ME 280a: HW 2

April Novak

October 11, 2016

1 Introduction and Objectives

The purpose of this study is to solve a simple finite element (FE) problem and perform a convergence study to determine the number of elements needed to reach a particular error relative to an analytical solution for different shape function orders. 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:

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

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, and it has been assumed that E is a constant. 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:

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

It has been assumed that E is not a function of x, and hence can be treated as constant in the integration. Integrating once more:

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

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

$$\begin{aligned} u(0) &= 3 \\ u(L) &= -1 \end{aligned} \tag{4}$$

By applying these boundary conditions:

$$C_1 = \frac{1}{L}\left[(-1)E + \frac{Lk^3}{\gamma^2}\cos(\gamma L) - \frac{2k^3}{\gamma^3}\sin(\gamma L) - C_2\right] \qquad C_2 = 3 \tag{5}$$

The purpose of this assignment is to solve Eq. (1) with the finite element method (FEM) and then to determine how many elements are needed to obtain a specified error as a function of the order of the shape functions. The solutions for various numbers of elements will also be compared to illustrate how increasing the number of elements "hones in" on the analytical solution. In addition, the convergence rates of the different shape element orders will be plotted to determine the relationship between element order and the rate at which the error decreases as a function of the mesh spacing.

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:

$$u \approx u^N = \sum_{j=1}^{N} a_j \phi_j \qquad \psi = \sum_{i=1}^{N} b_i \phi_i \tag{6}$$

Galerkin's method is stated as:

$$r^N \cdot u^N = 0 \tag{7}$$

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, $d\Omega$.

$$\int_\Omega \frac{d}{dx}\left(E(x)\frac{du}{dx}\right)\psi d\Omega - \int_\Omega xk^3\cos(\gamma x)\psi d\Omega = 0 \tag{8}$$

Applying integration by parts to the first term:

$$-\int_\Omega E(x)\frac{du}{dx}\frac{d\psi}{dx}d\Omega + \int_{\partial\Omega} E(x)\frac{du}{dx}\psi d(\partial\Omega) - \int_\Omega xk^3\cos(\gamma x)\psi d\Omega = 0 \tag{9}$$

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:

$$-\int_0^L E(x)\frac{du}{dx}\frac{d\psi}{dx}dx + E(x)\frac{du}{dx}\psi\Big|_0^L - \int_0^L xk^3\cos(\gamma x)\psi dx = 0$$
$$\int_0^L E(x)\frac{du}{dx}\frac{d\psi}{dx}dx = -\int_0^L xk^3\cos(\gamma x)\psi dx + E(x)\frac{du}{dx}\psi\Big|_0^L \tag{10}$$

Inserting the approximation described in Eq. (6):

$$\int_0^L E(x)\frac{d\left(\sum_{j=1}^N a_j\phi_j\right)}{dx}\frac{d\left(\sum_{i=1}^N b_i\phi_i\right)}{dx}dx = -\int_0^L xk^3\cos(\gamma x)\sum_{i=1}^N b_i\phi_i dx + E(x)\frac{du}{dx}\sum_{i=1}^N b_i\phi_i\Big|_0^L \tag{11}$$

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).

$$\int_0^L E(x)\frac{d\left(\sum_{j=1}^N a_j\phi_j\right)}{dx}\frac{d\phi_i}{dx}dx = -\int_0^L xk^3\cos(\gamma x)\phi_i dx + E(x)\frac{du}{dx}\phi_i\Big|_0^L \tag{12}$$

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

$$\int_0^L E(x)\frac{d\left(a_j\phi_j\right)}{dx}\frac{d\phi_i}{dx}dx = -\int_0^L xk^3\cos(\gamma x)\phi_i dx + E(x)\frac{du}{dx}\phi_i\Big|_0^L \tag{13}$$

This produces a system of matrix equations of the form:

$$\mathbf{K}\vec{a} = \vec{F} \tag{14}$$

where:

$$\begin{aligned} K_{ij} &= \int_0^L E(x)\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx \\ a_j &= a_j \\ F_i &= -\int_0^L xk^3\cos(\gamma x)\phi_i dx + E(x)\frac{du}{dx}\phi_i\Big|_0^L \end{aligned} \tag{15}$$

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

$$\begin{aligned} K_{ij} = \int_0^L E(x)\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx \rightarrow \int_{-1}^1 E(x(\xi))\frac{d\phi_i}{dx}\frac{d\phi_j}{dx}dx\left(\frac{dx}{d\xi}\frac{dx}{d\xi}\frac{d\xi}{dx}\right) \rightarrow \int_{-1}^1 E(x(\xi))\frac{d\phi_i}{d\xi}\frac{d\phi_j}{d\xi}dx\left(\frac{d\xi}{dx}\right) \\ a_j = a_j \\ F_i = -\int_0^L xk^3\cos(\gamma x)\phi_i dx \rightarrow -\int_{-1}^1 x(\xi)k^3\cos(\gamma x(\xi))\phi_i\frac{dx}{d\xi}d\xi \end{aligned} \tag{16}$$

where the second term in F_i has been dropped because there are no Neumann boundary conditions in this assignment. Three different shape function orders are used in this assignment - linear, quadratic, and cubic. All of these shape functions are Lagrangian shape functions that are determined by requiring them to go to zero at all nodes except the node to which they pertain. For linear elements, the shape functions have the following form and derivative over the master element:

$$\begin{aligned} \phi_1(\xi) &= \frac{1-\xi}{2}, \quad \frac{d\phi_1(\xi)}{d\xi} = -1/2 \\ \phi_2(\xi) &= \frac{1+\xi}{2}, \quad \frac{d\phi_2(\xi)}{d\xi} = +1/2 \end{aligned} \tag{17}$$

For quadratic elements, the shape functions have the following form and derivative over the master element:

$$\begin{aligned} \phi_1(\xi) &= \frac{(\xi-1)\xi}{2}, \quad \frac{d\phi_1(\xi)}{d\xi} = \xi - 1/2 \\ \phi_2(\xi) &= (1-\xi)(1+\xi), \quad \frac{d\phi_2(\xi)}{d\xi} = -2\xi \\ \phi_3(\xi) &= \frac{(\xi+1)\xi}{2}, \quad \frac{d\phi_3(\xi)}{d\xi} = \xi + 1/2 \end{aligned} \tag{18}$$

For cubic elements, the shape functions have the following form and derivative over the master element:

$$\begin{aligned}
\phi_1(\xi) &= \frac{9}{16}\left[(1-\xi)(1/3-\xi)(-1/3-\xi)\right], \quad \frac{d\phi_1(\xi)}{d\xi} = \frac{9}{16}\left(-3\xi^2+2\xi+\frac{1}{9}\right) \\
\phi_2(\xi) &= \frac{-27}{16}\left[(1-\xi)(1/3-\xi)(-1-\xi)\right], \quad \frac{d\phi_2(\xi)}{d\xi} = \frac{-27}{16}\left(-3\xi^2+\frac{2\xi}{3}+1\right) \\
\phi_3(\xi) &= \frac{27}{26}\left[(1-\xi)(-1/3-\xi)(-1-\xi)\right], \quad \frac{d\phi_3(\xi)}{d\xi} = \frac{27}{16}\left(-3\xi^2-\frac{2\xi}{3}+1\right) \\
\phi_4(\xi) &= \frac{-9}{16}\left[(1/3-\xi)(-1/3-\xi)(-1-\xi)\right], \quad \frac{d\phi_4(\xi)}{d\xi} = \frac{-9}{16}\left(-3\xi^2-2\xi+\frac{1}{9}\right)
\end{aligned} \tag{19}$$

where the nodes are spaced equally over $-1 \leq \xi \leq 1$ (i.e. nodes at $\pm 1/3$). The transformation from the physical domain (x) to the parent domain (ξ) is done with an isoparametric mapping:

$$x(\xi) = \sum_{i=1}^{N} X_i \phi_i(\xi) \tag{20}$$

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. (20) by differentiation:

$$\frac{dx(\xi)}{d\xi} = \sum_{i=1}^{N} X_i \frac{d\phi_i(\xi)}{d\xi} \tag{21}$$

With all these transformations from the physical domain to the isoparametric domain, Gaussian quadrature can be used. The quadrature used depends on the order of the shape functions. An n point quadrature rule can exactly integrate a polynomial of order $2n-1$. From Eq. (16), the integrand in K_{ij} will be of order $(n-1)(n-1)$, while the integrand in F_i will be of (technically) infinite order since cosine is not a polynomial, and it would require a summation of an infinite number of polynomials to approximate cosine as a polynomial (i.e. Taylor series). Neglecting the presence of the cosine function, then the integrand in F_i would be of order n^2. So, the selection of the order of the quadrature rule is not straightforward. Table 2 shows the order of the different matrices and vectors in the governing equations in order to guide the selection of a quadrature rule. The number of quadrature points is selected according to the desire to integrate correctly F_i (neglecting the cosine term), since this is more restrictive than correctly integrating the integrand in K_{ij}. When applicable, the number of quadrature points is rounded up.

Table 1. Number of quadrature points needed for exact integration of terms appearing in Eq. (16). The "order of integrand in F_i" neglects the order of the cosine term.

ϕ order	order of integrand in K_{ij}	order of integrand in F_i	Quadrature points needed
1	0	2	2
2	1	4	3
3	4	9	5

Hence, for cubic shape functions, a five-point rule is used, for quadratic a three-point rule, and for linear a two-point rule. These Gauss-Legendre rules are as follows. For the two-point quadrature rule, the weights w and sampling points x are:

$$\begin{aligned}
w &= [1.0, 1.0] \\
x &= [-\sqrt{1/3}, \sqrt{1/3}]
\end{aligned} \tag{22}$$

For the three-point quadrature rule, the weights w and sampling points x are:

$$\begin{aligned}
w &= [5/9, 8/9, 5/9] \\
x &= [-\sqrt{3/5}, 0, \sqrt{3/5}]
\end{aligned} \tag{23}$$

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.

In order to apply the boundary conditions within the numerical context of the finite element method, the matrix equation in Eq. (14) 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. (14) 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 by simple Gaussian elimination, 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). Convergence is defined to have been achieved for a given k and number of elements N once:

$$e^N = \frac{\|u-u^N\|}{\|u\|} \leq 0.04 \tag{32}$$

2.4 Solution Results and Discussion

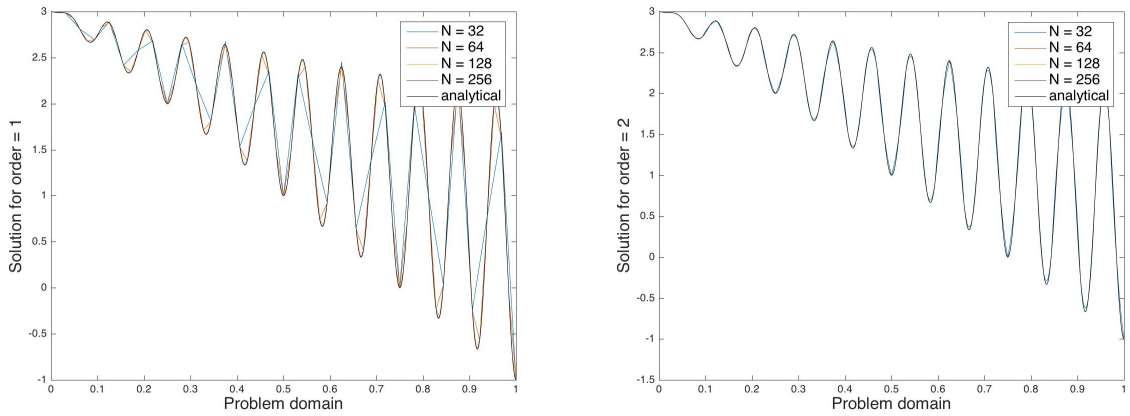
Table 2 shows the number of elements needed to reach the tolerance given by Eq. (32) as a function of the polynomial order of the approximation functions. The higher the element order, the less elements that are needed to reach the specified convergence criteria, as expected. The higher the polynomial order of the shape functions, the more degrees of freedom that are available to capture the behavior of the true solution.

It is important to take the results shown in Table 2 with a "grain of salt" because they depend on how exactly I compute the energy norm - I use `trapz()`, which is sensitive to the mesh I use. I use a very fine mesh, which is perhaps why the numbers of intervals over which I compute the integrals in the energy norm provides a lower energy norm than for students who use quadrature or the $a^T\mathbf{K}a$ method for computing the energy norm. The answer also depends on the quadrature rule used, but I believe the method I used is more accurate because the mesh is made incredibly fine to perform the integration.

Table 2. Number of elements needed to reach a tolerance given by Eq. (32) based on the polynomial order of the approximating functions.

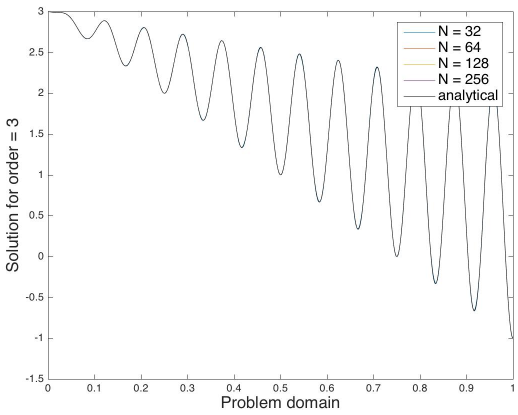
ϕ order	Number of elements needed
1	543
2	73
3	24

Fig. 1 shows the solutions for linear, quadratic, and cubic approximation functions as a function of various choices for number of elements. The results in this figure confirm the results in the table above - the higher the polynomial order of the approximation functions, the more accurate the answer (relative to the exact solution), and hence the fewer numbers of elements that are needed.



(a)

(b)



(c)

Figure 1. Finite element and exact (analytical) solutions for $N = 32, 64, 128, 256$ for (a) linear, (b) quadratic, and (c) cubic approximation functions.

Fig. 2 shows the error as a function of the element size $h \equiv L/(\text{number of elements})$ for linear (order $= 1$), quadratic (order $= 2$), and cubic (order $= 3$) shape functions. As can be seen, the higher the approximation function order, both the lower the error, and the faster the error decreases.

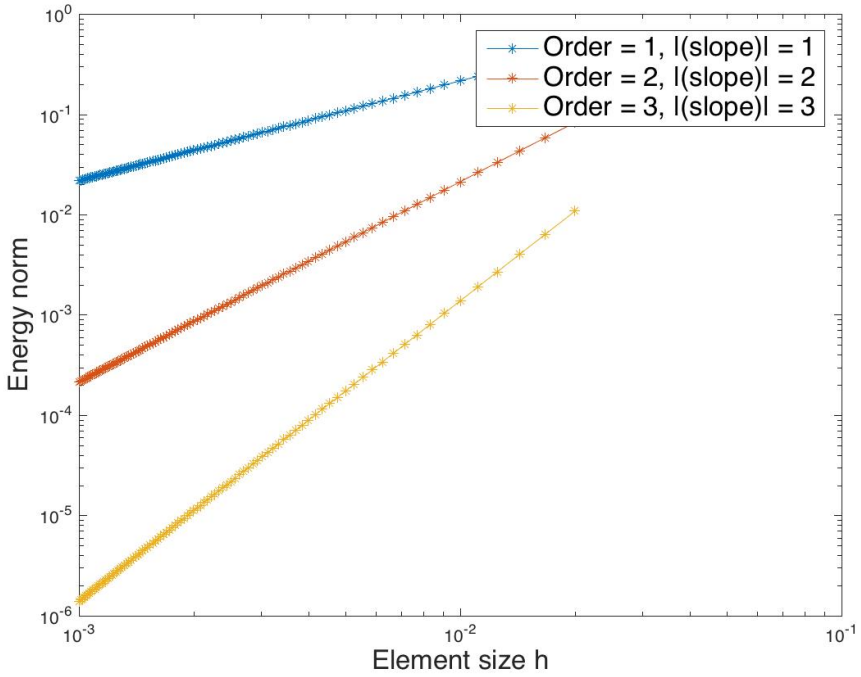


Figure 2. Energy norm e^N as a function of h for different polynomial orders.

Fig. 3 essentially shows the same information as Fig. 2, except that the independent variable is the number of degrees of freedom. For a linear element, there are two degrees of freedom, for a quadratic element there are three, and for a cubic element there are four. As can seen, the smaller the h, or the greater the number of degrees of freedom, the lower the error and the faster the error decrease.

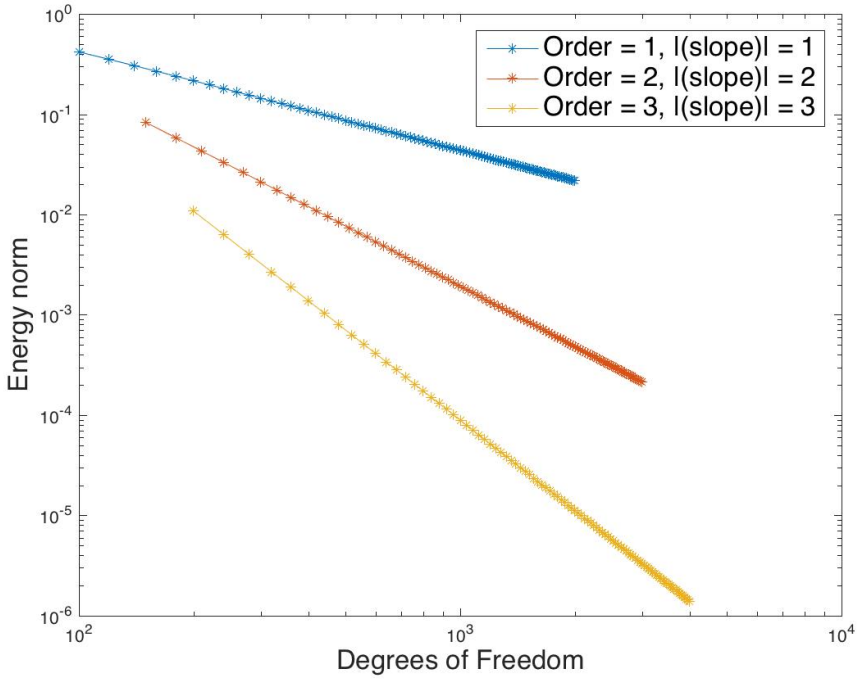


Figure 3. Energy norm e^N as a function of the number of degrees of freedom for different polynomial orders.

The theoretical rates of convergence are dictated by the fundamental interpolation theorem:

$$\|u - u^h\|_{E(\Omega)} \leq C(u,p)h^{\min(r-1,p)} \tag{33}$$

where C is an unknown constant, r is the regularity, and p is the polynomial order of the approximation functions. Hence, when plotted on log-log coordinates, the slopes of the energy norm as a function of h (Fig. 2) have values equal to the polynomial order of the approximation function. The slops of the energy norm as a function of the number of degrees of freedom (Fig. 3) simply has the negative of this slope. Hence, this homework assignment has shown the validity of Eq. (33), and has shown the relationship between the error and the element size for each p. For example, by doubling the number of elements, the error decreases by a factor of 2 for linear elements, 4 for quadratic elements, and 8 for cubic elements.

3 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.

3.1 `FEProgram.m`

```
clear all

% select which type of plot you want to make - at least one flag must equal 1
k_plot_flag = 0;                % 1 - plot the error as a function of order
k_plot_flag_dof = 0;            % 1 - plot the error as a function of DOF
N_plot_flag = 1;                % 1 - plot the solutions for various N

L = 1.0;                        % problem domain
```

```
k_freq = 12;                        % forcing frequency
num_elem = 5;                       % number of finite elements (initial guess)
shape_order = 2;                    % number of nodes per element
E = 0.2;                            % elastic modulus
left = 'Dirichlet';                 % left boundary condition
left_value = 3.0;                   % left Dirichlet boundary condition value
right = 'Dirichlet';                % right boundary condition type
right_value = -1.0;                 % right Dirichlet boundary condition value
tolerance = 0.04;                   % convergence tolerance
energy_norm = tolerance + 1;        % arbitrary initialization value
fontsize = 16;                      % fontsize for plots

if (N_plot_flag)
     N_elem = [32, 64, 128, 256];              % num_elem to cycle through for
        ↪  soln plots
elseif (k_plot_flag || k_plot_flag_dof)
     N_elem = 50:10:1000;          % num_elem to cycle through for e_N vs. N
else
    disp('Either N_plot_flag or k_plot_flag has to equal 1.');
end

Order = [2, 3, 4];                  % shape function (orders - 1) to cycle thru

for shape_order = Order
    clearvars permutation

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

% index for collecting error
e = 1;

% initial guess for determining number of elements to reach error tol
%N_elem = 100;

for num_elem = N_elem

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

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

    % --- ANALYTICAL SOLUTION --- % (over the physical domain)
    parent_domain = -1:0.1:1;
    physical_domain = linspace(0, L, num_elem * length(parent_domain) - (
       ↪ num_elem - 1));
    gamma = 2 * pi * k_freq ./ L;
    term1 = 2 * (k_freq .^ 3) * sin(gamma .* physical_domain) ./ (E .* gamma
       ↪ .^3);
    term2 = (k_freq .^ 3) * physical_domain .* cos(gamma .* physical_domain)
       ↪ ./ (E .* gamma .^ 2);
```

```
C_1 = left_value;
C_2 = (right_value - C_1 - (2 * (k_freq .^ 3) * sin(gamma .* L) ./ (E .*
    ↪ gamma .^3)) + ((k_freq .^ 3) * L .* cos(gamma .* L) ./ (E .* gamma
    ↪ .^ 2))) ./ L;
solution_analytical = C_1 + term1 - term2 + C_2 .* physical_domain;
term1_1 = 2 * (k_freq .^ 3) * cos(gamma .* physical_domain) .* gamma ./ (E
    ↪  .* gamma .^3);
term2_1 = ((k_freq .^ 3) ./ (E .* 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 .* gamma .^ 2) + C_2;

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

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

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) * 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 * dN(i) * dN(j) / dx_dxe;
             end
         end
     end

     % place the elemental k matrix into the global K matrix
     m = 1;
     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;
sprintf('energy_norm: %f', energy_norm)

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

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

end
```

```
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);
    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_for_order_%i', shape_order - 1), 'jpeg')
    close all
end

if (k_plot_flag || k_plot_flag_dof)
    if (k_plot_flag)
        independent_var = L ./ N_elem;
        independent_var_str = 'Element size h';
        filename = 'eN_vs_h';
    else
        independent_var = shape_order * N_elem;
        independent_var_str = 'Degrees of Freedom';
        filename = 'eN_vs_dof';
    end

    loglog(independent_var, e_N, '*-')
    hold on
    xlabel(independent_var_str, 'FontSize', fontsize)
    ylabel('Energy norm', 'FontSize', fontsize)
end

end

if (k_plot_flag || k_plot_flag_dof)
    txt = cell(length(Order),1);
    for i = 1:(length(Order))
        txt{i} = sprintf('Order = %i, |(slope)| = %i', Order(i) - 1, Order(i)
           ↪ - 1);
    end
    h2 = legend(txt);
    set(h2, 'FontSize', fontsize);
    saveas(gcf, filename, 'jpeg')
end

% uncomment to find out how many elements are needed to reach the error tol
%sprintf('For order = %i, number elements: %i', shape_order - 1, num_elem)
```

3.2 `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
```

3.3 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
```

3.4 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
```

3.5 `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
```

3.6 quadrature.m

This function selects the quadrature rule.

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

switch shape_order
    case 2
```

```
        wt = [1.0, 1.0];
        qp = [-sqrt(1/3), sqrt(1/3)];
    case 3
        wt = [5/9, 8/9, 5/9];
        qp = [-sqrt(3/5), 0, sqrt(3/5)];
    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
```

3.7 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
```

3.8 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
```