Homework 4

Due Date: November 21, 2025

2D Heat Equation with Finite Element Method

Consider the time-dependent heat equation as follows:

$$\frac{\partial u}{\partial t} - \nu \Delta u = f(x, y, t) \quad \text{in} \quad \Omega = (-2, 2) \times (-2, 2), \quad t \in (0, 1]$$

with diffusion coefficient $\nu = 0.05$ and corresponding homogeneous Dirichlet boundary conditions:

$$u(x, y, \cdot) = 0 \quad \text{for} \quad (x, y) \in \partial\Omega$$

We assume the exact solution is given by:

$$u_{exact}(x, y, t) = e^{-8\pi^2 \nu t} \sin(2\pi x) \sin(2\pi y)$$

We will be using rectangular elements. You will use the bilinear Q_1 element as your basis function to solve the heat equation. The corresponding four shape functions defined in the reference element $(\xi, \eta) \in (-1, 1) \times (-1, 1)$ are:

$$\Phi_1(\xi, \eta) = \frac{1}{4}(1 - \xi)(1 - \eta), \quad \Phi_2(\xi, \eta) = \frac{1}{4}(1 + \xi)(1 - \eta),$$

$$\Phi_3(\xi, \eta) = \frac{1}{4}(1 + \xi)(1 + \eta), \quad \Phi_4(\xi, \eta) = \frac{1}{4}(1 - \xi)(1 + \eta)$$

Preliminary Setup

This is a general outline of the finite element method for solving the 2D heat equation. The specifics for our problem will be addressed in the subsequent questions.

Weak Formulation

Let v be a test function belonging to the function space:

$$V = \{v \in H_0^1(\Omega) \mid v, v' \in L^2(\Omega)\}$$

Note that $v = 0$ on $\partial\Omega$. Multiplying the PDE by v and integrating over the domain Ω, we have:

$$\int_\Omega u_t v - \nu v \Delta u \ dx = \int_\Omega f v \ dx$$

Integrating by parts on the second term of the left-hand side:

$$\int_\Omega u_t v \ dx + \nu[\int_\Omega \nabla u \cdot \nabla v \ dx - \int_{\partial\Omega} v(\nabla u \cdot n) \ ds] = \int_\Omega f v \ dx$$

Since $v = 0$ on $\partial\Omega$, the boundary integral vanishes. Therefore, the weak formulation is given by:

$$\int_\Omega u_t v \ dx + \nu \int_\Omega \nabla u \cdot \nabla v \ dx = \int_\Omega f v \ dx$$

Where $u, v \in V$. Denote the left hand side as $a(u, v)$ and the right hand side as $L(v)$.

Discretization and Global System

We discretize the spatial domain Ω into rectangular elements. Let $V_h \subset V$ be the finite-dimensional subspace spanned by the basis functions $\{\phi_i\}_{i=1}^N$, where N is the total number of nodes in the mesh (will be reduced later on based on BC). Each bilinear ϕ_n corresponds to some node (x_i, y_j) satisfying $\phi_{i,j} = \delta_{i,j}$. We assume $u_h \in V_h$ satisfies the weak formulation $a(u_h, v_h) = L(v_h)$ for all $v_h \in V_h$.

Approximate the solution of u as:

$$u_h = \sum_{j=1}^N U_j(t)\phi_j(x, y)$$

where U_j are time-dependent coefficients to be determined. The test function v is also chosen from the same space and discretized similarly:

$$v_h = \sum_{i=1}^N V_i(t)\phi_i(x, y)$$

Substituting these approximations into the weak formulation, we obtain the system:

$$\sum_{i,j=1}^N V_j \left(\int_\Omega \phi_i\phi_j \ dx\right) \frac{dU_i}{dt} + \nu \sum_{i,j=1}^N V_j \left(\int_\Omega \nabla\phi_i \cdot \nabla\phi_j \ dx\right) U_i = \sum_{j=1}^N V_j \int_\Omega f\phi_j \ dx$$

We may factor out the V_j to give us:

$$\sum_{i,j=1}^N \left(\int_\Omega \phi_i\phi_j \ dx\right) \frac{dU_i}{dt} + \nu \sum_{i,j=1}^N \left(\int_\Omega \nabla\phi_i \cdot \nabla\phi_j \ dx\right) U_i = \sum_{j=1}^N \int_\Omega f\phi_j \ dx$$

More compactly, let:

$$U = [U_1, U_2, \ldots, U_N]^T, \quad M_{ij} = \int_\Omega \phi_i \phi_j \, dx, \quad K_{ij} = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, dx, \quad F_j = \int_\Omega f\phi_j \, dx$$

such that $M = (M_{ij})$, $K = (K_{ij})$, and $F = [F_1, F_2, \ldots, F_N]^T$. We can rewrite the system as:

$$M\frac{dU}{dt} + \nu KU = F$$

where M is the mass matrix, K is the stiffness matrix, and F is the force vector.

Elemental-Level Systems and Assembly

We suppose element-wise, each $u^{(e)}$ satisfies the weak formulation over its own domain $\Omega^{(e)}$:

$$\int_{\Omega^{(e)}} u_t^{(e)} v^{(e)} \, dx + \nu \int_{\Omega^{(e)}} \nabla u^{(e)} \cdot \nabla v^{(e)} \, dx = \int_{\Omega^{(e)}} f v^{(e)} \, dx$$

We suppose the local approximations are given by:

$$u_h^{(e)} = \sum_{j=1}^{4} U_j^{(e)} \phi_j^{(e)}(x,y), \quad v_h^{(e)} = \sum_{i=1}^{4} V_i^{(e)} \phi_i^{(e)}(x,y)$$

since we have rectangular elements with four nodes each. Using the same process as before, we can derive the elemental system:

$$M^{(e)}\frac{dU^{(e)}}{dt} + \nu K^{(e)} U^{(e)} = F^{(e)}$$

We can express the global matrices and vector as sums over all elements:

$$M = \sum_{e=1}^{E} M^{(e)}, \quad K = \sum_{e=1}^{E} K^{(e)}, \quad F = \sum_{e=1}^{E} F^{(e)}$$

where E is the total number of elements, and the elemental matrices and vector are defined as:

$$M_{ij}^{(e)} = \int_{\Omega^{(e)}} \phi_i^{(e)} \phi_j^{(e)} \, dx, \quad K_{ij}^{(e)} = \int_{\Omega^{(e)}} \nabla\phi_i^{(e)} \cdot \nabla\phi_j^{(e)} \, dx, \quad F_j^{(e)} = \int_{\Omega^{(e)}} f\phi_j^{(e)} \, dx$$

Here, $\Omega^{(e)}$ is the domain of element e, and $\phi_i^{(e)}$ are the local shape functions associated with element e.

We can use quadrature to numerically compute the integrals for $M^{(e)}$, $K^{(e)}$, and $F^{(e)}$ on each element, then assemble them into the global system.

Question 1

By substituting u_{exact} into the PDE, determine the forcing term $f(x, y, t)$ such that:

$$\frac{\partial u_{exact}}{\partial t} - \nu \Delta u_{exact} = f(x, y, t)$$

Solution. For the purposes of this question, denote $u = u_{exact}$. Where u is the exact solution given by:

$$u(x, y, t) = e^{-8\pi^2 \nu t} \sin(2\pi x) \sin(2\pi y)$$

Let us first compute the time derivative:

$$\frac{\partial u}{\partial t} = -8\pi^2 \nu e^{-8\pi^2 \nu t} \sin(2\pi x) \sin(2\pi y)$$

Next, we want to find the Laplacian. Computing the first and second derivative with respect to x:

$$\frac{\partial u}{\partial x} = 2\pi e^{-8\pi^2 \nu t} \cos(2\pi x) \sin(2\pi y)$$

$$\frac{\partial^2 u}{\partial x^2} = -4\pi^2 e^{-8\pi^2 \nu t} \sin(2\pi x) \sin(2\pi y)$$

The 2nd derivative with respect to y is the same:

$$\frac{\partial^2 u}{\partial y^2} = -4\pi^2 e^{-8\pi^2 \nu t} \sin(2\pi x) \sin(2\pi y)$$

Therefore, the Laplacian is:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -8\pi^2 e^{-8\pi^2 \nu t} \sin(2\pi x) \sin(2\pi y)$$

Substituting these results into the heat equation, we have:

$$\frac{\partial u}{\partial t} - \nu \Delta u = -8\pi^2 \nu e^{-8\pi^2 \nu t} \sin(2\pi x) \sin(2\pi y) - \nu(-8\pi^2 e^{-8\pi^2 \nu t} \sin(2\pi x) \sin(2\pi y)) = 0$$

So our forcing term is:

$$f(x, y, t) = 0$$

and we are working with the homogeneous heat equation.

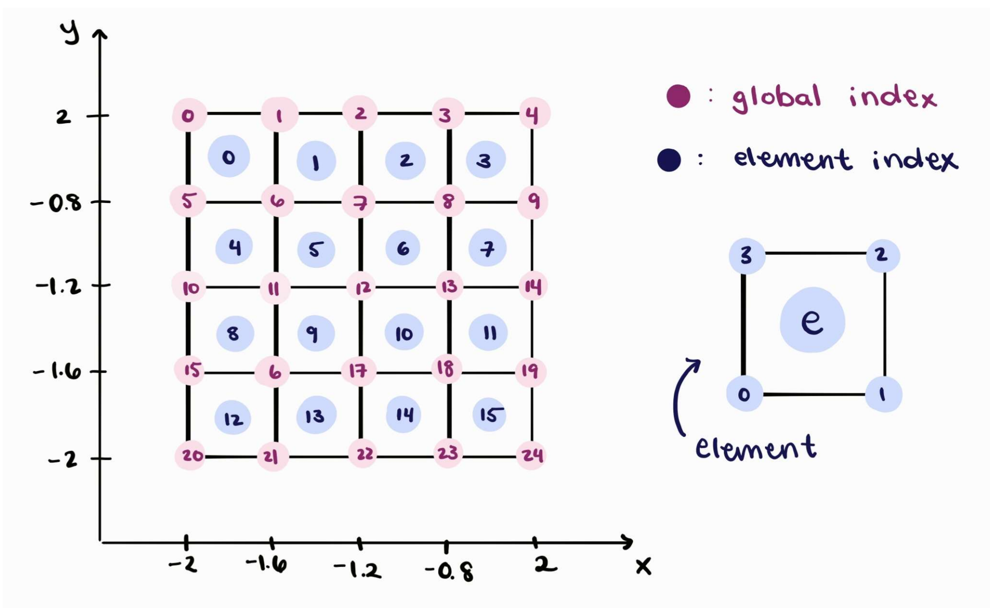
Question 2

Discretize the spacial domain Ω into 16 square elements arranged in a 4×4 grid, with the node coordinates:

$$(x, y) \in \{-2, -1.6, -1.2, -0.8, 2\} \times \{-2, -1.6, -1.2, -0.8, 2\}$$

Draw this mesh, define your own global numbering, label all global node numbers, and generate corresponding elemental connectivities.

Solution. All indexing used will start from 0. The mesh is as follows:



Globally, we have 25 nodes numbered from 0 to 24. Numbering starts from the top-left corner and goes row-wise. The elements are numbered from 0 to 15, also row-wise in the same fashion. For each element, its indicies $(0, 1, 2, 3)$ start from the bottom-left and go counter-clockwise to match the definition of the bilinear $Q1$ element. The folowing code was used to generate the connectivity matrix:

Listing 1: Question 2 Code

```
def global_indexing(width, height=None):
    if height is None:
        height = width
    return np.arange(width * height).reshape((height, width))

def generate_connectivity_matrix(global_indices):
    total_elements = (global_indices.shape[0] - 1) * (global_indices.shape[1] - 1)
    connectivity_matrix = np.zeros((total_elements, 4), dtype=int)
    element = 0
    for i in range(global_indices.shape[0] - 1):
```

```
        for j in range(global_indices.shape[1] - 1):
            # Element node ordering: bottom-left, bottom-right,
            #top-right, top-left
            connectivity_matrix[element, 0] = global_indices[i+1, j]
            connectivity_matrix[element, 1] = global_indices[i+1, j+1]
            connectivity_matrix[element, 2] = global_indices[i, j+1]
            connectivity_matrix[element, 3] = global_indices[i, j]
            element += 1
    return connectivity_matrix

if __name__ == "__main__":
    width = 5  # Number of nodes along one dimension
    global_indices = global_indexing(width)
    connectivity_matrix = generate_connectivity_matrix(global_indices)

    with open("./outputs_4/matrices.txt", "w") as f:
        # print(connectivity_matrix)
        latex_matrix = sp.latex(sp.Matrix(connectivity_matrix))
        f.write("Connectivity Matrix:\n")
        f.write(latex_matrix + "\n\n")
```

The elemental connectivities are as follows:

Element	Node 0	Node 1	Node 2	Node 3
0	5	6	1	0
1	6	7	2	1
2	7	8	3	2
3	8	9	4	3
4	10	11	6	5
5	11	12	7	6
6	12	13	8	7
7	13	14	9	8
8	15	16	11	10
9	16	17	12	11
10	17	18	13	12
11	18	19	14	13
12	20	21	16	15
13	21	22	17	16
14	22	23	18	17
15	23	24	19	18

Question 3

For one physical element $u^{(e)}$, write the mapping from the reference element $(\xi, \eta) \in (-1, 1) \times (-1, 1)$ to the physical coordinates (x, y) in terms of the nodal coordinates (x_n, y_n) and the shape functions $\Phi_n(\xi, \eta)$. Also, derive the Jacobian matrix $J(\xi, \eta)$ of this mapping.

Solution. Reindexing the four shape functions to fit our indexing scheme for the element nodes, we have:

$$\Phi_0(\xi, \eta) = \frac{1}{4}(1-\xi)(1-\eta), \quad \Phi_1(\xi, \eta) = \frac{1}{4}(1+\xi)(1-\eta),$$
$$\Phi_2(\xi, \eta) = \frac{1}{4}(1+\xi)(1+\eta), \quad \Phi_3(\xi, \eta) = \frac{1}{4}(1-\xi)(1+\eta)$$

On a physical element e, it is a rectangle of the region $[x_0, x_1] \times [y_0, y_1]$ where (x_0, y_0) is the bottom-left corner and (x_1, y_1) is the top-right corner. The mapping from $(\xi, \eta) \mapsto (x, y)$ should be given by the standard change of variables:

$$\begin{bmatrix} x(\xi, \eta) \\ y(\xi, \eta) \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(x_1 - x_0)\xi + \frac{1}{2}(x_1 + x_0) \\ \frac{1}{2}(y_1 - y_0)\eta + \frac{1}{2}(y_1 + y_0) \end{bmatrix} \tag{1}$$

We will verify this using the shape functions. We assume that:

$$x(\xi, \eta) = \sum_{n=0}^{3} x_n^{(e)} \Phi_n^{(e)}(\xi, \eta), \quad y(\xi, \eta) = \sum_{n=0}^{3} y_n^{(e)} \Phi_n^{(e)}(\xi, \eta)$$

where $(x_n^{(e)}, y_n^{(e)})$ are the nodal coordinates of element e. Note that by our indexing scheme:

$$(x_0^{(e)}, y_0^{(e)}) = (x_0, y_0), \quad (x_1^{(e)}, y_1^{(e)}) = (x_1, y_0),$$
$$(x_2^{(e)}, y_2^{(e)}) = (x_1, y_1), \quad (x_3^{(e)}, y_3^{(e)}) = (x_0, y_1)$$

Expanding $x(\xi, \eta)$:

$$\begin{aligned} x(\xi, \eta) &= x_0^{(e)}\Phi_0^{(e)} + x_1^{(e)}\Phi_1^{(e)} + x_2^{(e)}\Phi_2^{(e)} + x_3^{(e)}\Phi_3^{(e)} \\ &= x_0\frac{1}{4}(1-\xi)(1-\eta) + x_1\frac{1}{4}(1+\xi)(1-\eta) + x_1\frac{1}{4}(1+\xi)(1+\eta) + x_0\frac{1}{4}(1-\xi)(1+\eta) \\ &= \frac{1}{4}\left[x_0(1-\xi)(1-\eta+1+\eta) + x_1(1+\xi)(1-\eta+1+\eta)\right] \\ &= \frac{1}{4}\left[2x_0(1-\xi) + 2x_1(1+\xi)\right] \\ &= \frac{x_1 - x_0}{2}\xi + \frac{x_1 + x_0}{2} \end{aligned}$$

Similarly for $y(\xi, \eta)$, we get:

$$y(\xi, \eta) = \frac{y_1 - y_0}{2}\eta + \frac{y_1 + y_0}{2}$$

Therefore, the mapping from reference to physical coordinates is given by equation 1

Now we can derive the Jacobian matrix, $J(\xi, \eta)$, of this mapping, which is defined as:

$$J(\xi, \eta) = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

Let us compute each partial derivative:

$$\frac{\partial x}{\partial \xi} = \frac{x_1 - x_0}{2}, \quad \frac{\partial x}{\partial \eta} = 0$$

$$\frac{\partial y}{\partial \xi} = 0, \quad \frac{\partial y}{\partial \eta} = \frac{y_1 - y_0}{2}$$

Therefore, our Jacobian is:

$$J(\xi, \eta) = \begin{bmatrix} \frac{x_1 - x_0}{2} & 0 \\ 0 & \frac{y_1 - y_0}{2} \end{bmatrix}$$

Note that if our elements were all equally sized, the Jacobian would be identical for all elements.

Question 4

Using the basis functions from the reference element, compute the following derivatives:

$$\frac{\partial \Phi_i}{\partial \xi}, \frac{\partial \Phi_i}{\partial \eta} \quad \text{for} \quad i = 0, 1, 2, 3$$

Then express the physical gradients $\nabla \Phi_i = \left[\frac{\partial \Phi_i}{\partial x}, \frac{\partial \Phi_i}{\partial y}\right]^T$ using the Jacobian.

Solution. Note that:

$$\Phi_i(x, y) = \Phi_i(\xi(x, y), \eta(x, y))$$

We know that:

$$\nabla \Phi_i(\xi, \eta) = J(\xi, \eta)\ \nabla \Phi_i(x, y)$$

where J is the Jacobian matrix derived in the previous question. Therefore, we can express the physical gradients as:

$$\nabla \Phi_i(x, y) = J^{-1}(\xi, \eta)\ \nabla \Phi_i(\xi, \eta)$$

Given the Jacobian from before:

$$J(\xi, \eta) = \begin{bmatrix} \frac{x_1 - x_0}{2} & 0 \\ 0 & \frac{y_1 - y_0}{2} \end{bmatrix}$$

Its inverse is given by:

$$J^{-1}(x, y) = \begin{bmatrix} \frac{2}{x_1 - x_0} & 0 \\ 0 & \frac{2}{y_1 - y_0} \end{bmatrix}$$

Then for each $\Phi_i(x, y)$, we have:

$$\begin{aligned}
\nabla \Phi_i(x, y) &= J^{-1}(\xi, \eta)\ \nabla \Phi_i(\xi, \eta) \\
&= \begin{bmatrix} \frac{2}{x_1 - x_0} & 0 \\ 0 & \frac{2}{y_1 - y_0} \end{bmatrix} \begin{bmatrix} \frac{\partial \Phi_i}{\partial \xi} \\ \frac{\partial \Phi_i}{\partial \eta} \end{bmatrix} \\
&= \begin{bmatrix} \frac{2}{x_1 - x_0} & 0 \\ 0 & \frac{2}{y_1 - y_0} \end{bmatrix} \begin{bmatrix} \frac{\partial \Phi_i}{\partial \xi} \\ \frac{\partial \Phi_i}{\partial \eta} \end{bmatrix} \\
&= \begin{bmatrix} \frac{2}{x_1 - x_0} \frac{\partial \Phi_i}{\partial \xi} \\ \frac{2}{y_1 - y_0} \frac{\partial \Phi_i}{\partial \eta} \end{bmatrix} \\
&= 2 \begin{bmatrix} \frac{1}{x_1 - x_0} \frac{\partial \Phi_i}{\partial \xi} \\ \frac{1}{y_1 - y_0} \frac{\partial \Phi_i}{\partial \eta} \end{bmatrix}
\end{aligned}$$

Let us first calculate $\nabla \Phi_i(\xi, \eta)$. For reference, the shape functions are:

$$\Phi_0(\xi, \eta) = \frac{1}{4}(1 - \xi)(1 - \eta), \quad \Phi_1(\xi, \eta) = \frac{1}{4}(1 + \xi)(1 - \eta),$$

$$\Phi_2(\xi,\eta) = \frac{1}{4}(1+\xi)(1+\eta), \quad \Phi_3(\xi,\eta) = \frac{1}{4}(1-\xi)(1+\eta)$$

Starting with derivatives with respect to ξ:

$$\frac{\partial \Phi_0}{\partial \xi} = -\frac{1}{4}(1-\eta), \quad \frac{\partial \Phi_1}{\partial \xi} = \frac{1}{4}(1-\eta),$$

$$\frac{\partial \Phi_2}{\partial \xi} = \frac{1}{4}(1+\eta), \quad \frac{\partial \Phi_3}{\partial \xi} = -\frac{1}{4}(1+\eta)$$

Then for derivatives with respect to η:

$$\frac{\partial \Phi_0}{\partial \eta} = -\frac{1}{4}(1-\xi), \quad \frac{\partial \Phi_1}{\partial \eta} = -\frac{1}{4}(1+\xi),$$

$$\frac{\partial \Phi_2}{\partial \eta} = \frac{1}{4}(1+\xi), \quad \frac{\partial \Phi_3}{\partial \eta} = \frac{1}{4}(1-\xi)$$

Therefore, the gradients in reference coordinates are:

$$\nabla\Phi_0(\xi,\eta) = \frac{1}{4}\begin{bmatrix} -(1-\eta) \\ -(1-\xi) \end{bmatrix}, \quad \nabla\Phi_1(\xi,\eta) = \frac{1}{4}\begin{bmatrix} (1-\eta) \\ -(1+\xi) \end{bmatrix}$$

$$\nabla\Phi_2(\xi,\eta) = \frac{1}{4}\begin{bmatrix} (1+\eta) \\ (1+\xi) \end{bmatrix}, \quad \nabla\Phi_3(\xi,\eta) = \frac{1}{4}\begin{bmatrix} -(1+\eta) \\ (1-\xi) \end{bmatrix}$$

So we have the physical gradients:

$$\nabla\Phi_0(x,y) = \frac{1}{2}\begin{bmatrix} -\frac{1}{x_1-x_0}(1-\eta) \\ -\frac{1}{y_1-y_0}(1-\xi) \end{bmatrix}, \quad \nabla\Phi_1(x,y) = \frac{1}{2}\begin{bmatrix} \frac{1}{x_1-x_0}(1-\eta) \\ -\frac{1}{y_1-y_0}(1+\xi) \end{bmatrix}$$

$$\nabla\Phi_2(x,y) = \frac{1}{2}\begin{bmatrix} \frac{1}{x_1-x_0}(1+\eta) \\ \frac{1}{y_1-y_0}(1+\xi) \end{bmatrix}, \quad \nabla\Phi_3(x,y) = \frac{1}{2}\begin{bmatrix} -\frac{1}{x_1-x_0}(1+\eta) \\ \frac{1}{y_1-y_0}(1-\xi) \end{bmatrix}$$

Question 5

Use the following formulas for the elemental mass matrix $M^{(e)}$, and stiffness matrix $K^{(e)}$:

$$M_{ij}^{(e)} = \int_{\Omega^{(e)}} \Phi_i^{(e)} \Phi_j^{(e)} \, dx, \quad K_{ij}^{(e)} = \int_{\Omega^{(e)}} \nabla \Phi_i^{(e)} \cdot \nabla \Phi_j^{(e)} \, dx$$

to evaluate these integrals explicitly for an arbitrary square element in this mesh. Your $M^{(e)}$ and $K^{(e)}$ should be 4×4 matrices.

Solution. For ease of notation let $\Phi_i^{(e)} = \Phi_i$.

Let us denote $\Phi = [\Phi_0, \Phi_1, \Phi_2, \Phi_3]^T$ as the vector of shape functions for element e. Note that $M^{(e)}$ can also be expressed a $M^{(e)} = \int_{\Omega_e} \Phi \cdot \Phi^T \, dx$. Using the change of variables from physical to reference coordinates, we have:

$$M^{(e)} = \int_{-1}^{1} \int_{-1}^{1} \Phi \cdot \Phi^T |\det J(\xi, \eta)| \, d\xi d\eta$$

where $|\det J(\xi, \eta)|$ is the absolute value of the determinant of our Jacobian. Let the width of the node be $w = (x_1 - x_0)$ and the height be $h = (y_2 - y_1)$. From question 5, we have:

$$|\det J(\xi, \eta)| = \left| \frac{w}{2} \cdot \frac{h}{2} \right| = \frac{|w| \cdot |h|}{4}$$

When corrected for our local indexing scheme.

Since this constant, we can factor it out of the integral:

$$M^{(e)} = \frac{|w| \cdot |h|}{4} \int_{-1}^{1} \int_{-1}^{1} \Phi \cdot \Phi^T \, d\xi d\eta$$

Note that:

$$\Phi \cdot \Phi^T = \begin{bmatrix} \Phi_0 \\ \Phi_1 \\ \Phi_2 \\ \Phi_3 \end{bmatrix} \begin{bmatrix} \Phi_0 & \Phi_1 & \Phi_2 & \Phi_3 \end{bmatrix}$$

$$= \begin{bmatrix} \Phi_0\Phi_0 & \Phi_0\Phi_1 & \Phi_0\Phi_2 & \Phi_0\Phi_3 \\ \Phi_1\Phi_0 & \Phi_1\Phi_1 & \Phi_1\Phi_2 & \Phi_1\Phi_3 \\ \Phi_2\Phi_0 & \Phi_2\Phi_1 & \Phi_2\Phi_2 & \Phi_2\Phi_3 \\ \Phi_3\Phi_0 & \Phi_3\Phi_1 & \Phi_3\Phi_2 & \Phi_3\Phi_3 \end{bmatrix}$$

We will compute some auxiliary integrals with dummy variables first:

$$\int_{-1}^{1} (1 \pm z)^2 \, dz = \pm \frac{(1 \pm z)^3}{3} \bigg|_{-1}^{1} = \frac{8}{3}$$

$$\int_{-1}^{1} (1 + z)(1 - z) \, dz = \int_{-1}^{1} (1 - z^2) \, dz = z - \frac{z^3}{3} \bigg|_{-1}^{1} = \frac{4}{3}$$

By the nature of the calculations, the integral matrix will be symmetric. We will show one sample calculation for each main-diagonal and off-diagonal entries. For reference, the shape functions are:

$$\Phi_0(\xi,\eta) = \frac{1}{4}(1-\xi)(1-\eta), \quad \Phi_1(\xi,\eta) = \frac{1}{4}(1+\xi)(1-\eta),$$

$$\Phi_2(\xi,\eta) = \frac{1}{4}(1+\xi)(1+\eta), \quad \Phi_3(\xi,\eta) = \frac{1}{4}(1-\xi)(1+\eta)$$

First note that the product of any pair of shape functions will always have a factor of $\frac{1}{16}$. Also note that each product pair will contain two factors in some combination of the forms shown in the auxiliary integrals above.

Case: Main-diagonal entry ($i = j = 0$):

$$\begin{aligned}
\int_{-1}^{1}\int_{-1}^{1} \Phi_0\Phi_0 \; d\xi d\eta &= \int_{-1}^{1}\int_{-1}^{1} \frac{1}{16}(1-\xi)^2(1-\eta)^2 \; d\xi d\eta \\
&= \frac{1}{16}\int_{-1}^{1}(1-\eta)^2\int_{-1}^{1}(1-\xi)^2 \; d\xi d\eta \\
&= \frac{1}{16}\cdot\frac{8}{3}\int_{-1}^{1}(1-\eta)^2 \; d\eta \\
&= \frac{1}{16}\cdot\frac{8}{3}\cdot\frac{8}{3} \\
&= \frac{4}{9}
\end{aligned}$$

Case: 1st Off-diagonal entry ($i = 0, j = 1$):

$$\begin{aligned}
\int_{-1}^{1}\int_{-1}^{1} \Phi_0\Phi_1 \; d\xi d\eta &= \int_{-1}^{1}\int_{-1}^{1} \frac{1}{16}(1-\xi)(1-\eta)(1+\xi)(1-\eta) \; d\xi d\eta \\
&= \frac{1}{16}\int_{-1}^{1}(1-\eta)^2\int_{-1}^{1}(1-\xi^2) \; d\xi d\eta \\
&= \frac{1}{16}\cdot\frac{4}{3}\int_{-1}^{1}(1-\eta)^2 \; d\eta \\
&= \frac{1}{16}\cdot\frac{4}{3}\cdot\frac{8}{3} \\
&= \frac{2}{9}
\end{aligned}$$

Case: 2nd Off-diagonal entry ($i = 0, j = 2$):

$$\begin{aligned}
\int_{-1}^{1}\int_{-1}^{1} \Phi_0\Phi_2 \; d\xi d\eta &= \int_{-1}^{1}\int_{-1}^{1} \frac{1}{16}(1-\xi)(1-\eta)(1+\xi)(1+\eta) \; d\xi d\eta \\
&= \frac{1}{16}\int_{-1}^{1}(1-\eta^2)\int_{-1}^{1}(1-\xi^2) \; d\xi d\eta \\
&= \frac{1}{16}\cdot\frac{4}{3}\int_{-1}^{1}(1-\eta^2) \; d\eta \\
&= \frac{1}{16}\cdot\frac{4}{3}\cdot\frac{4}{3} \\
&= \frac{1}{9}
\end{aligned}$$

Case: Skew-Diagonal entry ($i = 0, j = 3$):

Same as 1st off-diagonal by symmetry, so the result is $\frac{2}{9}$.

Substituting all these results back into the integral matrix, we have:

$$\int_{-1}^{1}\int_{-1}^{1} \Phi \cdot \Phi^T \, d\xi d\eta = \frac{1}{9}\begin{bmatrix} 4 & 2 & 1 & 2 \\ 2 & 4 & 2 & 1 \\ 1 & 2 & 4 & 2 \\ 2 & 1 & 2 & 4 \end{bmatrix}$$

Therefore, the elemental mass matrix is:

$$M^{(e)} = \frac{|w| \cdot |h|}{4} \cdot \frac{1}{9}\begin{bmatrix} 4 & 2 & 1 & 2 \\ 2 & 4 & 2 & 1 \\ 1 & 2 & 4 & 2 \\ 2 & 1 & 2 & 4 \end{bmatrix}$$

Next, we compute the elemental stiffness matrix $K^{(e)}$. Denote $\nabla\Phi = [\nabla\Phi_0, \nabla\Phi_1, \nabla\Phi_2, \nabla\Phi_3]$. Note that $K^{(e)}$ can also be expressed as:

$$K^{(e)} = \int_{\Omega_e} \nabla\Phi(x, y) \cdot \nabla\Phi^T(x, y) \, dx$$

Using the change of variables, we have:

$$K^{(e)} = \int_{-1}^{1}\int_{-1}^{1} (J^{-1}\nabla\Phi(\xi, \eta)) \cdot (J^{-1}\nabla\Phi(\xi, \eta))^T |\det J(\xi, \eta)| \, d\xi d\eta$$

With the Jacobian determinant factored out, we have:

$$K^{(e)} = \frac{|w| \cdot |h|}{4} \int_{-1}^{1}\int_{-1}^{1} (J^{-1}\nabla\Phi(\xi, \eta)) \cdot (J^{-1}\nabla\Phi(\xi, \eta))^T \, d\xi d\eta$$

As reference from question 4, when corrected for our indexing scheme, we have the physical gradients:

$$\nabla\Phi_0(x, y) = \frac{1}{2}\begin{bmatrix} -\frac{1}{w}(1-\eta) \\ -\frac{1}{h}(1-\xi) \end{bmatrix}, \quad \nabla\Phi_1(x, y) = \frac{1}{2}\begin{bmatrix} \frac{1}{w}(1-\eta) \\ -\frac{1}{h}(1+\xi) \end{bmatrix}$$

$$\nabla\Phi_2(x, y) = \frac{1}{2}\begin{bmatrix} \frac{1}{w}(1+\eta) \\ \frac{1}{h}(1+\xi) \end{bmatrix}, \quad \nabla\Phi_3(x, y) = \frac{1}{2}\begin{bmatrix} -\frac{1}{w}(1+\eta) \\ \frac{1}{h}(1-\xi) \end{bmatrix}$$

so note that $J^{-1}\nabla\Phi(\xi, \eta) = \frac{1}{2} \cdot 2\nabla\Phi(\xi, \eta)$. In other words we rewrite can each physical gradient above in terms of reference coordinates without the $\frac{1}{2}$ factor. Until further notice we denote $\nabla\Phi$ instead of $2\nabla\Phi$ for ease of notation. Therefore:

$$K^{(e)} = \frac{|w| \cdot |h|}{16} \int_{-1}^{1}\int_{-1}^{1} \nabla\Phi(\xi, \eta) \cdot \nabla\Phi^T(\xi, \eta) \, d\xi d\eta$$

Also note that :

$$\nabla\Phi\cdot\nabla\Phi^T = \begin{bmatrix}\nabla\Phi_0\cdot\nabla\Phi_0 & \nabla\Phi_0\cdot\nabla\Phi_1 & \nabla\Phi_0\cdot\nabla\Phi_2 & \nabla\Phi_0\cdot\nabla\Phi_3\\ \nabla\Phi_1\cdot\nabla\Phi_0 & \nabla\Phi_1\cdot\nabla\Phi_1 & \nabla\Phi_1\cdot\nabla\Phi_2 & \nabla\Phi_1\cdot\nabla\Phi_3\\ \nabla\Phi_2\cdot\nabla\Phi_0 & \nabla\Phi_2\cdot\nabla\Phi_1 & \nabla\Phi_2\cdot\nabla\Phi_2 & \nabla\Phi_2\cdot\nabla\Phi_3\\ \nabla\Phi_3\cdot\nabla\Phi_0 & \nabla\Phi_3\cdot\nabla\Phi_1 & \nabla\Phi_3\cdot\nabla\Phi_2 & \nabla\Phi_3\cdot\nabla\Phi_3\end{bmatrix}$$

The stiffness matrix is also symmetric, so we will only show one sample calculation for each unique entry.

Case: Main-diagonal entry ($i = j = 0$):

$$\begin{aligned}\int_{-1}^{1}\int_{-1}^{1}\nabla\Phi_0\cdot\nabla\Phi_0\;d\xi d\eta &= \int_{-1}^{1}\int_{-1}^{1}\left(-\frac{1}{w}(1-\eta)\right)^2+\left(-\frac{1}{h}(1-\xi)\right)^2\;d\xi d\eta\\ &= \frac{1}{w^2}\int_{-1}^{1}\int_{-1}^{1}(1-\eta)^2\;d\xi d\eta+\frac{1}{h^2}\int_{-1}^{1}\int_{-1}^{1}(1-\xi)^2\;d\xi d\eta\\ &= \frac{2}{w^2}\int_{-1}^{1}(1-\eta)^2\;d\eta+\frac{2}{h^2}\int_{-1}^{1}(1-\xi)^2\;d\xi\\ &= \frac{2}{w^2}\cdot\frac{8}{3}+\frac{2}{h^2}\cdot\frac{8}{3}\\ &= \frac{16}{3w^2}+\frac{16}{3h^2}\\ &= \frac{16}{3}\left(\frac{1}{w^2}+\frac{1}{h^2}\right)\end{aligned}$$

Case: Skew-Diagonal entry ($i = 0, j = 3$):

$$\begin{aligned}\int_{-1}^{1}\int_{-1}^{1}\nabla\Phi_0\cdot\nabla\Phi_3\;d\xi d\eta &= \int_{-1}^{1}\int_{-1}^{1}\frac{1}{w^2}(1-\eta)(1+\eta)-\frac{1}{h^2}(1-\xi)(1-\xi)\;d\xi d\eta\\ &= \frac{1}{w^2}\int_{-1}^{1}\int_{-1}^{1}(1-\eta^2)\;d\xi d\eta-\frac{1}{h^2}\int_{-1}^{1}\int_{-1}^{1}(1-\xi)^2\;d\xi d\eta\\ &= \frac{2}{w^2}\int_{-1}^{1}(1-\eta)^2\;d\eta-\frac{2}{h^2}\int_{-1}^{1}(1-\xi)^2\;d\xi\\ &= \frac{2}{w^2}\cdot\frac{4}{3}-\frac{2}{h^2}\cdot\frac{8}{3}\\ &= \frac{8}{3w^2}-\frac{16}{3h^2}\\ &= \frac{8}{3}\left(\frac{1}{w^2}-\frac{2}{h^2}\right)\end{aligned}$$

Case: First Off-Diagonal entry ($i = 0, j = 1$):

$$\begin{aligned}
\int_{-1}^{1}\int_{-1}^{1} \nabla\Phi_0 \cdot \nabla\Phi_1 \; d\xi d\eta &= \int_{-1}^{1}\int_{-1}^{1} -\frac{1}{w^2}(1-\eta)^2 + \frac{1}{h^2}(1-\xi)(1+\xi) \; d\xi d\eta \\
&= -\frac{1}{w^2}\int_{-1}^{1}\int_{-1}^{1}(1-\eta)^2 \; d\xi d\eta + \frac{1}{h^2}\int_{-1}^{1}\int_{-1}^{1}(1-\xi^2) \; d\xi d\eta \\
&= -\frac{2}{w^2}\int_{-1}^{1}(1-\eta)^2 \; d\eta + \frac{2}{h^2}\int_{-1}^{1}(1-\xi^2) \; d\xi \\
&= -\frac{2}{w^2}\cdot\frac{8}{3} + \frac{2}{h^2}\cdot\frac{4}{3} \\
&= -\frac{16}{3w^2} + \frac{8}{3h^2} \\
&= -\frac{8}{3}\left(\frac{2}{w^2} - \frac{1}{h^2}\right)
\end{aligned}$$

Case: Second Off-Diagonal entry ($i = 0, j = 2$):

$$\begin{aligned}
\int_{-1}^{1}\int_{-1}^{1} \nabla\Phi_0 \cdot \nabla\Phi_2 \; d\xi d\eta &= \int_{-1}^{1}\int_{-1}^{1} -\frac{1}{w^2}(1-\eta)(1+\eta) - \frac{1}{h^2}(1-\xi)(1+\xi) \; d\xi d\eta \\
&= -\frac{1}{w^2}\int_{-1}^{1}\int_{-1}^{1}(1-\eta^2) \; d\xi d\eta - \frac{1}{h^2}\int_{-1}^{1}\int_{-1}^{1}(1-\xi^2) \; d\xi d\eta \\
&= -\frac{2}{w^2}\int_{-1}^{1}(1-\eta^2) \; d\eta - \frac{2}{h^2}\int_{-1}^{1}(1-\xi^2) \; d\xi \\
&= -\frac{2}{w^2}\cdot\frac{4}{3} - \frac{2}{h^2}\cdot\frac{4}{3} \\
&= -\frac{8}{3w^2} - \frac{8}{3h^2} \\
&= -\frac{8}{3}\left(\frac{1}{w^2} + \frac{1}{h^2}\right)
\end{aligned}$$

Now let us compute the unique elemental and stiffness matrices for our mesh using our formulas above.

There will be 3 unique element mass matrices for the following combinations of width $w = (x_1 - x_0)$ and height $h = (y_2 - y_1)$:

1. Element with width 0.4 and height 0.4
2. Element with width 0.4 and height 2.8 or vice versa
3. Element with width 2.8 and height 2.8

Case 1: Element with width 0.4 and height 0.4

$$M^{(e)} = \frac{(\frac{2}{5})^2}{4}\cdot\frac{1}{9}\begin{bmatrix} 4 & 2 & 1 & 2 \\ 2 & 4 & 2 & 1 \\ 1 & 2 & 4 & 2 \\ 2 & 1 & 2 & 4 \end{bmatrix} = \frac{1}{225}\begin{bmatrix} 4 & 2 & 1 & 2 \\ 2 & 4 & 2 & 1 \\ 1 & 2 & 4 & 2 \\ 2 & 1 & 2 & 4 \end{bmatrix}$$

Case 2: Element with width 0.4 and height 2.8

$$M^{(e)} = \frac{(\frac{2}{5})\cdot(\frac{14}{5})}{4}\cdot\frac{1}{9}\begin{bmatrix}4 & 2 & 1 & 2\\ 2 & 4 & 2 & 1\\ 1 & 2 & 4 & 2\\ 2 & 1 & 2 & 4\end{bmatrix} = \frac{7}{225}\begin{bmatrix}4 & 2 & 1 & 2\\ 2 & 4 & 2 & 1\\ 1 & 2 & 4 & 2\\ 2 & 1 & 2 & 4\end{bmatrix}$$

Case 3: Element with width 2.8 and height 2.8

$$M^{(e)} = \frac{(\frac{14}{5})^2}{4}\cdot\frac{1}{9}\begin{bmatrix}4 & 2 & 1 & 2\\ 2 & 4 & 2 & 1\\ 1 & 2 & 4 & 2\\ 2 & 1 & 2 & 4\end{bmatrix} = \frac{49}{225}\begin{bmatrix}4 & 2 & 1 & 2\\ 2 & 4 & 2 & 1\\ 1 & 2 & 4 & 2\\ 2 & 1 & 2 & 4\end{bmatrix}$$

There will be 3 unique element stiffness matrices for the following combinations of width $w = (x_1 - x_0)$ and height $h = (y_2 - y_1)$:

1. Square element with width 0.4 and height 0.4 or width 2.8 and height 2.8
2. Element with width 0.4 and height 2.8
3. Element with width 2.8 and height 0.4

Case 1: Square element (ex. width 0.4 and height 0.4)

Main diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot\frac{16}{3}\left(\frac{1}{w^2}+\frac{1}{h^2}\right) = \frac{(\frac{2}{5})^2}{16}\cdot\frac{16}{3}\left(\frac{1}{(\frac{2}{5})^2}+\frac{1}{(\frac{2}{5})^2}\right) = \frac{2}{3} = \frac{4}{6}$$

Skew diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot\frac{8}{3}\left(\frac{1}{w^2}-\frac{2}{h^2}\right) = \frac{(\frac{2}{5})^2}{16}\cdot\frac{8}{3}\left(\frac{1}{(\frac{2}{5})^2}-\frac{2}{(\frac{2}{5})^2}\right) = -\frac{1}{6}$$

First off-diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot-\frac{8}{3}\left(\frac{2}{w^2}-\frac{1}{h^2}\right) = \frac{(\frac{2}{5})^2}{16}\cdot-\frac{8}{3}\left(\frac{2}{(\frac{2}{5})^2}-\frac{1}{(\frac{2}{5})^2}\right) = -\frac{1}{6}$$

Second off-diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot-\frac{8}{3}\left(\frac{1}{w^2}+\frac{1}{h^2}\right) = \frac{(\frac{2}{5})^2}{16}\cdot-\frac{8}{3}\left(\frac{1}{(\frac{2}{5})^2}+\frac{1}{(\frac{2}{5})^2}\right) = -\frac{1}{3} = -\frac{2}{6}$$

Therefore the elemental stiffness matrix is:

$$K^{(e)} = \frac{1}{6}\begin{bmatrix}4 & -1 & -2 & -1\\ -1 & 4 & -1 & -2\\ -2 & -1 & 4 & -1\\ -1 & -2 & -1 & 4\end{bmatrix}$$

Case 2: Element with width 0.4 and height 2.8

Main diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot\frac{16}{3}\left(\frac{1}{w^2}+\frac{1}{h^2}\right)=\frac{(\frac{2}{5})\cdot(\frac{14}{5})}{16}\cdot\frac{16}{3}\left(\frac{1}{(\frac{2}{5})^2}+\frac{1}{(\frac{14}{5})^2}\right)=\frac{50}{21}=\frac{100}{42}$$

Skew-diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot\frac{8}{3}\left(\frac{1}{w^2}-\frac{2}{h^2}\right)=\frac{(\frac{2}{5})\cdot(\frac{14}{5})}{16}\cdot\frac{8}{3}\left(\frac{1}{(\frac{2}{5})^2}-\frac{2}{(\frac{14}{5})^2}\right)=\frac{47}{42}$$

First off-diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot-\frac{8}{3}\left(\frac{2}{w^2}-\frac{1}{h^2}\right)=\frac{(\frac{2}{5})\cdot(\frac{3}{5})}{16}\cdot-\frac{8}{3}\left(\frac{2}{(\frac{2}{5})^2}-\frac{1}{(\frac{14}{5})^2}\right)=-\frac{97}{42}$$

Second off-diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot-\frac{8}{3}\left(\frac{1}{w^2}+\frac{1}{h^2}\right)=\frac{(\frac{2}{5})\cdot(\frac{3}{5})}{16}\cdot-\frac{8}{3}\left(\frac{1}{(\frac{2}{5})^2}+\frac{1}{(\frac{14}{5})^2}\right)=-\frac{25}{21}=-\frac{50}{42}$$

Therefore the elemental stiffness matrix is:

$$K^{(e)}=\frac{1}{42}\begin{bmatrix}100 & -97 & -50 & 47\\ -97 & 100 & 47 & -50\\ -50 & 47 & 100 & -97\\ 47 & -50 & -97 & 100\end{bmatrix}$$

Case 3: Element with width 2.8 and height 0.4

Main diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot\frac{16}{3}\left(\frac{1}{w^2}+\frac{1}{h^2}\right)=\frac{(\frac{14}{5})\cdot(\frac{2}{5})}{16}\cdot\frac{16}{3}\left(\frac{1}{(\frac{14}{5})^2}+\frac{1}{(\frac{2}{5})^2}\right)=\frac{50}{21}=\frac{100}{42}$$

Skew-diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot\frac{8}{3}\left(\frac{1}{w^2}-\frac{2}{h^2}\right)=\frac{(\frac{14}{5})\cdot(\frac{2}{5})}{16}\cdot\frac{8}{3}\left(\frac{1}{(\frac{14}{5})^2}-\frac{2}{(\frac{2}{5})^2}\right)=-\frac{97}{42}$$

First off-diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot-\frac{8}{3}\left(\frac{2}{w^2}-\frac{1}{h^2}\right)=\frac{(\frac{14}{5})\cdot(\frac{2}{5})}{16}\cdot-\frac{8}{3}\left(\frac{2}{(\frac{14}{5})^2}-\frac{1}{(\frac{2}{5})^2}\right)=\frac{47}{42}$$

Second off-diagonal entry:

$$\frac{|w|\cdot|h|}{16}\cdot-\frac{8}{3}\left(\frac{1}{w^2}+\frac{1}{h^2}\right)=\frac{(\frac{14}{5})\cdot(\frac{2}{5})}{16}\cdot-\frac{8}{3}\left(\frac{1}{(\frac{14}{5})^2}+\frac{1}{(\frac{2}{5})^2}\right)=-\frac{25}{21}=-\frac{50}{42}$$

Therefore the elemental stiffness matrix is:

$$K^{(e)}=\frac{1}{42}\begin{bmatrix}100 & 47 & -50 & -97\\ 47 & 100 & -97 & -50\\ -50 & -97 & 100 & 47\\ -97 & -50 & 47 & 100\end{bmatrix}$$

Question 6

Assemble the global mass matrix M and global stiffness matrix K for the entire 2×2 mesh using the connectivity determined in question 2. Impose homogeneous Dirichlet boundary conditions on all boundary nodes. Write the resulting discretized ODE system in the following format:

$$M\frac{dU}{dt} + KU = F(t)$$

Solution.

We will adjust the indexing in the preliminary setup to match this scenario. From before we had:

$$\sum_{i,j=1}^{N} \left(\int_\Omega \Phi_i \Phi_j \; dx \right) \frac{dU_i}{dt} + \nu \sum_{i,j=1}^{N} \left(\int_\Omega \nabla\Phi_i \cdot \nabla\Phi_j \; dx \right) U_i = \sum_{j=1}^{N} \int_\Omega f \Phi_j \; dx$$

Was our global discretized system, where N is the total number of nodes. Rewriting this terms of the elemental components, we have:

$$\left(\sum_{e=1}^{E} M^{(e)} \right) \frac{dU}{dt} + \nu \left(\sum_{e=1}^{E} K^{(e)} \right) U = \sum_{e=1}^{E} F^{(e)}(t)$$

Where E is the total number of elements.

Correcting for our indexing scheme, we have:

$$\left(\sum_{e=0}^{15} M^{(e)} \right) \frac{dU}{dt} + \nu \left(\sum_{e=0}^{15} K^{(e)} \right) U = \sum_{e=0}^{15} F^{(e)}(t)$$

Such that that $M = \sum_{e=0}^{15} M^{(e)}$ and $K = \sum_{e=0}^{15} K^{(e)}$. We could lump the ν into K to get the form:

$$M\frac{dU}{dt} + KU = F(t)$$

Also, since we have the homogeneous heat equation, $F^{(e)}(t) = 0$ for all e.

In addition to methods defined in Question 2, the following code was used to assemble our global matrices:

Listing 2: Question 6 Python

```python
import numpy as np
import sympy as sp

def element_mass_matrix(w, h):
    area = w * h
    Me = (area/36) * np.array([[4, 2, 1, 2],
                               [2, 4, 2, 1],
```

```
                                [1, 2, 4, 2],
                                [2, 1, 2, 4]])
    return Me

def element_stiffness_matrix(w, h):
    # When multiplying each entry formula by w * h /16, we get the following
    reduced formulas:
    wh = w/h
    hw = h/w
    # Main diagonal
    a = (1/3) * (hw + wh)
    # Skew diagonal
    b = (1/6) * (hw - 2*wh)
    # First off-diagonal
    c = -(1/6) * (2*hw - wh)
    # Second off-diagonal
    d = -(1/6) * (hw + wh)
    # Construct element stiffness matrix
    Ke = np.array([[a, c, d, b],
                   [c, a, b, d],
                   [d, b, a, c],
                   [b, d, c, a]])
    return Ke

def generate_global_coordinates(x_nodes, y_nodes=None):
    if y_nodes is None:
        y_nodes = x_nodes

    y_nodes = y_nodes[::-1]
    global_coordinates = []
    for y in y_nodes:
        for x in x_nodes:
            global_coordinates.append([x, y])
    global_coordinates = np.array(global_coordinates)
    return global_coordinates


def global_assembly(x_nodes, y_nodes=None):
    if y_nodes is None:
        y_nodes = x_nodes

    global_coordinates = generate_global_coordinates(x_nodes, y_nodes)

    width = len(x_nodes)
    height = len(y_nodes)

    global_indices = global_indexing(width, height)
```

```
    connectivity_matrix = generate_connectivity_matrix(global_indices)

    num_nodes = width * height
    M_global = np.zeros((num_nodes, num_nodes))
    K_global = np.zeros((num_nodes, num_nodes))

    for element in connectivity_matrix:

        # Bottom-left (node 0)
        x0 = global_coordinates[element[0], 0]
        y0 = global_coordinates[element[0], 1]

        # Bottom-right (node 1)
        x1 = global_coordinates[element[1], 0]

        # Top-left (node 3)
        y3 = global_coordinates[element[3], 1]

        # Element width and height
        w = abs(x1 - x0)
        h = abs(y3 - y0)

        # Get element matrices
        Me = element_mass_matrix(w, h)
        Ke = element_stiffness_matrix(w, h)

        # Add element contributions to global matrices
        for i_local in range(4):
            i_global = element[i_local]
            for j_local in range(4):
                j_global = element[j_local]
                M_global[i_global, j_global] += Me[i_local, j_local]
                K_global[i_global, j_global] += Ke[i_local, j_local]

    return global_coordinates, M_global, K_global

if __name__ == "__main__":
    x_nodes = np.array([-2, -1.6, -1.2, -0.8, 2])
    _, M, K = global_assembly(x_nodes)

    with open("./outputs_4/matrices.txt", "w") as f:
        M = np.round(M, 4)
        M_split = np.array_split(M, 3, axis=1)
        latex_1_matrix = sp.latex(sp.Matrix(M_split[0]))
        latex_2_matrix = sp.latex(sp.Matrix(M_split[1]))
        latex_3_matrix = sp.latex(sp.Matrix(M_split[2]))
        f.write("Mass Matrix (Part 1):\n")
```

```
        f.write(latex_1_matrix + "\n\n")
        f.write("Mass Matrix (Part 2):\n")
        f.write(latex_2_matrix + "\n\n")
        f.write("Mass Matrix (Part 3):\n")
        f.write(latex_3_matrix + "\n\n")

        K = np.round(K, 4)
        K_split = np.array_split(K, 3, axis=1)
        latex_1_matrix = sp.latex(sp.Matrix(K_split[0]))
        latex_2_matrix = sp.latex(sp.Matrix(K_split[1]))
        latex_3_matrix = sp.latex(sp.Matrix(K_split[2]))
        f.write("Stiffness Matrix (Part 1):\n")
        f.write(latex_1_matrix + "\n\n")
        f.write("Stiffness Matrix (Part 2):\n")
        f.write(latex_2_matrix + "\n\n")
        f.write("Stiffness Matrix (Part 3):\n")
        f.write(latex_3_matrix + "\n\n")
```

The code was constructed such that non-uniform meshes of arbitrary widths and heights could be used.

Our mass matrix M will be of size 25×25 and was found to be:

$$
\begin{bmatrix}
0.1244 & 0.0622 & 0.0 & 0.0 & 0.0 & 0.0622 & 0.0311 & 0.0 & 0.0 \\
0.0622 & 0.2489 & 0.0622 & 0.0 & 0.0 & 0.0311 & 0.1244 & 0.0311 & 0.0 \\
0.0 & 0.0622 & 0.2489 & 0.0622 & 0.0 & 0.0 & 0.0311 & 0.1244 & 0.0311 \\
0.0 & 0.0 & 0.0622 & 0.9956 & 0.4356 & 0.0 & 0.0 & 0.0311 & 0.4978 \\
0.0 & 0.0 & 0.0 & 0.4356 & 0.8711 & 0.0 & 0.0 & 0.0 & 0.2178 \\
0.0622 & 0.0311 & 0.0 & 0.0 & 0.0 & 0.1422 & 0.0711 & 0.0 & 0.0 \\
0.0311 & 0.1244 & 0.0311 & 0.0 & 0.0 & 0.0711 & 0.2844 & 0.0711 & 0.0 \\
0.0 & 0.0311 & 0.1244 & 0.0311 & 0.0 & 0.0 & 0.0711 & 0.2844 & 0.0711 \\
0.0 & 0.0 & 0.0311 & 0.4978 & 0.2178 & 0.0 & 0.0 & 0.0711 & 1.1378 \\
0.0 & 0.0 & 0.0 & 0.2178 & 0.4356 & 0.0 & 0.0 & 0.0 & 0.4978 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0089 & 0.0044 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0044 & 0.0178 & 0.0044 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0044 & 0.0178 & 0.0044 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0044 & 0.0711 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0311 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix} \cdots
$$

$$\cdots \begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.2178 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.4356 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0089 & 0.0044 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0044 & 0.0178 & 0.0044 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0044 & 0.0178 & 0.0044 & 0.0 & 0.0 & 0.0 \\
0.4978 & 0.0 & 0.0 & 0.0044 & 0.0711 & 0.0311 & 0.0 & 0.0 \\
0.9956 & 0.0 & 0.0 & 0.0 & 0.0311 & 0.0622 & 0.0 & 0.0 \\
0.0 & 0.0356 & 0.0178 & 0.0 & 0.0 & 0.0 & 0.0089 & 0.0044 \\
0.0 & 0.0178 & 0.0711 & 0.0178 & 0.0 & 0.0 & 0.0044 & 0.0178 \\
0.0 & 0.0 & 0.0178 & 0.0711 & 0.0178 & 0.0 & 0.0 & 0.0044 \\
0.0311 & 0.0 & 0.0 & 0.0178 & 0.2844 & 0.1244 & 0.0 & 0.0 \\
0.0622 & 0.0 & 0.0 & 0.0 & 0.1244 & 0.2489 & 0.0 & 0.0 \\
0.0 & 0.0089 & 0.0044 & 0.0 & 0.0 & 0.0 & 0.0356 & 0.0178 \\
0.0 & 0.0044 & 0.0178 & 0.0044 & 0.0 & 0.0 & 0.0178 & 0.0711 \\
0.0 & 0.0 & 0.0044 & 0.0178 & 0.0044 & 0.0 & 0.0 & 0.0178 \\
0.0 & 0.0 & 0.0 & 0.0044 & 0.0711 & 0.0311 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0311 & 0.0622 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0089 & 0.0044 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0044 & 0.0178 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0044 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix} \cdots$$

$$\cdots \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0044 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0178 & 0.0044 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0044 & 0.0711 & 0.0311 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0311 & 0.0622 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0089 & 0.0044 & 0.0 & 0.0 & 0.0 \\ 0.0178 & 0.0 & 0.0 & 0.0044 & 0.0178 & 0.0044 & 0.0 & 0.0 \\ 0.0711 & 0.0178 & 0.0 & 0.0 & 0.0044 & 0.0178 & 0.0044 & 0.0 \\ 0.0178 & 0.2844 & 0.1244 & 0.0 & 0.0 & 0.0044 & 0.0711 & 0.0311 \\ 0.0 & 0.1244 & 0.2489 & 0.0 & 0.0 & 0.0 & 0.0311 & 0.0622 \\ 0.0 & 0.0 & 0.0 & 0.0178 & 0.0089 & 0.0 & 0.0 & 0.0 \\ 0.0044 & 0.0 & 0.0 & 0.0089 & 0.0356 & 0.0089 & 0.0 & 0.0 \\ 0.0178 & 0.0044 & 0.0 & 0.0 & 0.0089 & 0.0356 & 0.0089 & 0.0 \\ 0.0044 & 0.0711 & 0.0311 & 0.0 & 0.0 & 0.0089 & 0.1422 & 0.0622 \\ 0.0 & 0.0311 & 0.0622 & 0.0 & 0.0 & 0.0 & 0.0622 & 0.1244 \end{bmatrix}$$

Our stiffness matrix K will also be of size 25×25 and was found to be (without ν):

$$
\begin{bmatrix}
2.381 & -2.3095 & 0.0 & 0.0 & 0.0 & 1.119 & -1.1905 & 0.0 & 0.0 \\
-2.3095 & 4.7619 & -2.3095 & 0.0 & 0.0 & -1.1905 & 2.2381 & -1.1905 & 0.0 \\
0.0 & -2.3095 & 4.7619 & -2.3095 & 0.0 & 0.0 & -1.1905 & 2.2381 & -1.1905 \\
0.0 & 0.0 & -2.3095 & 3.0476 & -0.1667 & 0.0 & 0.0 & -1.1905 & 0.9524 \\
0.0 & 0.0 & 0.0 & -0.1667 & 0.6667 & 0.0 & 0.0 & 0.0 & -0.3333 \\
1.119 & -1.1905 & 0.0 & 0.0 & 0.0 & 3.0476 & -2.4762 & 0.0 & 0.0 \\
-1.1905 & 2.2381 & -1.1905 & 0.0 & 0.0 & -2.4762 & 6.0952 & -2.4762 & 0.0 \\
0.0 & -1.1905 & 2.2381 & -1.1905 & 0.0 & 0.0 & -2.4762 & 6.0952 & -2.4762 \\
0.0 & 0.0 & -1.1905 & 0.9524 & -0.3333 & 0.0 & 0.0 & -2.4762 & 6.0952 \\
0.0 & 0.0 & 0.0 & -0.3333 & -0.1667 & 0.0 & 0.0 & 0.0 & 0.9524 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.1667 & -0.3333 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.3333 & -0.3333 & -0.3333 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.3333 & -0.3333 & -0.3333 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.3333 & -2.4762 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.1905 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix} \cdots
$$

$$\cdots\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-0.3333 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-0.1667 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -0.1667 & -0.3333 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -0.3333 & -0.3333 & -0.3333 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & -0.3333 & -0.3333 & -0.3333 & 0.0 & 0.0 & 0.0 \\
0.9524 & 0.0 & 0.0 & -0.3333 & -2.4762 & -1.1905 & 0.0 & 0.0 \\
3.0476 & 0.0 & 0.0 & 0.0 & -1.1905 & -2.3095 & 0.0 & 0.0 \\
0.0 & 1.3333 & -0.3333 & 0.0 & 0.0 & 0.0 & -0.1667 & -0.3333 \\
0.0 & -0.3333 & 2.6667 & -0.3333 & 0.0 & 0.0 & -0.3333 & -0.3333 \\
0.0 & 0.0 & -0.3333 & 2.6667 & -0.3333 & 0.0 & 0.0 & -0.3333 \\
-1.1905 & 0.0 & 0.0 & -0.3333 & 6.0952 & 2.2381 & 0.0 & 0.0 \\
-2.3095 & 0.0 & 0.0 & 0.0 & 2.2381 & 4.7619 & 0.0 & 0.0 \\
0.0 & -0.1667 & -0.3333 & 0.0 & 0.0 & 0.0 & 1.3333 & -0.3333 \\
0.0 & -0.3333 & -0.3333 & -0.3333 & 0.0 & 0.0 & -0.3333 & 2.6667 \\
0.0 & 0.0 & -0.3333 & -0.3333 & -0.3333 & 0.0 & 0.0 & -0.3333 \\
0.0 & 0.0 & 0.0 & -0.3333 & -2.4762 & -1.1905 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & -1.1905 & -2.3095 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.1667 & -0.3333 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.3333 & -0.3333 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.3333 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix}\cdots$$

$$\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-0.3333 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-0.3333 & -0.3333 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-0.3333 & -2.4762 & -1.1905 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -1.1905 & -2.3095 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -0.1667 & -0.3333 & 0.0 & 0.0 & 0.0 \\
-0.3333 & 0.0 & 0.0 & -0.3333 & -0.3333 & -0.3333 & 0.0 & 0.0 \\
2.6667 & -0.3333 & 0.0 & 0.0 & -0.3333 & -0.3333 & -0.3333 & 0.0 \\
-0.3333 & 6.0952 & 2.2381 & 0.0 & 0.0 & -0.3333 & -2.4762 & -1.1905 \\
0.0 & 2.2381 & 4.7619 & 0.0 & 0.0 & 0.0 & -1.1905 & -2.3095 \\
0.0 & 0.0 & 0.0 & 0.6667 & -0.1667 & 0.0 & 0.0 & 0.0 \\
-0.3333 & 0.0 & 0.0 & -0.1667 & 1.3333 & -0.1667 & 0.0 & 0.0 \\
-0.3333 & -0.3333 & 0.0 & 0.0 & -0.1667 & 1.3333 & -0.1667 & 0.0 \\
-0.3333 & -2.4762 & -1.1905 & 0.0 & 0.0 & -0.1667 & 3.0476 & 1.119 \\
0.0 & -1.1905 & -2.3095 & 0.0 & 0.0 & 0.0 & 1.119 & 2.381
\end{bmatrix} \cdots$$

As expected, both M and K are symmetric, banded matrices.

Question 7

Using the following initial condition

$$u(x, y, 0) = \sin(2\pi x)\sin(2\pi y)$$

to solve the reduced ODE system from $t = 0$ to $t = 1$. (Show Code)

Solution. Given our system:

$$M\frac{dU}{dt} + \nu KU = 0$$

We can rearrange this to get:

$$\frac{dU}{dt} = -\nu M^{-1}KU$$

Implicit: Backward Euler

Let us find U^{n+1} implicitly using backward difference in time:

$$\frac{U^{n+1} - U^n}{\Delta t} = -\nu M^{-1}KU^{n+1}$$

Rearranging for U^{n+1}, we have:

$$\left(I + \nu\Delta t M^{-1}K\right)U^{n+1} = U^n$$

The following code was used to implement the implicit backward Euler method:

Listing 3: implicit_euler.py

```
import numpy as np
def classify_boundary_nodes(global_indexing):
    # Create a boundary indicator array where 1 indicates a boundary node
    boundary_indicator = np.zeros_like(global_indexing)
    boundary_indicator[0, :] = 1  # Top boundary
    boundary_indicator[-1, :] = 1  # Bottom boundary
    boundary_indicator[:, 0] = 1  # Left boundary
    boundary_indicator[:, -1] = 1  # Right boundary
    return boundary_indicator.flatten()

def implement_dirichlet_bc(M, K,boundary_indicator):
    num_nodes = M.shape[0]
    for i in range(num_nodes):
        if boundary_indicator[i] == 1:
            M[i, :] = 0
            M[:, i] = 0
            M[i, i] = 1
            K[i, :] = 0
            K[:, i] = 0
```

```python
            K[i, i] = 1
    return M, K

def u_0(coordinates, boundary_indicator):
    x = coordinates[:, 0]
    y = coordinates[:, 1]
    u = np.sin(2*np.pi*x) * np.sin(2*np.pi*y)
    u[boundary_indicator == 1] = 0.0  # Apply Dirichlet BCs
    num_nodes = coordinates.shape[0]
    return u.reshape((num_nodes,1))

def implicit_heat_solver(x_nodes, dt, t_final, nu=0.05):
    global_coordinates, M, K = global_assembly(x_nodes)
    width = len(x_nodes)
    global_indexing_array = global_indexing(width)
    boundary_indicator = classify_boundary_nodes(global_indexing_array)

    M, K = implement_dirichlet_bc(M, K, boundary_indicator)

    A = M + nu * dt * K

    num_time_steps = math.ceil(t_final / dt)
    dt = t_final / num_time_steps
    prev_U = u_0(global_coordinates, boundary_indicator)
    U = np.array([prev_U])

    for n in range(num_time_steps):
        b = np.matmul(M, U[n])
        next_U = np.linalg.solve(A, b)
        U = np.append(U, [next_U], axis=0)

    return boundary_indicator, U.T, global_coordinates
```

Question 8

Write your own solver for solving linear systems. You can freely choose the methods from Gaussian, Jacobi, or Gauss-Seidel.

Solution.

Jacobi is defined interatively as:

$$x_{n+1} = D^{-1}(-R \cdot x_n + b)$$

Where D is the diagonal of A and $R = A - D$ is the remainder matrix. We will use an initial guess of $x_0 = 0$ and iterate until tolerance is met (with l_∞ norm) or maximum iterations are reached.

The following code was used to implement the Jacobi method:

Listing 4: Jacobi Method

```python
import numpy as np

def jacobi(A, b, x0=None, tol=1e-12, max_iterations=10000):
    x_prev = np.zeros_like(b) if x0 is None else x0.copy()
    diagonal_list = np.diag(A)
    D = np.diagflat(diagonal_list)
    R = A - D
    D_inverse = np.linalg.inv(D)

    for _ in range(max_iterations):
        rx_b = -np.dot(R, x_prev) + b
        x_next = np.dot(D_inverse, rx_b)
        if np.linalg.norm(x_next - x_prev, ord=np.inf) < tol:
            return x_next
        x_prev = x_next

    raise ValueError("Jacobi method did not converge within the maximum number
   of iterations")
```

Extra: See Appendix for past implementations of Gaussian Elimination, Jacobi, and Gauss-Seidel in Maple.

Question 9

Perform a convergence study with different refinements on time steps. Plot the log-log plot of error vs time step size.

Solution. The following code was used to perform the convergence study:

Listing 5: Convergence Study

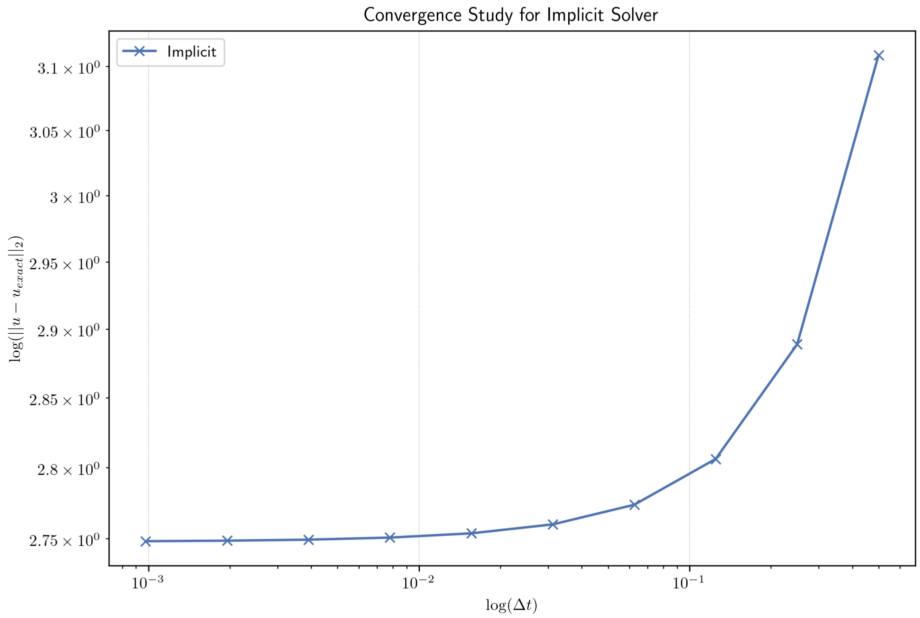
```python
import numpy as np
import matplotlib.pyplot as plt
def convergence_study_timesteps(x_nodes, t_final, dt_values, y_nodes=None):
    if y_nodes is None:
        y_nodes = x_nodes
    errors_implicit = []
    global_coordinates, _, _ = global_assembly(x_nodes, y_nodes)
    width = len(x_nodes)
    global_indexing_array = global_indexing(width)
    boundary = classify_boundary_nodes(global_indexing_array)
    x_coords = global_coordinates[:, 0].reshape(width**2, 1)
    y_coords = global_coordinates[:, 1].reshape(width**2, 1)
    u_exact_values = u_exact(x_coords, y_coords, t_final, boundary)

    for dt in dt_values:
        _, U, _ = implicit_heat_solver(x_nodes, dt, t_final)
        u_numerical_values = U[:, :, -1]
        error = np.linalg.norm(u_numerical_values - u_exact_values, ord=2)
        errors_implicit.append(error)

    return errors_implicit

if __name__ == "__main__":
    # <some matplotlib styling and enable latex>
    x_nodes = np.array([-2,-1.6,-1.2,-0.8,2])
    t_final = 1
    dt_values = np.logspace(-1, -10, 10, base=2)
    errors_implicit = convergence_study_timesteps(x_nodes, t_final, dt_values)
    fig, ax = plt.subplots(figsize=(9,6))
    ax.loglog(dt_values, errors_implicit, marker='x', label='Implicit')
    ax.set_xlabel(r'$\log(\Delta t)$')
    ax.set_ylabel(r'$\log(||u - u_{exact}||_2)$')
    ax.set_title('Convergence Study for Implicit Solver')
    ax.legend()
    fig.savefig("./outputs_4/implicit_convergence_study.png", dpi=300)
```

For efficiency, we use numpy's built in linear algebra. We get the resulting error plot below:

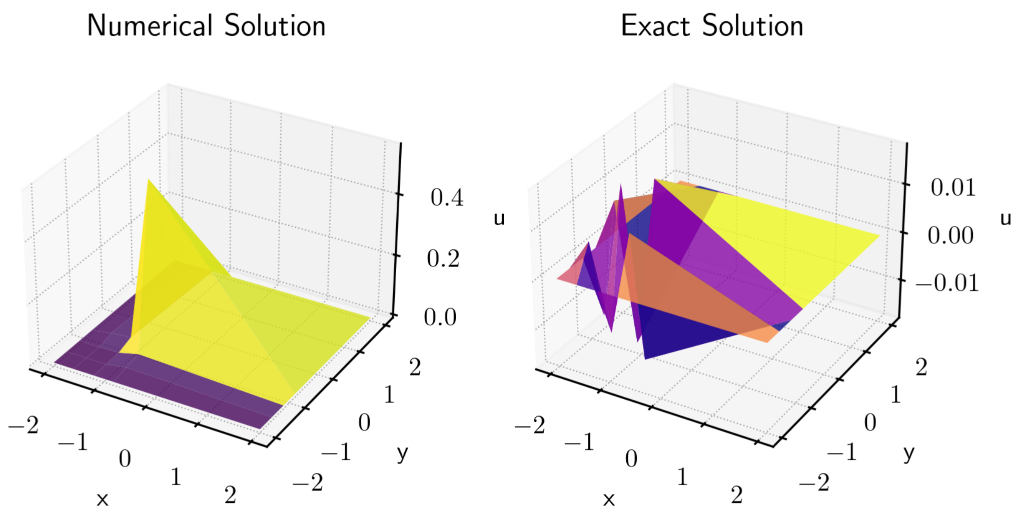


We can see that with smaller time steps, the error decreases.

Question 10

Plot $U(t)$ at $T = 1$.

Solution. The following plot was generated using the implicit solver at $t = 1$:



Appendix

1 Maple Implementations of Linear Solvers

These are implementations of Gaussian Elimination, Jacobi Method, and Gauss-Seidel Method in Maple I have done in the past.

These were the questions being answered:

3. Use LU factorization to solve the system:

$$\begin{aligned} 6x_1 - 2x_2 + 2x_3 + 4x_4 &= 16 \\ 12x_1 - 8x_2 + 6x_3 + 10x_4 &= 26 \\ 3x_1 - 13x_2 + 9x_3 + 3x_4 &= -19 \\ -6x_1 + 4x_2 + x_3 - 18x_4 &= -34 \end{aligned}$$

   Be sure to state the matrices L and U.

4. Use the Jacobi iterative method and the Gauss-Seidel iterative method to find the solution to the following set of equations within 10^{-4} in the ℓ_∞ norm using $\mathbf{x}^{(0)} = \mathbf{0}$ as your initial condition. Show theoretically whether or not both methods will converge in this case.

$$\begin{aligned} 4x_1 + x_2 + x_3 + x_4 &= -5 \\ x_1 + 8x_2 + 2x_3 + 3x_4 &= 23 \\ x_1 + 2x_2 - 5x_3 &= 9 \\ -x_1 + 2x_3 + 4x_4 &= 4 \end{aligned}$$

The following pages contain the Maple implementations and sample outputs.

Note: the Matrix Solver is Gaussian Elimination with addition of LU decomposition. No row exchanges were implemented.

Matrix Solver

```
with(LinearAlgebra) :
MatrixSolve := proc(A, b)
  local i, j, n, L, U, v, const, det;
    n := RowDimension(A);
    L := Matrix(n);
    U := A;
    v := b;
    det := 1;
    print(A, b);
  for i from 1 to n − 1 do
    for j from i + 1 to n do

      if (U[i, i] = 0) then
        error "zero along main diagonal";
      end if;
      const := U[j, i]/U[i, i];
      if (const ≠ 0) then
        L[j, i] := const;
        U := RowOperation(U, [j, i], −const);
        v[j] := v[j] − const·v[i];
        print(R(j) − const·R(i), U, v);
      end if;

    end do;
  end do;

  for i from 1 to n do
    det := det·U[i, i];
    L[i, i] := 1;
  end do;

  print(L, U, v, det);

end proc:
```

③ **Problem #3**

$A := Matrix([[6,-2,2,4],[12,-8,6,10],[3,-13,9,3],[-6,4,1,-18]]);$

$b := Vector([16,26,-19,-34]);$

$$A := \begin{bmatrix} 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix}$$

$$b := \begin{bmatrix} 16 \\ 26 \\ -19 \\ -34 \end{bmatrix} \tag{6}$$

$MatrixSolve(A, b);$

$$\begin{bmatrix} 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix}, \begin{bmatrix} 16 \\ 26 \\ -19 \\ -34 \end{bmatrix}$$

$$R(2) - 2\,R(1), \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix}, \begin{bmatrix} 16 \\ -6 \\ -19 \\ -34 \end{bmatrix}$$

$$R(3) - \frac{R(1)}{2}, \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & -12 & 8 & 1 \\ -6 & 4 & 1 & -18 \end{bmatrix}, \begin{bmatrix} 16 \\ -6 \\ -27 \\ -34 \end{bmatrix}$$

$$R(4) + R(1), \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & -12 & 8 & 1 \\ 0 & 2 & 3 & -14 \end{bmatrix}, \begin{bmatrix} 16 \\ -6 \\ -27 \\ -18 \end{bmatrix}$$

$$R(3) - 3\,R(2), \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 2 & 3 & -14 \end{bmatrix}, \begin{bmatrix} 16 \\ -6 \\ -9 \\ -18 \end{bmatrix}$$

$$R(4)+\frac{R(2)}{2},\ \begin{bmatrix}6&-2&2&4\\0&-4&2&2\\0&0&2&-5\\0&0&4&-13\end{bmatrix},\ \begin{bmatrix}16\\-6\\-9\\-21\end{bmatrix}$$

$$R(4)-2\,R(3),\ \begin{bmatrix}6&-2&2&4\\0&-4&2&2\\0&0&2&-5\\0&0&0&-3\end{bmatrix},\ \begin{bmatrix}16\\-6\\-9\\-3\end{bmatrix}$$

$$\boxed{\underbrace{\begin{bmatrix}1&0&0&0\\2&1&0&0\\\frac{1}{2}&3&1&0\\-1&-\frac{1}{2}&2&1\end{bmatrix}}_{L},\ \underbrace{\begin{bmatrix}6&-2&2&4\\0&-4&2&2\\0&0&2&-5\\0&0&0&-3\end{bmatrix}}_{U}},\ \begin{bmatrix}16\\-6\\-9\\-3\end{bmatrix},\ 144 \tag{7}$$

Ly=b

$y1 := 16 :$

$y2 := -2 \cdot y1 + 26 :$

$y3 := -\frac{1}{2} \cdot y1 - 3 \cdot y2 - 19 :$

$y4 := y1 + \frac{1}{2} \cdot y2 - 2 \cdot y3 - 34 :$

$y := Vector([y1, y2, y3, y4]);$

$$y := \begin{bmatrix}16\\-6\\-9\\-3\end{bmatrix} \tag{8}$$

Ux=y

$x4 := -\frac{1}{3}(-3) :$

$x3 := \frac{1}{2}(5 \cdot x4 - 9) :$

$x2 := -\frac{1}{4}(-2 \cdot x3 - 2 \cdot x4 - 6) :$

$x1 := \frac{1}{6}(2 \cdot x2 - 2 \cdot x3 - 4 \cdot x4 + 16) :$

$x := Vector([x1, x2, x3, x4]);$

$$\boxed{x := \begin{bmatrix} 3 \\ 1 \\ -2 \\ 1 \end{bmatrix}} \tag{9}$$

Jacobi

```
with(Student[LinearAlgebra]) :
Jacobi :=proc(T, c, α, ε, N)
    local i, x, err, temp;
      x := α;

    for i from 1 to N do
      temp := T · x + c;
      err := Norm(temp − x, infinity);
      x := temp;
      if (err < ε) then
        break;
      end if;
    end do;

 print(evalf(x), i);
 end proc:
```

$T := Matrix\left(\left[\left[0, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}\right], \left[-\frac{1}{8}, 0, -\frac{1}{4}, -\frac{3}{8}\right], \left[\frac{1}{5}, \frac{2}{5}, 0, 0\right], \left[\frac{1}{4}, 0, -\frac{1}{2}, 0\right]\right]\right);$

$$T := \begin{bmatrix} 0 & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{8} & 0 & -\frac{1}{4} & -\frac{3}{8} \\ \frac{1}{5} & \frac{2}{5} & 0 & 0 \\ \frac{1}{4} & 0 & -\frac{1}{2} & 0 \end{bmatrix} \tag{1}$$

$c := Vector\left(\left[-\frac{5}{4}, \frac{23}{8}, -\frac{9}{5}, 1\right]\right);$

$$c := \begin{bmatrix} -\frac{5}{4} \\ \frac{23}{8} \\ -\frac{9}{5} \\ 1 \end{bmatrix} \tag{2}$$

$\alpha := Vector([0, 0, 0, 0]);$

$$\alpha := \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3}$$

$Jacobi(T, c, \alpha, 10^{-4}, 100);$

$$\boxed{\begin{bmatrix} -1.999988563 \\ 3.000005627 \\ -1.000036062 \\ 0.9999687134 \end{bmatrix}}, 20 \tag{4}$$

Gauss-Seidel

```
with(LinearAlgebra) :
GaussSeidel :=proc(A, b, α, ε, N)
  local i, j, n, L, D, U, T, c, x, err, temp;
    n := RowDimension(A);
    L := Matrix(n);
    D := Matrix(n);
    U := Matrix(n);
    x := α;

  # Create L
  for j from 1 to n − 1 do
    for i from j + 1 to n do
      L[i, j] := A[i, j];
    end do;
  end do;

  # Create D
  for i from 1 to n do
    D[i, i] := A[i, i];
  end do;

  # Create U
  for i from 1 to n − 1 do
    for j from i + 1 to n do
      U[i, j] := A[i, j];
    end do;
  end do;

  T := − (L + D)^(−1) • U;
  c := (L + D)^(−1) • b;

  for i from 1 to N do
    temp := T • x + c;
    err := Norm(temp − x, infinity);
    x := temp;
    if (err < ε) then
      break;
    end if;
  end do;

 print(evalf(x), i);
 return T;
 end proc:
```

$A := Matrix([[4, 1, 1, 1], [1, 8, 2, 3], [1, 2, -5, 0], [-1, 0, 2, 4]]);$

$$A := \begin{bmatrix} 4 & 1 & 1 & 1 \\ 1 & 8 & 2 & 3 \\ 1 & 2 & -5 & 0 \\ -1 & 0 & 2 & 4 \end{bmatrix} \tag{5}$$

$b := Vector([-5, 23, 9, 4]);$

$$b := \begin{bmatrix} -5 \\ 23 \\ 9 \\ 4 \end{bmatrix} \tag{6}$$

$T2 := GaussSeidel\left(A, b, \alpha, 10^{-4}, 100\right);$

$$\boxed{\begin{bmatrix} -2.000010143 \\ 2.999995446 \\ -1.000003850 \\ 0.9999993894 \end{bmatrix}}, 8$$

$$T2 := \begin{bmatrix} 0 & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ 0 & \frac{1}{32} & -\frac{7}{32} & -\frac{11}{32} \\ 0 & -\frac{3}{80} & -\frac{11}{80} & -\frac{3}{16} \\ 0 & -\frac{7}{160} & \frac{1}{160} & \frac{1}{32} \end{bmatrix} \tag{7}$$

Jacobi

$evalf(Eigenvalues(T));$

$$\begin{bmatrix} 0.3637795831 \\ 0.0960484587 \\ -0.2299140209 + 0.5521692700\,\mathrm{I} \\ -0.2299140209 - 0.5521692700\,\mathrm{I} \end{bmatrix} \quad \textbf{(8)}$$

$eval3 := \mathrm{sqrt}(0.2299140209^2 + 0.552169270^2);$

$eval4 := \mathrm{sqrt}(0.2299140209^2 + 0.552169270^2);$

$$eval3 := 0.5981231978$$

$$eval4 := 0.5981231978 \quad \textbf{(9)}$$

ρ(T) =0.5981231978 < 1, therefore the sequence will converge.

Gauss-Seidel

$evalf(Eigenvalues(T2));$

$$\begin{bmatrix} 0. \\ 0. \\ 0.1388341998 \\ -0.2138341998 \end{bmatrix} \quad \textbf{(10)}$$

$$eval3 := 0.2053959591$$

$$eval4 := 0.2053959591 \quad \textbf{(11)}$$

ρ(T) =0.2138341998 < 1, therefore the sequence will converge.

Actual

$with(Student[NumericalAnalysis]):$

$evalf\left(IterativeApproximate\left(A, b, method = jacobi, initialapprox = \alpha, tolerance = 10^{-4}, maxiterations = 100\right)\right);$

$evalf\left(IterativeApproximate\left(A, b, method = gaussseidel, initialapprox = \alpha, tolerance = 10^{-4}, maxiterations = 100\right)\right);$

$$\begin{bmatrix} -1.999988563 \\ 3.000005627 \\ -1.000036062 \\ 0.9999687134 \end{bmatrix}$$

$$\begin{bmatrix} -2.000010143 \\ 2.999995446 \\ -1.000003850 \\ 0.9999993894 \end{bmatrix} \quad \textbf{(12)}$$

2 Assignment Code

The complete code used for this assignment is provided in the appendix for reference. Files can be accessed directly at this GitHub repository.

Listing 6: Assembly

```
from copy import deepcopy
import numpy as np
import sympy as sp

def global_indexing(width, height=None):
    if height is None:
        height = width
    return np.arange(width * height).reshape((height, width))

def generate_connectivity_matrix(global_indices):
    total_elements = (global_indices.shape[0] - 1) * (global_indices.shape[1] - 1)
    connectivity_matrix = np.zeros((total_elements, 4), dtype=int)
    element = 0
    for i in range(global_indices.shape[0] - 1):
        for j in range(global_indices.shape[1] - 1):
            connectivity_matrix[element, 0] = global_indices[i+1, j]
            connectivity_matrix[element, 1] = global_indices[i+1, j+1]
            connectivity_matrix[element, 2] = global_indices[i, j+1]
            connectivity_matrix[element, 3] = global_indices[i, j]
            element += 1
    return connectivity_matrix

def element_mass_matrix(w, h):
    area = w * h
    Me = (area/36) * np.array([[4, 2, 1, 2],
                               [2, 4, 2, 1],
                               [1, 2, 4, 2],
                               [2, 1, 2, 4]])
    return Me

def element_stiffness_matrix(w, h):
    # When multiplying each entry formula by w * h /16, we get the following reduced formulas:
    wh = w/h
    hw = h/w
    # Main diagonal
    a = (1/3) * (hw + wh)
    # Skew diagonal
    b = (1/6) * (hw - 2*wh)
    # First off-diagonal
```

```
    c = -(1/6) * (2*hw - wh)
    # Second off-diagonal
    d = -(1/6) * (hw + wh)
    # Construct element stiffness matrix
    Ke = np.array([[a, c, d, b],
                   [c, a, b, d],
                   [d, b, a, c],
                   [b, d, c, a]])
    return Ke

def generate_global_coordinates(x_nodes, y_nodes=None):
    if y_nodes is None:
        y_nodes = x_nodes

    y_nodes = y_nodes[::-1]
    global_coordinates = []
    for y in y_nodes:
        for x in x_nodes:
            global_coordinates.append([x, y])
    global_coordinates = np.array(global_coordinates)
    return global_coordinates


def global_assembly(x_nodes, y_nodes=None):
    if y_nodes is None:
        y_nodes = x_nodes

    global_coordinates = generate_global_coordinates(x_nodes, y_nodes)

    width = len(x_nodes)
    height = len(y_nodes)

    global_indices = global_indexing(width, height)
    connectivity_matrix = generate_connectivity_matrix(global_indices)

    num_nodes = width * height
    M_global = np.zeros((num_nodes, num_nodes))
    K_global = np.zeros((num_nodes, num_nodes))

    for element in connectivity_matrix:

        # Bottom-left (node 0)
        x0 = global_coordinates[element[0], 0]
        y0 = global_coordinates[element[0], 1]

        # Bottom-right (node 1)
        x1 = global_coordinates[element[1], 0]
```

```

        # Top-left (node 3)
        y3 = global_coordinates[element[3], 1]

        # Element width and height
        w = abs(x1 - x0)
        h = abs(y3 - y0)

        # Get element matrices
        Me = element_mass_matrix(w, h)
        Ke = element_stiffness_matrix(w, h)

        # Add element contributions to global matrices
        for i_local in range(4):
            i_global = element[i_local]
            for j_local in range(4):
                j_global = element[j_local]
                M_global[i_global, j_global] += Me[i_local, j_local]
                K_global[i_global, j_global] += Ke[i_local, j_local]

    return global_coordinates, M_global, K_global

def classify_boundary_nodes(global_indexing):
    # Create a boundary indicator array where 1 indicates a boundary node
    boundary_indicator = np.zeros_like(global_indexing)
    boundary_indicator[0, :] = 1  # Top boundary
    boundary_indicator[-1, :] = 1  # Bottom boundary
    boundary_indicator[:, 0] = 1  # Left boundary
    boundary_indicator[:, -1] = 1  # Right boundary
    return boundary_indicator.flatten()

def implement_dirichlet_bc(M, K,boundary_indicator):
    num_nodes = M.shape[0]
    for i in range(num_nodes):
        if boundary_indicator[i] == 1:
            M[i, :] = 0
            M[:, i] = 0
            M[i, i] = 1
            K[i, :] = 0
            K[:, i] = 0
            K[i, i] = 1
    return M, K

def u_0(coordinates, boundary_indicator):
    x = coordinates[:, 0]
    y = coordinates[:, 1]
    u = np.sin(2*np.pi*x) * np.sin(2*np.pi*y)
```

```
    u[boundary_indicator == 1] = 0.0  # Apply Dirichlet BCs
    num_nodes = coordinates.shape[0]
    return u.reshape((num_nodes,1))

if __name__ == "__main__":
    width = 5  # Number of nodes along one dimension
    global_indices = global_indexing(width)
    connectivity_matrix = generate_connectivity_matrix(global_indices)

    x_nodes = np.array([-2, -1.6, -1.2, -0.8, 2])
    _, M, K = global_assembly(x_nodes)

    coordinates = generate_global_coordinates(x_nodes)
    boundary_indicator = classify_boundary_nodes(global_indices)
    u = u_0(coordinates, boundary_indicator).reshape(5,5)
    print(u)

    # avoid forming explicit inverse; use solve for M^{-1} K
    M_inv_K = np.linalg.solve(M, K)
    evals = np.linalg.eigvals(M_inv_K)
    max_eigenvalue = np.max(np.abs(evals))
    M_evals = np.linalg.eigvals(M)
    K_evals = np.linalg.eigvals(K)

    with open("./outputs_4/matrices.txt", "w") as f:
        latex_matrix = sp.latex(sp.Matrix(coordinates))
        f.write("Global Coordinates:\n")
        f.write(latex_matrix + "\n\n")

        # print(global_indices)
        latex_matrix = sp.latex(sp.Matrix(global_indices))
        f.write("Global Indices Matrix:\n")
        f.write(latex_matrix + "\n\n")

        # print(connectivity_matrix)
        latex_matrix = sp.latex(sp.Matrix(connectivity_matrix))
        f.write("Connectivity Matrix:\n")
        f.write(latex_matrix + "\n\n")

        M = np.round(M, 4)
        M_split = np.array_split(M, 3, axis=1)
        latex_1_matrix = sp.latex(sp.Matrix(M_split[0]))
        latex_2_matrix = sp.latex(sp.Matrix(M_split[1]))
        latex_3_matrix = sp.latex(sp.Matrix(M_split[2]))
        f.write("Mass Matrix (Part 1):\n")
        f.write(latex_1_matrix + "\n\n")
        f.write("Mass Matrix (Part 2):\n")
```

```
        f.write(latex_2_matrix + "\n\n")
        f.write("Mass Matrix (Part 3):\n")
        f.write(latex_3_matrix + "\n\n")

        K = np.round(K, 4)
        K_split = np.array_split(K, 3, axis=1)
        latex_1_matrix = sp.latex(sp.Matrix(K_split[0]))
        latex_2_matrix = sp.latex(sp.Matrix(K_split[1]))
        latex_3_matrix = sp.latex(sp.Matrix(K_split[2]))
        f.write("Stiffness Matrix (Part 1):\n")
        f.write(latex_1_matrix + "\n\n")
        f.write("Stiffness Matrix (Part 2):\n")
        f.write(latex_2_matrix + "\n\n")
        f.write("Stiffness Matrix (Part 3):\n")
        f.write(latex_3_matrix + "\n\n")

        latex_matrix = sp.latex(sp.Matrix(M_evals))
        f.write("Mass Matrix Evals:\n")
        f.write(latex_matrix + "\n\n")

        latex_matrix = sp.latex(sp.Matrix(K_evals))
        f.write("Stiffness Matrix Evals:\n")
        f.write(latex_matrix + "\n\n")

        latex_matrix = sp.latex(sp.Matrix(evals))
        f.write("M^-1 * K Evals:\n")
        f.write(latex_matrix + "\n\n")

        f.write(f"Maximum Eigenvalue of M^-1 * K: {max_eigenvalue}\n")
```

Listing 7: Implicit Euler Method

```
from assembly import global_assembly, global_indexing, classify_boundary_nodes,
    u_0, implement_dirichlet_bc
import numpy as np
import math
import matplotlib.pyplot as plt

def u_exact(x, y, t, boundary_indicator,nu=0.05):
    u = np.exp(-8 * (np.pi**2) * nu * t) * np.sin(2*np.pi*x) * np.sin(2*np.pi*y
    )
    boundary_indicator = boundary_indicator.reshape(u.shape)
    u[boundary_indicator == 1] = 0.0  # Apply Dirichlet BCs
    return u

def implicit_heat_solver(x_nodes, dt, t_final, nu=0.05):
    global_coordinates, M, K = global_assembly(x_nodes)
    width = len(x_nodes)
```

```
    global_indexing_array = global_indexing(width)
    boundary_indicator = classify_boundary_nodes(global_indexing_array)

    M, K = implement_dirichlet_bc(M, K, boundary_indicator)

    A = M + nu * dt * K

    num_time_steps = math.ceil(t_final / dt)
    dt = t_final / num_time_steps
    prev_U = u_0(global_coordinates, boundary_indicator)
    U = np.array([prev_U])

    for n in range(num_time_steps):
        b = np.matmul(M, U[n])
        next_U = np.linalg.solve(A, b)
        U = np.append(U, [next_U], axis=0)

    return boundary_indicator, U.T, global_coordinates

def convergence_study_timesteps(x_nodes, t_final, dt_values, y_nodes=None):
    if y_nodes is None:
        y_nodes = x_nodes
    errors_implicit = []
    global_coordinates, _, _ = global_assembly(x_nodes, y_nodes)
    width = len(x_nodes)
    global_indexing_array = global_indexing(width)
    boundary = classify_boundary_nodes(global_indexing_array)
    x_coords = global_coordinates[:, 0].reshape(width**2, 1)
    y_coords = global_coordinates[:, 1].reshape(width**2, 1)
    u_exact_values = u_exact(x_coords, y_coords, t_final, boundary)

    for dt in dt_values:
        _, U, _ = implicit_heat_solver(x_nodes, dt, t_final)
        u_numerical_values = U[:, :, -1]
        error = np.linalg.norm(u_numerical_values - u_exact_values, ord=2)
        errors_implicit.append(error)

    return errors_implicit

if __name__ == "__main__":
    plt.rcParams["text.usetex"] = True
    plt.rcParams["axes.grid"] = True
    plt.rc("grid", color="#a6a6a6", linestyle="dotted", linewidth=0.5)
    plt.style.use("seaborn-v0_8-deep")

    x_nodes = np.array([-2, -1.6, -1.2, -0.8, 2])
    # x_nodes = chebyshev_nodes(7) * 2  # Scale to [-2, 2]
```

```
    width = len(x_nodes)
    dt = 0.01
    t_final = 1
    boundary, U, global_coordinates = implicit_heat_solver(x_nodes, dt, t_final
)
    u_approx = U[:,:,-1].reshape((width, width))

    # create plotting mesh consistent with assembly (y reversed in
generate_global_coordinates)
    x_mesh = global_coordinates[:,0].reshape((width, width))
    y_mesh = global_coordinates[:,1].reshape((width, width))
    u_exact_values = u_exact(x_nodes, y_mesh, 1, boundary)

    fig = plt.figure()
    ax1 = fig.add_subplot(121, projection='3d')
    ax1.plot_surface(x_mesh, y_mesh, u_approx, cmap='viridis', alpha=0.8)
    ax1.set_xlabel('x')
    ax1.set_ylabel('y')
    ax1.set_zlabel('u')
    ax1.set_title('Numerical Solution')

    ax2 = fig.add_subplot(122, projection='3d')
    ax2.plot_surface(x_mesh, y_mesh, u_exact_values, cmap='plasma', alpha=0.8)
    ax2.set_xlabel('x')
    ax2.set_ylabel('y')
    ax2.set_zlabel('u')
    ax2.set_title('Exact Solution')

    fig.savefig("./outputs_4/implicit_solution.png", dpi=300)
    plt.show()

    dt_values = np.logspace(-1, -10, 10, base=2)
    errors_implicit = convergence_study_timesteps(x_nodes, t_final, dt_values)
    fig, ax = plt.subplots(figsize=(9,6))
    ax.loglog(dt_values, errors_implicit, marker='x', label='Implicit')
    ax.set_xlabel(r'$\log(\Delta t)$')
    ax.set_ylabel(r'$\log(||u - u_{exact}||_2)$')
    ax.set_title('Convergence Study for Implicit Solver')
    ax.legend()
    fig.savefig("./outputs_4/implicit_convergence_study.png", dpi=300,
bbox_inches='tight')
```

Listing 8: Jacobi Method

```
import numpy as np

def jacobi(A, b, x0=None, tol=1e-12, max_iterations=10000):
    x_prev = np.zeros_like(b) if x0 is None else x0.copy()
```

```
    diagonal_list = np.diag(A)
    D = np.diagflat(diagonal_list)
    R = A - D
    D_inverse = np.linalg.inv(D)

    for _ in range(max_iterations):
        rx_b = -np.dot(R, x_prev) + b
        x_next = np.dot(D_inverse, rx_b)
        if np.linalg.norm(x_next - x_prev, ord=np.inf) < tol:
            return x_next
        x_prev = x_next

    raise ValueError("Jacobi method did not converge within the maximum number of iterations")
```