & 48 & 45 & 0 & 0 & 0 & 0
\end{bmatrix} \tag{49}$$

This is not the final form for the connectivity matrix (a.k.a. location matrix). Because the slices lay exactly on top of one another, the front nodes of the second slice are exactly the back nodes on the previous slice. With this knowledge, the back nodes for each element can be assigned based on the frontal nodes of the following slice. Then, the last $N_c N_t$ rows in the location matrix above can be deleted, since they refer to the frontal nodes of a slice that does not technically exist (there are only 2 slices, but 3 planes defining those slices). With this information, the final form of the location matrix becomes, for $N_\theta = 2, N_t = 3, N_c = 4$ for example:

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

Each row in the location matrix corresponds to an elements, and each column to a local node number, so that $LM(1,4)$ indicates the global node number of local node number 4 in element 1. This method is extended to the case for $N_\theta = 8, N_c = 4, N_t = 3$, where the purpose of the previous discussion for a fewer number of circumferential elements was simply to illustrate the process by which the location matrix is generated. So, for the problem statement in this homework assignment ($N_\theta = 8, N_c = 4, N_t = 3$):

$$\mathbf{LM} = \begin{bmatrix} LM_1 \\ LM_2 \end{bmatrix} \tag{51}$$

where, in order to be able to print the matrix, the following components are defined to simply be stacked on top of each other as in Eq. (51).

$$LM_1 = \begin{bmatrix}
1 & 2 & 5 & 6 & 17 & 18 & 21 & 22 \\
2 & 3 & 6 & 7 & 18 & 19 & 22 & 23 \\
3 & 4 & 7 & 8 & 19 & 20 & 23 & 24 \\
4 & 1 & 8 & 5 & 20 & 17 & 24 & 21 \\
5 & 6 & 9 & 10 & 21 & 22 & 25 & 26 \\
6 & 7 & 10 & 11 & 22 & 23 & 26 & 27 \\
7 & 8 & 11 & 12 & 23 & 24 & 27 & 28 \\
8 & 5 & 12 & 9 & 24 & 21 & 28 & 25 \\
9 & 10 & 13 & 14 & 25 & 26 & 29 & 30 \\
10 & 11 & 14 & 15 & 26 & 27 & 30 & 31 \\
11 & 12 & 15 & 16 & 27 & 28 & 31 & 32 \\
12 & 9 & 16 & 13 & 28 & 25 & 32 & 29 \\
17 & 18 & 21 & 22 & 33 & 34 & 37 & 38 \\
18 & 19 & 22 & 23 & 34 & 35 & 38 & 39 \\
19 & 20 & 23 & 24 & 35 & 36 & 39 & 40 \\
20 & 17 & 24 & 21 & 36 & 33 & 40 & 37 \\
21 & 22 & 25 & 26 & 37 & 38 & 41 & 42 \\
22 & 23 & 26 & 27 & 38 & 39 & 42 & 43 \\
23 & 24 & 27 & 28 & 39 & 40 & 43 & 44 \\
24 & 21 & 28 & 25 & 40 & 37 & 44 & 41 \\
25 & 26 & 29 & 30 & 41 & 42 & 45 & 46 \\
26 & 27 & 30 & 31 & 42 & 43 & 46 & 47 \\
27 & 28 & 31 & 32 & 43 & 44 & 47 & 48 \\
28 & 25 & 32 & 29 & 44 & 41 & 48 & 45 \\
33 & 34 & 37 & 38 & 49 & 50 & 53 & 54 \\
34 & 35 & 38 & 39 & 50 & 51 & 54 & 55 \\
35 & 36 & 39 & 40 & 51 & 52 & 55 & 56 \\
36 & 33 & 40 & 37 & 52 & 49 & 56 & 53 \\
37 & 38 & 41 & 42 & 53 & 54 & 57 & 58 \\
38 & 39 & 42 & 43 & 54 & 55 & 58 & 59 \\
39 & 40 & 43 & 44 & 55 & 56 & 59 & 60 \\
40 & 37 & 44 & 41 & 56 & 53 & 60 & 57 \\
41 & 42 & 45 & 46 & 57 & 58 & 61 & 62 \\
42 & 43 & 46 & 47 & 58 & 59 & 62 & 63 \\
43 & 44 & 47 & 48 & 59 & 60 & 63 & 64 \\
44 & 41 & 48 & 45 & 60 & 57 & 64 & 61 \\
49 & 50 & 53 & 54 & 65 & 66 & 69 & 70 \\
50 & 51 & 54 & 55 & 66 & 67 & 70 & 71 \\
51 & 52 & 55 & 56 & 67 & 68 & 71 & 72 \\
52 & 49 & 56 & 53 & 68 & 65 & 72 & 69 \\
53 & 54 & 57 & 58 & 69 & 70 & 73 & 74 \\
54 & 55 & 58 & 59 & 70 & 71 & 74 & 75 \\
55 & 56 & 59 & 60 & 71 & 72 & 75 & 76 \\
56 & 53 & 60 & 57 & 72 & 69 & 76 & 73 \\
57 & 58 & 61 & 62 & 73 & 74 & 77 & 78 \\
58 & 59 & 62 & 63 & 74 & 75 & 78 & 79 \\
59 & 60 & 63 & 64 & 75 & 76 & 79 & 80 \\
60 & 57 & 64 & 61 & 76 & 73 & 80 & 77 \\
65 & 66 & 69 & 70 & 81 & 82 & 85 & 86
\end{bmatrix} \tag{52}$$

$$LM_2 = \begin{bmatrix}
66 & 67 & 70 & 71 & 82 & 83 & 86 & 87 \\
67 & 68 & 71 & 72 & 83 & 84 & 87 & 88 \\
68 & 65 & 72 & 69 & 84 & 81 & 88 & 85 \\
69 & 70 & 73 & 74 & 85 & 86 & 89 & 90 \\
70 & 71 & 74 & 75 & 86 & 87 & 90 & 91 \\
71 & 72 & 75 & 76 & 87 & 88 & 91 & 92 \\
72 & 69 & 76 & 73 & 88 & 85 & 92 & 89 \\
73 & 74 & 77 & 78 & 89 & 90 & 93 & 94 \\
74 & 75 & 78 & 79 & 90 & 91 & 94 & 95 \\
75 & 76 & 79 & 80 & 91 & 92 & 95 & 96 \\
76 & 73 & 80 & 77 & 92 & 89 & 96 & 93 \\
81 & 82 & 85 & 86 & 97 & 98 & 101 & 102 \\
82 & 83 & 86 & 87 & 98 & 99 & 102 & 103 \\
83 & 84 & 87 & 88 & 99 & 100 & 103 & 104 \\
84 & 81 & 88 & 85 & 100 & 97 & 104 & 101 \\
85 & 86 & 89 & 90 & 101 & 102 & 105 & 106 \\
86 & 87 & 90 & 91 & 102 & 103 & 106 & 107 \\
87 & 88 & 91 & 92 & 103 & 104 & 107 & 108 \\
88 & 85 & 92 & 89 & 104 & 101 & 108 & 105 \\
89 & 90 & 93 & 94 & 105 & 106 & 109 & 110 \\
90 & 91 & 94 & 95 & 106 & 107 & 110 & 111 \\
91 & 92 & 95 & 96 & 107 & 108 & 111 & 112 \\
92 & 89 & 96 & 93 & 108 & 105 & 112 & 109 \\
97 & 98 & 101 & 102 & 113 & 114 & 117 & 118 \\
98 & 99 & 102 & 103 & 114 & 115 & 118 & 119 \\
99 & 100 & 103 & 104 & 115 & 116 & 119 & 120 \\
100 & 97 & 104 & 101 & 116 & 113 & 120 & 117 \\
101 & 102 & 105 & 106 & 117 & 118 & 121 & 122 \\
102 & 103 & 106 & 107 & 118 & 119 & 122 & 123 \\
103 & 104 & 107 & 108 & 119 & 120 & 123 & 124 \\
104 & 101 & 108 & 105 & 120 & 117 & 124 & 121 \\
105 & 106 & 109 & 110 & 121 & 122 & 125 & 126 \\
106 & 107 & 110 & 111 & 122 & 123 & 126 & 127 \\
107 & 108 & 111 & 112 & 123 & 124 & 127 & 128 \\
108 & 105 & 112 & 109 & 124 & 121 & 128 & 125 \\
113 & 114 & 117 & 118 & 129 & 130 & 133 & 134 \\
114 & 115 & 118 & 119 & 130 & 131 & 134 & 135 \\
115 & 116 & 119 & 120 & 131 & 132 & 135 & 136 \\
116 & 113 & 120 & 117 & 132 & 129 & 136 & 133 \\
117 & 118 & 121 & 122 & 133 & 134 & 137 & 138 \\
118 & 119 & 122 & 123 & 134 & 135 & 138 & 139 \\
119 & 120 & 123 & 124 & 135 & 136 & 139 & 140 \\
120 & 117 & 124 & 121 & 136 & 133 & 140 & 137 \\
121 & 122 & 125 & 126 & 137 & 138 & 141 & 142 \\
122 & 123 & 126 & 127 & 138 & 139 & 142 & 143 \\
123 & 124 & 127 & 128 & 139 & 140 & 143 & 144 \\
124 & 121 & 128 & 125 & 140 & 137 & 144 & 141
\end{bmatrix} \tag{53}$$

# 4 Computational Cost

The cost of a 3-D finite element simulation of a scalar-valued equation such as the diffusion-reaction equation results in element stiffness matrices of size $8 \times 8$ when using trilinear shape functions. For a cubical mesh, with $M \times M$ elements, there are $(M+1)^3$ total unknowns in the mesh, so the global stiffness matrix

is of size $((M+1)^3) \times ((M+1)^3)$. If the symmetry of the local stiffness matrices is taken into account (they will always be symmetric for linear differential equations so long as the Bubnov-Galerkin approach is used), then the required storage per element drops from 64 to 28.

If the Conjugate Gradient (CG) method is used to solve the matrix system $\mathbf{Ka} = \mathbf{R}$, then this represents repeated application of a $8 \times 8$ matrix and a $8 \times 1$ vector, which is an $\mathcal{O}(N)$ operation. This is performed for each iteration, so the cost for the CG method scales as $\mathcal{O}(IN)$, where $I$ is the number of iterations needed to reach a particular tolerance in the solve.

The cost of a solve refers to both the storage required and the number of floating point operations required. There are three ways to perform the storage for a mesh consisting of $M \times M \times M$ linear elements:

1. Direct storage - all zeros are stored, and no shortcuts are made by saving element-by-element. This requires $((M+1)^3) \times ((M+1)^3) \approx M^6$
2. Element-by-element storage: no zeros are stored. This requires $8 \times 8 \times M^3 = 64M^3$
3. Element-by-element storage, taking advantage of the symmetry of the element stiffness matrices. This requires $28M^3$

So, the storage scales cubically when using element-by-element storage as opposed to direct storage. The number of floating point operations is also significantly reduced when using an iterative solver. The total number of unknowns is $(M+1)^3$, so the ratio of the direct solver (Gaussian elimination) to the CG method is:

$$\frac{\mathcal{O}(N^3)}{I\mathcal{O}(N)} = \frac{\left((M+1)^3\right)^2}{I} = \frac{(M+1)^6}{I} \tag{54}$$

where $N = (M+1)^3$ is the total number of unknowns. For $N_t \times N_c \times N_\theta$ elements, there are $(N_t + 1)N_c(N_\theta + 1)$ total nodes. So, if there are 3 unknowns per node, and if a CG solver takes $I\mathcal{O}(N)$ operations, then the total number of operations required is:

$$\text{Operations with CG method} = I(N_t+1)N_c(N_\theta+1) \tag{55}$$

where $I$ is the number of iterations.

# 5 Time-Dependent Implementation

This section will describe the FE solution to the transient problem in Eq. (3), beginning from Eq. (56). The time term that was last included in Eq. (16) can simply be added back into the stiffness matrix portion:

$$\begin{aligned} &\text{Find } \Phi \in \mathbf{H}^{\phi}(\Omega) \subset \mathbf{H}^1(\Omega) \text{ so that } \Phi|_{\Gamma_d} = \bar{\Phi}, \Phi(t=0) = \bar{c_0} \\ &\text{and so that } \forall\ \Psi \in \mathbf{H}^{\psi}(\Omega) \subset \mathbf{H}^1(\Omega) \text{ and for } q \in L^2(\Gamma_q) \text{ and } f \in L^2(\Omega) \\ &\int_\Omega \mathbf{N}^T\mathbf{N}\dot{\Phi}d\Omega + \int_\Omega \mathbf{B}^T D\mathbf{B}\Phi d\Omega + \int_\Omega \tau\mathbf{N}^T\mathbf{N}\Phi d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T\mathbf{N}\Phi d\Gamma = \\ &\qquad - \int_{\Gamma_q} q\mathbf{N}^T d\Gamma + \int_\Omega f\mathbf{N}^T d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T\bar{c} d\Gamma \end{aligned} \tag{56}$$

where now the only difference is that the time term is included in the weak form, and included in the weak form is the requirement that the solution satisfy the initial condition. All computations occur over each individual element, and the weak form above can be transformed to integrals in the master domain following the procedure described previously. Then, the matrix system to be solved becomes:

$$\mathbf{M}\dot{\Phi} + \mathbf{K}\Phi = \mathbf{R} \tag{57}$$

where these matrices are defined as follows over the entire domain:

$$
\begin{aligned}
\mathbf{M} &\equiv \int_{\Omega} \mathbf{N}^T \mathbf{N} d\Omega \\
\mathbf{K} &\equiv \int_{\Omega} \mathbf{B}^T D \mathbf{B} d\Omega + \int_{\Omega} \tau \mathbf{N}^T \mathbf{N} d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T \mathbf{N} d\Gamma \\
\mathbf{R} &\equiv -\int_{\Gamma_q} q \mathbf{N}^T d\Gamma + \int_{\Omega} f \mathbf{N}^T d\Omega + P^* \int_{\Gamma_d} \mathbf{N}^T \bar{c} d\Gamma
\end{aligned} \tag{58}
$$

Over each element, these matrices are as follows:

$$
\begin{aligned}
\mathbf{M}^e &\equiv \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \mathbf{N}^T \mathbf{N} |\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 \\
\mathbf{K}^e &\equiv \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} (\mathbf{F}^{-1}\mathcal{B})^T D (\mathbf{F}^{-1}\mathcal{B}) |\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \tau \mathbf{N}^T \mathbf{N} |\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \\
&\quad P^* \int_{\Gamma_{d,e}} \mathbf{N}^T \mathbf{N} (\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e \\
\mathbf{R}^e &\equiv -\int_{\Gamma_{q,e}} q \mathbf{N}^T (\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e + \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f \mathbf{N}^T |\mathbf{F}| d\xi_1 d\xi_2 d\xi_3 + \\
&\quad P^* \int_{\Gamma_{d,e}} \mathbf{N}^T \bar{c} (\mathcal{J}\ \mathbf{F}^{-T} \cdot \hat{N}) \cdot \hat{n} d\hat{A}_e
\end{aligned} \tag{59}
$$

As can be seen, the element stiffness matrix and load vectors do not change, but rather a mass matrix $\mathbf{M}$ must now be computed for each element. The local mass matrices are assembled into this matrix by the same procedure as described in Section 2.3.2, and the location matrix that is used to populate the global stiffness matrix is equivalently used to populate the global mass matrix. However, the procedure up until now is not fully complete, since an approximation is needed to describe how the value of the FE solution changes in time, since separable space-time finite elements are used. Taylor series are used to derive several common finite difference approximations to the first derivative that appears in this equation. Expanding a quantity $a$ in a Taylor series about $t + \Delta t$:

$$
a(t + \Delta t) \approx a(t) + \frac{da}{dt}\Delta t + \cdots \tag{60}
$$

If the higher order terms indicates by the ellipses above are neglected, an approximation to the first derivative is obtained:

$$
\frac{da}{dt} \approx \frac{a(t + \Delta t) - a(t)}{\Delta t} \tag{61}
$$

Because the highest neglected term is of order $\Delta t^2$, this discretization scheme is a first-order scheme. The above discretization scheme is termed an Euler method, and can be either implicit or explicit, depending on the point in time at which $da/dt$ is to be evaluated. If $da/dt$ is evaluated at $t+\Delta t$, then the scheme is implicit, and is unconditionally stable. If evaluated at $t$, then the scheme is explicit, which is conditionally stable for sufficiently small time steps, which may be a large limitation on the computational cost of a simulation. Using this first-order approximation to the first derivative in the governing equations, some options for time discretization are:

$$
\begin{aligned}
\mathbf{M}\frac{\Phi^{k+1} - \Phi^k}{\Delta t} + \mathbf{K}\Phi^k = \mathbf{R} \qquad &\text{explicit} \\
\mathbf{M}\frac{\Phi^{k+1} - \Phi^k}{\Delta t} + \mathbf{K}\Phi^{k+1} = \mathbf{R} \qquad &\text{implicit}
\end{aligned} \tag{62}
$$

where $k$ indicates the time step index. The implicit method is to be used in this assignment, so rearranging the second line above in order to isolate $\Phi^{k+1}$

$$
\left(\frac{1}{\Delta t}\mathbf{M} + \mathbf{K}\right)\phi^{k+1} = R + \frac{1}{\Delta t}\mathbf{M}\Phi^k \tag{63}
$$

This allows a time-marching scheme, where it is assumed that $\Phi^k$ is always known because an initial condition is given. So, the general algorithm is a small modification to the assignment developed for the first homework. First, apply the initial condition to find $\Phi^1$. Then, solve the above equation in a loop to find the $\Phi$ corresponding to each subsequent time step, until the end of the simulation. The above equation is solved for the global matrices, so for each time step, the entire finite element implementation is performed.

For the 1-D problem for which this is to be applied, the element mass and stiffness matrices and the element load vector simplify considerably. For simplicity, this problem is not solved with the penalty method (static condensation, or the process of removing Dirichlet nodes from the matrix system and load vector, is performed instead). For 1-D and with no forcing term, Eq. (59) simplifies to:

$$\begin{aligned} \mathbf{M}^e_{ij} &\equiv \int_{-1}^{1} \phi_i \phi_j \mathcal{J} d\xi \\ \mathbf{K}^e_{ij} &\equiv \int_{-1}^{1} \frac{d\phi_i}{d\xi} D \frac{d\phi_j}{d\xi} \frac{1}{\mathcal{J}} d\xi + \int_{-1}^{1} \tau \phi_i \phi_j \mathcal{J} d\xi \\ \mathbf{R}^e_j &\equiv -q\phi_j \Big|_0^L \end{aligned} \tag{64}$$

where the Jacobian $\mathcal{J} = dx/d\xi$. When not using the penalty method, the Dirichlet nodes must be removed from the solve, or else a singular system will result. So, the above is solved as follows, where "u" indicates a node whose value is unknown (not a Dirichlet node), while "k" indicates a node whose value is known (a Dirichlet node):

$$\left(\frac{1}{\Delta t} M_{uu} + K_{uu}\right) \phi_u^{k+1} + \left(\frac{1}{\Delta t} M_{uk} + K_{uk}\right) \phi_k^{k+1} = R_u + \frac{1}{\Delta t}\left(M_{uu}\phi_u^k + M_{uk}\phi_k^k\right) \tag{65}$$

where the $k$ that appears in the superscript indicates iteration number. Rearranging the above gives the actual equation that is solved at each time step:

$$\phi_u^{k+1} = \left(\frac{1}{\Delta t} M_{uu} + K_{uu}\right)^{-1} \left[R_u + \frac{1}{\Delta t}\left(M_{uu}\phi_u^k + M_{uk}\phi_k^k\right) - \left(\frac{1}{\Delta t} M_{uk} + K_{uk}\right) \phi_k^{k+1}\right] \tag{66}$$

Because the material properties are constant in time, and because separable space-time finite elements are used, the matrices above only need to be computed once, and then simply are re-applied in a loop until reaching the final time.

# 6 Results

This section shows the results for the 1-D diffusion-reaction equation for an end time of 6.5e3 seconds. The three plots that follow show the time evolution. Fig. 5 shows the results for 100 time steps, Fig. 6 for 1000 time steps, and Fig. 7 for 10000 time steps. The time progression is shown beginning with red color early in the transient and progressing to purple/blue towards the end of the time period. The blue line shows the initial condition, and the black line shows the solution at the last time step (the final time solution). Also plotted is the flux $-Ddc/dx$ as a function of time to show how the Neumann boundary condition is satisfied.

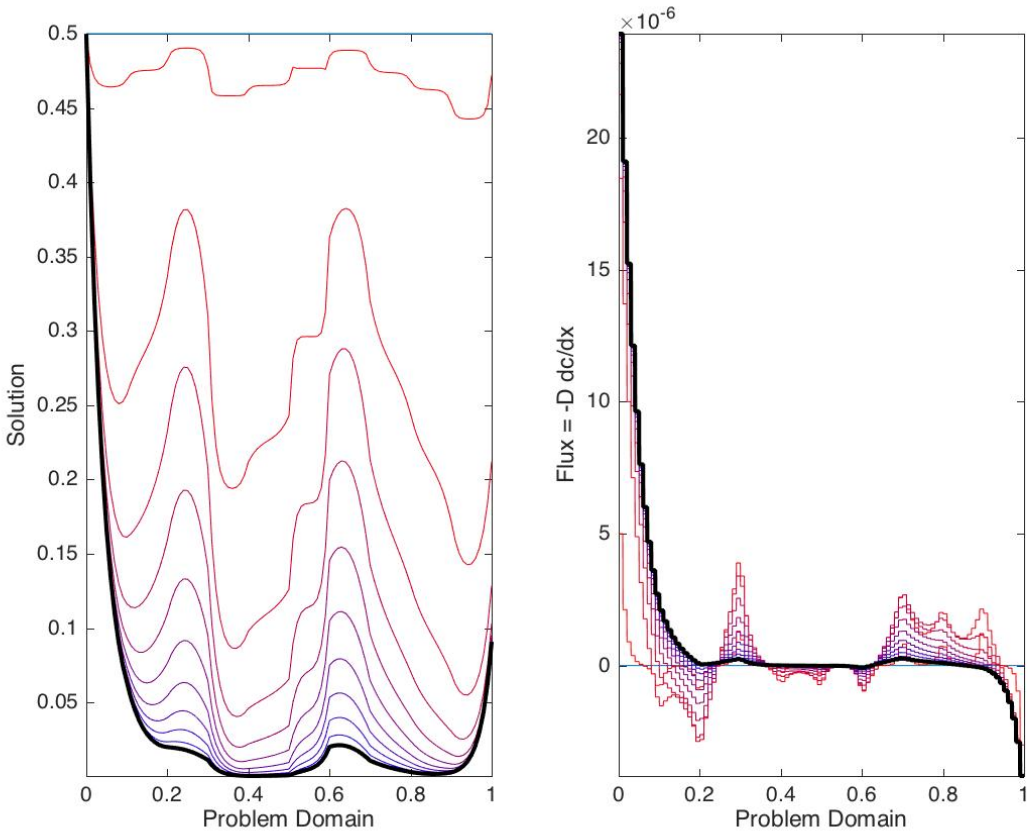


**Figure 5.** Time-dependent solution for concentration (red = early in transient, purple/blue = late in transient, black = at final time step) for 100 time steps for every 10 time steps.

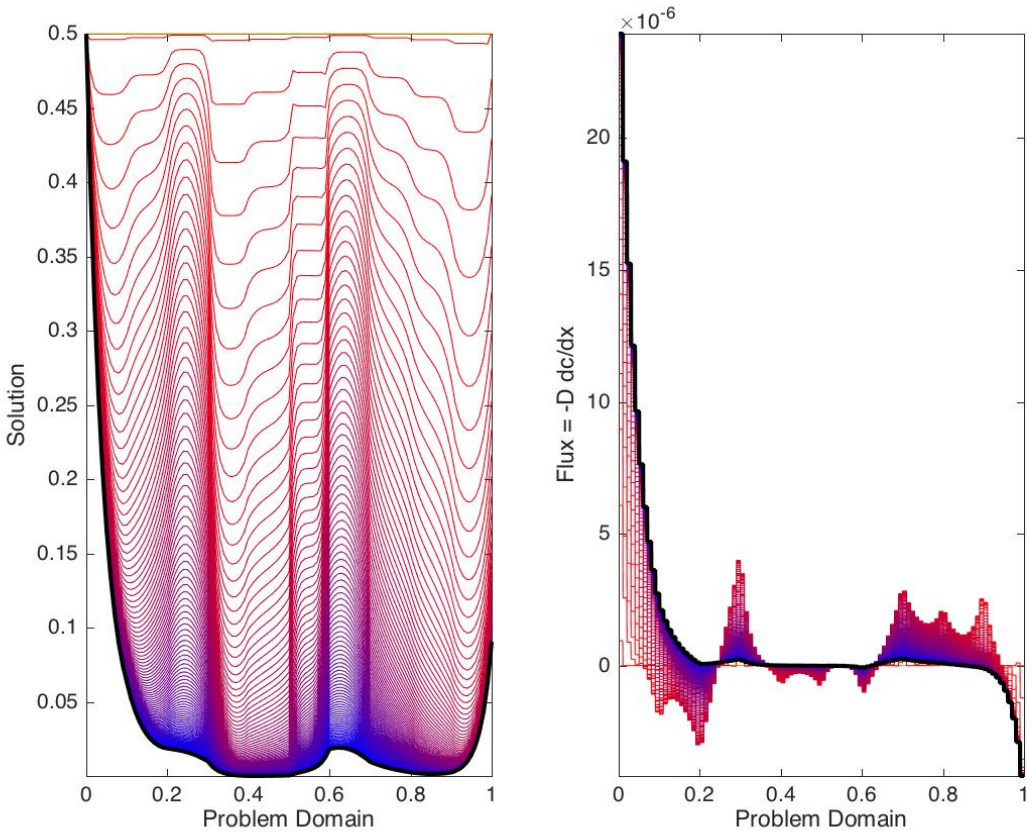


**Figure 6.** Time-dependent solution for concentration (red = early in transient, purple/blue = late in transient, black = at final time step) for 1000 time steps for every 10 time steps.

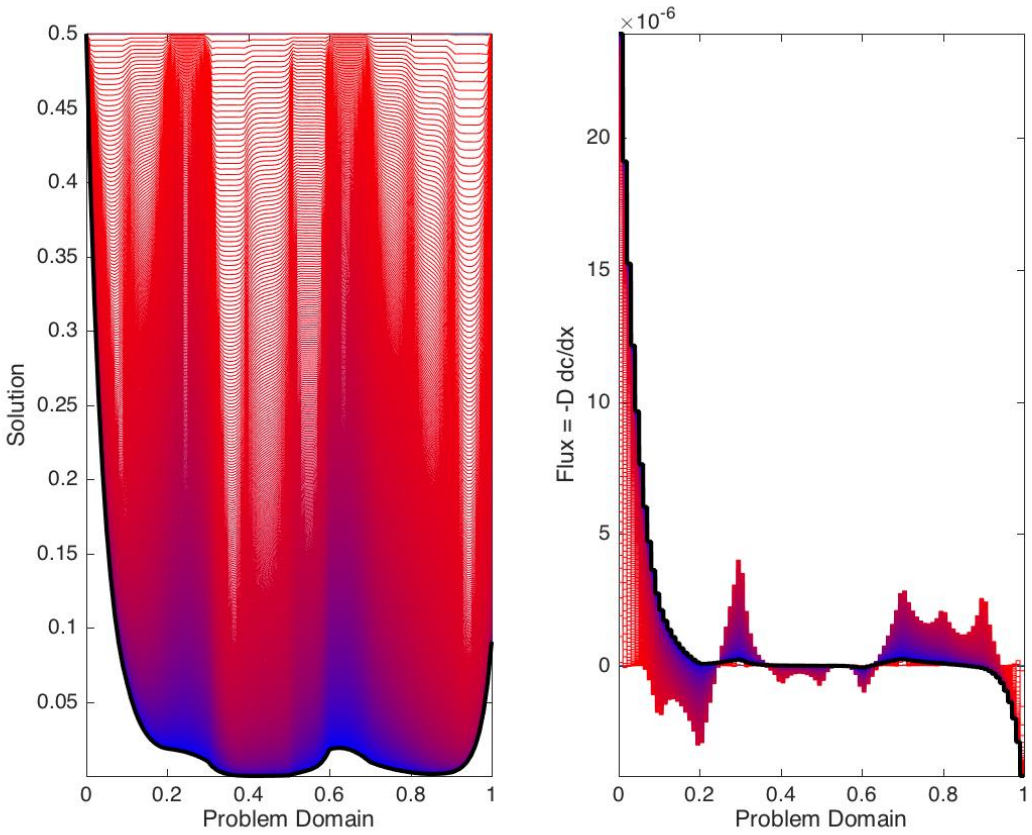


**Figure 7.** Time-dependent solution for concentration (red = early in transient, purple/blue = late in transient, black = at final time step) for 10000 time steps for every 10 time steps.

Because there are so many time steps for the 10000 time step case, Fig. 7 is re-plotted every 100 times steps instead of every 10 time steps in Fig. 8.

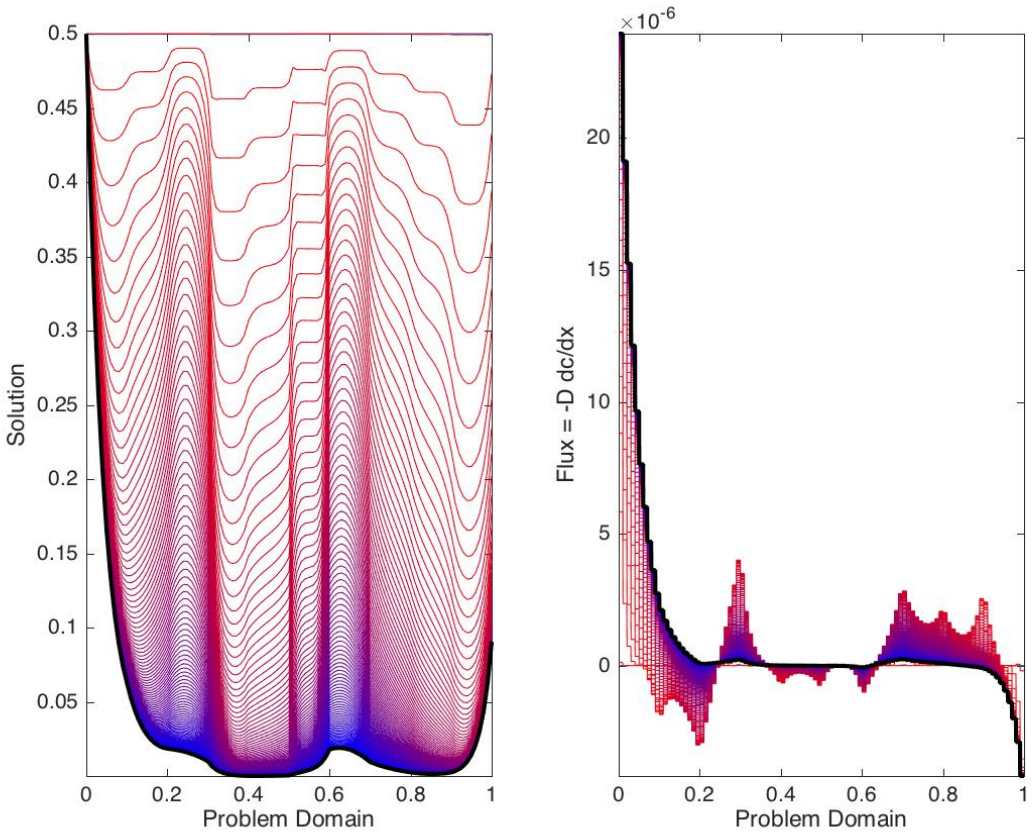


**Figure 8.** Time-dependent solution for concentration (red = early in transient, purple/blue = late in transient, black = at final time step) for 10000 time steps for every 100 time steps.

As can be seen, there is relatively little difference in the results obtained for the three different time steps, except that with a finer time step, the solution results are obtained at more instances in time. This is beneficial early in the transient, especially because, as can be seen from Fig. 5, in the first few time steps the solution changes very rapidly, but as time progresses, begins to approach a steady state. Adaptive time stepping would therefore be very beneficial for this approach to equilibrium case, since then large numbers of time steps could be used early in the transient, and fewer later as the solution stops changing as much.

# 7 Appendix

This section contains the complete code used in this assignment.

## 7.1 `FEProgram.m`

This program contains the master program for running the FE problem.

```
clear all

L = 1.0;                          % problem domain
left = 'Dirichlet';               % left boundary condition
left_value = 0.5;                 % left Dirichlet boundary condition value
right = 'Neumann';              % right boundary condition type
right_value = -(5e-6);              % right Dirichlet boundary condition value
```

```
fontsize = 16;                          % fontsize for plots
num_elem = 100;                         % number of finite elements
shape_order = 2;                        % linear elements
end_time = 6.5e3;                       % end simulation time
num_steps = 100;                        % number of time steps
dt = end_time / num_steps;              % time step size
ic = 0.5;                               % initial condition
discr = 100;                             % plot every discr time steps

% specify D and Tau over the domain in a block structure
D_blocks    = [2.4 2.0 1.5 0.6 1.3 0.14 1.1 2.2 2.0 1.5].* (10^(-6));
Tau_blocks  = [1.2 0.8 0.3 1.4 1.15 0.75 0.35 0.85 1.25 2.0].* (10^(-3));
space_blocks = 0.1:0.1:L;

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

parent_domain = -1:0.1:1;
physical_domain = linspace(0, L, num_elem * length(parent_domain) - (num_elem
    ↪ - 1));

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

% interpolate D and Tau into the physical domain
[D_physical_domain] = PhysicalInterpolation(physical_domain, space_blocks,
    ↪ D_blocks);
[Tau_physical_domain] = PhysicalInterpolation(physical_domain, space_blocks,
    ↪ Tau_blocks);

% perform the meshing
[num_nodes, num_nodes_per_element, LM, coordinates] = mesh(L, num_elem,
    ↪ shape_order);

% specify the boundary conditions
[dirichlet_nodes, neumann_nodes, a_k] = BCnodes(left, right, left_value,
    ↪ right_value, num_nodes);
num_dirichlet = length(dirichlet_nodes(1,:));

n = 1; % index for the time step

% apply the initial condition
soln_condensed_cell = cell([1, num_steps]);
soln_condensed_cell{1, n} = ic .* ones(1, num_nodes - num_dirichlet)';

% initialize the solution and derivative cells
soln_FE_cell = cell([1, length(physical_domain)]);
soln_derivative_FE_cell = cell([1, length(physical_domain)]);

soln_FE_cell{1, n} = ic .* ones(1, length(physical_domain));
soln_derivative_FE_cell{1, n} = zeros(1, length(physical_domain));

% interpolate D and Tau into the an elemental basis
[D_elem, right_endpoint_index, right_endpoint_coordinate] =
```

```
    ↪ ElementInterpolation(coordinates, num_elem, num_nodes_per_element,
    ↪ space_blocks, D_blocks);
[Tau_elem, right_endpoint_index, right_endpoint_coordinate] =
    ↪ ElementInterpolation(coordinates, num_elem, num_nodes_per_element,
    ↪ space_blocks, Tau_blocks);

K_cell = cell([1, num_elem]);
M_cell = cell([1, num_elem]);
F_cell = cell([1, num_elem]);

K = zeros(num_nodes, num_nodes);
M = zeros(num_nodes, num_nodes);
F = zeros(num_nodes, 1);

for elem = 1:num_elem
    k = zeros(num_nodes_per_element);
    m = zeros(num_nodes_per_element);
    f = zeros(num_nodes_per_element, 1);

     for l = 1:length(qp)
         for i = 1:num_nodes_per_element
             [N, dN, x_xe, dx_dxe] = shapefunctions(qp(l), shape_order,
                 ↪ coordinates, LM, elem);

             % assemble the (elemental) forcing vector
             if strcmp(right, 'Neumann')
                 if ((neumann_nodes(1,1) == (elem + 1)) && (i ==
                     ↪ num_nodes_per_element) && (l == 1))
                     f(i) = f(i) - neumann_nodes(2, 1);
                 end
             end

             for j = 1:num_nodes_per_element
                 % assemble the (elemental) stiffness matrix
                 k(i,j) = k(i,j) + wt(l) * (D_elem(elem) * dN(i) * dN(j) /
                     ↪ dx_dxe + Tau_elem(elem) * N(i) * N(j) * dx_dxe);

                 % assemble the (elemental) mass matrix
                 m(i,j) = m(i,j) + wt(l) * N(i) * N(j) * dx_dxe;
             end
         end
     end

     % store elemental values into cells
     K_cell{1, elem} = k;
     M_cell{1, elem} = m;
     F_cell{1, elem} = f;
end

for elem = 1:num_elem % assemble into the global matrices
     m = 1;
     for m = 1:length(permutation(:,1))
         i = permutation(m,1);
         j = permutation(m,2);
```

```
            K(LM(elem, i), LM(elem, j)) = K_cell{1, elem}(i, j) + K(LM(elem, i),
                ↪ LM(elem, j));
            M(LM(elem, i), LM(elem, j)) = M_cell{1, elem}(i, j) + M(LM(elem, i),
                ↪ LM(elem, j));
        end

        for i = 1:length(f)
            F(LM(elem, i)) = F((LM(elem, i))) + F_cell{1,elem}(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);
[M_uu, M_uk, F_u, F_k] = condensation(M, F, num_nodes, dirichlet_nodes);

for n = 1:num_steps
    % perform the very first solve using Gaussian elimination (time-dependent
        ↪ case)
    A_mat = (1 / dt) * M_uu + K_uu;
    b_mat = F_u + (1 / dt) * (M_uu * soln_condensed_cell{1,n} + M_uk *
        ↪ dirichlet_nodes(2,:)') - ((1 / dt) * M_uk + K_uk) * dirichlet_nodes
        ↪ (2,:)';
    a_u_condensed = A_mat \ b_mat;
    soln_condensed_cell{1, n + 1} = a_u_condensed;

    % 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
    [solution_FE, solution_derivative_FE] = postprocess(num_elem,
        ↪ parent_domain, a, LM, num_nodes_per_element, shape_order,
        ↪ coordinates, physical_domain);
    soln_FE_cell{n+1} = solution_FE;
    soln_derivative_FE_cell{n+1} = solution_derivative_FE;

    n = n + 1;
end

[none] = PlotInTime(1, soln_FE_cell, physical_domain, 'Solution', discr,
    ↪ num_steps);
```

```
[none] = PlotInTime(-D_physical_domain, soln_derivative_FE_cell,
    ↪ physical_domain, 'Flux = -D dc/dx', discr, num_steps);
%saveas(gcf, '10000_100', 'jpeg')
```

## 7.2 MeshGenerator.m

This program generates the mesh and connectivity matrix.

```
% Mesh generator, ME 280a HW 5
clear all

Nt = 3;                                 % number of layers
No = 8;                                 % number of elements in theta
Nc = 4;                                 % number of elements in circum

if mod(No, 2) ~= 0
    disp('No must be even!')
end

num_nodes_per_elem = 8;                 % linear elements

R = 1;                                  % radius of each arch
r = 0.3;                                % radius of inner hole
t = 0.2;                                % thickness of the tube wall

layer_thickness = t / (Nt);             % thickness of each ring
angle = (2*pi) / Nc;                    % angle in horizontal plane

% each row represents one coordinate of a global node
coordinates = zeros(Nt * Nc, 3);
Angle = pi / (No / 2);

k = 1; % index for coordinate row
x = 0;
y = 0;
z = 0;

Theta = 0;  % angle in each plane

% find the x-coordinates of the No + 1 slices
x_centers = zeros(1, No + 1);
x_centers(1) = x;
theta_inc = pi / (No / 2);
theta = theta_inc;

% in the first half of the tube
for i = 2:((No/2) + 1)
    x_centers(i) = x_centers(1) + (R + t + r) * (1 - cos(theta));
    theta = theta + theta_inc;
end

% in the second half of the tube
j = i + 1;
second_part_start = x_centers(i);
```

```
for l = 2:((No/2) + 1)
    x_centers(j) = second_part_start + x_centers(l);
    j = j + 1;
end

% create a vector of the y-coordinates
z_centers = zeros(1, No + 1);
z_centers(1) = z;
theta = theta_inc;

% in the first half of the tube
for i = 2:((No/2) + 1)
    z_centers(i) = (R + t + r) * sin(theta);
    theta = theta + theta_inc;
end

% in the second half of the tube
j = i + 1;
for l = 2:((No/2) + 1)
    z_centers(j) = - z_centers(l);
    j = j + 1;
end

for l = 1:(No + 1) % mesh in the theta direction

    % for each plane
    theta = 0;
    dt = 0;

    % meshes in a plane perpendicular to tube axis
    for j = 1:(Nt + 1) % create all layers of rings
        for i = 1:Nc % create a single ring

            % x-coordinate
            coordinates(k, 1) = x_centers(l) + (r + dt) * cos(theta);

            % compute tilting parameters
            w = sin(pi/2 - Theta) * ((r + dt) * cos(theta)) / sin(pi/2);
            h = w * sin(Theta);
            p = w * cos(Theta);

            % y-coordinate
            coordinates(k, 2) = y + (r + dt) * sin(theta);

            % z-coordinate
            coordinates(k,3) = z_centers(l);

            % tilt for each half of the tube
            if l > No/2
                sn = -1;
            else
                sn = 1;
            end
```

```
            % tilt the z-coordinate for off-symmetric planes
            if find([1, 2, 8], i)
                coordinates(k,3) = coordinates(k,3) - sn*h;
            else
                coordinates(k,3) = coordinates(k,3) + sn*h;
            end

            % tilt the symmetric planes (the peaks)
            if (l == 3) || (l == 7)
                if find([1, 2, 8], i)
                    coordinates(k,3) = coordinates(k,3) - ((r + dt) * cos(
                        ↪ theta));
                else
                    coordinates(k,3) = coordinates(k,3) + ((r + dt) * cos(
                        ↪ theta));
                end
                coordinates(k,1) = x_centers(l);
            end

            % tilt the x-coordinate
            if find([1, 2, 8], i)
                coordinates(k,1) = coordinates(k,1) + p;
            else
                coordinates(k,1) = coordinates(k,1) - p;
            end

            k = k + 1;
            theta = theta + angle;
        end
        dt = dt + layer_thickness; % reset radius
        theta = 0; % reset angle
    end

    % move the angle (along length) and centers for the next plane
    Theta = Theta + Angle;
    x = x + (R + t + r) * (1 - cos(Theta));
    y = y;
    z = z + (R + t + r) * sin(Theta);
end




X = coordinates(:,1);
Y = coordinates(:,2);
Z = coordinates(:,3);

% generate the connectivity matrix
num_elem = No * Nc * Nt;
LM = zeros(num_elem, num_nodes_per_elem);

% apply in a single slice
j = 1;
k = 1;
```

```
e = 1;
for l = 1:(No + 1) % for each slice, assign the front node values
    for elem = e:(e + num_elem / No - 1) % for each element in the slice
        LM(elem, 1) = j;
        LM(elem, 3) = j + Nc;

        if (mod(elem, Nc) == 0)
            LM(elem, 2) = k;
            LM(elem, 4) = k + Nc;
            k = k + Nc;
        else
            LM(elem, 2) = j + 1;
            LM(elem, 4) = j + Nc + 1;
        end
        j = j + 1;
    end

    % update for next slice frontal values
    k = k + Nc;
    j = j + Nc;
    e = e + Nc * Nt;
end

% assign the back face values - front values for second slice are
% back values for first slice, etc.
i = 1;
for j = 1:No
    for elem = i:(i + Nc*Nt - 1)
        LM(elem, (Nc + 1):end) = LM(elem + Nc * Nt, 1:Nc);
    end
    i = i + Nc*Nt;
end

% delete the unecessary last "chunk" in the LM
LM = LM(1:num_elem, :);
```

### 7.3 BCnodes.m

This function applies the 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)

% 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;
j = 1;
switch left
```

```
    case 'Dirichlet'
        dirichlet_nodes(1, i) = 1;
        dirichlet_nodes(2, i) = left_value;
        i = i + 1;
    case 'Neumann'
        neumann_nodes(1, j) = 1;
        j = j + 1;
    otherwise
        disp('You entered an incorrect field for the type of BC on the left
            ↪ boundary.');
end

switch right
    case 'Dirichlet'
        dirichlet_nodes(1, i) = num_nodes;
        dirichlet_nodes(2, i) = right_value;
    case 'Neumann'
        neumann_nodes(1, j) = num_nodes;
        neumann_nodes(2, j) = right_value;
    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
```

## 7.4 ElementInterpolation.m

This function interpolates the material property data into an element basis.

```
function [var_elem, right_endpoint_index, right_endpoint_coordinate] =
    ↪ ElementInterpolation(coordinates, num_elem, num_nodes_per_element,
    ↪ space_blocks, var_blocks)

% This function interpolates a block-structued variable into the physical
% domain. By default, values at the ends of boundaries are put into the
% previous domain.

    s = 1;
    var_elem = zeros(1, length(num_elem));
    right_endpoint_index = zeros(1, length(num_elem));
    right_endpoint_coordinate = zeros(1, length(num_elem));

    for elem = 1:num_elem
        right_endpoint_index(elem) = elem * (num_nodes_per_element - 1);
        right_endpoint_coordinate(elem) = coordinates(right_endpoint_index(
            ↪ elem) + 1, 1);
```

```
        if (abs(right_endpoint_coordinate(elem) - space_blocks(s)) < 1e-10) ||
            ↪ (right_endpoint_coordinate(elem) <= space_blocks(s))

        else
            s = s + 1;
        end
        var_elem(elem) = var_blocks(s);
    end


end
```

## 7.5 PhysicalInterpolation.m

This function interpolates the material property data into the physical domain.

```
function [var_physical_domain] = PhysicalInterpolation(physical_domain,
    ↪ space_blocks, var_blocks)

var_physical_domain = zeros(1,length(physical_domain));

j = 1;
for i = 1:length(physical_domain)

    if (physical_domain(i) <= space_blocks(j))
    else
        j = j + 1;
    end

    var_physical_domain(i) = var_blocks(j);
end

end
```

## 7.6 PlotInTime.m

This function performs the plotting for this assignment.

```
function [none] = PlotInTime(param, func, physical_domain, y_label, discr,
    ↪ num_steps)

linewidth = 2;

current_max = max(param .* func{1,1});
current_min = min(param .* func{1,1});

for m = 2:num_steps
    maximum = max(param .* func{1,m});
    minimum = min(param .*func{1,m});

    if maximum > current_max
        current_max = maximum;
    end
```

```
    if minimum < current_min
        current_min = minimum;
    end
end

plot(physical_domain, param .* func{1, 1})
ylim([minimum, maximum])
ylabel(y_label)
xlabel('Problem Domain')
hold on
dc = 0.0;

for n = [2:discr:num_steps, num_steps]
    if n == num_steps
        plot(physical_domain, param .* func{1, n}, 'k-', 'LineWidth',
            ↪ linewidth)
    else
        plot(physical_domain, param .* func{1, n}, 'Color', [1.0 - dc, 0.0, dc
            ↪ ])
    end
    drawnow
    dc = dc + 0.01;
end

none = 0;
end
```

## 7.7 condensation.m

This function performs the static condensation to remove Dirichlet nodes from the solve.

```
% 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

%K_uu = sparse(K_uu);
```

### 7.8 mesh.m

This function meshes the domain.

```
function [num_nodes, num_nodes_per_element, LM, coordinates] = mesh(L,
    ↪ num_elem, shape_order)

num_nodes = (shape_order - 1) * num_elem + 1;

% for evenly-spaced nodes, on a 3-D mesh. Each row corresponds to a node.
coordinates = zeros(num_nodes, 3);

% in 1-D, the first node starts at (0,0), and the rest are evenly-spaced
for i = 2:num_nodes
   coordinates(i,:) = [coordinates(i - 1, 1) + L/(num_nodes - 1), 0, 0];
end

% Which nodes correspond to which elements depends on the shape function
% used. Each row in the LM corresponds to one element.
num_nodes_per_element = shape_order;

LM = zeros(num_elem, num_nodes_per_element);

for i = 1:num_elem
    for j = 1:num_nodes_per_element
        LM(i,j) = num_nodes_per_element * (i - 1) + j - (i - 1);
    end
end

end
```

### 7.9 permutation.m

This function meshes the domain.

```
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
```

## 7.10 postprocess.m

This function postprocesses the results into the physical domain.

```
function [solution_FE, solution_derivative_FE] = postprocess(num_elem,
    ↪ parent_domain, a, LM, num_nodes_per_element, shape_order, coordinates,
    ↪ physical_domain)

b = zeros(1, shape_order);
A = zeros(shape_order);
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_element
        b(i) = a(LM(elem, i));
    end

    for j = 1:shape_order % loop over the rows of A
        coordinate = coordinates(LM(elem, j));
        for l = 1:shape_order % loop over the columns of A
            A(j,l) = coordinate .^ (l - 1);
        end
    end

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

    % 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_element)), length(parent_domain));
```

```
    for i = 1:num_nodes_per_element
        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_element % 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
```

## 7.11 quadrature.m

This function selects the quadrature rule.

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

shape_order = 4; % forces a five-point quadrature rule for this problem
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)];
    case 4
        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))];
    otherwise
        disp('You entered an unsupported shape function order for the
            ↪ quadrature rule.');
end
```

## 7.12 shapefunctions.m

This function provides the shape functions.

```
% N          : shape functions in the master domain
% dN         : derivative of the shape functions with respect to xe
% x_xe       : x as a function of xe
% dx_dxe     : derivative of x with respect to xe
function [N, dN, x_xe, dx_dxe] = shapefunctions(xe, shape_order, coordinates,
    ↪ LM, elem)

% shape functions and their derivatives
N = zeros(shape_order, 1);
dN = zeros(shape_order, 1);

switch shape_order
    case 2
        N(1) = (1 - xe) ./ 2;
        N(2) = (1 + xe) ./ 2;
        dN(1) = - 1/2;
        dN(2) = 1/2;g
    otherwise
        disp('You entered an unsupported shape function order.');
end

% check that the sum of the shape functions adds up to 1
sum = 0;
for j = 1:shape_order
    sum = sum + N(j);
end

if (abs(sum - 1.0) > 1e-10)
    disp('Sum of the shape functions does not add up to 1.');
end

% x(xe) transformation to the parametric domain
x_xe = 0.0;
dx_dxe = 0.0;
for i = 1:shape_order
    x_xe = x_xe + coordinates(LM(elem, i)) * N(i);
    dx_dxe = dx_dxe + coordinates(LM(elem,i)) * dN(i);
end
```