SF1693 - Laboration 2
Heat Conduction and Deformation of Membranes

February 18, 2024

1 Heat Conduction in a Rod

Introduction In this study, we investigate the heat conduction in a rod with a given length and conductivity, under the influence of a heat source $f(x)$. Our goal is to model the steady-state temperature distribution $u(x)$ along the rod, constrained by two boundary conditions: a fixed temperature of 0 at one end and a defined heat flux g at the other. This scenario is given as the following boundary value problem (BVP):

$$\begin{aligned} -(a(x)u'(x))' &= f(x), \quad x \in (0,1) \\ u(0) &= 0 \\ a(1)u'(1) &= g \end{aligned} \tag{1}$$

where $a(x)$ represents varying conductivity.

This BVP is suitable for several reasons: Firstly, the function $a(x)$ allows for variations in conductivity, which is essential for accurately representing materials with variable conductivity properties. The steady-state assumption allows the model to be simplified such that one can focus on equilibrium conditions. The boundary conditions mirror common physical setups, with one end of the rod in equilibrium with an environment or object at a fixed temperature, and the other end under the influence of a known heat source, simulating realistic scenarios.

Moreover, the assumption that circumferential heat flow is negligible further simplifies the analysis without significantly impacting accuracy for rods with small diameters or minimal radial temperature gradients. The governing differential equation, derived from Fourier's law, ensures the model's foundation on well-established principles of heat conduction, relating heat transfer to conductivity and temperature gradient.

1.1 Problem 1

The objective extends to numerically solving this boundary value problem using the Finite Element Method (FEM) in MATLAB. Employing piecewise linear elements and Gaussian quadrature for integral computations, the approach is designed to be flexible, accommodating arbitrary conductivity functions $a(x)$, heat sources $f(x)$, and heat flux g.

1.1.1 Modeling the Problem

The foundation of our model is the heat conduction equation, which in its general form for a three-dimensional space is expressed as[1]:

$$\frac{\partial u}{\partial t} = \nabla \cdot (\kappa \nabla u) + f(x,t)$$

Where:

- u represents the temperature
- κ denotes the thermal conductivity, and
- $f(x,t)$ is the heat generation per unit volume.

For our specific scenario, considering the time-independent (steady-state) condition, the term $\frac{\partial u}{\partial t}$ becomes zero. Moreover, due to the one-dimensional nature of the problem, the divergence operator $\nabla \cdot (\kappa \nabla u)$ simplifies.

The first boundary condition is set at one end of the rod, where the temperature is fixed at zero.

$$u(0) = 0$$

The second boundary condition concerns the opposite end of the rod.

$$a(1)u'(1) = g$$

Thus we know that:

$$\nabla \cdot (\kappa \nabla u) + f(x,t) = \frac{\partial u}{\partial t}$$
$$\nabla \cdot (a(x) \nabla u) + f(x) = 0$$

[1]See Page 16: *Partial Differential Equations: An Introduction* by Walter A. Strauss, 2nd edition, John Wiley & Sons, 2007, ISBN 0470054565.

$$(a(x)u'(x))' = -f(x)$$

Starting with the integral form of the differential equation:

$$\int_0^1 (a(x)u')'\,\phi(x)\,dx = -\int_0^1 f(x)\phi(x)\,dx$$

Represents a balance between the heat conducted in the rod and the heat generated within it, where $\phi(x)$ is our test function.

Applying integration by parts:

$$-a(x)u'\phi(x)\Big|_0^1 + \int_0^1 a(x)u'\phi'(x)\,dx = \int_0^1 f(x)\phi(x)\,dx$$

Incorporating the Neumann boundary condition $a(1)u'(1) = g$:

$$-g\phi(1) + \int_0^1 a(x)u'\phi'(x)\,dx = \int_0^1 f(x)\phi(x)\,dx$$

Arriving at the final form of the variational formulation:

$$\underbrace{\int_0^1 a(x)u'\phi'(x)\,dx}_{\text{Stiffness matrix: } A(u,\phi)} = \underbrace{g\phi(1) + \int_0^1 f(x)\phi(x)\,dx}_{\text{Load vector in the FEM context: } L(\phi)}$$

This equation will form the basis for the numerical solution using FEM.

Physics tells us that in heat conduction, the rate of heat transfer through a material is directly proportional to the cross-sectional area through which heat is being transferred. Thus a should increase proportionally with the cross-sectional area, however note that in model is one-dimensional so the cross-sectional = 0. For the sake of this specific model we have to conclude that the area is irrelevant. So we have to assume that a represents thermal conductivity, which comes from Fourier's law of heat conduction.

Fourier's law in its one-dimensional form is often stated as $q = -k\frac{dT}{dx}$,

1.1.2 Implementing the FEM for the Rod's Temperature Distribution

- **Selection of Basis Functions**
  For our problem, we use piece-wise linear functions.
  $\phi_i(j) = \delta_{ij}$, where δ_{ij} is the Kronecker delta function.

- **Discretization into finite elements**.

- **Assembly of the System Matrix and Load Vector**
  We translate the variational formulation into a system of linear equations. First we construct the matrix A and the vector L.

- **Incorporating Boundary Conditions**
  For $u(0) = 0$ we set the first row of A to 0.
  Note: The first element of the diagonal in A should still be 1*!*
  First element of L to 0. This will give us that the $u(0) = 0$.

  For $a(1)u'(1) = g$ we dont have to do anything since we already have it in the $L(\phi) = \int_0^1 f(x)\phi(x)dx + g\phi(1)$ as we have the term $g\phi(1)$.
  Note: In L we have:

$$g\phi_j(1) = 0 \text{ for all } j \neq n$$

$$g\phi_j(1) = g \text{ if } j = n$$

- **Solving the Linear System**

$$Au = L$$

  Where the solution vector u gives the approximate temperatures at the discretized points along the rod. Note that the FEM solution, denoted as u, is constructed as a linear combination of the basis functions weighted by the elements of $\vec{u}$.

1.2 Problem 2 - Finding the Exact solution

Exact solution In this section, we derive the exact solution to the boundary value problem characterized by Eq. (1).

$$-(a(x)u'(x))' = f(x),$$

we start by integrating both sides with respect to x from y to 1, where y is a variable that will be integrated over later

$$\int_y^1 -(a(x)u'(x))'\,dx = \int_y^1 f(x)\,dx.$$

The integration of the left-hand side yields

$$-(a(x)u'(x))\Big|_y^1$$

which represents the change in heat flux from y to 1.
Applying the boundary conditions, we obtain:

$$-\underbrace{a(1)u'(1)}_{g} + a(y)u'(y) = \int_y^1 f(x)\,dx.$$

According to the Neumann boundary condition, we know that $a(1)u'(1) = g$, so we can substitute this into the equation, which yields

$$a(y)u'(y) = g + \int_y^1 f(z)dz$$

Solving for $u'(y)$, we find

$$u'(y) = \frac{g + \int_y^1 f(x)\,dx}{a(y)}.$$

To find $u(x)$, we now integrate with respect to y from 0 to x, setting z as the integration variable

$$u(x) = \int_0^x \frac{g + \int_y^1 f(z)\,dz}{a(y)}\,dy. \tag{2}$$

1.3 Problem 3

Introduction In this section, we delve into further analysis of the boundary value problem characterized by Eq. (1). Specifically, we consider two distinct cases: firstly, where the coefficient $a(x)$ is linearly dependent on x as $1 + x$ with a zero source term $f(x) = 0$, and secondly, where both $a(x)$ and $f(x)$ are exponentially dependent on x as e^x. These cases are evaluated within the domain $x \in [0, 1]$, with the boundary condition at the right end set to $g = 1$.

1.3.1 Case 1

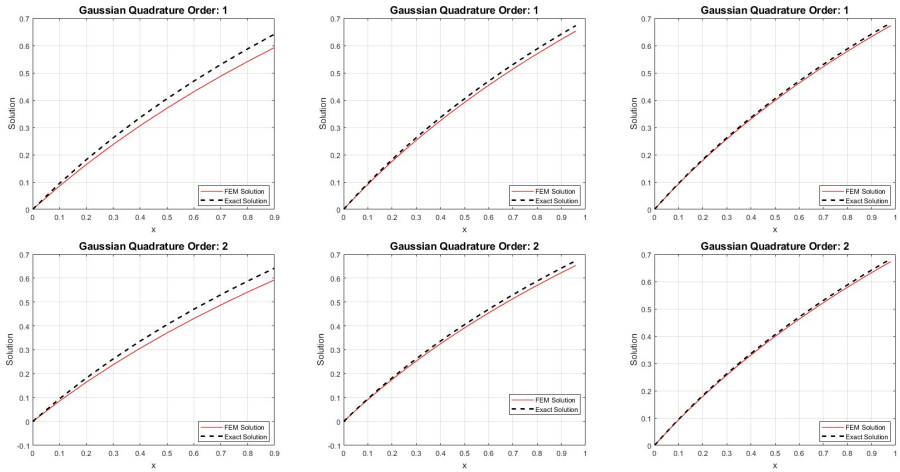
The functions are given as

$$a_1(x) = 1 + x, \quad f_1(x) = 0.$$

We begin by analytically deriving the exact solution by applying the given functions to Eq. (2) which yields

$$u(x) = \ln(1 + x).$$

The exact solution can be seen in comparison to the numerical solution in Fig. 1.



(a) $n = 10$ (b) $n = 25$ (c) $n = 50$

Figure 1: Result of FEM compared to exact solution for case 1

1.3.2 Case 2

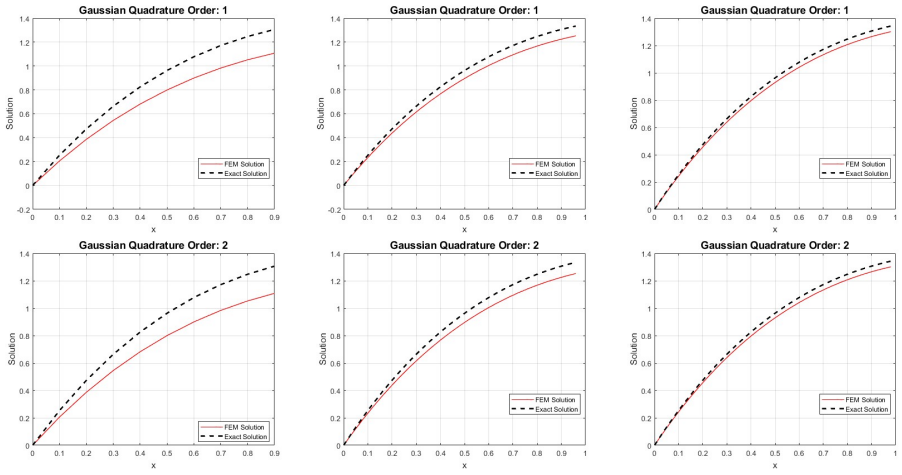
The functions are given as

$$a_2(x) = f_2(x) = e^x.$$

We begin by analytically deriving the exact solution by applying the given functions to Eq. (2) which yields

$$u(x) = (1 + e)(1 - e^{-x}) - x$$

The exact solution can be seen in comparison to the numerical solution in Fig. 2.



(a) $n = 10$ (b) $n = 25$ (c) $n = 50$

Figure 2: Result of FEM compared to exact solution for case 2

1.3.3 Comparison

Visual inspection of the solution curves in Fig. 1 and Fig. 2, where both one-point and two-point Gaussian quadrature was used, shows no noticeable difference in the appearance of the graphs for either set of boundary conditions.

We found no significant difference in efficiency between the one-point and two-point methods for this specific model during experimentation. Approximately the same number of elements are needed in both methods, and the additional function evaluations in the two-point method are relatively inexpensive. We found that the second set of exponential functions consistently required a finer resolution in the form of more elements for the same level of accuracy, as can be seen in Fig. 3 in comparison to Fig. 4, which depict the number of elements in relation to the size of a given L^2 norm of the error.

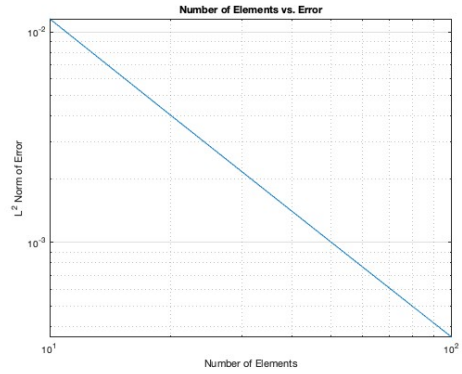


Figure 3: $a_1(x) = 1 + x, \quad f_1(x) = 0.$

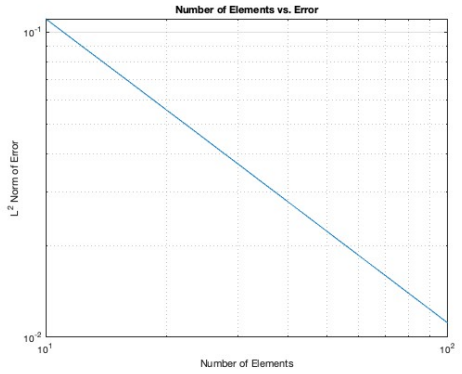


Figure 4: $a_2(x) = f_2(x) = e^x$

We can observe a negative linear relationship in both cases, where the exponential case has a steeper slope, which aligns with the expected behaviour due to the increase of complexity in the functions.

2 Deformation in a Membrane

2.1 Problem 4

Introduction In this section, we present the analytical solution for a model involving an almost conical tent modeled as a stretched membrane. The membrane's height above the ground, represented by the function $u(x_1, x_2)$ at the point $(x_1, x_2) \in \mathbb{R}^2$, adheres to specific boundary conditions within a defined domain given by

$$\begin{aligned} -\Delta u(x_1, x_2) &= 0, \quad \epsilon^2 < x_1^2 + x_2^2 < 1, \\ u(x_1, x_2) &= 1, \quad x_1^2 + x_2^2 = \epsilon^2, \\ u(x_1, x_2) &= 0, \quad x_1^2 + x_2^2 = 1, \end{aligned} \tag{3}$$

where $0 < \epsilon < 1$ and $\Omega_\epsilon := \{x \in \mathbb{R}^2 : \epsilon \leq |x| \leq 1\}$.

Our goal is to demonstrate that $u : \Omega_\epsilon \to \mathbb{R}$ not only satisfies the given differential equation but also minimizes a particular integral J within a set V_ϵ such that

$$\min_{v \in V_\epsilon} \int_{\epsilon<|x|<1} |\nabla v(x)|^2 \, dx = \int_{\epsilon<|x|<1} |\nabla u(x)|^2 \, dx$$

where:

$$V_\epsilon := \left\{ v : \Omega_\epsilon \to \mathbb{R} : \int_{\epsilon<|x|<1} |\nabla v(x)|^2 \, dx < \infty \right\},$$

with $v(x) = 0$ for $|x| = 1$ and $v(x) = 1$ for $|x| = \epsilon$.

2.1.1 Showing that u minimizes the integral:

Given a function $u(x_1, x_2)$ that represents the height of a tent membrane over the ground within a domain D in $\mathbb{R}^2$, this function must satisfy certain boundary conditions and a differential equation indicative of its harmonic nature. The boundary conditions are given by:

$$\begin{aligned} u(x_1, x_2) &= 1, \quad \text{for } x_1^2 + x_2^2 = \epsilon^2, \\ u(x_1, x_2) &= 0, \quad \text{for } x_1^2 + x_2^2 = 1, \end{aligned}$$

where $0 < \epsilon < 1$, defining the circular domain D with an inner boundary at ϵ and an outer boundary at 1.

The differential equation that $u(x_1, x_2)$ satisfies within the annular region $\epsilon^2 < x_1^2 + x_2^2 < 1$ is:

$$\nabla^2 u = 0,$$

This implies that u is a harmonic function within this domain.
Dirichlet's principle asserts that the function which minimizes the energy of the system, defined by the integral:

$$\min_{v \in V_\epsilon} \int_{\epsilon<|x|<1} |\nabla v(x)|^2 \, dx,$$

is the harmonic function $u(x_1, x_2)$ that satisfies the given boundary conditions!

Alternative Solution:
We can also arrive to the same conclusion through another method. Assume that we have some Integral $I(v)$ that accepts a function $v \in V_\epsilon$:

$$I(v) = \int_{\epsilon<|x|<1} |\nabla v(x)|^2 \, dx.$$

For functions v within V_ϵ we expand $I(v)$ such that

$$I(v) = \int_{\epsilon<|x|<1} \underbrace{\nabla v(x) \nabla v(x)}_{|\nabla v(x)|^2} \, dx.$$

Since our goal is to find a function within V_ϵ that minimizes the integral and it was suggested that it is the function $u(x)$ we still need to consider and show some variations around $u(x)$ and show that those variations do not lead to the lower value of our integral! This is why we need to consider some other function $f(x) = u(x) + \alpha \cdot g(x)$. This function $f(x)$ is just a perturbation of $u(x)$ where $g(x)$ is some function that also satisfies the boundary conditions and α is some scalar parameter that will control the magnitude and direction of our perturbation. Basically we explore how the integral will change if we slightly vary $u(x)$. Therefore we have

$$I(f) = \int_{\epsilon<|x|<1} \nabla f(x) \nabla f(x) dx.$$

Using the "proper" notation this would be written as:

$$I\left(f_\alpha\right) = \int_{\epsilon<|x|<1} |\nabla(u(x) + \alpha g(x))|^2 dx$$

Using the standard formula for $(a+b)^2 = a^2 + 2ab + b^2$ we calculate:

$$I\left(f_\alpha\right) = \int_{\epsilon<|x|<1} \left(\nabla u \nabla u + 2\alpha \nabla u \nabla g + \alpha^2 \nabla g \nabla g\right) dx$$

Now if we observe this expression we can note that these are just three integrals where we recognise that:

$$I\left(f_\alpha\right) = \underbrace{\int_{\epsilon<|x|<1} \nabla u \nabla u}_{I(u)} + \int_{\epsilon<|x|<1} 2\alpha \nabla u \nabla g + \int_{\epsilon<|x|<1} \alpha^2 \nabla g \nabla g dx$$

Since we can see that:

$$\int_{\epsilon<|x|<1} \alpha^2 \nabla g \nabla g dx \geq 0$$

We can write that:

$$I(u) + \int_{\epsilon<|x|<1} 2\alpha \nabla u \nabla g + \underbrace{\int_{\epsilon<|x|<1} \alpha^2 \nabla g \nabla g dx}_{\geq 0} \geq I(u) + \int_{\epsilon<|x|<1} 2\alpha \nabla u \cdot \nabla g$$

Now we remember that the function u satisfies the equation $A(u,v) = L(v)$ for all functions v belonging to the function space V_ϵ, in variational problem form:

- Specific bi-linear form: $A(u,g) = \int_{\epsilon<|x|<1} \nabla u \cdot \nabla g \; dx$
- Linear functional: $L(g) = \int_{\epsilon<|x|<1} 0 \cdot g \; dx = 0$

$$A(u,g) = L(g) \Longleftrightarrow A(u,g) = 0$$

$$I(u) + \int_{\epsilon<|x|<1} 2\alpha \nabla u \cdot \nabla g + \underbrace{\int_{\epsilon<|x|<1} \alpha^2 \nabla g \cdot \nabla g dx}_{\geq 0} \geq I(u) + 2\alpha \underbrace{\int_{\epsilon<|x|<1} \nabla u \cdot \nabla g}_{A(u,g)=0}$$

$$\underbrace{I(u) + \int_{\epsilon<|x|<1} 2\alpha \nabla u \cdot \nabla g + \underbrace{\int_{\epsilon<|x|<1} \alpha^2 \nabla g \cdot \nabla g dx}_{\geq 0}}_{I(f)} \geq I(u)$$

Which leads us to conclude that u minimises the integral:

$$\min_{v \in V_\epsilon} \int_{\epsilon<|x|<1} |\nabla v(x)|^2 \, dx = \int_{\epsilon<|x|<1} |\nabla u(x)|^2 \, dx$$

2.1.2 Analytical Solution

We know that that u minimizes the integral within V_ϵ.

$$\min_{v \in V_\epsilon} \int_{\epsilon<|x|<1} |\nabla v(x)|^2 \, dx = \int_{\epsilon<|x|<1} |\nabla u(x)|^2 \, dx.$$

To solve the Laplace equation $\Delta u = 0$, we transform it into polar coordinates, assuming rotational symmetry. This simplification leads to

$$\Delta = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} = \frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r} + \frac{1}{r^2}\frac{\partial^2}{\partial \theta^2}$$

$$u''(r) + \frac{1}{r}u'(r) = 0.$$

By setting $v(r) = u'(r)$, we reduce the order of our equation to

$$rv'(r) + v(r) = 0,$$

This is just a separable equation yielding $v(r) = \frac{C}{r}$.

Integrating this result gives us

$$u(r) = C\ln(r) + D.$$

Applying the boundary conditions $u(\epsilon) = 1$ and $u(1) = 0$, we determine

$$C = \frac{1}{\ln(\epsilon)} \text{ and } D = 0$$

which results in the final solution

$$u(r) = \frac{\ln(r)}{\ln(\epsilon)}. \tag{4}$$

2.1.3 Analysis of the Solution

The limit As we study the behaviour of ϵ as it approaches the limit of 0^+ in the analytical solution given by Eq. (4), we observe that the tent's structure undergoes a notable sharpening, becoming increasingly pronounced at its apex. This effect is amplified as ϵ draws closer to zero, with the tent's height at points near ϵ experiencing a significant rise.

Regarding the tent's height, at the center, corresponding to $|x| = \epsilon$, the tent's height steadfastly remains at 1, despite the diminishing ϵ. This consistency, however, morphs into a singular peak as ϵ verges on zero, creating a point of contrast. Meanwhile, at the tent's boundary, where $|x| = 1$, the height holds at 0, unaffected by the variations in ϵ.

The result of this is that we see gradual flattening of the tent's structure, with the exception of the pointed peak that continues to define its center. This extreme scenario reveals a fundamental breakdown in the model physical plausibility when pushed to its theoretical extremities.

The Maximum Principle The boundary value behavior of u is such that it attains its maximum and minimum values at the boundary of the domain $\Omega_\epsilon \backslash \partial\Omega_\epsilon$. Within the domain, for points where $|x|$ falls between ϵ and 1, the value of $u(x)$ is bounded between 0 and 1. This bounded nature is a consequence of the logarithmic formulation of $u(x)$, which ensures that $u(x)$ remains confined within these limits across the domain.

The maximum principle tells us that the height of the tent fabric, u, transitions smoothly from 1 at the center to 0 at the edge, in alignment with the established boundary conditions. In the limiting scenario, as ϵ approaches 0^+, the relevant boundary converges to the set of points where $|x| = 1$. Even in this case, the maximum principle remains applicable.

In essence, the maximum principle underscores the determination of the tent fabric's height u by its boundary values, facilitating a smooth gradation within the domain.

2.2 Problem 5

$$\min_{v\in V_\epsilon}\int_{\epsilon<|x|<1}|\nabla v(x)|^2dx = 2\pi\min_{v\in\hat{V}_\epsilon}\int_\epsilon^1 r\left(v'(r)\right)^2dr,$$

2.2.1 Lax-Milgram's Theorem

According to Sats 1.3 in *"A short introduction to the finite element method"*

$$A(u,v) = L(v) \quad \forall v \in V$$

$$A(v,v) = 2\pi\int_\epsilon^1 v'(r)^2rdr \quad |v|^2 = \int_\epsilon^1 v(r)^2 + v'(r)^2dr$$

$$v(r) = \begin{cases} 5-\frac{5r}{y} & \text{if } r \le y, \\ 0 & \text{if } r > y, \end{cases}$$

We have that $v'(r) = -\frac{5}{y}$ thus we calculate:

$$A(v,v) = 2\pi\int_0^y \frac{25r}{y^2}dr = 2\pi\cdot\frac{25}{2y^2}r^2\Big|_0^y = \frac{25}{2} = 25\pi$$

$$v(r)^2 = \left(5-\frac{5r}{y}\right)^2 \Rightarrow \int_0^y v(r)^2 = \frac{25y}{3}$$

$$v'(r)^2 = \frac{25}{y^2} \Rightarrow \int_0^y v'(r)^2 = \frac{25}{y}$$

$$\|v\|^2 = \frac{25y}{3}+\frac{25}{y} = \frac{25y^2+75}{3y}$$

For a given vector space V and a bi-linear form A, there exists a positive constant α such that for all vectors (or functions) v in V :

$$A(v,v) \ge \alpha\|v\|^2$$

So we have:

$$25\pi \ge \alpha\frac{25y^2+75}{3y}$$

However this is not always true!

$$\lim_{y\to0}\frac{25y^2+75}{3y} = \lim_{y\to0}\frac{25y+\frac{75}{y}}{3} \to \infty$$

More precisely the limit is indeterminate since fraction grow without bound, suggesting the limit tends towards infinity. The Lax-Miligram conditions are not satisfied and we cant say anything about existence or uniqueness.

Alternative Solution:
We can think of two ways to show that a solution w belonging to the set $\hat{V}_0$ cannot exist for our minimization problem.

- Method 1: $\lim_{\epsilon \to 0^+}$ and observing how a solution u behaves in $\hat{V}_\epsilon$.
- Method 2: Assume the existence of a solution $w \in \hat{V}_0$

Method 1:
$\lim_{\epsilon \to 0^+}$ as the inner boundary vanishes in the limit, u becomes zero on the entire boundary, following the maximum principle for harmonic functions.

Method 2:
Lets assume that there exists some $w \in \hat{V}_0$. This means that w is continuous since w is harmonic function. Thus we know that $\Delta w = 0$ for all $|x| < 1$. However, when applying Poisson's formula, we have a contradiction where $w(0)$ should be 1, but it is found to be 0 because w is zero over the entire boundary!

Conclusion:
Conditions required for Lax-Milgram's theorem are not met. Since there is no $w \in \hat{V}_0$ that solves our minimization problem.

2.3 Problem 6

Introduction In this section, we numerically approximate a specific integral, central to understanding the energy distribution within a given domain. This integral,

$$\int_{\epsilon<|x|<1} |\nabla u(x)|^2 \, \mathrm{d}x = 2\pi \min_{v \in \hat{V}_\epsilon} \int_\epsilon^1 \left(v'(r)\right)^2 r \, \mathrm{d}r,$$

captures the essence of the energy within the domain bounded by $\epsilon < |x| < 1$, where ϵ is a parameter that adjusts the domain's size, and $\hat{V}_\epsilon$ represents a functional space defined by specific boundary conditions.

To achieve this numerical approximation, we leverage the FEM. The function $U(x)$ is expressed as a linear combination of piecewise linear basis functions $\varphi_i(x)$, given by:

$$U(x) = \sum_{i=0}^{N} u_i \varphi_i(x), \tag{5}$$

where $u_0 = 1$ and $u_N = 0$ are the boundary conditions at the endpoints of the domain, and $\varphi_i(x)$ are designed to satisfy $\varphi_i(x_j) = \delta_{ij}$ (where δ_{ij} is the Kronecker delta), ensuring that each basis function is peaked at its respective nodal point and zero at all others. The nodal points $x(i)$ are strategically distributed across the domain, from c to 1, divided into N intervals.

Our focus then shifts to the minimization problem expressed as:

$$\min_{(u_1,\ldots,u_{N-1})\in\mathbb{R}^{N-1}} \int_\epsilon^1 x \left(U'(x)\right)^2 \, \mathrm{d}x, \tag{6}$$

which can be reformulated into a quadratic form involving the coefficients u_i of the basis functions. This reformulation yields

$$\min_{(u_1,\ldots,u_{N-1})\in\mathbb{R}^{N-1}} \sum_{i=0}^{N} \sum_{j=0}^{N} A_{ij} u_i u_j, \tag{7}$$

leading to a system of equations that must be solved to find the coefficients u_i that minimize the integral. The matrix A_{ij}, representing the interaction between basis functions φ_i and φ_j, is defined by:

$$A_{ij} = \int_\epsilon^1 x \varphi_i'(x) \varphi_j'(x) \mathrm{d}x,$$

encapsulating the weighted contribution of each pair of basis functions to the overall energy integral.

2.3.1 Analytical Derivation

We want to find the minimum of the integral

$$\min_{(u_1,\ldots,u_{N-1})\in\mathbb{R}^{N-1}} \int_\epsilon^1 x\left(U'(x)\right)^2 dx.$$

Here, $U'(x)$ represents the derivative of the function $U(x)$. The goal is to minimize this integral by appropriately choosing values for the parameters $u_1, \ldots, u_{N-1}$.

First we transform $U'(x)$ into a linear combination of basis functions

$$U'(x)^2 = \sum_{i=0}^{N} u_i \varphi_i'(x)$$

where $\varphi_i'(x)$ are the derivatives of given basis functions $\varphi_i(x)$. The term $\sum_{i=0}^{N} u_i\varphi_i'(x)$ is the representation of $U'(x)$ as a weighted sum of these basis function derivatives. Each u_i is a coefficient that multiplies the corresponding basis function derivative $\varphi_i'(x)$.

We now move on to choose the coefficients $u_1, \ldots, u_{N-1}$ in such a way that this new representation of the integral is minimized. We begin by expanding the square of the sum such that

$$\left(\sum_{i=0}^{N} u_i\varphi_i'(x)\right)^2 = \left(\sum_{i=0}^{N} u_i\varphi_i'(x)\right)\cdot\left(\sum_{j=0}^{N} u_j\varphi_j'(x)\right)$$

which yields

$$\int_\epsilon^1 x\left(\sum_{i=0}^{N} u_i\varphi_i'(x)\right)\left(\sum_{i=0}^{N} u_j\varphi_j'(x)\right)dx =$$

$$= \int_\epsilon^1 x\left(\sum_{j=0}^{N}\sum_{i=0}^{N} u_i\varphi_i' u_j\varphi_j'\right)dx$$

This double sum represents all combinations of the pairwise products of the terms $u_i\varphi_i'(x)$ and $u_j\varphi_j'(x)$. Essentially, every term in the original sum is multiplied by every other term, including itself.

We now separate the integral from the coefficients u_i and u_j such that

$$\int_\epsilon^1 x\left(\sum_{j=0}^{N}\sum_{i=0}^{N} u_i\varphi_i' u_j\varphi_j'\right)dx = \sum_{i=0}^{N}\sum_{j=0}^{N} u_i u_j \int_\epsilon^1 x\varphi_i'\varphi_j' dx.$$

Finally, we introduce the matrix $A_{ij} = \int_\epsilon^1 x\varphi_i'(x)\varphi_j'(x)dx$, which yields

$$\sum_{i=0}^{N}\sum_{j=0}^{N} u_i u_j \int_\epsilon^1 x\varphi_i'\varphi_j' dx = \sum_{i=0}^{N}\sum_{j=0}^{N} u_i u_j A_{ij}$$

To minimize the resulting expression, we compute the gradient:

$$\frac{\partial}{\partial u_\phi}\left(\sum_{i=0}^{N}\sum_{j=0}^{N} u_i u_j A_{ij}\right)$$

for each term $u_i\ u_j\ A_{ij}$ where either $i = \phi$ or $j = \phi$ or $i = j = \phi$ will contribute to the derivative. The derivative of u_ϕ is 1, and the derivative of any u_i where $i \neq \phi$ is 0, which yields:

$$2\sum_{i=0}^{N} u_i A_{\phi i} = 2u_0A_{\phi 0} + 2u_1A_{\phi 1} + 2u_2A_{\phi 2} + \ldots + 2u_{N-1}A_{\phi(N-1)} + 2u_NA_{\phi N}$$

where the first and last term represent our boundary conditions. We can rewrite the expansion as:

$$2\sum_{i=0}^{N} u_i A_{\phi i} = 2\sum_{i=1}^{N-1} u_i A_{\phi i} + \underbrace{2u_0A_{\phi 0}}_{u_0=1} + \underbrace{2u_NA_{\phi N}}_{u_N=0}$$

We know that our boundary conditions are $u_0 = 1$ and $u_N = 0$, which yields

$$2\sum_{i=1}^{N-1} u_i A_{\phi i} + 2u_0A_{\phi 0} + 2u_NA_{\phi N} = 2\sum_{i=1}^{N-1} u_i A_{\phi i} + 2A_{\phi 0}.$$

To minimize this expression, we set the gradient to zero

$$\frac{\partial}{\partial u_\phi}\left(\sum_{i=0}^{N}\sum_{j=0}^{N} u_i u_j A_{ij}\right) = 2\sum_{i=1}^{N-1} u_i A_{\phi i} + 2A_{\phi 0} = 0$$

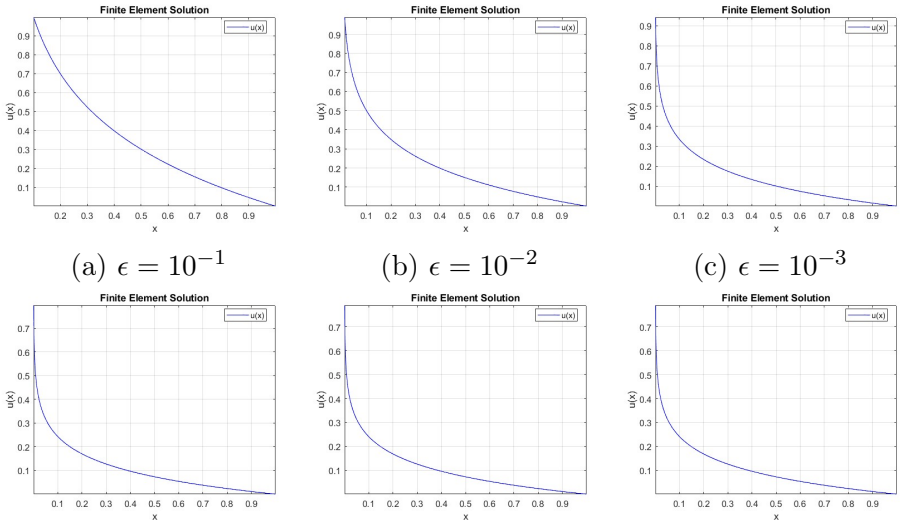
which is equivalent to

$$\sum_{i=1}^{N-1} u_i A_{\phi i} = -A_{\phi 0}.$$

This holds for all ϕ in $\{1, \ldots, N-1\}$, leading to a system of linear equations to find the minimum of the original integral.

2.3.2 Numerical Approximation

To illustrate the effectiveness and versatility of this numerical approach, we present plots showcasing the solution U for various values of ϵ in Fig. 5, providing a visual representation of how the domain's size influences the energy distribution and the overall shape of the function $U(x)$ within the domain.



(a) $\epsilon = 10^{-1}$ (b) $\epsilon = 10^{-2}$ (c) $\epsilon = 10^{-3}$

(d) $\epsilon = 10^{-5}$ (e) $\epsilon = 10^{-8}$ (f) $\epsilon = 10^{-10}$

Figure 5: Result of $N = 2000$

2.4 Problem 7

Introduction In this section, we numerically approximate the function $U(x)$, as defined by Eq. (5) and Eq. (6) and in turn discretized by Eq. (7).

The Gradient Method To tackle this minimization problem, we employ an iterative strategy known as the gradient method. This approach iteratively refines the coefficients u_k using the update rule:

$$u_k[n+1] = u_k[n] - \delta \frac{\partial}{\partial u_k} \left(\sum_{i=0}^{N} \sum_{j=0}^{N} \Lambda_{ij} u_i[n] u_j[n] \right), \quad n = 1, \ldots, N-1, \quad (8)$$

where δ is a carefully chosen positive step size, and n denotes the iteration step.

A critical aspect of this iterative process is the selection of the step size δ, which must strike a balance between computational efficiency and the stability of the method. The iteration process bears resemblance to finite difference methods used for solving the heat equation, wherein stability is contingent upon the λ ratio, defined by $\lambda = \frac{\delta t}{\delta x^2} \leq \frac{1}{2}$. This ensures that the method is stable and the solution does not diverge as time progresses.

In the context of the given minimization problem, the iterative method for updating u_k is analogous to the explicit time-stepping method for the heat equation. To ensure stability, the step size δ must adhere to a condition that prevents the iterative process from diverging, thereby guaranteeing convergence to a solution.

Stability condition The iterative update rule we're applying for the gradient descent method is given by Eq. (8). The partial derivative with respect to u_k in the context of the minimization problem is

$$\frac{\partial}{\partial u_k} \left(\sum_{i=0}^{N} \sum_{j=0}^{N} A_{ij} u_i[n] u_j[n] \right) = 2 \sum_{j=0}^{N} A_{kj} u_j[n]$$

where we multiply by 2 since the derivative of $u_i u_j$ with respect to u_k yields two terms when $i = k$ or $j = k$ and $i \neq j$. We can now simplify the update

rule by plugging in the partial derivative, which gives us

$$u_k[n+1] = u_k[n] - 2\delta \sum_{j=0}^{N} A_{kj} u_j[n].$$

For the iterative process to be stable, the magnitude of the updated value $u_k[n+1]$ can not exceed the current magnitude of $u_k[n]$. This can be expressed as

$$|2\delta \sum_{j=0}^{N} A_{kj} u_j[n]| \leq |u_k[n]|.$$

Now, consider the worst-case scenario where the sum of terms $\sum_{j=0}^{N} A_{kj} u_j[n]$ yields the maximum value, let's denote this maximum value as M. We express this scenario as

$$|2\delta M| \leq |2u_{\bar{m}}[n]|$$

where $u_{\bar{m}}[n]$ is current value of $u_k[n]$ that will be updated with the maximum possible change. Since δ is inherently positive we finally arrive at the stability condition

$$0 < \delta \leq \frac{|u_{\bar{m}}[n]|}{\left|\sum_{j=0}^{N} A_{\bar{m}j} u_j[n]\right|}.$$

which ensures that the iterative gradient descent process remains stable and converges to a solution.

Result In Fig. 6 we can observe the relationship between the step size δ and the number of iterations required to achieve convergence for the numerical approximation of $U(x)$ using the gradient method as defined above. The convergence threshold is set at 1×10^{-6}, with nodal points $N = 25$ and $\epsilon = 0.1$.

As depicted, there is an inverse correlation between δ and the required number of iterations to achieve convergence. As δ increases, the number of iterations required decreases exponentially. This is an expected characteristic of gradient descent methods, where a smaller step size typically leads to more accurate, more computationally demanding convergence due to the higher resolution.

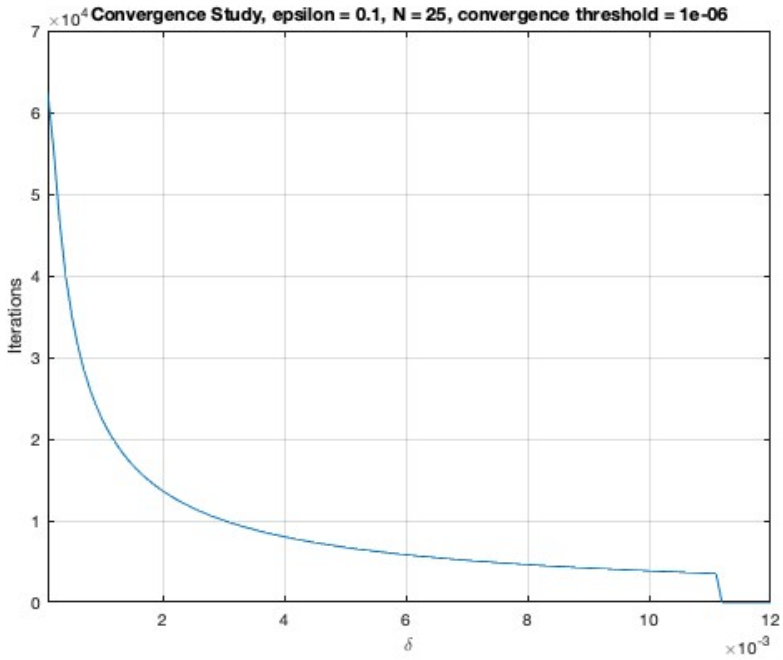


Figure 6: Convergence study of gradient method

However, the plot also reveals an interesting plateau effect for values of $\delta > 6 \times 10^{-3}$, where the number of iterations levels off. This plateau may suggest that below a certain δ, further reductions do not significantly affect the convergence rate, possibly due to reaching the limits of numerical precision or the algorithm's inherent sensitivity to changes in δ.

Finally, the sharp cut-off in iteration counts at $\delta \approx 11 \times 10^{-3}$, indicates a point where the larger size of δ introduces instability to the method such that it does not converge anymore.

In conclusion, for this specific problem with the chosen parameters, picking a value for delta such that $6 \times 10^{-3} < \delta < 11 \times 10^{-3}$ will offer a good balance between accuracy and computational efficiency.

3 Appendix

3.1 Code

Problem 3

finiteelementmethod.m

```
clear all;
%-----------------------------------------------------------------
% Define functions f(x) and a(x), as well as the boundary value g
%-----------------------------------------------------------------

% Alternative => 1
% f_x = @(x) exp(x); a_x = @(x) exp(x);
% real_solution = @(x) (1+exp(1))*(1-exp(-1*x)) - x; % Exact solution for alt 1
%-----------------------------------------------------------------

% Alternative => 2
 f_x = @(x) 0; a_x = @(x) 1 + x;
 real_solution =@(x) log(1+x);
%-----------------------------------------------------------------

boundary_value = 1; % Boundary value at x=1

%-----------------------------------------------------------------
% Define the number of mesh points
num_mesh_points = 100; % Number of mesh points
x_values = linspace(0, 1, num_mesh_points + 1); % Equally spaced mesh points

for gauss_order = 1:2
    % Define weights and nodes for Gaussian quadrature on [-1, 1]
    if gauss_order == 2
        gauss_nodes = [-1/sqrt(3), 1/sqrt(3)];
        gauss_weights = [1, 1];
```

```
    elseif gauss_order == 1
        gauss_nodes = [0];
        gauss_weights = [2];
    else
        disp('Invalid number of points for Gaussian quadrature!
            ↪ ');
        return;
    end

    % Initialize coefficient matrix (A) and load vector (L)
    A = zeros(num_mesh_points, num_mesh_points); % Coefficient
        ↪ matrix
    L = zeros(num_mesh_points, 1); % Load vector

    for element = 1:num_mesh_points-1
        left_endpoint = x_values(element); % Left endpoint of
            ↪ the element
        right_endpoint = x_values(element+1); % Right endpoint
            ↪ of the element

        for gauss_point = 1:gauss_order
            % Gaussian quadrature points and weights
            xi = (right_endpoint - left_endpoint) / 2 *
                ↪ gauss_nodes(gauss_point) + ...
                 (left_endpoint + right_endpoint) / 2;
            weight = gauss_weights(gauss_point) * (
                ↪ right_endpoint - left_endpoint) / 2;

            % Shape functions and their derivatives
            phi_left = 1 - (xi - left_endpoint) / (
                ↪ right_endpoint - left_endpoint);
            phi_right = 1 - (right_endpoint - xi) / (
                ↪ right_endpoint - left_endpoint);
            phi_left_prime = -1 / (right_endpoint -
                ↪ left_endpoint);
            phi_right_prime = 1 / (right_endpoint -
                ↪ left_endpoint);

```

```
56            % Evaluate f(x) and a(x) at the Gaussian point
57            f_xi = f_x(xi);
58            a_xi = a_x(xi);
59
60            % Update the coefficient matrix A and the load
                  ↪ vector L
61            A(element, element) = A(element, element) + weight *
                  ↪  a_xi * phi_left_prime * phi_left_prime;
62            A(element, element+1) = A(element, element+1) +
                  ↪ weight * a_xi * phi_left_prime *
                  ↪ phi_right_prime;
63            A(element+1, element) = A(element+1, element) +
                  ↪ weight * a_xi * phi_right_prime *
                  ↪ phi_left_prime;
64            A(element+1, element+1) = A(element+1, element+1) +
                  ↪ weight * a_xi * phi_right_prime *
                  ↪ phi_right_prime;
65            L(element) = L(element) + weight * phi_left * f_xi;
66            L(element+1) = L(element+1) + weight * phi_right *
                  ↪ f_xi;
67        end
68
69        % Apply boundary condition a(1)u'(1) = g:
70        if element+1 == num_mesh_points
71            L(element+1) = L(element+1) + boundary_value;
72        end
73    end
74
75    % Apply boundary condition u(0) = 0:
76    A(1, :) = 0;
77    A(1, 1) = 1;
78    L(1) = 0;
79
80    % Solve the system of equations
81    u = A \ L;
82
83    % Plot the results
84    num_plot_points = num_mesh_points;
```

```
    plot_step = 1 / num_plot_points;

    xx = zeros(num_plot_points, 1);
    yy = zeros(num_plot_points, 1);

    node_step = 1 / (num_mesh_points - 1);
    for j = 1:num_plot_points
        x_j = (j-1) * plot_step;
        xx(j) = x_j;
        yy(j) = sum(arrayfun(@(i) max(0, 1 - abs(x_j - (i-1) *
            ↪ node_step) / node_step) * u(i), 1:num_mesh_points
            ↪ ));
    end
    figure;
    hold on;

    % Plot FEM Solution
    plot(xx, yy, 'LineWidth', 1, 'LineStyle', '-', 'Color', [1,
        ↪  0, 0], 'DisplayName', 'FEM Solution'); % Red

    % Plot Exact Solution
    plot(xx, arrayfun(real_solution, xx), 'LineWidth', 2, '
        ↪ LineStyle', '--', 'Color', [0, 0, 0], 'DisplayName',
        ↪ 'Exact Solution'); % Black

    % Enhance the plot with labels, legend, and grid
    xlabel('x', 'FontSize', 12);
    ylabel('Solution', 'FontSize', 12);
    title(['Gaussian Quadrature Order: ', num2str(gauss_order)
        ↪ ], 'FontSize', 14);
    legend('Location', 'best');
    grid on;
    box on;

% Calculate the difference between FEM and exact solution
diff = yy' - arrayfun(real_solution, x_values(1:end-1)); %
    ↪ Ensure diff has size n
```

integrate.m

```
function result = integrate(funct, start, stop)

stepSize = 10^(-3);
    result = 0;

    for x = start:stepSize:stop
        result = result + funct(x) * stepSize;
    end
end
```

generatefiniteelements.m

```
% Initialization of Finite Element and Derivative Arrays
finite_elements = {};
finite_elements_derivative = {};
num_elements = 50;
small_offset = 1e-9; % Small epsilon to avoid boundary issues
step_size = (1 - small_offset) / num_elements;

% Generation of Finite Elements and their Derivatives
for i = 0:num_elements
    point = small_offset + i * step_size;
    finite_elements{i+1} = @(x) max(0, 1 - abs((x - point) /
        ↪ step_size));
    finite_elements_derivative{i+1} = @(x) (1 / step_size) * (x
        ↪  < point & x > (point - step_size)) - ...
                                        (1 / step_size) * (x >
                                            ↪ point & x < (point
                                            ↪ + step_size));
end

% Construct Coefficient Matrix A
A = zeros(num_elements + 1, num_elements + 1);
for i = 0:num_elements
    for j = 0:num_elements
        A(i+1, j+1) = integrate(@(x) finite_elements_derivative
            ↪ {i+1}(x) .* finite_elements_derivative{j+1}(x) .*
```

```
            ↪ x, small_offset, 1);
    end
end

% Iterative Solver Parameters
max_iterations = 1e6;
tolerance = 1e-5;
adjustment_factor = 0.005;

% Initial Guess for Iterative Solver
solution_guess = rand(num_elements + 1, 1);
solution_guess(1) = 1; % Boundary Condition at start
solution_guess(num_elements + 1) = 0; % Boundary Condition at
    ↪ end

% Iterative Solver Loop
difference = ones(num_elements + 1, 1);
iterations = 0;
while norm(difference) > tolerance && iterations <
    ↪ max_iterations
    derivatives = 2 * (A * solution_guess);
    difference = adjustment_factor * derivatives;
    difference(1) = 0; % Fixing the boundary values
    difference(num_elements + 1) = 0;

    solution_guess = solution_guess - difference;
    iterations = iterations + 1;
end

% Display Results
final_norm_of_difference = norm(difference);
disp(['Final norm of difference: ', num2str(
    ↪ final_norm_of_difference)]);
disp(['Iterations: ', num2str(iterations)]);

% Prepare for Plotting
coefficients = solution_guess;
num_plot_points = 100;
```

```
plot_step_size = (1 - small_offset) / num_plot_points;

% Generate Plot Points
xx = zeros(num_plot_points + 1, 1);
yy = zeros(num_plot_points + 1, 1);
for j = 0:num_plot_points
    result = 0;
    x_point = small_offset + j * plot_step_size;
    for i = 1:num_elements
        result = result + finite_elements{i}(x_point) *
            ↪ coefficients(i);
    end
    xx(j+1) = x_point;
    yy(j+1) = result;
end

% Plotting the Solution
plot(xx, yy);
title('Finite Element Method Solution');
xlabel('x');
ylabel('Solution y');
```

l2_error.m

```
clear all;

num_elements_array = [];
computation_times = [];
l2_norm_errors = [];

for num_mesh_points = 10:10:100

    % f_x = @(x) exp(x);
    % a_x = @(x) exp(x);
    % real_solution = @(x) (1 + exp(1)) * (1 - exp(-x)) - x;

    f_x = @(x) 0;
    a_x = @(x) 1 + x;
```

```
15      real_solution = @(x) log(1+x);
16
17      boundary_value = 1;
18
19      x_values = linspace(0, 1, num_mesh_points + 1);
20
21      A = zeros(num_mesh_points, num_mesh_points);
22      L = zeros(num_mesh_points, 1);
23
24
25      gauss_nodes = 0;
26      gauss_weights = 2;
27
28      % Assembly process
29      for element = 1:num_mesh_points-1
30          left_endpoint = x_values(element);
31          right_endpoint = x_values(element+1);
32
33          for gauss_point = 1:length(gauss_nodes)
34              xi = (right_endpoint - left_endpoint) / 2 *
                    ↪ gauss_nodes(gauss_point) + (left_endpoint +
                    ↪ right_endpoint) / 2;
35              weight = gauss_weights(gauss_point) * (
                    ↪ right_endpoint - left_endpoint) / 2;
36
37              phi_left = 1 - (xi - left_endpoint) / (
                    ↪ right_endpoint - left_endpoint);
38              phi_right = 1 - (right_endpoint - xi) / (
                    ↪ right_endpoint - left_endpoint);
39              phi_left_prime = -1 / (right_endpoint -
                    ↪ left_endpoint);
40              phi_right_prime = 1 / (right_endpoint -
                    ↪ left_endpoint);
41
42              f_xi = f_x(xi);
43              a_xi = a_x(xi);
44
45              A(element, element) = A(element, element) + weight *
```

```
            ↪  a_xi * phi_left_prime * phi_left_prime;
        A(element, element+1) = A(element, element+1) +
            ↪ weight * a_xi * phi_left_prime *
            ↪ phi_right_prime;
        A(element+1, element) = A(element+1, element) +
            ↪ weight * a_xi * phi_right_prime *
            ↪ phi_left_prime;
        A(element+1, element+1) = A(element+1, element+1) +
            ↪ weight * a_xi * phi_right_prime *
            ↪ phi_right_prime;
        L(element) = L(element) + weight * phi_left * f_xi;
        L(element+1) = L(element+1) + weight * phi_right *
            ↪ f_xi;
    end
end

% Apply boundary conditions
L(end) = L(end) + boundary_value; % Neumann BC
A(1, :) = 0; A(1, 1) = 1; L(1) = 0; % Dirichlet BC

u = A \ L;

fem_solution = zeros(size(x_values));
for i = 1:length(x_values)
    if i < length(x_values)
        fem_solution(i) = u(i);
    else
        fem_solution(i) = u(end);
    end
end

% Compute the L^2 norm of the error
exact_solution = arrayfun(real_solution, x_values);
error = fem_solution - exact_solution;
l2_norm = sqrt(trapz(x_values, error.^2));

num_elements_array(end+1) = num_mesh_points;
l2_norm_errors(end+1) = l2_norm;
```

```
end

figure;
loglog(num_elements_array, l2_norm_errors);
ylabel('L^2 Norm of Error');
xlabel('Number of Elements');
title('Number of Elements vs. Error');
grid on;
```

Problem 6

main.m

```
N = 2000;
eps = 0.0000000001;

[x, u] = finite_element(eps, N);
plot_solution(x, u);

%123
```

finite_element.m

```
function [x_result, u_result] = finite_element(epsilon,
    ↪ num_intervals)
    % Inputs:
    % epsilon: Lower bound such that epsilon <= r <= 1
    % num_intervals: Number of intervals (one less than the
        ↪ number of x values)

    % Generate evenly spaced x values from epsilon to 1
    x_result = linspace(epsilon, 1, num_intervals + 1);

    % Calculate the left and right neighbor x values
    x_left_neighbor = [x_result(2:end), 1];
    x_right_neighbor = [epsilon, x_result(1:end - 1)];

```

```
    % Define the main, super, and sub-diagonals of the sparse
        ↪ matrix A
    main_diagonal = (num_intervals / (1 - epsilon))^2 / 2 .* (
        ↪ x_left_neighbor.^2 - x_right_neighbor.^2);
     sub_diagonal = -(num_intervals / (1 - epsilon))^2 / 2 .* (
         ↪ x_left_neighbor.^2 - x_result.^2);
    super_diagonal = -(num_intervals / (1 - epsilon))^2 / 2 .*
        ↪ (x_result.^2 - x_right_neighbor.^2);


    % Create the sparse matrix A using the diagonals
    A = spdiags([sub_diagonal', main_diagonal', super_diagonal
        ↪ '], -1:1, num_intervals + 1, num_intervals + 1);

    % Remove the boundary values and corresponding rows/columns
    L = A(2:end - 1, 1);
    A = A(2:end - 1, 2:end - 1);
    x_result = x_result(2:end - 1);

    % Solve for u_result using A and L
    u_result = A \ -L;
end
```

plot_solution.m

```
function plot_solution(x_values, u_values)
    figure; % Create a new figure window
    plot(x_values, u_values, 'b', 'LineWidth', 1); % Plot the
        ↪ solution with a blue line and thicker line width
    xlabel('x'); % Label the x-axis
    ylabel('u(x)'); % Label the y-axis
    title('Finite Element Solution'); % Add a title to the plot
    grid on; % Display grid lines
    axis tight; % Fit the axis limits to the data
    set(gca, 'FontSize', 12); % Increase font size for better
        ↪ readability
    legend('u(x)', 'Location', 'Best'); % Add a legend
end
```

3.1.1 Problem 7

main7.m

```
epsilon = 0.1;
N = 25;
delta_values = 0.0001:0.0001:0.015;
convergence_threshold = 1e-6;
max_iterations = 100000;

x = linspace(epsilon, 1, N+1)';
u = zeros(N+1, length(delta_values));
u(1,:) = 1; % Boundary condition u_0 = 1
u(end,:) = 0; % Boundary condition u_N = 0
iterations = zeros(length(delta_values), 1);

A = compute_matrix_A(N, epsilon);

for d = 1:length(delta_values)
    delta = delta_values(d);
    for n = 1:max_iterations
        u_old = u(:,d);
        for k = 2:N % Exclude boundary points
            gradient = 2 * (A(k-1,:) * u_old(2:N)) - A(k-1,1) -
                ↪ A(k-1,end-1)*u_old(N+1);
            u(k,d) = u(k,d) - delta * gradient;
        end
        if norm(u(2:N,d) - u_old(2:N), 2) <
            ↪ convergence_threshold
            iterations(d) = n;
            break;
        end
    end
end

figure;
plot(delta_values, iterations);
xlabel('\delta');
```

```
ylabel('Iterations');
xlim([0.0001, 0.012])
title(['Convergence Study, epsilon = ', num2str(epsilon), ', N = ', num2str(N), ', convergence threshold = ', num2str(convergence_threshold)]);
grid on;
```

compute_matrix_A.m

```
function A = compute_matrix_A(N, epsilon)
    x = linspace(epsilon, 1, N);

    A = zeros(N-1, N-1);

    for i = 1:N-1
        for j = 1:N-1
            A(i,j) = integral(@(x) x .* dphi(x,i,epsilon,N) .* dphi(x,j,epsilon,N), epsilon, 1);
        end
    end
end
```

dphi.m

```
function dphi_val = dphi(x, i, epsilon, N)
    xi = epsilon + (i-1)*(1-epsilon)/N;
    xi_minus = epsilon + (i-2)*(1-epsilon)/N;
    xi_plus = epsilon + i*(1-epsilon)/N;

    dphi_val = zeros(size(x));

    left_interval = (x >= xi_minus) & (x < xi);
    dphi_val(left_interval) = 1 / (xi - xi_minus);

    right_interval = (x >= xi) & (x < xi_plus);
    dphi_val(right_interval) = -1 / (xi_plus - xi);
end
```