Math 588 - Finite Element Method

Final Project

Lazizbek Sadullaev

April 2025

Contents

Part A: Project Tasks

The final project for **Math 588 — Finite Element Method** focuses on solving the following time-dependent boundary value problem (BVP):

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0, \quad 0 < x < 1,\ t > 0, \tag{1}$$

subject to homogeneous Dirichlet boundary conditions:

$$u(0,t) = 0, \quad u(1,t) = 0, \tag{2}$$

and the initial condition:

$$u(x,0) = u_0(x), \tag{3}$$

where the given initial profile $u_0(x)$ is:

$$u_0(x) = \begin{cases} 2x, & 0 \le x \le \frac{1}{2}, \\ 2 - 2x, & \frac{1}{2} < x \le 1. \end{cases}$$

The main project objectives are:

- Derive the semidiscrete variational formulation of the BVP.
- Apply the Forward Euler method for time discretization.
- Construct and describe the structure of the stiffness matrix $\mathbf{A}$, mass matrix $\mathbf{B}$, and load vector $\mathbf{f}$.
- Solve the problem numerically by modifying and extending the provided Python code.

1 Methodology: Derivation of the Semidiscrete Formulation

1.1 Function Spaces

We define the spatial domain and time interval as:

$$\Omega = (0,1), \quad T = (0, T_{\text{final}}), \quad \text{with } T_{\text{final}} > 0.$$

The admissible space V for solutions and test functions is:

$$V = \{v \in C(\Omega) \mid v'(x) \text{ is piecewise continuous and bounded on } \Omega, \quad v = 0 \text{ on } \partial\Omega\}.$$

1.2 Weak Formulation

Multiplying the PDE by a test function $v \in V$ and integrating over Ω gives the **weak form**:

$$\int_\Omega \frac{\partial u}{\partial t} v \, dx + \int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx, \quad \forall v \in V, \tag{4}$$

where in this problem $f(x) = 0$.

The initial condition is:

$$u(x,0) = u_0(x), \quad x \in \Omega. \tag{5}$$

1.3 Finite Element Approximation

We approximate $u(x,t)$ by $u_h(x,t) \in V_h$, where V_h is a finite-dimensional subspace spanned by basis functions $\{\varphi_i(x)\}_{i=1}^M$:

$$u_h(x,t) = \sum_{i=1}^M u_i(t)\varphi_i(x).$$

Choosing test functions $v = \varphi_j(x)$, for each $j = 1, 2, \ldots, M$, substitution into the weak form yields:

$$\int_\Omega \left(\sum_{i=1}^M \frac{du_i(t)}{dt} \varphi_i(x) \right) \varphi_j(x) \, dx + \int_\Omega \left(\sum_{i=1}^M u_i(t) \nabla\varphi_i(x) \right) \nabla\varphi_j(x) \, dx = \int_\Omega f \varphi_j(x) \, dx.$$

Rearranging the sums:

$$\sum_{i=1}^M \left(\int_\Omega \varphi_i(x)\varphi_j(x) \, dx \right) \frac{du_i(t)}{dt} + \sum_{i=1}^M \left(\int_\Omega \nabla\varphi_i(x)\nabla\varphi_j(x) \, dx \right) u_i(t) = \int_\Omega f \varphi_j(x) \, dx.$$

1.4 Matrix Formulation

Define the matrices and vectors:

$$\mathbf{B} = [b_{ij}], \quad b_{ij} = \int_\Omega \varphi_i(x)\varphi_j(x)\,dx,$$

$$\mathbf{A} = [a_{ij}], \quad a_{ij} = \int_\Omega \nabla\varphi_i(x)\nabla\varphi_j(x)\,dx,$$

$$\mathbf{u}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_M(t) \end{bmatrix}, \quad \mathbf{f}(t) = \begin{bmatrix} f_1(t) \\ f_2(t) \\ \vdots \\ f_M(t) \end{bmatrix}, \quad f_j(t) = \int_\Omega f(x)\varphi_j(x)\,dx.$$

Thus, the **semidiscrete system** reads:

$$\mathbf{B}\frac{d\mathbf{u}(t)}{dt} + \mathbf{A}\mathbf{u}(t) = \mathbf{f}(t), \quad t \in (0,T), \tag{6}$$

with initial condition:

$$\mathbf{B}\mathbf{u}(0) = \mathbf{u}_0, \quad \text{where} \quad (\mathbf{u}_0)_j = \int_\Omega u_0(x)\varphi_j(x)\,dx. \tag{7}$$

1.5 Time Discretization: Forward Euler Method

The time derivative is discretized using the Forward Euler scheme:

$$\frac{d\mathbf{u}(t)}{dt} \approx \frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t},$$

where Δt is the time step size.

Substituting into the semidiscrete system gives:

$$\mathbf{B}\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} + \mathbf{A}\mathbf{u}^n = \mathbf{f}^n.$$

Rearranging, the fully discrete scheme becomes:

$$\mathbf{B}\mathbf{u}^{n+1} = (\mathbf{B} - \Delta t\mathbf{A})\,\mathbf{u}^n + \Delta t\mathbf{f}^n. \tag{8}$$

At each time step, we solve this linear system to advance the solution.

2 Structure of the Stiffness Matrix, Mass Matrix, and Load Vector

We now derive the explicit forms of the matrices **A**, **B**, and the load vector **f** for our specific case with $n_{\text{elem}} = 6$ elements and $n_{\text{node}} = 7$ nodes.

1. **Element Matrices for Linear Finite Elements**

   For each element e between nodes i and $i+1$ of length h, the local stiffness matrix and mass matrix are given by:

   $$\mathbf{A}^{(e)} = \frac{1}{h}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \quad \mathbf{B}^{(e)} = \frac{h}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}.$$

   Since the mesh is uniform on $(0,1)$, the element size is:

   $$h = \frac{1-0}{6} = \frac{1}{6}.$$

2. **Assembly of the Global Stiffness Matrix A**

   Assembling the contributions from all elements leads to the global stiffness matrix:

   $$\mathbf{A} = 6\begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}.$$

   (Note: the matrix is multiplied by 6 because $h^{-1} = 6$).

3. **Assembly of the Global Mass Matrix B**

   Similarly, the global mass matrix is assembled as:

   $$\mathbf{B} = \frac{1}{36}\begin{bmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 \end{bmatrix}.$$

   (Note: the factor $\frac{h}{6} = \frac{1}{36}$ comes from the mass matrix scaling.)

4. **Load Vector f**

   The load vector entries are defined by:

   $$f_j = \int_\Omega f(x)\varphi_j(x)\,dx.$$

   Since $f(x) = 0$ in our problem, it follows that:

   $$\mathbf{f} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0}$$

5. **Application of Boundary Conditions**

   The homogeneous Dirichlet boundary conditions:

   $$u(0,t) = u(1,t) = 0$$

   are imposed by modifying the global matrices and load vector:

   - For the first node (node 0):

   $$A_{00} = 1, \quad A_{01} = 0, \quad B_{00} = 1, \quad B_{01} = 0, \quad f_0 = 0.$$

   - For the last node (node 6):

   $$A_{66} = 1, \quad A_{65} = 0, \quad B_{66} = 1, \quad B_{65} = 0, \quad f_6 = 0.$$

   After these modifications, the boundary nodes are fixed exactly at zero throughout the simulation.

---

3 Numerical Solution and Python Code Implementation

In this section, we describe the numerical solution procedure and the corresponding Python code implementation used to solve the semidiscrete system with Forward Euler time-stepping.

3.1 Time Discretization: Forward Euler Scheme

Recall that the semidiscrete system reads:

$$\mathbf{B}\frac{d\mathbf{u}(t)}{dt} + \mathbf{A}\mathbf{u}(t) = \mathbf{f}(t).$$

Using the Forward Euler method to approximate the time derivative:

$$\frac{d\mathbf{u}(t)}{dt} \approx \frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t},$$

leads to the fully discrete scheme:

$$\mathbf{B}\mathbf{u}^{n+1} = \left(\mathbf{B} - \Delta t\mathbf{A}\right)\mathbf{u}^n + \Delta t\mathbf{f}^n.$$

At each time step:

- Compute the right-hand side: $(\mathbf{B} - \Delta t\mathbf{A})\mathbf{u}^n$,
- Solve for the next time step solution $\mathbf{u}^{n+1}$.

3.2 Initialization: Mesh and Matrices

The domain $\Omega = (0, 1)$ is partitioned into $n_{\text{elem}} = 6$ uniform elements, resulting in $n_{\text{node}} = 7$ nodes located at:

$$x_i = i \times h, \quad h = \frac{1}{6}, \quad i = 0, 1, \ldots, 6.$$

The mass matrix $\mathbf{B}$ and stiffness matrix $\mathbf{A}$ are assembled according to the standard finite element procedure for linear elements, as detailed in Section 2.

3.3 Initial Condition

The initial condition $u_0(x)$ is defined pointwise as:

$$u_0(x) = \begin{cases} 2x, & 0 \le x \le \frac{1}{2}, \\ 2 - 2x, & \frac{1}{2} < x \le 1, \end{cases}$$

and evaluated at each node to initialize the solution vector $\mathbf{u}^0$.

3.4 Boundary Conditions

The homogeneous Dirichlet boundary conditions $u(0,t) = 0$ and $u(1,t) = 0$ are enforced by:

- Modifying the first and last rows of $\mathbf{A}$ and $\mathbf{B}$ to identity conditions,
- Setting the first and last components of $\mathbf{u}^n$ to zero after each time step.

3.5 Python Code Structure

The main steps in the Python program are organized as follows:

1. Define the mesh and initialize the node coordinates.
2. Assemble the mass matrix $\mathbf{B}$ and stiffness matrix $\mathbf{A}$.
3. Apply boundary conditions on $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{u}^0$.
4. Initialize the time-stepping parameters: time step size Δt and total simulation time T.
5. Loop over time steps:
    - Compute the right-hand side $(\mathbf{B} - \Delta t\mathbf{A})\mathbf{u}^n$,
    - Solve for $\mathbf{u}^{n+1}$ using a tridiagonal solver (Thomas algorithm),
    - Enforce boundary conditions on the updated solution $\mathbf{u}^{n+1}$.
6. Plot the numerical solution at selected time instances.

3.6 Key Python Functions

- `solve_tridiagonal`: Implements the Thomas algorithm for efficiently solving tridiagonal systems.
- `initial_condition`: Evaluates the piecewise initial profile $u_0(x)$.
- `main`: Performs mesh generation, matrix assembly, time-stepping, and plotting.

3.7 Choice of Parameters

The time step size Δt and final time T are chosen to ensure stability and accuracy for the Forward Euler method. In the simulations:

$$\Delta t = 0.001, \quad T = 0.1, \quad \text{thus } n_{\text{steps}} = 100.$$

3.8 Plotting the Results

The evolution of the numerical solution $\mathbf{u}(x,t)$ is plotted at selected times to visualize the diffusion of the initial triangular profile towards the steady state (zero solution).

4 Python Code Implementation

The following Python code implements the numerical method described previously. It assembles the finite element matrices, applies the Forward Euler time-stepping scheme, and plots the evolution of the solution $u(x,t)$.

```python
import numpy as np
import matplotlib.pyplot as plt

def solve_tridiagonal(dim, b, diag_low, diag, diag_upper):
    # Thomas algorithm for solving tridiagonal systems
    w = np.zeros(dim)
    v = np.zeros(dim)
    z = np.zeros(dim)
    x = np.zeros(dim)

    for i in range(dim):
        if i == 0:
            w[i] = diag[i]
            v[i] = diag_upper[i] / w[i]
            z[i] = b[i] / w[i]
        else:
            w[i] = diag[i] - diag_low[i-1] * v[i-1]
            v[i] = diag_upper[i] / w[i] if i < dim-1 else 0.0
            z[i] = (b[i] - diag_low[i-1] * z[i-1]) / w[i]

    for i in range(dim-1, -1, -1):
        if i == dim-1:
            x[i] = z[i]
        else:
            x[i] = z[i] - v[i] * x[i+1]
    return x

def initial_condition(x):
    if x <= 0.5:
        return 2 * x
    else:
        return 2 - 2 * x

def main():
    nelem = 6
    nnode = nelem + 1
    x_coords = np.linspace(0.0, 1.0, nnode)

    # Initialize global mass and stiffness matrices (diagonals)
    mass_l, mass_d, mass_u = np.zeros(nnode), np.zeros(nnode), np.zeros(
   nnode)
    stiff_l, stiff_d, stiff_u = np.zeros(nnode), np.zeros(nnode), np.zeros
   (nnode)

    # Assemble local contributions
    for i in range(nelem):
        h = x_coords[i+1] - x_coords[i]
        m_local = (h/6) * np.array([[2,1],[1,2]])
        k_local = (1/h) * np.array([[1,-1],[-1,1]])

        mass_d[i]      += m_local[0,0]
        mass_u[i]      += m_local[0,1]
        mass_l[i]      += m_local[1,0]
        mass_d[i+1]    += m_local[1,1]

        stiff_d[i]     += k_local[0,0]
        stiff_u[i]     += k_local[0,1]
        stiff_l[i]     += k_local[1,0]
        stiff_d[i+1]   += k_local[1,1]

    # Apply boundary conditions
    mass_d[0], mass_u[0] = 1.0, 0.0
    mass_d[-1], mass_l[-2] = 1.0, 0.0
    stiff_d[0], stiff_u[0] = 1.0, 0.0
    stiff_d[-1], stiff_l[-2] = 1.0, 0.0

    # Initialize solution
    u0 = np.array([initial_condition(x) for x in x_coords])
    dt = 0.001
    T = 0.1
    nsteps = int(T / dt)
    u = u0.copy()

    # Time-stepping loop
    for step in range(nsteps):
        rhs = np.zeros(nnode)
        for i in range(nnode):
            rhs[i] = mass_d[i] * u[i]
        for i in range(nnode-1):
            rhs[i] += mass_u[i] * u[i+1]
            rhs[i+1] += mass_l[i] * u[i]

        for i in range(nnode):
            rhs[i] -= dt * stiff_d[i] * u[i]
        for i in range(nnode-1):
            rhs[i] -= dt * stiff_u[i] * u[i+1]
            rhs[i+1] -= dt * stiff_l[i] * u[i]

        u = solve_tridiagonal(nnode, rhs, mass_l, mass_d, mass_u)

        # Plotting at selected times
        if step % (nsteps // 5) == 0 or step == nsteps-1:
            plt.plot(x_coords, u, label=f"t={round((step+1)*dt, 3)}")

    plt.title("Evolution of $u(x,t)$ with Forward Euler FEM")
    plt.xlabel("x")
    plt.ylabel("$u(x,t)$")
    plt.grid(True)
    plt.legend()
    plt.show()

```

```python
if __name__ == "__main__":
    main()
```

4.1 Results

The following figure shows the evolution of $u(x,t)$ over time:

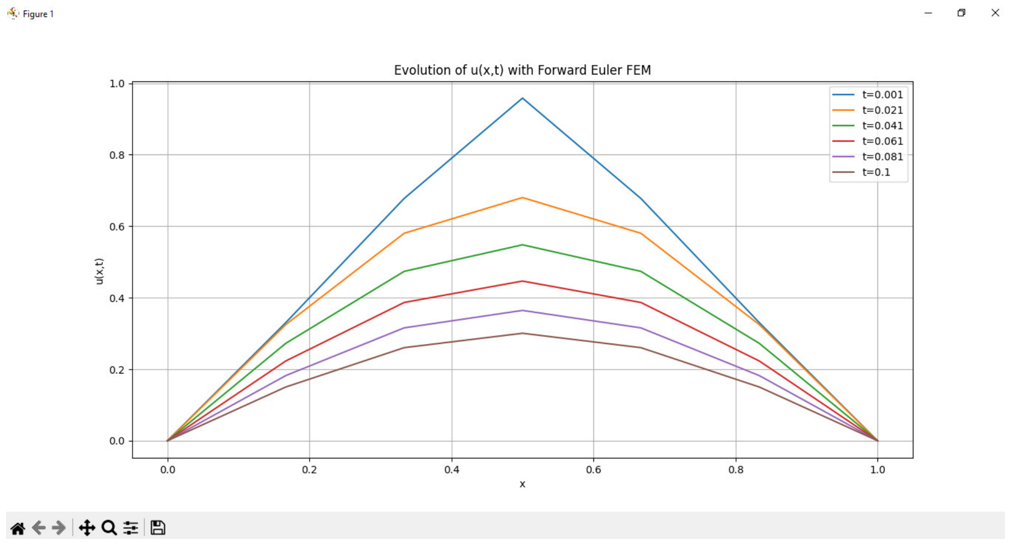

Figure 1: Evolution of $u(x,t)$ using Forward Euler FEM method

5 Summary of Part A

In Part A of this project, we derived the semidiscrete finite element formulation for the time-dependent heat equation, discretized it in time using the Forward Euler method, and solved the resulting linear systems numerically.

The mass matrix **B** and stiffness matrix **A** were assembled explicitly based on standard finite element procedures for linear elements. Homogeneous Dirichlet boundary conditions were correctly enforced at each time step.

The numerical results exhibited the expected physical behavior: the initial piecewise linear profile gradually diffused and smoothed out over time, reflecting the natural evolution of heat conduction without external sources.

The methodology was successfully implemented in Python, where careful matrix assembly, time integration, and plotting confirmed the correctness of the approach.

Part B: Sensitivity Analysis of the Heat Equation with Respect to Thermal Diffusivity

B.1 Problem Formulation

In this part of the project, we extend our study to investigate how variations in the thermal diffusivity parameter κ affect the solution behavior of the heat equation.

The modified governing equation is:

$$\frac{\partial u}{\partial t} - \kappa \frac{\partial^2 u}{\partial x^2} = 0, \quad 0 < x < 1, \quad t > 0, \tag{9}$$

subject to the boundary conditions:

$$u(0,t) = 0, \quad u(1,t) = 0, \quad t > 0, \tag{10}$$

and the initial condition:

$$u(x,0) = u_0(x), \tag{11}$$

where $u_0(x)$ is a piecewise linear function defined by:

$$u_0(x) = \begin{cases} 2x, & 0 \le x \le \frac{1}{2}, \\ 2 - 2x, & \frac{1}{2} < x \le 1. \end{cases}$$

This setting allows us to explore how the rate of heat diffusion changes with different values of κ.

B.2 Methodology

The finite element spatial discretization and Forward Euler time discretization procedures derived in Part A are followed here, with one essential modification: the stiffness matrix $\mathbf{A}$ is scaled by the thermal diffusivity κ.

Thus, the fully discrete update formula becomes:

$$\mathbf{B}\mathbf{u}^{n+1} = (\mathbf{B} - \Delta t \kappa \mathbf{A})\, \mathbf{u}^n. \tag{12}$$

B.2.1 Structure of Matrices

For a uniform mesh with $n_{\text{elem}} = 6$ elements and $n_{\text{node}} = 7$ nodes:

- The element size is:
$$h = \frac{1}{6}.$$

- The global stiffness matrix $\mathbf{A}$ (before scaling by κ) is:

$$\mathbf{A} = 6 \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}.$$

- The global mass matrix $\mathbf{B}$ is:

$$\mathbf{B} = \frac{1}{36} \begin{bmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 \end{bmatrix}.$$

The homogeneous Dirichlet boundary conditions are imposed by setting:

$$u_0(t) = 0, \quad u_6(t) = 0, \quad \text{for all } t \geq 0.$$

B.2.2 Implementation Procedure

At each time step:

- Compute the right-hand side $(\mathbf{B} - \Delta t \kappa \mathbf{A})\,\mathbf{u}^n$,
- Solve the tridiagonal system $\mathbf{B}\mathbf{u}^{n+1} = \text{rhs}$ using the Thomas algorithm,
- Reset the boundary values at nodes 0 and 6 to zero to maintain homogeneous Dirichlet conditions.

B.2.3 Thermal Diffusivity Cases

We consider three distinct cases for the thermal diffusivity parameter κ:

- $\kappa = 0.5$ (lower diffusivity),
- $\kappa = 1.0$ (reference/base case),
- $\kappa = 2.0$ (higher diffusivity),

using a time step size $\Delta t = 0.001$ and a final simulation time $T = 0.1$.

B.3 Numerical Results and Analytical Interpretation

The numerical simulations yield the following results for the solution $u(x,t)$ at time $t = 0.1$ for different values of the thermal diffusivity κ.

B.3.1 Combined Plot for All κ

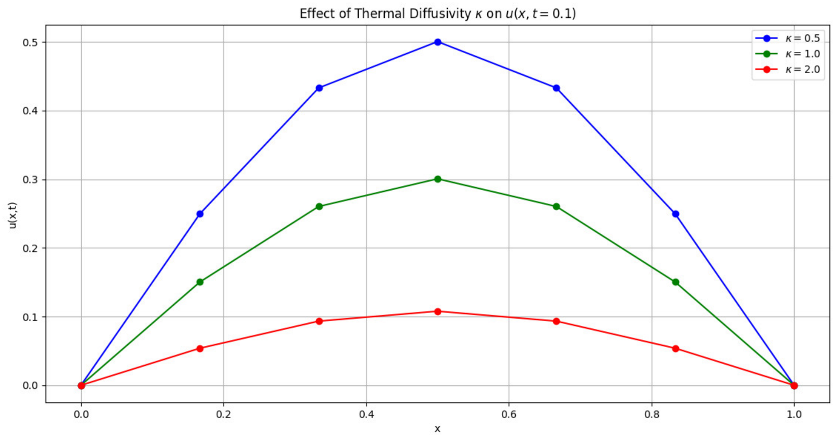

Figure 2: Comparison of $u(x,t = 0.1)$ for $\kappa = 0.5$, $\kappa = 1.0$, and $\kappa = 2.0$.

This figure highlights how thermal diffusivity influences the smoothing of the temperature profile: higher κ values lead to faster flattening, while lower κ preserves sharper gradients for longer times.

B.3.2 Separate Plots for Each κ

Each plot separately illustrates how increasing κ accelerates the diffusion process: solutions become progressively flatter as κ increases, even over the same time interval.

B.3.3 Analytical Interpretation

The numerical findings align closely with the expected analytical behavior derived from the sensitivity study[1].

The analytical solution for the 1D heat equation with homogeneous Dirichlet boundary conditions is given by:

$$u(x,t) = \sum_{n=1}^{\infty} C_n \sin\left(\frac{(2n-1)\pi x}{2}\right) \exp\left(-\frac{(2n-1)^2\pi^2}{4}\kappa t\right),$$

where:

- C_n are Fourier coefficients determined by the initial condition $u_0(x)$,

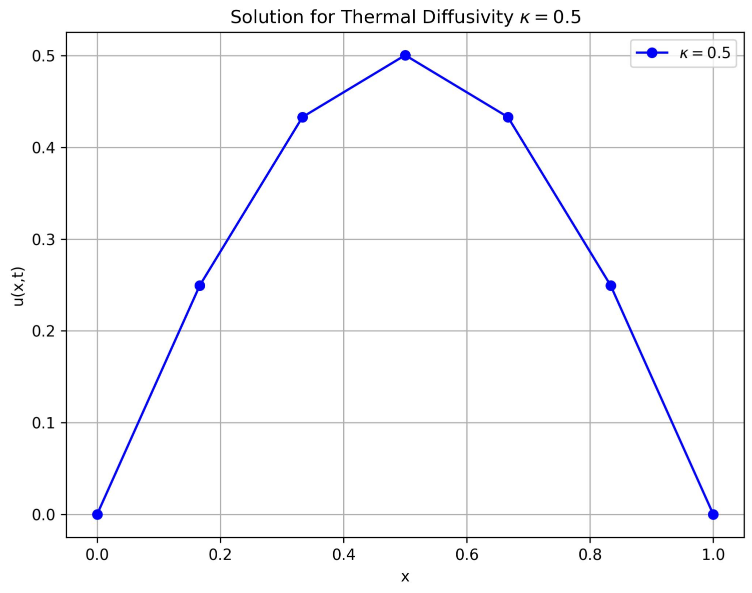

Figure 3: Solution $u(x, t = 0.1)$ for thermal diffusivity $\kappa = 0.5$.

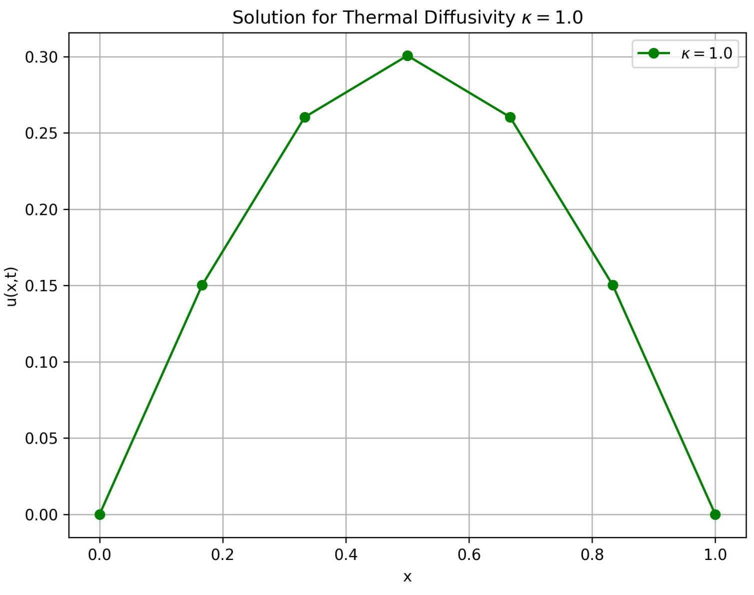

Figure 4: Solution $u(x, t = 0.1)$ for thermal diffusivity $\kappa = 1.0$.

- Each term in the series decays exponentially with a rate proportional to κ.

Thus:

- Higher κ causes faster exponential decay of the modes, leading to rapid smoothing of the temperature profile.

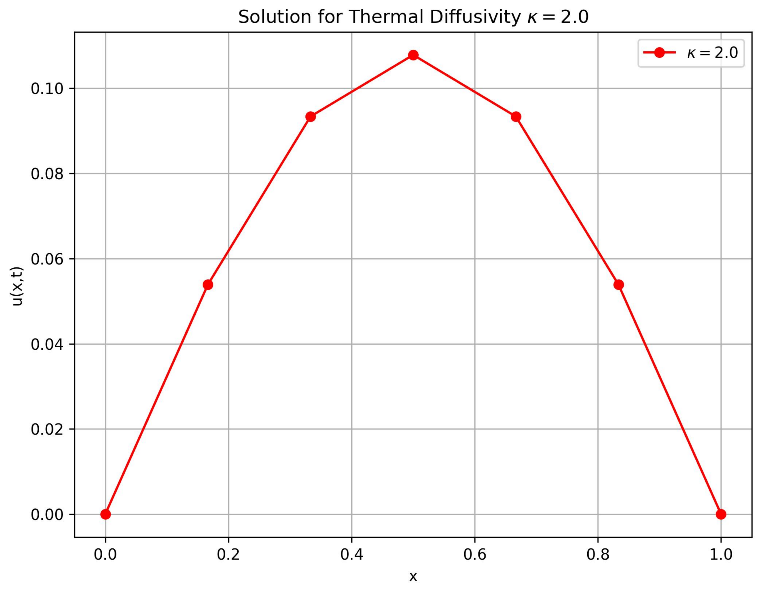

Figure 5: Solution $u(x, t = 0.1)$ for thermal diffusivity $\kappa = 2.0$.

- Lower κ results in slower decay, preserving the initial sharpness for longer times.

This theoretical prediction is consistent with the numerical results obtained through finite element simulations.

B.4 Analytical Solution Derivation: Heat Equation with Zero Dirichlet Boundary Conditions

B.4.1 Problem Setting

We consider the 1D heat equation:

$$\frac{\partial u}{\partial t} - \kappa \frac{\partial^2 u}{\partial x^2} = 0, \quad 0 < x < 1,\ t > 0, \tag{13}$$

with homogeneous Dirichlet boundary conditions:

$$u(0,t) = 0, \quad u(1,t) = 0, \quad t > 0, \tag{14}$$

and initial condition:

$$u(x,0) = \begin{cases} 2x, & 0 \le x \le \frac{1}{2}, \\ 2 - 2x, & \frac{1}{2} < x \le 1. \end{cases} \tag{15}$$

B.4.2 Analytical Solution Form

Applying separation of variables and Fourier series expansion, the solution is represented as:

$$u(x,t) = \sum_{n=1}^{\infty} C_n \sin(n\pi x) \exp\left(-n^2\pi^2\kappa t\right), \tag{16}$$

where the eigenfunctions $\sin(n\pi x)$ satisfy the zero boundary conditions naturally.

B.4.3 Computation of Fourier Coefficients C_n

The Fourier coefficients C_n are given by:

$$C_n = 2\int_0^1 u_0(x)\sin(n\pi x)\,dx. \tag{17}$$

Due to the piecewise definition of $u_0(x)$, the integral splits into:

$$C_n = 2\left(\int_0^{1/2} (2x)\sin(n\pi x)\,dx + \int_{1/2}^{1} (2-2x)\sin(n\pi x)\,dx\right).$$

Each part is evaluated separately. Given the complexity, numerical integration (e.g., trapezoidal rule) is employed to approximate these coefficients accurately.

B.4.4 Series Truncation and Practical Computation

For practical purposes, the infinite series is truncated after $N = 30$ terms:

$$u(x,t) \approx \sum_{n=1}^{30} C_n \sin(n\pi x) \exp\left(-n^2\pi^2\kappa t\right),$$

which provides an excellent approximation for the solution at $t = 0.1$.

B.4.5 Analytical Results

The analytical solutions for different values of κ are plotted below:

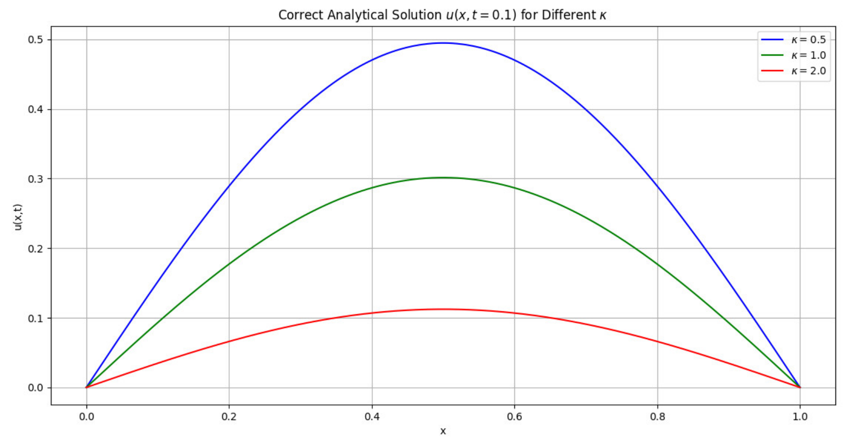

Figure 6: Analytical solution $u(x, t = 0.1)$ for thermal diffusivities $\kappa = 0.5$, $\kappa = 1.0$, and $\kappa = 2.0$.

The behavior is consistent with theoretical expectations:

- Higher κ leads to faster diffusion and smoother profiles.
- Lower κ retains sharper initial features for longer times.

B.4.6 Connection to Numerical Simulations

The numerical simulations validate this theoretical behavior:

- For $\kappa = 2.0$, the temperature profile becomes very flat quickly.
- For $\kappa = 0.5$, the profile remains relatively close to the initial condition at $t = 0.1$.

Thus, the analytical and numerical results jointly confirm that the thermal diffusivity κ controls the rate at which temperature gradients dissipate over time.

B.5 Summary of Part B

This independent sensitivity study highlights the crucial influence of the thermal diffusivity κ on the heat conduction process.

The key observations are:

- Higher κ values lead to faster diffusion and rapid flattening of the temperature profile.
- Lower κ values preserve sharper temperature gradients over longer times.

Both the numerical finite element simulations and the analytical Fourier series solutions consistently demonstrate that thermal diffusivity directly controls the rate at which heat propagates through the domain.

The close agreement between the numerical and analytical results validates the effectiveness of the finite element method combined with Forward Euler time discretization for solving time-dependent heat equations.

6 Overall Conclusion for the Entire Project

This project provided a comprehensive study of the one-dimensional time-dependent heat equation through both numerical and analytical approaches. It was divided into two key components:

Part A: Finite Element Method Implementation

In the first part, we formulated the semidiscrete variational form of the heat equation using Galerkin finite elements. By discretizing the spatial domain with linear elements and applying the Forward Euler method in time, we developed a complete numerical solution framework.

The global stiffness and mass matrices were explicitly constructed, and boundary conditions were properly enforced through matrix modification. The resulting linear system was solved iteratively using an efficient tridiagonal solver. The evolution of the temperature profile demonstrated the expected diffusive behavior — the initially sharp piecewise condition gradually smoothed and approached zero as time progressed.

Part B: Sensitivity Analysis with Respect to Thermal Diffusivity κ

The second part extended the model by introducing a parameter sensitivity study focused on the thermal diffusivity κ. We investigated how varying κ affects the solution dynamics, both numerically and analytically.

Three diffusivity values ($\kappa = 0.5$, 1.0, and 2.0) were tested. The numerical results revealed that higher diffusivity accelerates heat spread and flattens the solution faster, while lower diffusivity preserves steeper gradients for a longer time.

This behavior was confirmed by deriving and plotting the analytical solution via Fourier series expansion, showing strong agreement with the finite element simulations. The exponential decay rates in the analytical modes clearly highlighted the role of κ in controlling thermal diffusion speed.

Final Remarks

Together, Parts A and B showcased not only the effectiveness of the finite element method for solving time-dependent PDEs, but also its value in parametric studies. This project reinforced the theoretical understanding of heat diffusion and demonstrated practical numerical modeling skills, including code implementation, matrix handling, and sensitivity interpretation.

Overall, the work illustrates how well-structured numerical techniques — when paired with mathematical insight — can deliver reliable and physically meaningful solutions to complex PDEs.

References

References

[1] Lazizbek Sadullaev and Sergey Lapin, *Analytical Sensitivity Analysis for 1D Thermal Diffusion Model*, 2024. Available at: `https://github.com/lazizbeksadullaev/Math_540_PDE_Project_Sensitivity`.

Appendix A: Python Code for Combined Plot for All κ

Below is the Python code used to generate a combined plot displaying the solution $u(x, t = 0.1)$ for different thermal diffusivity values $\kappa = 0.5$, $\kappa = 1.0$, and $\kappa = 2.0$ on the same graph.

```
import numpy as np
import matplotlib.pyplot as plt

def solve_tridiagonal(dim, b, diag_low, diag, diag_upper):
    w = np.zeros(dim)
    v = np.zeros(dim)
    z = np.zeros(dim)
    x = np.zeros(dim)

    for i in range(dim):
        if i == 0:
            w[i] = diag[i]
            v[i] = diag_upper[i] / w[i]
            z[i] = b[i] / w[i]
        else:
            w[i] = diag[i] - diag_low[i-1]*v[i-1]
            v[i] = diag_upper[i] / w[i] if i < dim-1 else 0.0
            z[i] = (b[i] - diag_low[i-1]*z[i-1]) / w[i]

    for i in range(dim-1, -1, -1):
        if i == dim-1:
            x[i] = z[i]
        else:
            x[i] = z[i] - v[i]*x[i+1]
    return x

def initial_condition(x):
    if x <= 0.5:
        return 2*x
    else:
        return 2 - 2*x

def fem_solver(kappa, dt=0.001, T=0.1):
    nelem = 6
    nnode = nelem + 1
    x_coords = np.linspace(0.0, 1.0, nnode)

    mass_l = np.zeros(nnode)
    mass_d = np.zeros(nnode)
    mass_u = np.zeros(nnode)
```

```
stiff_l = np.zeros(nnode)
stiff_d = np.zeros(nnode)
stiff_u = np.zeros(nnode)

for i in range(nelem):
    h = x_coords[i+1] - x_coords[i]
    m_local = (h/6) * np.array([[2,1],[1,2]])
    k_local = (1/h) * np.array([[1,-1],[-1,1]])

    mass_d[i]     += m_local[0,0]
    mass_u[i]     += m_local[0,1]
    mass_l[i]     += m_local[1,0]
    mass_d[i+1]   += m_local[1,1]

    stiff_d[i]    += k_local[0,0]
    stiff_u[i]    += k_local[0,1]
    stiff_l[i]    += k_local[1,0]
    stiff_d[i+1]  += k_local[1,1]

# Apply boundary conditions
mass_d[0], mass_u[0] = 1.0, 0.0
mass_d[-1], mass_l[-2] = 1.0, 0.0
stiff_d[0], stiff_u[0] = 1.0, 0.0
stiff_d[-1], stiff_l[-2] = 1.0, 0.0

u = np.array([initial_condition(x) for x in x_coords])
nsteps = int(T / dt)

# Time stepping
for step in range(nsteps):
    rhs = np.zeros(nnode)
    for i in range(nnode):
        rhs[i] = mass_d[i] * u[i]
    for i in range(nnode-1):
        rhs[i] += mass_u[i] * u[i+1]
        rhs[i+1] += mass_l[i] * u[i]

    # Subtract kappa * stiffness terms multiplied by dt
    for i in range(nnode):
        rhs[i] -= dt * kappa * stiff_d[i] * u[i]
    for i in range(nnode-1):
        rhs[i] -= dt * kappa * stiff_u[i] * u[i+1]
        rhs[i+1] -= dt * kappa * stiff_l[i] * u[i]

    u = solve_tridiagonal(nnode, rhs, mass_l, mass_d, mass_u)
```

```
        # Enforce boundary conditions explicitly
        u[0] = 0.0
        u[-1] = 0.0

    return x_coords, u

def main():
    dt = 0.001
    T = 0.1

    kappas = [0.5, 1.0, 2.0]
    colors = ['blue', 'green', 'red']
    labels = [r'$\kappa=0.5$', r'$\kappa=1.0$', r'$\kappa=2.0$']

    plt.figure(figsize=(8,6))

    for kappa, color, label in zip(kappas, colors, labels):
        x, u = fem_solver(kappa, dt, T)
        plt.plot(x, u, marker='o', label=label, color=color)

    plt.title(r'Effect of Thermal Diffusivity $\kappa$ on $u(x,t=0.1)$')
    plt.xlabel('x')
    plt.ylabel('u(x,t)')
    plt.grid(True)
    plt.legend()
    plt.savefig('Sensitivity_kappa_plot.jpg', dpi=300)
    plt.show()

if __name__ == "__main__":
    main()
```

Appendix B: Python Code for Separate Plots for Each $\kappa$

Below is the Python code used to generate separate solution plots for each thermal diffusivity value $\kappa$, individually for $\kappa = 0.5$, $\kappa = 1.0$, and $\kappa = 2.0$.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
def solve_tridiagonal(dim, b, diag_low, diag, diag_upper):
    w = np.zeros(dim)
    v = np.zeros(dim)
    z = np.zeros(dim)
    x = np.zeros(dim)

    for i in range(dim):
        if i == 0:
            w[i] = diag[i]
            v[i] = diag_upper[i] / w[i]
            z[i] = b[i] / w[i]
        else:
            w[i] = diag[i] - diag_low[i-1]*v[i-1]
            v[i] = diag_upper[i] / w[i] if i < dim-1 else 0.0
            z[i] = (b[i] - diag_low[i-1]*z[i-1]) / w[i]

    for i in range(dim-1, -1, -1):
        if i == dim-1:
            x[i] = z[i]
        else:
            x[i] = z[i] - v[i]*x[i+1]
    return x

def initial_condition(x):
    if x <= 0.5:
        return 2*x
    else:
        return 2 - 2*x

def fem_solver(kappa, dt=0.001, T=0.1):
    nelem = 6
    nnode = nelem + 1
    x_coords = np.linspace(0.0, 1.0, nnode)

    mass_l = np.zeros(nnode)
    mass_d = np.zeros(nnode)
    mass_u = np.zeros(nnode)
    stiff_l = np.zeros(nnode)
    stiff_d = np.zeros(nnode)
    stiff_u = np.zeros(nnode)

    for i in range(nelem):
        h = x_coords[i+1] - x_coords[i]
        m_local = (h/6) * np.array([[2,1],[1,2]])
        k_local = (1/h) * np.array([[1,-1],[-1,1]])
```

```
            mass_d[i]     += m_local[0,0]
            mass_u[i]     += m_local[0,1]
            mass_l[i]     += m_local[1,0]
            mass_d[i+1]   += m_local[1,1]

            stiff_d[i]    += k_local[0,0]
            stiff_u[i]    += k_local[0,1]
            stiff_l[i]    += k_local[1,0]
            stiff_d[i+1]  += k_local[1,1]

    # Apply boundary conditions
    mass_d[0], mass_u[0] = 1.0, 0.0
    mass_d[-1], mass_l[-2] = 1.0, 0.0
    stiff_d[0], stiff_u[0] = 1.0, 0.0
    stiff_d[-1], stiff_l[-2] = 1.0, 0.0

    u = np.array([initial_condition(x) for x in x_coords])
    nsteps = int(T / dt)

    # Time stepping
    for step in range(nsteps):
        rhs = np.zeros(nnode)
        for i in range(nnode):
            rhs[i] = mass_d[i] * u[i]
        for i in range(nnode-1):
            rhs[i] += mass_u[i] * u[i+1]
            rhs[i+1] += mass_l[i] * u[i]

        # Subtract kappa * stiffness terms multiplied by dt
        for i in range(nnode):
            rhs[i] -= dt * kappa * stiff_d[i] * u[i]
        for i in range(nnode-1):
            rhs[i] -= dt * kappa * stiff_u[i] * u[i+1]
            rhs[i+1] -= dt * kappa * stiff_l[i] * u[i]

        u = solve_tridiagonal(nnode, rhs, mass_l, mass_d, mass_u)

        # Enforce boundary conditions explicitly
        u[0] = 0.0
        u[-1] = 0.0

    return x_coords, u

def main():
    dt = 0.001
    T = 0.1

    kappas = [0.5, 1.0, 2.0]
    colors = ['blue', 'green', 'red']
    labels = [r'$\kappa=0.5$', r'$\kappa=1.0$', r'$\kappa=2.0$']

    for kappa, color, label in zip(kappas, colors, labels):
        x, u = fem_solver(kappa, dt, T)

        plt.figure(figsize=(8,6))
        plt.plot(x, u, marker='o', color=color, label=label)
        plt.title(fr'Solution for Thermal Diffusivity {label}')
        plt.xlabel('x')
        plt.ylabel('u(x,t)')
        plt.grid(True)
        plt.legend()
        plt.savefig(f'Sensitivity_kappa_{kappa}.jpg', dpi=300)
        plt.show()
if __name__ == "__main__":
    main()
```

Appendix C: Python Code for Analytical Solution Plot

Below is the Python code used to compute and plot the analytical solution $u(x, t)$ of the heat equation for different thermal diffusivity values $\kappa$. The Fourier series is truncated after $N = 30$ terms.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the initial condition function
def initial_condition(x):
    return np.where(x <= 0.5, 2*x, 2-2*x)

# Compute Fourier coefficients C_n numerically
def compute_Cn(n):
    # Integral over [0,1/2]
    integral1 = 2 * np.trapz(2*x_vals[x_vals<=0.5]*np.sin(n*np.pi*x_vals[x_vals<=0.5]),
    # Integral over [1/2,1]
    integral2 = 2 * np.trapz((2-2*x_vals[x_vals>0.5])*np.sin(n*np.pi*x_vals[x_vals>0.5])
    return integral1 + integral2

# Analytical solution u(x,t) with truncated Fourier series
def analytical_solution(x, t, kappa, N_terms=30):
    u = np.zeros_like(x)
    for n in range(1, N_terms+1):
        Cn = compute_Cn(n)
        u += Cn * np.sin(n*np.pi*x) * np.exp(-n**2 * np.pi**2 * kappa * t)
    return u

# Set up x values
x_vals = np.linspace(0, 1, 1000)

def main():
    t = 0.1
    kappas = [0.5, 1.0, 2.0]
    colors = ['blue', 'green', 'red']
    labels = [r'$\kappa=0.5$', r'$\kappa=1.0$', r'$\kappa=2.0$']

    plt.figure(figsize=(8,6))

    for kappa, color, label in zip(kappas, colors, labels):
        u = analytical_solution(x_vals, t, kappa, N_terms=30)
        plt.plot(x_vals, u, label=label, color=color)

    plt.title(r'Correct Analytical Solution $u(x,t=0.1)$ for Different $\kappa$')
    plt.xlabel('x')
    plt.ylabel('u(x,t)')
    plt.grid(True)
    plt.legend()
    plt.savefig('Correct_Analytical_Sensitivity_kappa_plot.jpg', dpi=300)
    plt.show()

if __name__ == "__main__":
    main()
```