ME 280a: HW 4

April Novak

November 1, 2016

1 Introduction and Objectives

The purpose of this study is to solve a simple finite element (FE) problem and perform Adaptive Mesh Refinement (AMR) in order to refine the mesh in locations where the solution is highly-varying. AMR is an important technique for reducing the necessary run-time of a simulation, and is especially important for large simulations where every method possible to reduce the computational cost is needed.

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:

$$\frac{d}{dx}\left(E(x)\frac{du}{dx}\right) = f(x) \tag{1}$$

where E is the modulus of elasticity, u is the solution, x is the spatial variable, and $f(x)$ is a forcing function whose form is to be determined in order to obtain the solution $u(x) = \cos(10\pi x^5)$. In order to determine the form of $f(x)$, substitute the manufactured solution into Eq. (1) to give:

$$\cancel{\frac{dE(x)}{dx}}\cancel{\frac{du}{dx}} + E(x)\frac{d^2u}{dx^2} = f(x) \quad \rightarrow \quad E\frac{d^2}{dx^2}\left(\cos(10\pi x^5)\right) = f(x) \tag{2}$$

where the fact that $E(x)$ is constant has been implemented. This gives the following form for $f(x)$:

$$f(x) = -E\left(200\pi x^3 \sin(10\pi x^5) + 2500\pi^2 x^8 \cos(10\pi x^5)\right) \tag{3}$$

The boundary conditions for this problem are Dirichlet at both endpoints such that the assumed solution is satisfied. At $x = 0$, $u(0) = \cos(0) = 1$. At $x = L$, $u(L) = \cos(10\pi L^5)$.

$$\begin{aligned} u(0) &= 1 \\ u(L) &= \cos(10\pi L^5) \end{aligned} \tag{4}$$

This problem will be solved using the FE method, and AMR will be performed in order to most effectively solve the problem.

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:

$$
\begin{aligned}
u \approx u^N &= \sum_{j=1}^{N} a_j \phi_j \\
\psi &= \sum_{i=1}^{N} b_i \phi_i
\end{aligned}
\tag{5}
$$

Galerkin's method is stated as:

$$
r^N \cdot u^N = 0 \tag{6}
$$

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, $d\Omega$.

$$
\int_\Omega \frac{d}{dx}\left(E(x)\frac{du}{dx}\right)\psi d\Omega - \int_\Omega f(x)\psi d\Omega = 0 \tag{7}
$$

Applying integration by parts to the first term:

$$
-\int_\Omega E(x)\frac{du}{dx}\frac{d\psi}{dx}d\Omega + \int_{\partial\Omega} E(x)\frac{du}{dx}\psi d(\partial\Omega) - \int_\Omega f(x)\psi d\Omega = 0 \tag{8}
$$

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:

$$
\begin{aligned}
-\int_0^L E(x)\frac{du}{dx}\frac{d\psi}{dx}dx + E(x)\frac{du}{dx}\psi\Big|_0^L - \int_0^L f(x)\psi dx = 0 \\
\int_0^L E(x)\frac{du}{dx}\frac{d\psi}{dx}dx = -\int_0^L f(x)\psi dx + E(x)\frac{du}{dx}\psi\Big|_0^L
\end{aligned}
\tag{9}
$$

Inserting the approximation described in Eq. (5):

$$
\int_0^L E(x)\frac{d\left(\sum_{j=1}^N a_j\phi_j\right)}{dx}\frac{d\left(\sum_{i=1}^N b_i\phi_i\right)}{dx}dx = -\int_0^L f(x)\sum_{i=1}^N b_i\phi_i dx + E(x)\frac{du}{dx}\sum_{i=1}^N b_i\phi_i\Big|_0^L \tag{10}
$$

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).

$$
\int_0^L E(x)\frac{d\left(\sum_{j=1}^N a_j\phi_j\right)}{dx}\frac{d\phi_i}{dx}dx = -\int_0^L f(x)\phi_i dx + E(x)\frac{du}{dx}\phi_i\Big|_0^L \tag{11}
$$

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

$$
\int_0^L E(x)\frac{d\left(a_j\phi_j\right)}{dx}\frac{d\phi_i}{dx}dx = -\int_0^L f(x)\phi_i dx + E(x)\frac{du}{dx}\phi_i\Big|_0^L \tag{12}
$$

This produces a system of matrix equations of the form:

$$
\mathbf{K}\vec{a} = \vec{F} \tag{13}
$$

where:

$$
\begin{aligned}
K_{ij} &= \int_0^L E(x)\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx \\
a_j &= a_j \\
F_i &= -\int_0^L f(x)\phi_i dx + E(x)\frac{du}{dx}\phi_i\Big|_0^L
\end{aligned}
\tag{14}
$$

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. (14). In order for these equations to be useful with Gaussian quadrature, they must be transformed to the master element which exists over $-1 \leq \xi \leq 1$:

$$\begin{aligned} K_{ij} = \int_0^L E(x)\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx \rightarrow \int_{-1}^1 E(x(\xi))\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx\left(\frac{dx}{d\xi}\frac{dx}{d\xi}\frac{d\xi}{dx}\right) \rightarrow \int_{-1}^1 E(x(\xi))\frac{d\phi_i}{d\xi}\frac{d\phi_j}{d\xi}dx\left(\frac{d\xi}{dx}\right) \\ a_j = a_j \\ F_i = -\int_0^L f(x)\phi_i dx \rightarrow -\int_{-1}^1 f(x(\xi))\phi_i\frac{dx}{d\xi}d\xi \end{aligned} \tag{15}$$

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:

$$\begin{aligned} \phi_1(\xi) &= \frac{1-\xi}{2}, \quad \frac{d\phi_1(\xi)}{d\xi} = -1/2 \\ \phi_2(\xi) &= \frac{1+\xi}{2}, \quad \frac{d\phi_2(\xi)}{d\xi} = +1/2 \end{aligned} \tag{16}$$

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

$$x(\xi) = \sum_{i=1}^{N} X_i \phi_i(\xi) \tag{17}$$

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. (17) by differentiation:

$$\frac{dx(\xi)}{d\xi} = \sum_{i=1}^{N} X_i \frac{d\phi_i(\xi)}{d\xi} \tag{18}$$

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

$$\begin{aligned} w &= \left[\frac{322-13\sqrt{70}}{900}, \frac{322+13\sqrt{70}}{900}, \frac{128}{225}, \frac{322+13\sqrt{70}}{900}, \frac{322-13\sqrt{70}}{900}\right] \\ x &= \left[-\frac{1}{3}\sqrt{5+2\sqrt{10/7}}, -\frac{1}{3}\sqrt{5-2\sqrt{10/7}}, 0, \frac{1}{3}\sqrt{5-2\sqrt{10/7}}, \frac{1}{3}\sqrt{5+2\sqrt{10/7}}\right] \end{aligned} \tag{19}$$

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. After the global matrix and vector are formed, the global matrix has a banded-diagonal structure.

In order to apply the boundary conditions within the numerical context of the finite element method, the matrix equation in Eq. (13) must be rewritten to reflect that some of the nodal values are actually already specified through the Dirichlet boundary conditions.

$$\begin{bmatrix} K_{kk} & K_{ku} \\ K_{uk} & K_{uu} \end{bmatrix} \begin{bmatrix} x_k \\ x_u \end{bmatrix} = \begin{bmatrix} F_k \\ F_u \end{bmatrix} \tag{20}$$

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

$$\begin{aligned} K_{kk}x_k + K_{ku}x_u &= F_k \\ K_{uk}x_k + K_{uu}x_u &= F_u \end{aligned} \tag{21}$$

Solving this matrix system is sometimes referred to as "static condensation," since the original matrix system in Eq. (13) 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.

Once the solution is obtained, 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:

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{22}$$

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:

$$e^N = \frac{\|u - u^N\|}{\|u\|} \tag{23}$$

where:

$$\|u\| = \sqrt{\int_\Omega \frac{du}{dx} E \frac{du}{dx}} \tag{24}$$

$$\|u - u^N\| = \sqrt{\int_\Omega \frac{d(u-u^N)}{dx} E \frac{d(u-u^N)}{dx}} = \sqrt{\int_\Omega \left(\frac{du}{dx} - \frac{du^N}{dx}\right) E \left(\frac{du}{dx} - \frac{du^N}{dx}\right)} \tag{25}$$

The derivatives of the FE solution are determined according to:

$$\frac{du^N}{dx} = \frac{d}{dx}\sum_{j=1}^{N} a_j \phi_j(x) = \sum_{j=1}^{N} a_j \frac{d\phi_j(x)}{dx} = \sum_{j=1}^{N} a_j \frac{d\phi_j(x)}{d\xi}\frac{d\xi}{dx} \tag{26}$$

while the derivative of the analytical solution is obtained from Eq. (2). The purpose of this assignment is to perform mesh refinement for elements that have high error relative to the manufactured solution $u(x) = \cos(10\pi x^5)$. In this assignment, an error estimate is determined for each element. This error estimate, A_i is defined for each element i as:

$$A_i^2 = \frac{\frac{1}{h_i}\|u - u^N\|^2_{E(\Omega_i)}}{\frac{1}{L}\|u\|^2_{E(\Omega)}} \tag{27}$$

If $A_i > 0.05$, then that element is subdivided into two, and the solution repeated until all elements have satisfactory A_i. Once reaching this point, the mesh is considered sufficiently fine that the error is acceptable.

It should be noted that while the energy norm converges as a first order method (proportional to h) for linear elements, because it is squared in the equation above, A_i^2 converges as first-order as well.

3 Solution Results and Discussion

Fig. 1 shows the FE and analytical solutions for various numbers of elements. To reach a tolerance of 0.05 in Eq. (23) ($e^N \leq 0.05$), 100 elements are needed.

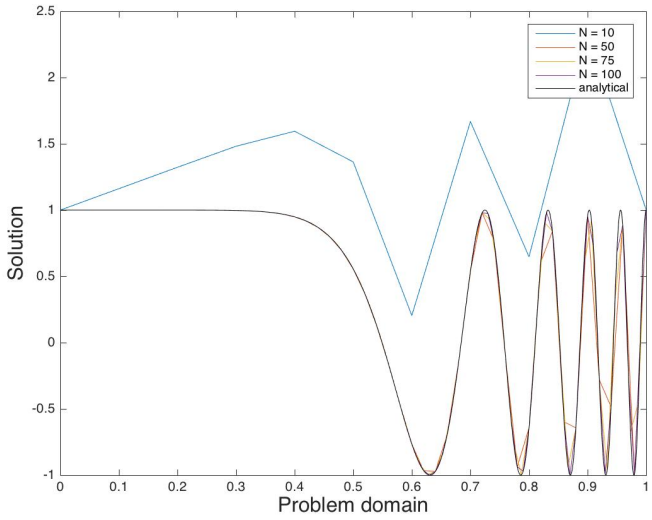

Figure 1. FE and analytical solutions for various numbers of elements.

Fig. 2 shows A_i as calculated from Eq. (27) for a mesh beginning with 20 elements. As can be seen, due to the oscillatory nature of the function towards the right end of the domain, A_i increases since the FE solution does not have a sufficient number of degrees of freedom to accurately capture the behavior of the solution. As the mesh is refined, it is expected that the mesh become increasingly fine when moving in the positive-x direction. With 20 elements, only the first 12 elements have $A_i < 0.05$, and hence the remaining 9 elements will be repeatedly refined until reaching an acceptable error.

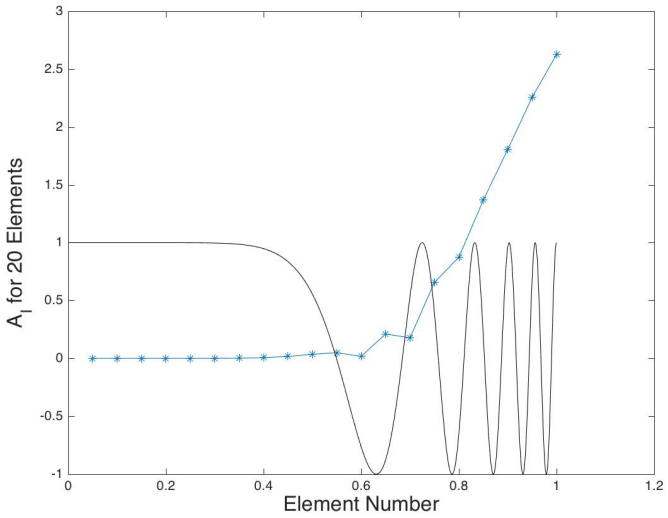

Figure 2. A_i for each element as calculated by Eq. (27).

With the AMR scheme discussed in the previous section, a total of 405 elements are needed to achieve $A_i \leq 0.05$ for each element i.

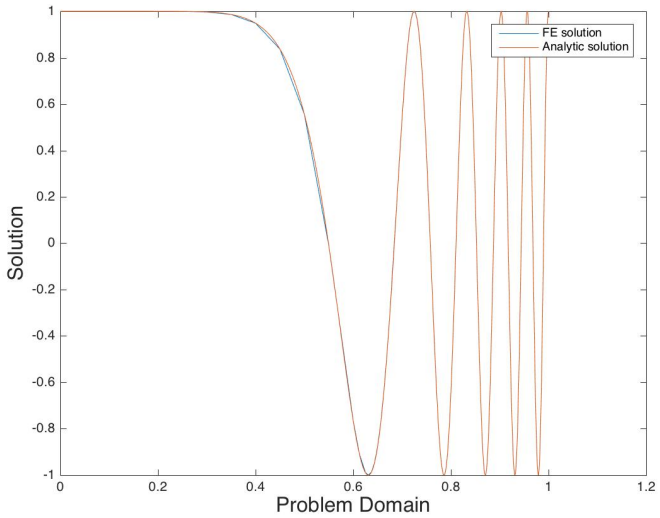

Figure 3. FE and analytic solutions for 405 elements, the minimum numbers of elements (with the particular refinement scheme discussed) such that each element satisfies $A_i \leq 0.05$.

Because the difference between the FE and analytic solutions are relatively difficult to perceive from the above figure, Fig. 4 shows the difference between the analytic and FE solutions. It is interesting to see that the greatest difference is observed in the earlier elements that likely had A_i only slightly lower than the tolerance in order to not be refined later.

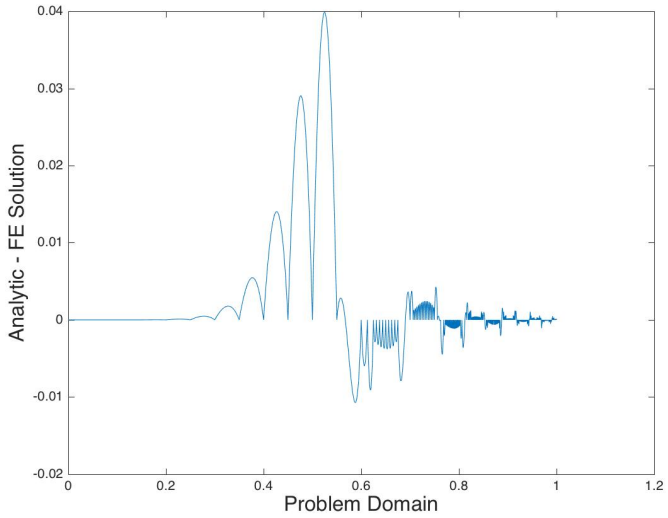

Figure 4. Analytic - FE solution for 405 elements.

Table 1 shows the final number of elements in each of the original element subdivisions. As expected, due to the sharp gradients that occur with increasing x, significantly more elements are needed towards the end of the domain to achieve the same A_i tolerance for each element. This table shows the great benefit of AMR - if the entire domain were refined "blindly" without recognizing that some regions of the domain don't need to be meshed, then each original element might have been meshed with 160 elements each, which would result in a total of 3200 elements assuming that a single element were responsible for "holding back" the remaining elements from achieving the desired error tolerance. Compared with the 405 elements actually needed, this represents about an order of magnitude savings in computational cost, and is hence why AMR is very important for obtaining the desired error levels.

Table 1. Final numbers of elements required for the original element domains.

Initial Element Domain	Final Number of Elements
$0.00 \leq x < 0.05$	1
$0.05 \leq x < 0.10$	1
$0.10 \leq x < 0.15$	1
$0.15 \leq x < 0.20$	1
$0.20 \leq x < 0.25$	1
$0.25 \leq x < 0.30$	1
$0.30 \leq x < 0.35$	1
$0.35 \leq x < 0.40$	1
$0.40 \leq x < 0.45$	1
$0.45 \leq x < 0.50$	1
$0.50 \leq x < 0.55$	1
$0.55 \leq x < 0.60$	1
$0.60 \leq x < 0.65$	6
$0.65 \leq x < 0.70$	5
$0.70 \leq x < 0.75$	15
$0.75 \leq x < 0.80$	22
$0.80 \leq x < 0.85$	39
$0.85 \leq x < 0.90$	53
$0.90 \leq x < 0.95$	93
$0.95 \leq x < 1.00$	160

Finally, Fig. 5 shows A_i for each element plotted as a function of the coordinate at the right endpoint of each element. As can be seen, $A_i < 0.05$ as required by the convergence criteria, and due to the highly-varying nature of the solution towards the right endpoint of the domain, A_i are closer the reaching the tolerance in that region than earlier in the domain where the true solution is slowly-varying.

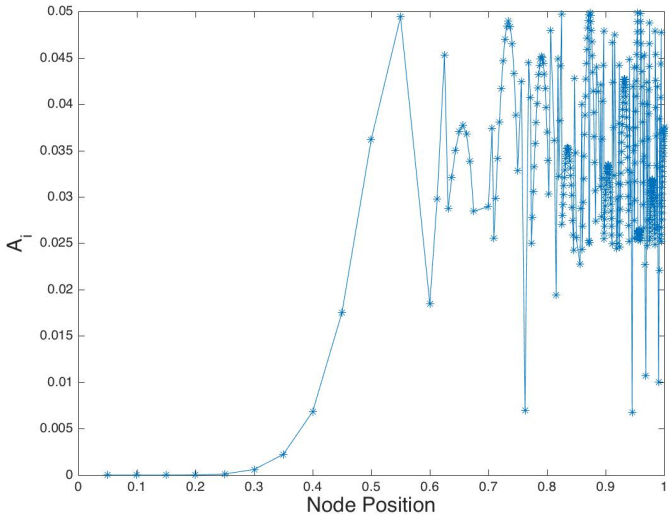

Figure 5. A_i plotted for each element as a function of the coordinate of the right endpoint of the element.

4 Appendix

This section contains the complete code used in this assignment.

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
N_plot_flag = 0;                    % 1 - plot the solutions for various N

L = 1.0;                            % problem domain
shape_order = 2;                    % number of nodes per element
E = 1.0;                            % elastic modulus
left = 'Dirichlet';                 % left BC
left_value = 1.0;                   % left Dirichlet BC value
right = 'Dirichlet';                % right BC type
right_value = cos(10 * pi * L^5);   % right Dirichlet BC value
tolerance = 0.05;                   % convergence tolerance
refine_tol = 0.05;                  % refinement tolerance
energy_norm = tolerance + 1;        % arbitrary initialization value
fontsize = 16;                      % fontsize for plots
num_refinements = 0;                % number of refinements to prevent loop
max_refinements = 10;               % maximum number of refinements + 1
num_elem_initial = 20;              % initial guess for number of elements
```

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

% index for collecting error
finished_refining = 0;
e = 1;

for num_elem = num_elem_initial
% uncomment to find how many elements are required to reach the error
% tolerance
 %while energy_norm > tolerance
%     num_elem = num_elem + 1;

    parent_domain = -1:0.01:1;
    physical_domain = zeros(1, num_elem * length(parent_domain) - (num_elem -
        ↪ 1));

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

    % --- ADAPTIVE MESH REFINEMENT --- %
    while ((finished_refining ~= 1) && (num_refinements <= max_refinements))

            % create a new physical domain
            j = 1;
            for elem = 1:num_elem
                discretization = linspace(coordinates(LM(elem, 1)),
                    ↪ coordinates(LM(elem, num_nodes_per_element)), length(
                    ↪ parent_domain));
                if elem == 1
                    physical_domain(j:length(parent_domain)) = discretization;
                    j = j + length(parent_domain);
                else
                    physical_domain(j:(j + length(parent_domain) - 2)) =
                        ↪ discretization(2:end);
                    j = j + length(parent_domain) - 1;
                end

            end

            % --- ANALYTICAL SOLUTION --- %
            solution_analytical = cos(10 .* pi .* physical_domain .^ 5);
            solution_analytical_derivative = - 10 .* pi .* 5 .*
                ↪ physical_domain .^ 4 .* sin(10 .* pi .* physical_domain .^
                ↪ 5);

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

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

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

for elem = 1:num_elem
    k = 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) * - E * (200 * pi * x_xe ^3 * sin
                 ↪ (10 * pi * x_xe ^ 5) + 2500 * pi^2 * x_xe^8 *
                 ↪ cos(10 * pi * x_xe^5)) * 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 * dN(i) * dN(j) /
                     ↪ dx_dxe;
             end
         end
     end

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

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

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

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

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

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

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

% assemble the solution in the physical domain
[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 .*
    ↪ solution_analytical_derivative));
energy_norm_top = sqrt(trapz(physical_domain, (
    ↪ solution_derivative_FE - solution_analytical_derivative) .*
    ↪ E .* (solution_derivative_FE -
    ↪ solution_analytical_derivative)));
energy_norm = energy_norm_top ./ energy_norm_bottom;

% determine the indices for the physical_domain to line up with
    ↪ coordinates
k = 1;
j = 1;
bounds = zeros(1, length(coordinates));
for i = 1:length(physical_domain)
    if (abs(coordinates(j) - physical_domain(i)) < 1e-6)
        bounds(k) = i;
        j = j + 1;
        k = k + 1;
    end
end

if (N_plot_flag)
    plot(physical_domain, solution_FE)
    hold on
end

% uncomment to find out how many elements are needed to reach the
    ↪ error
% tolerance
% end
e_N(e) = energy_norm;
e = e + 1;

if (N_plot_flag)
    plot(physical_domain, solution_analytical, 'k')
    txt = cell(length(N_elem),1);
```

```
    for i = 1:length(N_elem)
        txt{i}= sprintf('N = %i', N_elem(i));
    end
    txt{i+1} = 'analytical';
    h = legend(txt);
    xlabel('Problem domain', 'FontSize', fontsize)
    ylabel('Solution', 'FontSize', fontsize)
    saveas(gcf, 'Nplot', 'jpeg')
    close all
end

% compute the energy norm over each element
eN_per_elem = zeros(1, length(num_elem));
elem_length = zeros(1, length(num_elem));
for i = 1:num_elem
    elem_length(i) = coordinates(i+1, 1) - coordinates(i,1);
    spatial_domain = physical_domain(bounds(i):bounds(i+1));
    dFE = solution_derivative_FE(bounds(i):bounds(i+1));
    dAN = solution_analytical_derivative(bounds(i):bounds(i+1));
    eN_per_elem(i) = trapz(spatial_domain, (dFE - dAN) .* E .* (
        ↪ dFE - dAN));
end

% check that eN_per_elem is computed correctly
sprintf('Difference between sum and exact = %.6f', sum(eN_per_elem
    ↪ ) - energy_norm_top^2)

% plot A_I as a function of the element number
A_I = sqrt((1 ./ elem_length) .* eN_per_elem ./ ((1 ./ L) .*
    ↪ energy_norm_bottom .^ 2));
coords = coordinates(:,1);

if num_refinements == 0
    plot(coords(2:1:end), A_I, '*-', physical_domain,
        ↪ solution_analytical, 'k')
    xlabel('Element Number', 'FontSize', fontsize)
    ylabel(sprintf('A_I for %i Elements', num_elem), 'FontSize',
        ↪ fontsize)
    saveas(gcf, 'A_I_NoRefinement', 'jpeg')
    close all
end

% determine which elements need to be refined
clearvars refine
j = 1;
for i = 1:length(A_I)
    if (A_I(i) < refine_tol)
    else
        refine(j) = i;
        j = j + 1;
    end
end

if (j == 1)
```

```
        finished_refining = 1;
        sprintf('Finished refining, with %i total elements', num_elem
            ↪ )
        plot(physical_domain, solution_FE, physical_domain,
            ↪ solution_analytical)
        xlabel('Problem Domain', 'FontSize', fontsize)
        ylabel('Solution', 'FontSize', fontsize)
        legend('FE solution', 'Analytic solution')
        saveas(gcf, 'FinalSolution', 'jpeg')
        close all

        plot(physical_domain, solution_analytical - solution_FE)
        xlabel('Problem Domain', 'FontSize', fontsize)
        ylabel('Analytic - FE Solution', 'FontSize', fontsize)
        saveas(gcf, 'FinalSolutionDiff', 'jpeg')
        close all

        plot(coords(2:end), A_I, '*-')
        xlabel('Node Position', 'FontSize', fontsize)
        ylabel('A_i', 'FontSize', fontsize)
        saveas(gcf, 'Nodes_vs_Ai', 'jpeg')
        close all


        % determine the number of elements per initial range
        m = 1;
        num_elem_per_initial_elem2 = zeros(1, length(num_elem_initial)
            ↪ );
        initial_bounds = linspace(0, L, num_elem_initial + 1);
        for y = 1:length(coords)
            if (abs(coords(y) - initial_bounds(m)) < 1e-10)
                num_elem_per_initial_elem2(m) = y;
                m = m + 1;
            end
        end

        num_elem_per_initial_elem = zeros(1, length(num_elem_initial))
            ↪ ;
        for y = 1:(length(num_elem_per_initial_elem2) - 1)
            num_elem_per_initial_elem(y) = num_elem_per_initial_elem2(
                ↪ y+1) - num_elem_per_initial_elem2(y);
        end

        num_elem_per_initial_elem

    else
        num_refinements = num_refinements + 1;

        % update coordinates vector
        num_elem_new = num_elem + length(refine);
        num_nodes_new = (shape_order - 1) * num_elem_new + 1;
        coordinates_new = zeros(num_nodes_new, 3);

        j = 1;
```

```
                k = 1;
                l = 1;
                for i = 1:num_elem
                    if ((j <= length(refine)) && (refine(j) == i))
                        increment = 0.5 * (coordinates(k+1,1) - coordinates(k
                            ↪ ,1));
                        coordinates_new(l,1) = coordinates(k,1);
                        coordinates_new(l+1,1) = coordinates(k,1) + increment;
                        coordinates_new(l+2,1) = coordinates(k+1,1);
                        k = k + 1;
                        l = l + 2;
                        j = j + 1;
                    else
                        coordinates_new(l,1) = coordinates(k,1);
                        coordinates_new(l+1,1) = coordinates(k+1,1);
                        l = l + 1;
                        k = k + 1;
                    end
                end

                coordinates = coordinates_new;
                num_elem = num_elem_new;
                num_nodes = num_nodes_new;

                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
        end
    end

% uncomment to find out how many elements are needed to reach the error
% tolerance
%sprintf('Number elements needed: %i', num_elem)
```

4.2 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
```

4.3 `condensation.m`

This function separates out the matrix equation as in Eq. (20).

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

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

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


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

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


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


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


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

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

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

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

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

```
    end
end

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

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

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

4.4 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
```

4.5 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
```

4.6 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. (22).

```
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
```

4.7 quadrature.m

This function selects the quadrature rule.

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

shape_order = 4;

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
```

4.8 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;
    otherwise
        disp('You entered an unsupported shape function order.');
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
```