ME 280a: HW 3

April Novak

October 27, 2016

1 Introduction and Objectives

The purpose of this study is to solve a simple finite element (FE) problem using a Preconditioned Conjugate-Gradient (PCG) solver, with a diagonal preconditioner. This involves the storage of the information element-by-element. So, while the problem to be solved is similar to past assignments, the numerical solution method is modified. In addition, the elastic modulus is allowed to be a function of position. The Galerkin FE method is used, which for certain classes of problems possesses the "best approximation property," a characteristic that signifies that the FE solution obtained is the best possible solution for a given mesh refinement and choice of shape functions. The mathematical procedure and numerical implementation is described in Section 2.

2 Procedure

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

2.1 Problem Statement

This section describes the mathematical process used to solve the following problem:

$$\frac{d}{dx}\left(E(x)\frac{du}{dx}\right) = xk^3\cos\left(\frac{2\pi kx}{L}\right) \tag{1}$$

where E is the modulus of elasticity, u is the solution, k is a known constant, L is the problem domain length, and x is the spatial variable. For simplicity, a constant $\gamma = 2\pi k/L$ is defined. In order to verify that the program functions correctly, the FE solution to Eq. (1) will be compared with the analytical solution to Eq. (1). To determine the analytical solution, integrate Eq. (1) once to obtain:

$$E(x)\frac{du}{dx} = k^3\left[\frac{x}{\gamma}\sin(\gamma x) + \frac{1}{\gamma^2}\cos(\gamma x)\right] + C_1 \tag{2}$$

where, even though E is a function of x, it is constant over particular lengths of the domain. Hence, the analytical solution can be determined by splitting up the domain into sufficiently small regions where E is constant. So, for the remainder of this section, E is treated as constant, and it is understood that the analytical solutions obtained are exact only over each individual region where E is constant. Integrating once more:

$$u(x) = \frac{1}{E(x)}\left[\frac{-xk^3}{\gamma^2}\cos(\gamma x) + \frac{2k^3}{\gamma^3}\sin(\gamma x)\right] + \frac{C_1 x}{E(x)} + C_2 \tag{3}$$

The boundary conditions for this problem are Dirichlet at both endpoints, such that:

$$\begin{aligned} u(0) &= -0.1 \\ u(L) &= 0.7 \end{aligned} \tag{4}$$

From Eq. (1), $E(x)du/dx$ must be continuous in the domain, and must match at inter-block boundaries. For example, for two blocks that share a node at $x = x_{share}$:

$$\begin{aligned} k^3 \left[\frac{x_{share}}{\gamma}\sin(\gamma x_{share}) + \frac{1}{\gamma^2}\cos(\gamma x_{share})\right] + C_1(\text{block 1}) = \\ k^3 \left[\frac{x_{share}}{\gamma}\sin(\gamma x_{share}) + \frac{1}{\gamma^2}\cos(\gamma x_{share})\right] + C_1(\text{block 2a}) \end{aligned} \tag{5}$$

Because all constants besides E are constant in the domain, C_1 is the same for each block. The second requirement is that u be continuous across block edges. From Eq. (3):

$$\begin{aligned} \frac{1}{E(\text{block 1})}\underbrace{\left[\frac{-x_{share}k^3}{\gamma^2}\cos(\gamma x_{share}) + \frac{2k^3}{\gamma^3}\sin(\gamma x_{share})\right]}_{\text{Ⓐ}} + \frac{C_1(\text{block 1})x_{share}}{E(\text{block 1})} + C_2(\text{block 1}) = \\ \frac{1}{E(\text{block 2})}\left[\frac{-x_{share}k^3}{\gamma^2}\cos(\gamma x_{share}) + \frac{2k^3}{\gamma^3}\sin(\gamma x_{share})\right] + \frac{C_1(\text{block 2})x_{share}}{E(\text{block 2})} + C_2(\text{block 2}) \end{aligned} \tag{6}$$

Rearranging, and recognizing that C_1 is the same for all blocks, the following must be true at each shared location:

$$C_1\left(\frac{x_{share}}{E(\text{left})} - \frac{x_{share}}{E(\text{right})}\right) + C_2(\text{left}) - C_2(\text{right}) = -\text{Ⓐ}\left(\frac{1}{E(\text{left})} - \frac{1}{E(\text{right})}\right) \tag{7}$$

Then, at the left endpoint:

$$C_2(\text{block 1}) = u(0) \tag{8}$$

And at the right endpoint:

$$\frac{C_1(\text{last block})L}{E(L)} + C_2(\text{last block}) = u(L) - \frac{1}{E(x)}\left[\frac{-Lk^3}{\gamma^2}\cos(\gamma L) + \frac{2k^3}{\gamma^3}\sin(\gamma L)\right] \tag{9}$$

Eq. (8) and Eq. (9) provide two boundary conditions, while Eq. (7) provides one boundary condition at each shared location (locations at which E changes). This, combined with the fact that C_1 is the same for each element, provides a system of linear equations that can be solved to obtain C_1 and $C_2(\text{block 1}, \text{block 2}, \text{block 3}, \text{block 4}, \cdots, \text{block 10})$. This is the method used in this assignment to determine the analytical solution. The method is made very general so that any number of piecewise-constant regions of E can be solved both analytically and with the FEM. The analytical solution is plotted below, and the values for the coefficients C_1 and C_2 in each block are given in the following table.

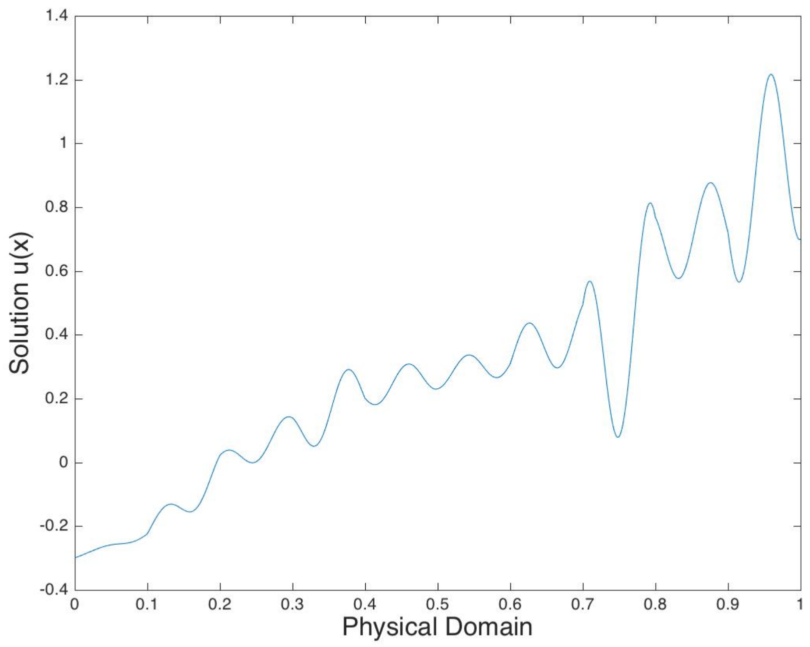

Figure 1. Analytical solution.

Table 1. Coefficients C_1 and C_2 over each domain of the problem.

Domain	C_1	C_2
$0.0 \leq x < 0.1$	1.9052	-0.3000
$0.1 \leq x < 0.2$	1.9052	-0.4133
$0.2 \leq x < 0.3$	1.9052	-0.2269
$0.3 \leq x < 0.4$	1.9052	-0.3733
$0.4 \leq x < 0.5$	1.9052	-0.0605
$0.5 \leq x < 0.6$	1.9052	0.0171
$0.6 \leq x < 0.7$	1.9052	-0.1774
$0.7 \leq x < 0.8$	1.9052	-1.5201
$0.8 \leq x < 0.9$	1.9052	-0.0900
$0.9 \leq x < 1.0$	1.9052	-0.9012

2.2 Finite Element Implementation

The Galerkin FEM achieves the best approximation property by approximating the true solution $u(x)$ as $u^N(x)$, where both $u^N(x)$ and the test function ψ are expanded in the same set of N basis functions ϕ:

$$\begin{gathered} u \approx u^N = \sum_{j=1}^{N} a_j \phi_j \\ \psi = \sum_{i=1}^{N} b_i \phi_i \end{gathered} \tag{10}$$

Galerkin's method is stated as:

$$r^N \cdot u^N = 0 \tag{11}$$

where r^N is the residual. Hence, to formulate the weak form to Eq. (1), multiply Eq. (1) through by ψ and integrate over all space, $d\Omega$.

$$\int_\Omega \frac{d}{dx}\left(E(x)\frac{du}{dx}\right)\psi d\Omega - \int_\Omega xk^3\cos(\gamma x)\psi d\Omega = 0 \tag{12}$$

Applying integration by parts to the first term:

$$-\int_\Omega E(x)\frac{du}{dx}\frac{d\psi}{dx}d\Omega + \int_{\partial\Omega} E(x)\frac{du}{dx}\psi d(\partial\Omega) - \int_\Omega xk^3\cos(\gamma x)\psi d\Omega = 0 \tag{13}$$

where $\partial\Omega$ refers to one dimension lower than Ω, which for this case refers to evaluation at the endpoints of the domain. Hence, for this particular 1-D problem, the above reduces to:

$$\begin{gathered}-\int_0^L E(x)\frac{du}{dx}\frac{d\psi}{dx}dx + E(x)\frac{du}{dx}\psi\bigg|_0^L - \int_0^L xk^3\cos(\gamma x)\psi dx = 0\\ \int_0^L E(x)\frac{du}{dx}\frac{d\psi}{dx}dx = -\int_0^L xk^3\cos(\gamma x)\psi dx + E(x)\frac{du}{dx}\psi\bigg|_0^L\end{gathered} \tag{14}$$

Inserting the approximation described in Eq. (10):

$$\int_0^L E(x)\frac{d\left(\sum_{j=1}^N a_j\phi_j\right)}{dx}\frac{d\left(\sum_{i=1}^N b_i\phi_i\right)}{dx}dx = -\int_0^L xk^3\cos(\gamma x)\sum_{i=1}^N b_i\phi_i dx + E(x)\frac{du}{dx}\sum_{i=1}^N b_i\phi_i\bigg|_0^L \tag{15}$$

Recognizing that b_i appears in each term, the sum of the remaining terms must also equal zero (i.e. basically cancel b_i from each term).

$$\int_0^L E(x)\frac{d\left(\sum_{j=1}^N a_j\phi_j\right)}{dx}\frac{d\phi_i}{dx}dx = -\int_0^L xk^3\cos(\gamma x)\phi_i dx + E(x)\frac{du}{dx}\phi_i\bigg|_0^L \tag{16}$$

This equation can be satisfied for each choice of j, and hence can be reduced to:

$$\int_0^L E(x)\frac{d\left(a_j\phi_j\right)}{dx}\frac{d\phi_i}{dx}dx = -\int_0^L xk^3\cos(\gamma x)\phi_i dx + E(x)\frac{du}{dx}\phi_i\bigg|_0^L \tag{17}$$

This produces a system of matrix equations of the form:

$$\mathbf{K}\vec{a} = \vec{F} \tag{18}$$

where:

$$\begin{gathered}K_{ij} = \int_0^L E(x)\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx\\ a_j = a_j\\ F_i = -\int_0^L xk^3\cos(\gamma x)\phi_i dx + E(x)\frac{du}{dx}\phi_i\bigg|_0^L\end{gathered} \tag{19}$$

where the second term in F_i is only applied at nodes that have Neumann boundary conditions (since ψ satisfies the homogeneous form of the essential boundary conditions). The above equation governs the entire domain. $\mathbf{K}$ is an $n \times n$ matrix, where n is the number of global nodes. The solution is contained within $\vec{a}$. This matrix system is solved in this assignment by simple Gaussian elimination, such that $\vec{a} = \mathbf{K}^{-1}\vec{F}$.

Quadrature is used to perform the numerical integration because it is much faster, and more general, than symbolic integration of the terms appearing in Eq. (19). In order for these equations to be useful with Gaussian quadrature, they must be transformed to the master element which exists over $-1 \leq \xi \leq 1$:

$$K_{ij} = \int_0^L E(x)\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx \rightarrow \int_{-1}^1 E(x(\xi))\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx\left(\frac{dx}{d\xi}\frac{dx}{d\xi}\frac{d\xi}{dx}\right) \rightarrow \int_{-1}^1 E(x(\xi))\frac{d\phi_i}{d\xi}\frac{d\phi_j}{d\xi}dx\left(\frac{d\xi}{dx}\right)$$
$$a_j = a_j \tag{20}$$
$$F_i = -\int_0^L xk^3\cos(\gamma x)\phi_i dx \rightarrow -\int_{-1}^1 x(\xi)k^3\cos(\gamma x(\xi))\phi_i\frac{dx}{d\xi}d\xi$$

where the second term in F_i has been dropped because there are no Neumann boundary conditions in this assignment. For linear elements, the shape functions have the following form and derivative over the master element:

$$\begin{aligned}\phi_1(\xi) &= \frac{1-\xi}{2}, \quad \frac{d\phi_1(\xi)}{d\xi} = -1/2\\ \phi_2(\xi) &= \frac{1+\xi}{2}, \quad \frac{d\phi_2(\xi)}{d\xi} = +1/2\end{aligned} \tag{21}$$

The transformation from the physical domain (x) to the parent domain (ξ) is done with an isoparametric mapping:

$$x(\xi) = \sum_{i=1}^N X_i\phi_i(\xi) \tag{22}$$

where X_i are the coordinates in each element. This mapping is performed for each element individually. The Jacobian $dx/d\xi$ is obtain from Eq. (22) by differentiation:

$$\frac{dx(\xi)}{d\xi} = \sum_{i=1}^N X_i\frac{d\phi_i(\xi)}{d\xi} \tag{23}$$

With all these transformations from the physical domain to the isoparametric domain, Gaussian quadrature can be used. A 5-point quadrature rule is used (given in the problem statement). For the five-point quadrature rule, the weights w and sampling points x are:

$$\begin{aligned}w &= \left[\frac{322-13\sqrt{70}}{900}, \frac{322+13\sqrt{70}}{900}, \frac{128}{225}, \frac{322+13\sqrt{70}}{900}, \frac{322-13\sqrt{70}}{900}\right]\\ x &= \left[-\frac{1}{3}\sqrt{5+2\sqrt{10/7}}, -\frac{1}{3}\sqrt{5-2\sqrt{10/7}}, 0, \frac{1}{3}\sqrt{5-2\sqrt{10/7}}, \frac{1}{3}\sqrt{5+2\sqrt{10/7}}\right]\end{aligned} \tag{24}$$

Transformation to the isoparametric domain therefore easily allows construction of the local stiffness matrix and local force matrix. The actual numerical algorithm computes the elemental $k(i,j)$ and $b(i)$ by looping over i, j, and the quadrature points. Because each calculation is computed over a single element, a connectivity matrix is used to populate the global stiffness matrix and the global forcing vector with the elemental matrices and vectors. See the Appendix for the full code used in this assignment. After the global matrix and vector are formed, the global matrix has a banded-diagonal structure. However, this approach is only needed for direct methods such as Gaussian elimination. Because a PCG method is used here, there is no need to assemble the global **K**. This will be explained in more detail in **??**.

In order to apply the boundary conditions within the numerical context of the finite element method, the matrix equation in Eq. (18) must be rewritten to reflect that some of the nodal values are actually already specified through the Dirichlet boundary conditions.

$$\begin{bmatrix}K_{kk} & K_{ku}\\ K_{uk} & K_{uu}\end{bmatrix}\begin{bmatrix}x_k\\ x_u\end{bmatrix} = \begin{bmatrix}F_k\\ F_u\end{bmatrix} \tag{25}$$

where k indicates a known quantity (specified through a boundary condition) and u indicates an unknown quantity. Explicitly expanding this equation gives:

$$\begin{aligned} K_{kk}x_k + K_{ku}x_u &= F_k \\ K_{uk}x_k + K_{uu}x_u &= F_u \end{aligned} \tag{26}$$

Solving this matrix system is sometimes referred to as "static condensation," since the original matrix system in Eq. (18) must be separated into its components. The nodes at which Dirichlet conditions are specified are "known," while all other nodes, including Neumann condition nodes, are "unknown," since it is the value of u that we are looking to find at each node. The second of these equations is the one that is solved in this assignment, since there are no natural boundary conditions. So, while the global **K** matrix is not formed, in principle the Dirichlet boundary conditions are stilled applied in this manner, by removing certain rows and columns of **K** and $\vec{F}$ from the PCG solve.

Once the solution is obtained by the PCG method, the solution is transformed back to the physical domain (from the isoparametric domain) by solving a linear matrix system to determine the coefficients on the basis functions over each element (in the physical domain). For example, for a quadratic shape function, over one element with coordinates x_1, x_2, and x_3, with solution values a_1, a_2, and a_3, the following linear system solves for the coordinates on the shape function in the physical domain, in that element:

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{27}$$

Each element is looped over to obtain the coefficients on the shape functions in the physical domain. This then transforms the solution back to the physical domain, and completes the FE solution.

2.3 Error Estimates and Convergence Criteria

The accuracy of the FE solution is estimated using the energy norm e^N, defined as:

$$e^N = \frac{\|u - u^N\|}{\|u\|} \tag{28}$$

where:

$$\|u\| = \sqrt{\int_\Omega \frac{du}{dx} E \frac{du}{dx}} \tag{29}$$

$$\|u - u^N\| = \sqrt{\int_\Omega \frac{d(u-u^N)}{dx} E \frac{d(u-u^N)}{dx}} = \sqrt{\int_\Omega \left(\frac{du}{dx} - \frac{du^N}{dx}\right) E \left(\frac{du}{dx} - \frac{du^N}{dx}\right)} \tag{30}$$

The derivatives of the FE solution are determined according to:

$$\frac{du^N}{dx} = \frac{d}{dx}\sum_{j=1}^{N} a_j \phi_j(x) = \sum_{j=1}^{N} a_j \frac{d\phi_j(x)}{dx} = \sum_{j=1}^{N} a_j \frac{d\phi_j(x)}{d\xi}\frac{d\xi}{dx} \tag{31}$$

while the derivative of the analytical solution is obtained from Eq. (2). The accuracy of the finite element solution can also be estimated by calculation of the potential energy $\mathfrak{J}$ of the solution:

$$\mathfrak{J}(u^N) = \frac{1}{2}B(u^N, u^N) - L(u^N) \tag{32}$$

where:

$$\begin{aligned} B(u^N, u^N) &= \int_0^L \frac{du^N}{dx} E \frac{du^N}{dx} dx \\ L(u^N) &= -\int_0^L x k^3 \cos(\gamma x) u^N dx \end{aligned} \tag{33}$$

The Neumann boundary component of $L(u^N)$ has been omitted because it is not present for this problem. The potential energy of the finite element solution must always be greater than that of the finite element solution. This statement is directly related to the "best approximation property" of finite elements that holds for symmetric stiffness matrices. So, as the mesh becomes finer and finer and the finite element solution better approximates the true solution, the potential energy of the finite element solution should decrease and asymptotically approach that of the true solution.

2.4 The Preconditioned Conjugate Gradient Method

The Preconditioned Conjugate Gradient (PCG) method is used to perform the numerical solution of the linear algebra system. This method is an iterative method, and computes the solution $\vec{a}$ at each iteration by adding an update $\vec{z}$ to the previous iterate, scaled by a constant λ:

$$\vec{a}^{i+1} = \vec{a}^{i} + \lambda^{i}\vec{z}^{i} \tag{34}$$

where i denotes the iteration number. This requires an initial guess for $\vec{a}^1$ (indexing starts at 1). Because the nodes at which Dirichlet boundary conditions apply have already been removed, this initial guess is set, for simplicity, to values of 1 at all nodes. Physics-based preconditioning is a more sophisticated method that takes into account more realistic solutions to provide the initial guess for the solution, but this is not pursued here. Due to the similarity between the PCG method and the Method of Steepest Descent, for simplicity, the initial guess for $\vec{z}^1$ is chosen to be the residual when evaluate with $\vec{a}^1$. The residual $\vec{r}$ is here defined to be:

$$\vec{r}^{i} = \vec{F} - \mathbf{K}\vec{a}^{i} \tag{35}$$

The quantity $\vec{z}^i$ is chosen to be K-conjugate to the previous iterate $\vec{z}^{i-1}$. This weighted inner product searches, in the next iterate, in the conjugate gradient direction of the previous iterate, while taking into account the structure of the governing equation. In other words, it is required that:

$$\vec{z}^{i,T}\mathbf{K}\vec{z}^{i-1} = 0 \tag{36}$$

This then gives the following form for θ:

$$\theta^{i} = -\frac{\vec{r}^{T,i}\mathbf{K}\vec{z}^{i-1}}{\vec{z}^{T,i-1}\mathbf{K}\vec{z}^{i-1}} \tag{37}$$

where θ is used in the update of $\vec{z}$ as:

$$\vec{z}^{i} = \vec{r}^{i} + \theta^{i}\vec{z}^{i-1} \tag{38}$$

By solving for λ^i in Eq. (34) and substituting this into the potential function that minimizes the potential energy of the FE solution, the following optimal form for λ^i is found:

$$\lambda^{i} = \frac{\vec{z}^{T,i}\vec{r}^{i}}{\vec{z}^{T,i}\mathbf{K}\vec{z}^{i}} \tag{39}$$

The point at which the iterations are stopped is based on the energy norm discussed in the previous section. The error tolerance for the PCG method is set to 0.000001, and so the iterations are halted once:

$$\left(\vec{a}^{T,i+1} - \vec{a}^{T,i}\right)\mathbf{K}\left(\vec{a}^{i+1} - \vec{a}^{i}\right) < 0.000001 \tag{40}$$

The PCG method has convergence behavior dictated by:

$$\|\vec{a} - \vec{a}^{i}\|_K \leq \left(\frac{\sqrt{C(\mathbf{K})}-1}{\sqrt{C(\mathbf{K})}+1}\right)^{i}\|\vec{a} - \vec{a}^{1}\|_K \tag{41}$$

In other words, the convergence speed is governed by the condition number. The identity matrix has the lowest possible condition number of 1, but real physical problems tend to have much larger condition numbers. The higher the condition number, then the slower the rate of convergence. Preconditioning can be used to transform the linear system $\mathbf{K}\vec{a} = \vec{R}$ to a different domain in which the condition number is higher. The preconditioning matrix $\mathbf{T}$ is used to introduce this change of variables:

$$\mathbf{K}\vec{a} = \vec{R} \quad \rightarrow \hat{\mathbf{K}}\hat{\vec{a}} = \hat{\vec{R}} \tag{42}$$

where

$$\begin{aligned} \hat{\vec{a}} &= \mathbf{T}^{-1}\vec{a} \\ \hat{\mathbf{K}} &= \mathbf{T}^T\mathbf{K}\mathbf{T} \\ \hat{\vec{R}} &= \mathbf{T}^T\vec{R} \end{aligned} \tag{43}$$

Then, it is the condition number of $\hat{\mathbf{K}}$ that controls the convergence. The best possible preconditioner is to set $\mathbf{T} = \mathbf{L}^{-T}$, where $\mathbf{L}\mathbf{L}^T = \mathbf{K}$ and $\mathbf{L}$ is a lower-triangular matrix. With this preconditioner, Eq. (42) becomes:

$$\underbrace{\mathbf{L}^{-1}\mathbf{L}}_{\mathbf{I}}\underbrace{\mathbf{L}^T\mathbf{L}^{-T}}_{\mathbf{I}}\hat{\vec{a}} = \hat{\vec{R}} \tag{44}$$

which then gives $\mathbf{I}\hat{\vec{a}} = \hat{\vec{R}}$, which is an incredibly easy system to solve numerically, since then $\hat{\vec{a}} = \hat{\vec{R}}$. However, the cost to form $\mathbf{L}$ is of order N^3, and hence defeats the purpose of using the PCG method, since standard Gaussian elimination also scales as approximately N^3. Hence, a simpler preconditioner is selected here. For this assignment, a diagonal preconditioner is used:

$$T_{ij} = \frac{1}{\sqrt{K_{ij}}}\delta_{ij} \tag{45}$$

where δ_{ij} is the Kronecker delta. With this preconditioner, the transformations described in Eq. (43) are made. Then, the PCG method described earlier is performed on this preconditioned system. Once reaching an acceptable error tolerance, the result is transformed back to the actual problem domain to give the final result.

It should be noted that this assignment was completed in two different ways. The first of these ways takes advantage of the sparse matrix system in Matlab to perform the PCG solve. The function to perform the PCG solve using this method is shown in `PCG.m`. On the other hand, because the problem prompted us to perform the multiplication element-by-element, that is also performed as an alternative to the sparse PCG solve. This is about two times slower than simply using the sparse system in Matlab, likely due to code inefficiencies. This element-by-element method is shown in `PCG_element_by_element.m`.

3 Solution Results and Discussion

Fig. 2 shows the nodal values (the solution) for $N = 100$, 1000, and 10000. Because it is very difficult to see any differences between the nodal values for these choices of elements, Fig. 3 shows the difference between the analytical solution and the solutions for $N = 100$, 1000, and 10000 elements. As can be seen, with more elements, the solution is much closer to the true solution. This is to be expected because the greater the number of elements, the greater the number of degrees of freedom that can be used to capture the behavior of the true solution.

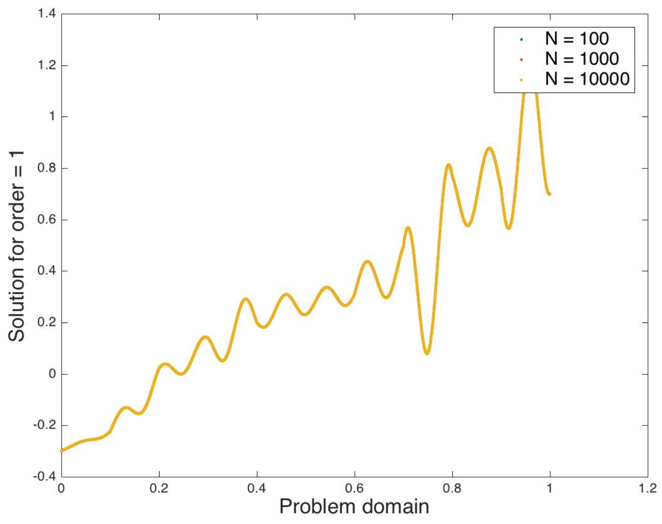

Figure 2. Nodal values (the solution) for $N = 100, 1000$, and 10000.

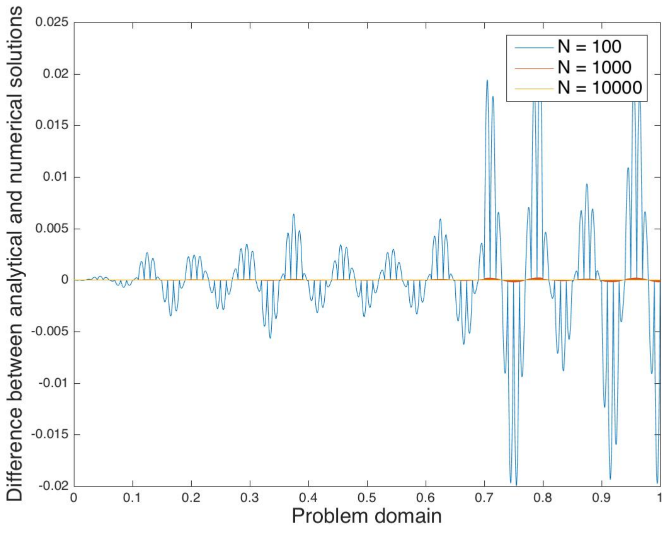

Figure 3. Difference between the analytical solution and the solutions for N = 100, 1000, and 10000 elements.

Fig. 4 shows the energy norm as a function of the number of elements. As can be seen, the energy norm decreases linearly with the number of elements when plotted on a log-log scale. The energy norm converges according to the fundamental interpolation theorem from functional analysis:

$$\|u - u^h\|_{E(\Omega)} \leq C(u,p)h^{\min{(r-1,p)}} \tag{46}$$

where C is an unknown constant, r is the regularity, and p is the polynomial order of the approximation functions. Hence, because linear elements are used in this assignment, the solution converges as a first-order method when measured in the energy norm, and hence the slope of the line in Fig. 4 should have a magnitude of 1 for linear elements. This slope is observed in the figure below, demonstrating the validity of the fundamental interpolation theorem.

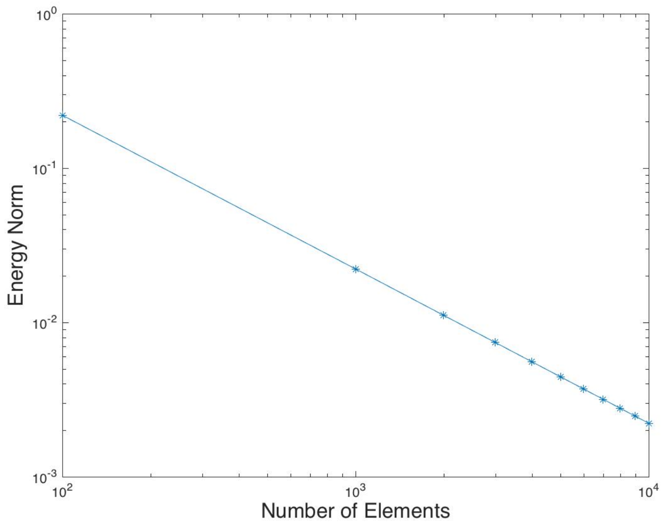

Figure 4. Energy norm as a function of the number of elements. For $N = 100$ elements, $e^N = 0.2199$, for $N = 1000$, $e^N = 0.0222$ and for $N = 10000$, $e^N = 0.0022$.

Fig. 5 shows the number of PCG iterations needed to reach a stopping tolerance of 0.000001 as a function of the number of elements. While used in this assignment as an iterative method, the PCG method is also technically a direct method when allowed to perform N iterations. After N iterations, the PCG method will give nearly the exact solution, and because we required such a stringent stopping tolerance, this was essentially requiring the PCG method to find the exact solution, hence why N iterations were required. So, for any value of N for a stopping tolerance of 0.000001, the PCG solver converges in N iterations. This behavior is even observed (with small deviations) with higher-order elements. For instance, with 100 quadratic elements, 211 PCG iterations are required, while there are 201 nodes. For cubic elements, a similar scaling is observed, but for linear elements, it was always observed that N iterations were required. This occurs due to the relatively poor choice of a preconditioner. Because the preconditioner is so poor, the PCG method then performs poorly, and requires all N theoretical iterations to converge to the solution. This is why, in the last iteration, there is a drop of about 10 orders of magnitude in the PCG error as measured by Eq. (40) - it is precisely due to the fact that the PCG method is a direct method if allowed to have N iterations.

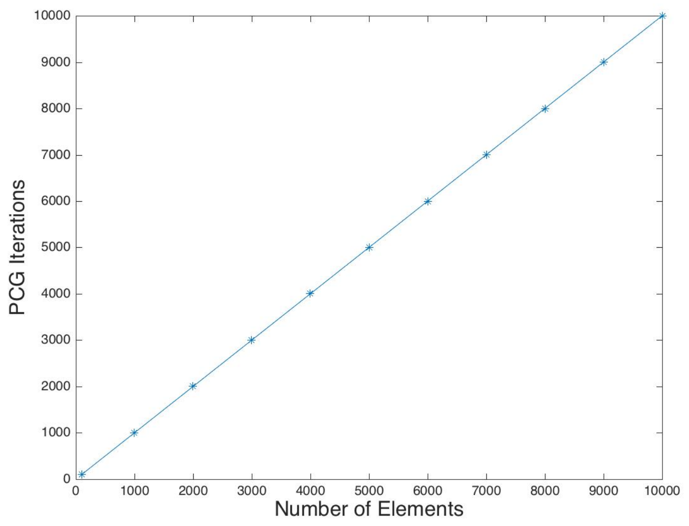

Figure 5. Number of PCG iterations required to reach a stopping tolerance of 0.000001. For $N = 100$, 100 iterations are required; for $N = 1000$, 1000 iterations, and for $N = 10000$, 10000 iterations are needed.

For example, for 1000 elements, Fig. 6 shows the error as computed from Eq. (40). As can be seen, in the last iteration, a significant drop in the error occurs because we're essentially requiring the PCG method to provide the exact answer, which requires N iterations because the PCG method is technically a direct method.

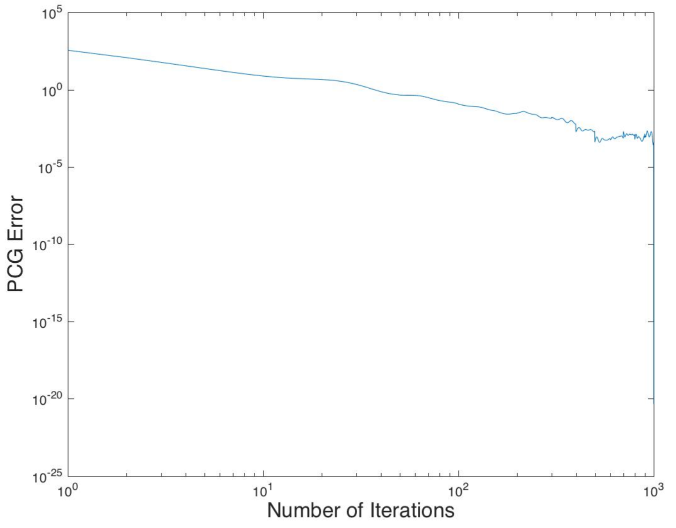

Figure 6. PCG error as computed from Eq. (40) as a function of the iteration number.

Fig. 7 shows the potential energy as calculated from Eq. (32) for the analytical and finite element solutions. As can be seen, as the accuracy of the finite element solution improves, the potential energy asymptotically approaches the potential energy of the true solution. This feature of the "best approximation property" is a significant advantage for problems with symmetric stiffness matrices because it means that the finite element solution will monotonically approach the exact solution as the mesh is refined, and that the error for any other kinematically admissible solution will be greater than that of the finite element solution (for the same number of unknowns).

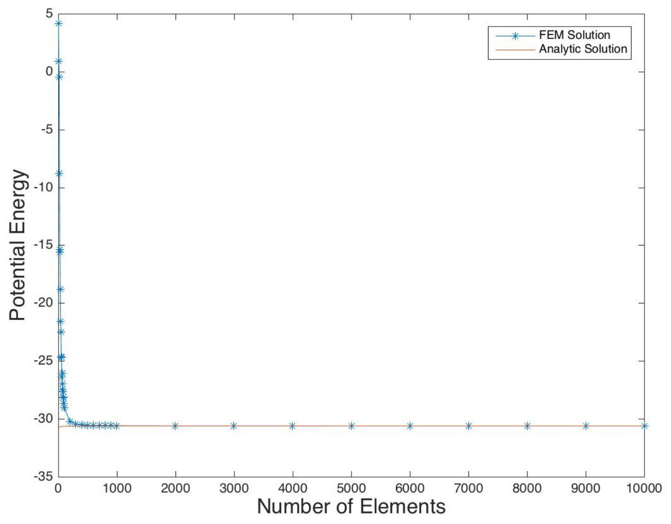

Figure 7. Potential energy as a function of number of elements.

When plotted on log-log coordinates, where the absolute value of the potential energies must be taken (in order to avoid problems associated with logarithms of negative numbers), the above plot can similarly be shown as follows. It is not clear why the potential energies of both the true and finite element solutions are negative, but the plots show the correct trend in that as the mesh is refined, the potential energy of the finite element solution approaches that of the analytical solution, but never becomes lower than that of the analytical solution.

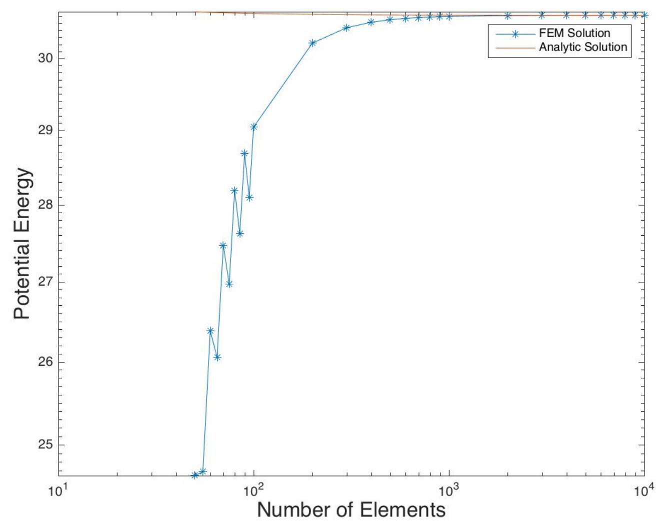

Figure 8. Potential energy as a function of number of elements.

To further illustrate how the difference between the analytical and finite element solution potential energies goes to zero as the mesh is refined, Fig. 9 shows the difference between the finite element and analytical potential energies (finite element - analytical) as the mesh is refined. Except for oscillations that occur at relatively coarse meshes that are not able to fully capture the material information, the difference in the potential energy decreases linearly when plotted on a log-log plot.

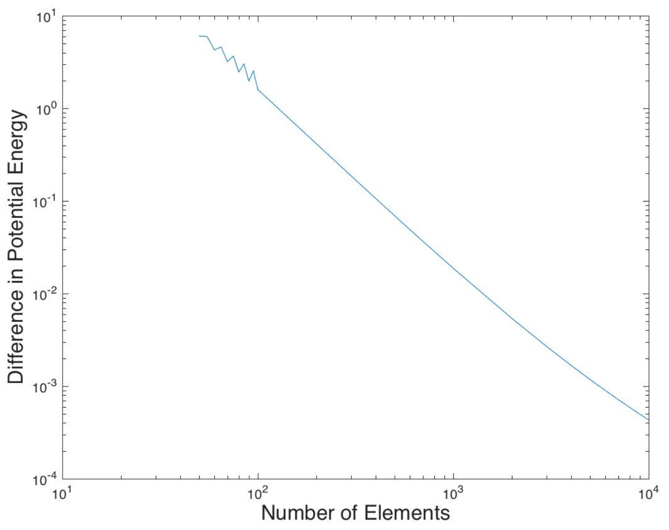

Figure 9. Difference between the finite element and analytical potential energies (finite element - analytical) as a function of number of elements.

Because the potential energy is defined such that:

$$\|u - u^N\|^2_{E(\Omega} = 2\left(\mathcal{J}(u^N) - \mathcal{J}(u)\right) \tag{47}$$

and from the fundamental interpolation theorem in Eq. (46), the difference in the potential energies can be related to h by:

$$2\left(\mathcal{J}(u^N) - \mathcal{J}(u)\right) \leq \left(C(u,p)h^{\min(r-1,p)}\right)^2 \tag{48}$$

This can equivalently be stated as:

$$2\left(\mathcal{J}(u^N) - \mathcal{J}(u)\right) \leq C(u,p)^2 h^{2\min(r-1,p)} \tag{49}$$

Since the quantity $\left(\mathcal{J}(u^N) - \mathcal{J}(u)\right)$ is plotted in Fig. 9, it is expected that for linear elements for a sufficiently smooth solution, that the difference in the potential energy converge with a slope of 2 when plotted on a log-log plot. This slope is observed in Fig. 9, and hence the utility of the potential energy in serving as a convergence estimate (if the true solution is known) has been shown to be valid.

To summarize, the potential energy, energy norm, and number of PCG iterations for 100, 1000, and 10000 elements is given in the table below. Again, the potential energy of the analytical solution was computed to be -30.61, and hence the finite element solution approaches this value as the mesh becomes more and more refined.

Table 2. Requested information for 100, 1000, and 10000 elements.

Number of elements	Potential energy	Energy norm	PCG iterations
100	-29.0434	0.2199	100
1000	-30.5996	0.0222	1000
10000	-30.6155	0.0022	10000

4 Appendix

This section contains the code used for this modeling. The main program is `FEProgram.m`, and functions perform specialized tasks for a high degree of modularity.

4.1 FEProgram.m

This is the main code used for the problem solving.

```
clear all

% select which type of plot you want to make - at least one flag must equal 1
pe_plot_flag = 1;                 % 1 - plot the potential energy as a function
    ↪ of N
e_plot_flag = 0;                  % 1 - plot the energy norm as a function of N
N_plot_flag = 0;                  % 1 - plot the solutions for various N

L = 1.0;                          % problem domain
k_freq = 12;                      % forcing frequency
left = 'Dirichlet';               % left boundary condition
left_value = -0.3;                % left Dirichlet boundary condition value
right = 'Dirichlet';              % right boundary condition type
right_value = 0.7;                % right Dirichlet boundary condition value
tolerance = 0.04;                 % convergence tolerance
energy_norm = tolerance + 1;      % arbitrary initialization value
fontsize = 16;                    % fontsize for plots
pcg_error_tol = 0.000001;         % error tolerance for PCG
precondition = 'precondition';    % 'nopreconditi' for no preconditioning

if (N_plot_flag)
     N_elem = [100, 1000, 10000];                 % num_elem to cycle through for
         ↪  soln plots
elseif (pe_plot_flag || e_plot_flag)
     N_elem = [50:5:100, 200:100:1000, 2000:1000:10000];          % num_elem to
         ↪ cycle through for e_N vs. N
else
    disp('Either␣N_plot_flag␣or␣pe_plot_flag␣has␣to␣equal␣1.');
end

Order = [2];                 % shape function (orders - 1) to cycle thru

% specify E over the domain in a block structure
E_blocks = [2.5, 1.0, 1.75, 1.25, 2.75, 3.75, 2.25, 0.75, 2.0, 1.0];
space_blocks = 0.1:0.1:L;

for shape_order = Order
    clearvars permutation

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

% initialize vectors for collecting error
e = 1;
pe = ones(1, length(N_elem));
pe_2 = ones(1, length(N_elem));
pe_a = ones(1, length(N_elem));
pe_a2 = ones(1, length(N_elem));

for num_elem = N_elem
```

```
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 E into the physical domain
[E_physical_domain] = PhysicalInterpolation(physical_domain, space_blocks,
    ↪  E_blocks);

% --- ANALYTICAL SOLUTION --- %
[solution_analytical, solution_analytical_derivative, gamma] =
    ↪ AnalyticalSolution(L, space_blocks, left_value, E_blocks,
    ↪ E_physical_domain, right_value, k_freq, physical_domain);

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

j = 1;
k = 1;
result = zeros(length(coordinates(:,1)), 1);
for i = 1:length(physical_domain)
    if (abs(physical_domain(i) - coordinates(j)) < 1e-10)
        result(k) = solution_analytical(i);
        k = k + 1;
        j = j + 1;
    else
    end
end

% interpolate E into the an elemental basis
[E_elem, right_endpoint_index, right_endpoint_coordinate] =
    ↪ ElementInterpolation(coordinates, num_elem, num_nodes_per_element,
    ↪ space_blocks, E_blocks);

% specifiy the boundary conditions
[dirichlet_nodes, neumann_nodes, a_k] = BCnodes(left, right, left_value,
    ↪ right_value, num_nodes);

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

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

for elem = 1:num_elem
    k = 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
                f(i) = f(i) - wt(l) * x_xe * (k_freq .^ 3) * cos(gamma * x_xe
                    ↪ ) * N(i) * dx_dxe;

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

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

% assemble into the global matrices
for elem = 1:num_elem
    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));
    end

    for i = 1:length(f)
        F(LM(elem, i)) = F((LM(elem, i))) + F_cell{1,elem}(i);
    end
end

K = sparse(K);

% 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 using Gaussian elimination for comparison
a_u_condensed = K_uu \ (F_u - K_uk * dirichlet_nodes(2,:)');

% perform the solve using the global matrices
[a_u_condensed, pcg_error] = PCG(K_uu, F_u, K_uk, dirichlet_nodes,
    ↪ pcg_error_tol, precondition);

% perform the solve element-by-element
%[a_u_condensed] = PCG_element_by_element(F_u, K_uk, K_cell, LM, num_nodes,
    ↪ dirichlet_nodes, num_elem, num_nodes_per_element, pcg_error_tol,
    ↪ precondition);

% 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);

% compute the energy norm
energy_norm_bottom = sqrt(trapz(physical_domain,
    ↪ solution_analytical_derivative .* E_physical_domain .*
    ↪ solution_analytical_derivative));
energy_norm_top = sqrt(trapz(physical_domain, (solution_derivative_FE -
    ↪ solution_analytical_derivative) .* E_physical_domain .* (
    ↪ solution_derivative_FE - solution_analytical_derivative)));
energy_norm = energy_norm_top ./ energy_norm_bottom;

% compute the potential energy - FE
B_uNuN = trapz(physical_domain, solution_derivative_FE .* E_physical_domain .*
    ↪  solution_derivative_FE);
L_uN = - trapz(physical_domain, physical_domain .* k_freq .^3 .* cos(gamma .*
    ↪ physical_domain) .* solution_FE);
potential_energy = 0.5 .* B_uNuN - L_uN;

% compute the potential energy - analytic
B_uu = trapz(physical_domain, solution_analytical_derivative .*
    ↪ E_physical_domain .* solution_analytical_derivative);
L_u = - trapz(physical_domain, physical_domain .* k_freq .^3 .* cos(gamma .*
    ↪ physical_domain) .* solution_analytical);
pe_a = 0.5 .* B_uu - L_u;

if (N_plot_flag)
    plot(coordinates(:,1), a, '.')
    hold on
end

e_N(e) = energy_norm;
pe(e) = potential_energy;
pea(e) = pe_a;
e = e + 1;

end

if (N_plot_flag)
```

```
    txt = cell(length(N_elem),1);
    for i = 1:length(N_elem)
       txt{i}= sprintf('N = %i', N_elem(i));
    end
    h = legend(txt);
    set(h, 'FontSize', fontsize - 2);
    xlabel('Problem domain', 'FontSize', fontsize)
    ylabel(sprintf('Solution for order = %i', shape_order - 1), 'FontSize',
       ↪ fontsize)

    saveas(gcf, sprintf('Nplot', shape_order - 1), 'jpeg')
    close all
end

if (pe_plot_flag || e_plot_flag)
    if (pe_plot_flag)
        plot(N_elem, pe, '*-', N_elem, pea, '-')
        legend('FEM Solution','Analytic Solution')
        ylabel_str = 'Potential Energy';
        filename = 'pe_vs_N';
    else
        loglog(N_elem, e_N, '*-')
        ylabel_str = 'Energy Norm';
        filename = 'eN_vs_N';
    end

    xlabel('Number of Elements', 'FontSize', fontsize)
    ylabel(ylabel_str, 'FontSize', fontsize)
    saveas(gcf, filename, 'jpeg')
    close all
end

end

% --- analytical solution plot --- %
% plot(physical_domain, solution_analytical)
% xlabel('Physical Domain', 'FontSize', fontsize)
% ylabel('Solution u(x)', 'FontSize', fontsize)
% saveas(gcf, 'AnalyticalSoln2', 'jpeg')
% close all


% --- plot of error as a function of iteration --- %
% loglog(1:1:length(pcg_error), pcg_error)
% xlabel('Iteration Number', 'FontSize', fontsize)
% ylabel('PCG Error','FontSize', fontsize)
% close all


% --- plot of number of iterations as a function of N_elem --- %
% plot([100, 1000:1000:10000], [100, 1000:1000:10000], '*-')
% xlabel('Number of Elements', 'FontSize', fontsize)
% ylabel('PCG Iterations', 'FontSize', fontsize)
% saveas(gcf, 'PCGIterations', 'jpeg')
% close all


% --- plot pcg_error as a function of number of iterations --- %
% loglog(1:1:length(pcg_error), pcg_error)
% xlabel('Number of Iterations', 'FontSize', fontsize)
% ylabel('PCG Error', 'FontSize', fontsize)
% saveas(gcf, 'PCGerror', 'jpeg')
% close all
```

4.2 `AddDirichlet.m`

This function adds in dummy Dirichlet node placeholders.

```
function [result] = AddDirichlet(vector, dirichlet_nodes)
% This function adds zero values for the locations of Dirichlet nodes

j = 1;
for i = 1:(length(vector) + length(dirichlet_nodes(1,:)))
    if (find(dirichlet_nodes(1,:) == i))
        result(i) = 0;
    else
        result(i) = vector(j);
        j = j + 1;
    end
end

result = result';

end
```

4.3 `AnalyticalSolution.m`

This function computes the analytical solution.

```
function [solution_analytical, solution_analytical_derivative, gamma] =
    ↪ AnalyticalSolution(L, space_blocks, left_value, E_blocks,
    ↪ E_physical_domain, right_value, k_freq, physical_domain)
gamma = 2 * pi * k_freq ./ L;

C_unknowns = zeros(1, length(space_blocks) + 1);
coeff_matrix = zeros(length(space_blocks) + 1, length(space_blocks) + 1);
coeff_vector = zeros(length(space_blocks) + 1, 1);

% left boundary condition
coeff_matrix(1,1) = 1;
coeff_vector(1) = left_value;

% right boundary condition
coeff_matrix(end, end) = L / E_blocks(end);
coeff_matrix(end, end - 1) = 1;
coeff_vector(end) = right_value - (1 / E_blocks(end)) * RHSFunction(gamma,
    ↪ k_freq, space_blocks, E_blocks, length(space_blocks));
```

```
% intermediate conditions
for i = 1:(length(space_blocks) - 1)
    coeff_matrix(i+1, i) = 1;
    coeff_matrix(i+1, i+1) = -1;
    coeff_matrix(i+1, end) = (space_blocks(i) / E_blocks(i) - space_blocks(i)
       ↪ / E_blocks(i+1));
    coeff_vector(i+1) = - RHSFunction(gamma, k_freq, space_blocks, E_blocks, i
       ↪ ) * (1/E_blocks(i) -1/E_blocks(i+1));
end

coeff_soln = coeff_matrix\coeff_vector;

% assemble block-oriented E and C_2 into physical_domain structure
[C_2_physical_domain2] = PhysicalInterpolation(physical_domain, space_blocks,
   ↪ coeff_soln(1:end-1));
C_1_physical_domain2 = coeff_soln(end) .* ones(1, length(physical_domain));

term1_2 = 2 * (k_freq .^ 3) * sin(gamma .* physical_domain) ./ (
   ↪ E_physical_domain .* gamma .^3);
term2_2 = (k_freq .^ 3) * physical_domain .* cos(gamma .* physical_domain) ./
   ↪ (E_physical_domain .* gamma .^ 2);
solution_analytical = C_2_physical_domain2 + term1_2 - term2_2 +
   ↪ C_1_physical_domain2 .* physical_domain ./ E_physical_domain;

term1_1 = 2 * (k_freq .^ 3) * cos(gamma .* physical_domain) .* gamma ./ (
   ↪ E_physical_domain .* gamma .^3);
term2_1 = ((k_freq .^ 3) ./ (E_physical_domain .* gamma .^ 2)) .* (
   ↪ physical_domain .* gamma .* - sin(gamma .* physical_domain) + cos(gamma
   ↪ .* physical_domain));
solution_analytical_derivative = (k_freq^3) * (gamma .* physical_domain .* sin
   ↪ (gamma .* physical_domain) + cos(gamma .* physical_domain)) ./ (
   ↪ E_physical_domain .* gamma .^ 2) + C_1_physical_domain2 ./
   ↪ E_physical_domain;



function [RHS_value] = RHSFunction(gamma, k_freq, space_blocks, E_blocks,
   ↪ node_number)
% function to repeatedly evaluate \circled{A} in the document

loc = space_blocks(node_number);
RHS_value = - loc * (k_freq^3) * cos(gamma * loc) / (gamma ^2) + 2 * k_freq ^
   ↪ 3 * sin(gamma * loc) / (gamma ^ 3);
end


end
```

4.4 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)

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

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

4.5 condensation.m

This function separates out the matrix equation as in Eq. (25).

```
% 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);
```

4.6 CutoffDirichlet.m

This function removes Dirichlet nodes from a vector.

```
function [result] = CutoffDirichlet(vector, dirichlet_nodes)
% This function removes Dirichlet nodes from a vector

j = 1;
for i = 1:length(vector)
    if (find(dirichlet_nodes(1,:) == i))
    else
        result(j) = vector(i);
        j = j + 1;
    end
end

result = result';

end
```

4.7 ElementInterpolation.m

This function interpolates the block-structure of E into an elemental basis (E as a function of the element number).

```
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
```

4.8 mesh.m

This function performs the meshing.

```
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
```

4.9 `Mult.m`

This functions performs multiplication element-by-element of a matrix and a vector.

```
function [result] = Mult(K_cell, num_elem, vector, num_nodes,
    ↪ num_nodes_per_element, dirichlet_nodes, LM)
% This function multiplies the global K by a vector without needing to
% explicitly form the global K and by avoiding unnecessary multiplication
% of zeros. You don't need to cut out the Dirichlet nodes.

result = zeros(num_nodes, 1);
for elem = 1:num_elem
    for i = 1:num_nodes_per_element
        for j = 1:num_nodes_per_element
            %if find(dirichlet_nodes(1,:) == LM(elem, j))
            if ((LM(elem, j) == 1) || (LM(elem, j) == num_nodes)) % works good
                ↪ enough for only 2 dirichlet points
            else
                result(LM(elem,i)) = K_cell{1,elem}(i,j) * vector(LM(elem, j))
                    ↪ + result(LM(elem,i));
            end
        end
    end
end

%result = result';
%result = CutoffDirichlet(result, dirichlet_nodes);

end
```

4.10 `NodalInterpolation.m`

This function interpolates a vector defined over the physical domain into one defined over a nodal domain.

```
function [result] = NodalInterpolation(vector, coordinates, physical_domain)
% This function interpolates a vector in the physical domain to a vector
% defined only at nodal values.

j = 1;
k = 1;
for i = 1:length(physical_domain)
    if (abs(physical_domain(i) - coordinates(j)) < 1e-10)
        result(k) = vector(i);
```

```
            k = k + 1;
            j = j + 1;
        else
        end
end

end
```

4.11 PCG.m

This function performs the Preconditioned Conjugate Gradient solve of the matrix system (not element-by-element).

```
function [a_u_condensed, pcg_error] = PCG(K_uu, F_u, K_uk, dirichlet_nodes,
    ↪ pcg_error_tol, precondition)

K = K_uu;
R = F_u - K_uk * dirichlet_nodes(2,:)';

% pick the initial guess for the solution
soln_iter = ones(1, length(K_uu))';

if (precondition == 'precondition')
    % diagonal preconditioner
    T = diag(ones(1,length(K)));
    for i = 1:length(K(1,:))
        T(i,i) = 1 ./ sqrt(K(i,i));
    end

    % transform to the preconditioned regime
    K = transpose(T) * K * T;
    R = transpose(T) * R;
end

% compute the initial z from the residual
r = R - K * soln_iter;
z = r;

% compute the initial lambda
lambda = transpose(z) * r / (transpose(z) * K * z);

% store the previous iteration for convergence estimates
soln_prev = soln_iter;

% perform the first update
soln_iter = soln_prev + lambda * z;

j = 1;
pcg_error(j) = transpose(soln_iter - soln_prev) * K * (soln_iter - soln_prev);
    ↪ %/(transpose(soln_prev) * K * soln_prev);
num_updates = 1;
% perform all subsequent updates
while pcg_error > pcg_error_tol
    z_prev = z;
    soln_prev = soln_iter;

    r = R - K * soln_iter;
    K_z_prev = K * z_prev;
    theta = - transpose(r) * K_z_prev / (transpose(z_prev) * K_z_prev);

    z = r + theta * z_prev;
    lambda = transpose(z) * r / (transpose(z) * K * z);
    soln_iter = soln_prev + lambda * z;

    pcg_error(j+1) = transpose(soln_iter - soln_prev) * K * (soln_iter -
       ↪ soln_prev);% / (transpose(soln_prev) * K * soln_prev);
    num_updates = num_updates + 1;
    j = j + 1;
end

a_u_condensed = soln_iter;

if (precondition == 'precondition')
    % transform back from the preconditioned regime
    a_u_condensed = T * a_u_condensed;
end

sprintf('Number␣of␣PCG␣iterations:␣%i', num_updates)

end
```

4.12 PCG_element_by_element.m

This function performs the Preconditioned Conjugate Gradient solve of the matrix system (element-by-element).

```
function [a_u_condensed] = PCG_element_by_element(F_u, K_uk, K_cell, LM,
    ↪ num_nodes, dirichlet_nodes, num_elem, num_nodes_per_element,
    ↪ pcg_error_tol, precondition)

% figure out how to eliminate the CutoffDirichlet and AddDirichlet
% functions

% This function solves the system K * a = R

R = F_u - K_uk * dirichlet_nodes(2,:)';
R = AddDirichlet(R, dirichlet_nodes);

% if (precondition == 'precondition')
%     % diagonal preconditioner
%     T = diag(ones(1, num_nodes));
%     for i = 1:length(K(1,:))
%         T(i,i) = 1 ./ sqrt(K(i,i));
%     end
%
%     % transform to the preconditioned regime
%     K = transpose(T) * K * T;
%     R = transpose(T) * R;
% end

% pick the initial guess for the solution
soln_iter = ones(1, num_nodes)';

% compute the initial z from the residual
r = R - Mult(K_cell, num_elem, soln_iter, num_nodes, num_nodes_per_element,
    ↪ dirichlet_nodes, LM);
z = r;

% compute the initial lambda
%lambda = transpose(CutoffDirichlet(z, dirichlet_nodes)) * CutoffDirichlet(r,
    ↪ dirichlet_nodes) / (transpose(CutoffDirichlet(z, dirichlet_nodes)) *
    ↪ CutoffDirichlet(Mult(K_cell, num_elem, z, num_nodes,
    ↪ num_nodes_per_element, dirichlet_nodes, LM), dirichlet_nodes));
lambda = transpose(z(2:(end-1))) * r(2:(end-1)) / (transpose(z(2:(end-1))) *
    ↪ CutoffDirichlet(Mult(K_cell, num_elem, z, num_nodes,
    ↪ num_nodes_per_element, dirichlet_nodes, LM), dirichlet_nodes));

% store the previous iteration for convergence estimates
soln_prev = soln_iter;

% perform the first update
soln_iter = soln_prev + lambda * z;


j = 1;
% only calculate error using the non-Dirichlet nodes
K_soln_iter = Mult(K_cell, num_elem, soln_iter - soln_prev, num_nodes,
    ↪ num_nodes_per_element, dirichlet_nodes, LM);
pcg_error(j) = transpose(soln_iter(2:(end-1)) - soln_prev(2:(end-1))) *
    ↪ K_soln_iter(2:(end-1));
num_updates = 1;

% perform all subsequent updates
while pcg_error(j) > pcg_error_tol
    z_prev = z;
    soln_prev = soln_iter;

    r = R - Mult(K_cell, num_elem, soln_iter, num_nodes, num_nodes_per_element
        ↪ , dirichlet_nodes, LM);
    K_z_prev = Mult(K_cell, num_elem, z_prev, num_nodes, num_nodes_per_element
        ↪ , dirichlet_nodes, LM);
    theta = - transpose(r(2:(end-1))) * K_z_prev(2:(end-1)) / (transpose(
        ↪ z_prev(2:(end-1))) * K_z_prev(2:(end-1)));

    z = r + theta * z_prev;
    K_z = Mult(K_cell, num_elem, z, num_nodes, num_nodes_per_element,
        ↪ dirichlet_nodes, LM);
    lambda = transpose(z(2:(end-1))) * r(2:(end-1)) / (transpose(z(2:(end-1)))
        ↪ * K_z(2:(end-1)));
    soln_iter = soln_prev + lambda * z;

    K_soln_iter_soln_prev = Mult(K_cell, num_elem, soln_iter - soln_prev,
        ↪ num_nodes, num_nodes_per_element, dirichlet_nodes, LM);
    pcg_error(j+1) = transpose(soln_iter(2:(end-1)) - soln_prev(2:(end-1))) *
        ↪ K_soln_iter_soln_prev(2:(end-1));
    num_updates = num_updates + 1;
    j = j + 1;
end

% if (precondition == 'precondition')
%     % transform back from the preconditioned regime
%     a_u_condensed = T * a_u_condensed;
% end

a_u_condensed = CutoffDirichlet(soln_iter, dirichlet_nodes);

sprintf('Number␣of␣PCG␣iterations:␣%i', num_updates)

end
```

4.13 permutation.m

This function determines the permutation matrix for use with the connectivity matrix.

```
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
```

4.14 PhysicalInterpolation.m

This function interpolates a block-structure E into the physical domain (E as a function of position).

```
function [var_physical_domain] = PhysicalInterpolation(physical_domain,
    ↪ 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.

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
```

4.15 `postprocess.m`

This function postprocesses the FE solution and transforms it back to the physical domain using a linear system solve as described in Eq. (27).

```
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
```

4.16 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
```

4.17 shapefunctions.m

This function contains the library of 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;
    case 3
        N(1) = xe .* (xe - 1) ./ 2;
        N(2) = - (xe - 1) .* (1 + xe);
        N(3) = xe .* (1 + xe) ./ 2;
        dN(1) = xe - 1/2;
        dN(2) = -2 .* xe;
        dN(3) = 1/2 + xe;
    case 4
        N(1) = (9/16) * (1 - xe) * (1/3 - xe) * (-1/3 - xe);
        N(2) = (-27/16) * (1 - xe) * (1/3 - xe) * (-1 - xe);
        N(3) = (27/16) * (1 - xe) * (-1/3 - xe) * (-1 - xe);
        N(4) = (-9/16) * (1/3 - xe) * (-1/3 - xe) * (-1 - xe);
        dN(1) = (9/16) * (-3 * xe.^ 2 + 2 * xe + 1/9);
        dN(2) = (-27/16) * (-3 * xe .^ 2 + 2 .* xe ./ 3 + 1);
        dN(3) = (27/16) * (-3 * xe .^ 2 - 2.* xe ./ 3 + 1);
        dN(4) = (-9/16) * (-3 * xe ^ 2 - 2 * xe + 1/9);
    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
```