HW 7

1a.

$$\begin{aligned} u''(x) &= -sin(x) \\ u'(x) &= \int -sin(x)dx = cos(x) + C_1 \\ u(x) &= \int cos(x) + C_1 dx = sin(x) + C_1 x + C_2 \\ u(0) &= 0 = sin(0) + C_1 \cdot 0 + C_2 \\ C_2 &= 0 \\ u'(\pi) &= \frac{1}{2} = cos(\pi) + C_1 \\ C_1 &= \frac{3}{2} \end{aligned} \tag{1}$$

1b.

$$\begin{aligned} \phi_0(x) &= \begin{cases} 1 - \frac{3x}{\pi} & 0 \le x \le \frac{\pi}{3} \\ 0 & \text{otherwise} \end{cases} \\ \phi_0'(x) &= -\frac{3}{\pi} \\ \phi_1(x) &= \begin{cases} \frac{3x}{\pi} & 0 \le x \le \frac{\pi}{3} \\ 2 - \frac{3x}{\pi} & \frac{\pi}{3} < x \le \frac{2\pi}{3} \\ 0 & \text{otherwise} \end{cases} \\ \phi_{1,0}'(x) &= \frac{3}{\pi} \\ \phi_{1,1}'(x) &= -\frac{3}{\pi} \\ \phi_2(x) &= \begin{cases} \frac{3x}{\pi} - 1 & \frac{\pi}{3} \le x \le \frac{2\pi}{3} \\ 3 - \frac{3x}{\pi} & \frac{2\pi}{3} < x \le \pi \\ 0 & \text{otherwise} \end{cases} \\ \phi_{2,1}'(x) &= \frac{3}{\pi} \\ \phi_{2,2}'(x) &= -\frac{3}{\pi} \\ \phi_3(x) &= \begin{cases} \frac{3x}{\pi} - 2 & \frac{2\pi}{3} \le x \le \pi \\ 0 & \text{otherwise} \end{cases} \\ \phi_3'(x) &= \frac{3}{\pi} \end{aligned} \tag{2}$$

$$\begin{aligned} K_{0,0}^1 &= \int_0^{\pi/3} (\frac{-3}{\pi})^2 dx = \frac{3}{\pi} \\ K_{0,1}^1 = K_{10}^1 &= \int_0^{\pi/3} (\frac{-3}{\pi})(\frac{3}{\pi})dx = -\frac{3}{\pi} \\ K^1 &= \begin{bmatrix} \frac{3}{\pi} & -\frac{3}{\pi} \\ -\frac{3}{\pi} & \frac{3}{\pi} \end{bmatrix} \end{aligned} \tag{3}$$

$$\begin{aligned} N_1^1 &= -\frac{3x}{\pi}+1 \\ N_2^1 &= \frac{3x}{\pi} \\ F_1 &= \begin{bmatrix} \int_0^{\pi/3}(sin(x))(-\frac{3x}{\pi}+1)dx \\ \int_0^{\pi/3}(sin(x))(\frac{3x}{\pi})dx \end{bmatrix} \\ &= \begin{bmatrix} 1-\frac{3\sqrt{3}}{2\pi} \\ -\frac{1}{2}+\frac{3\sqrt{3}}{2\pi} \end{bmatrix} \end{aligned} \tag{4}$$

$$\begin{aligned} K^1 = K^2 = K^3 &= \begin{bmatrix} \frac{3}{\pi} & -\frac{3}{\pi} \\ -\frac{3}{\pi} & \frac{3}{\pi} \end{bmatrix} \\ F_2 &= \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \\ F_3 &= \begin{bmatrix} -\frac{1}{2}+\frac{3\sqrt{3}}{2\pi} \\ 1-\frac{3\sqrt{3}}{2\pi} \end{bmatrix} \end{aligned} \tag{5}$$

$$\begin{aligned} K &= \begin{bmatrix} 3/\pi & -3/\pi & 0 & 0 \\ -3/\pi & 6/\pi & -3/\pi & 0 \\ 0 & -3/\pi & 6/\pi & -3/\pi \\ 0 & 0 & -3/\pi & 3/\pi \end{bmatrix} \\ F &= \begin{bmatrix} 0.17300666 \\ 0.82699334 \\ 0.82699334 \\ 0.17300666 \end{bmatrix} \end{aligned} \tag{6}$$

Now applying boundary conditions:

$$\begin{aligned} K[0:,] &= 0 \\ K[0,0] &= 1 \\ F[0] &= 1 \\ F[-1]+ &= 1/2 \end{aligned}$$

$$K = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3/\pi & 6/\pi & -3/\pi & 0 \\ 0 & -3/\pi & 6/\pi & -3/\pi \\ 0 & 0 & -3/\pi & 3/\pi \end{bmatrix} \tag{7}$$

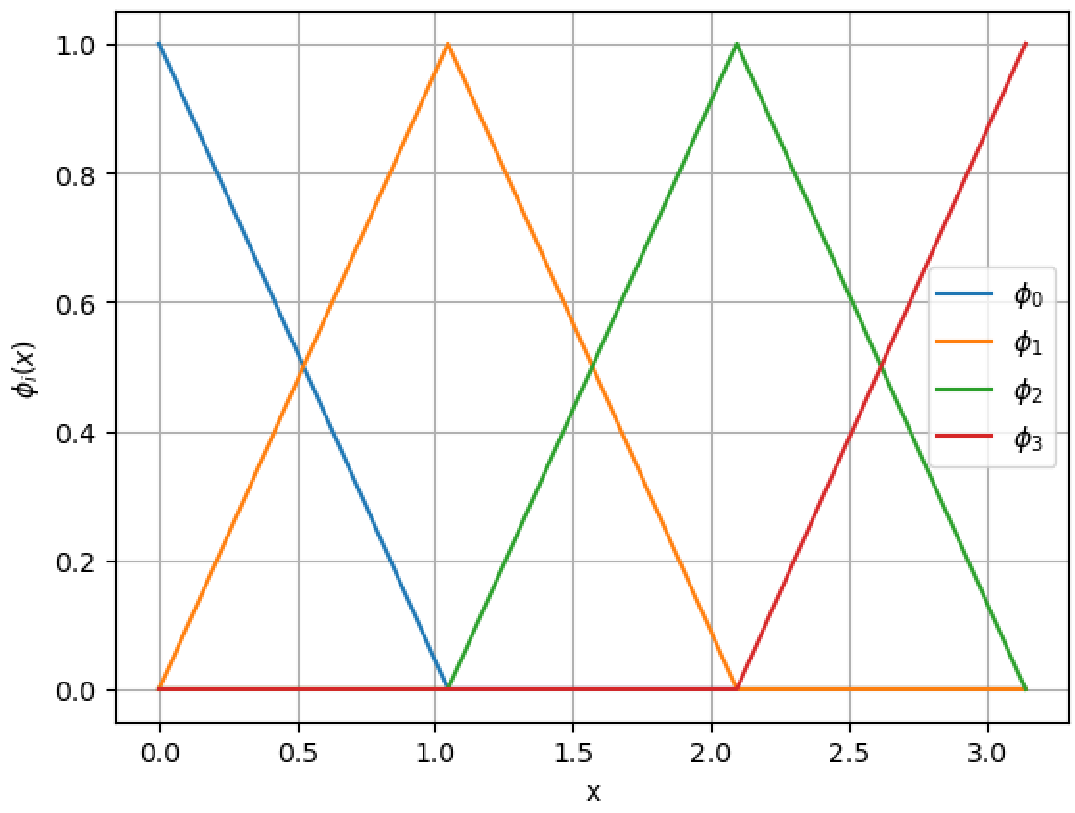
$$F = \begin{bmatrix} 1 \\ 0.82699334 \\ 0.82699334 \\ 0.67300666 \end{bmatrix}$$

And solving for u:

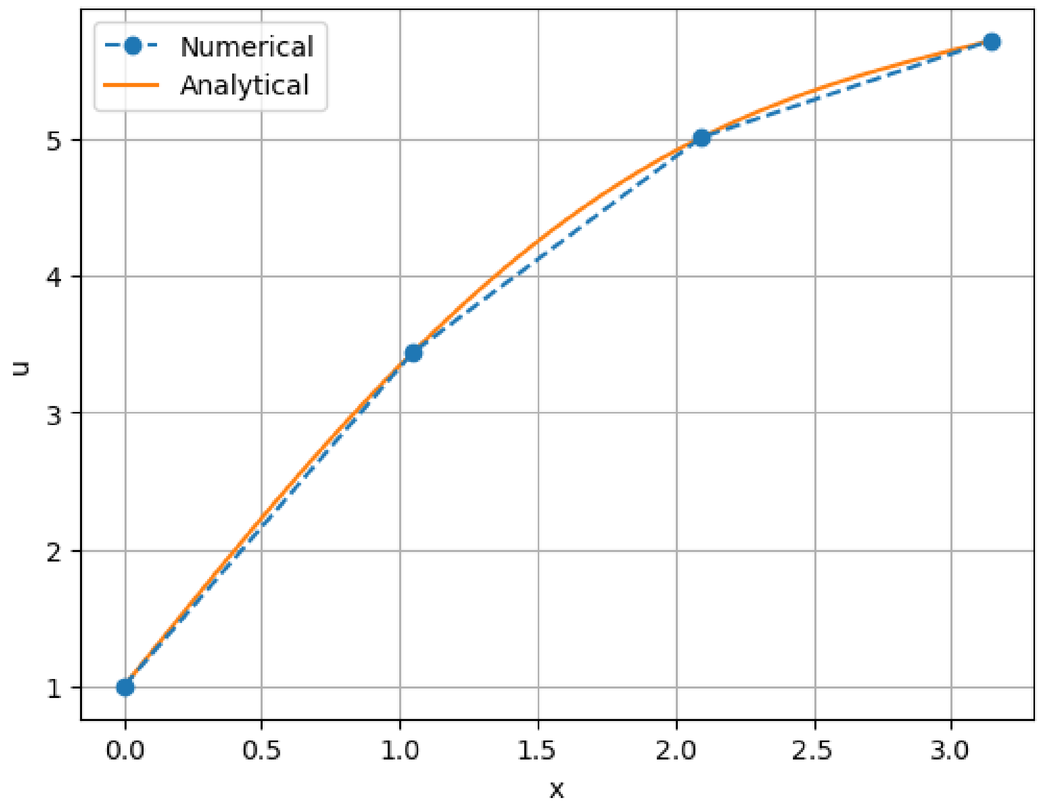
$$\begin{aligned} Ku &= F \\ u &= K^{-1}F \\ u &= \begin{bmatrix} 1.0 \\ 3.43682173 \\ 5.00761806 \\ 5.71238898 \end{bmatrix} \end{aligned} \tag{8}$$

1c.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad


def phi0(x):
    return np.where((0 <= x) & (x <= np.pi / 3), 1 - 3 * x / np.pi, 0)


def phi1(x):
    return np.where(
        (0 <= x) & (x <= np.pi / 3),
        3 * x / np.pi,
        np.where((np.pi / 3 < x) & (x <= 2 * np.pi / 3), 2 - 3 * x / np.pi, 0),
    )


def phi2(x):
    return np.where(
        (np.pi / 3 <= x) & (x <= 2 * np.pi / 3),
        3 * x / np.pi - 1,
        np.where((2 * np.pi / 3 < x) & (x <= np.pi), 3 - 3 * x / np.pi, 0),
    )


def phi3(x):
    return np.where((2 * np.pi / 3 <= x) & (x <= np.pi), 3 * x / np.pi - 2, 0)


x = np.linspace(0, np.pi, 100)
plt.plot(x, phi0(x), label=r"$\phi_0$")
plt.plot(x, phi1(x), label=r"$\phi_1$")
plt.plot(x, phi2(x), label=r"$\phi_2$")
plt.plot(x, phi3(x), label=r"$\phi_3$")
plt.xlabel("x")
plt.ylabel(r"$\phi_i(x)$")
plt.grid()
plt.legend()
plt.show()

mesh = np.array([0, np.pi / 3, 2 * np.pi / 3, np.pi])
# mesh = np.linspace(0, np.pi, 1000)
f = lambda x: np.sin(x)
u0 = 1
numel = len(mesh) - 1
numnp = len(mesh)
nen = 2
xmin = mesh[0]
xmax = mesh[-1]
x = mesh.copy()
LM = np.zeros((nen, numel), dtype=int)
for e in range(numel):
    LM[0, e] = e
    LM[1, e] = e + 1
print(LM)

K = np.zeros((numnp, numnp))
F = np.zeros(numnp)

for e in range(numel):
    i1 = LM[0, e]
    i2 = LM[1, e]
    x1 = x[i1]
    x2 = x[i2]
    dxe = x2 - x1
    Ke = np.array([[1, -1], [-1, 1]]) * (1 / dxe)
    N1 = lambda x: (x2 - x) / dxe
    N2 = lambda x: (x - x1) / dxe
    Fe = np.zeros(nen)
    F1 = quad(lambda x: N1(x) * f(x), x1, x2)[0]
    F2 = quad(lambda x: N2(x) * f(x), x1, x2)[0]
    Fe[0] = F1
    Fe[1] = F2

    for i in range(nen):
        I = LM[i, e]
        F[I] += Fe[i]
        for j in range(nen):
            J = LM[j, e]
            K[I, J] += Ke[i, j]

# Apply Dirichlet boundary condition at x = 0
K[0, :] = 0
K[0, 0] = 1
F[0] = u0

# Apply Neumann boundary condition at x = pi
F[-1] += 1 / 2
u = np.linalg.solve(K, F)

# Analytical solution
C1 = 3 / 2
C2 = 1
u_analytical = lambda x: np.sin(x) + C1 * x + C2
xfine = np.linspace(xmin, xmax, 100)
# Plot the solution
plt.plot(x, u, label="Numerical", marker="o", linestyle="--", zorder=10)
plt.plot(xfine, u_analytical(xfine), label="Analytical")
plt.xlabel("x")
plt.ylabel("u")
plt.legend()
plt.grid()
plt.show()
```

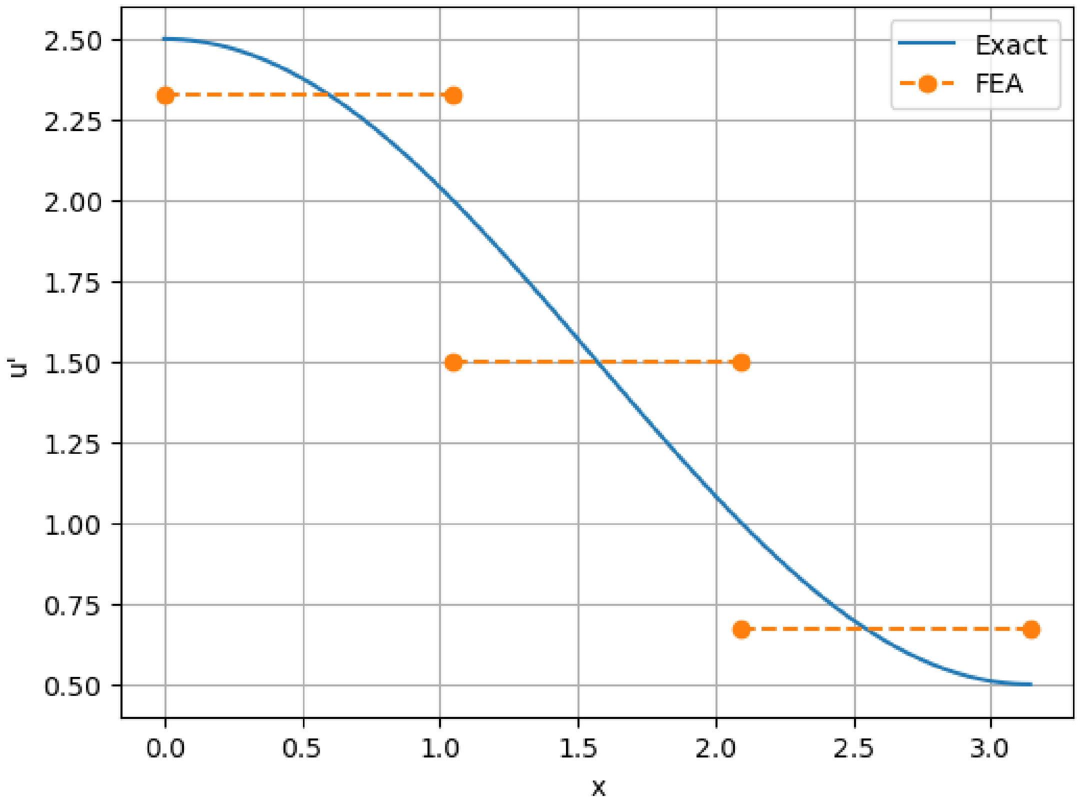


```
[[0 1 2]
 [1 2 3]]
```



1d.

```python
du_analytical = lambda x: np.cos(x) + C1
xfine = np.linspace(xmin, xmax, 100)

plt.figure()
plt.plot(xfine, du_analytical(xfine), label="Analytical")
for e in range(numel):
    # Global element nodes
    i1 = LM[0, e]
    i2 = LM[1, e]

    # Element coordinates
    x1 = x[i1]
    x2 = x[i2]

    # Element displacements
    u1 = u[i1]
    u2 = u[i2]

    # Element derivative: constant when using linear basis functions
    du_fea = (u2 - u1) / (x2 - x1)

    # Add to the plot
    plt.plot([x1, x2], [du_fea, du_fea], linestyle="--", marker="o", color="C1")

plt.xlabel("x")
plt.ylabel("u'")
plt.legend(["Exact", "FEA"])
plt.grid()
plt.show()
```



2.

```
import numpy as np


def heat2d_explicit(L, t_end, Nx, Ny, Nt, alpha=1):
    """
    Solve the 2D transient heat equation with the forward Euler method
    and return the solution as a 3D array (one for each time step).

    Parameters:
    -----------
    L: float
        Length of the square domain
    t_end: float
        End time
    Nx: int
        Number of grid points in the x direction
    Ny: int
        Number of grid points in the y direction
    Nt: int
        Number of time steps
    alpha: float
        Diffusion coefficient

    Returns:
    --------
    x: array, shape (Nx + 1,)
        Grid points in the x direction
    y: array, shape (Ny + 1,)
        Grid points in the y direction
    t: array, shape (Nt + 1,)
        Time points
    T: array, shape (Nx + 1, Ny + 1, Nt + 1)
        Temperature at each point in space and time
    """

    # Spatial discretization
    dx = L / Nx
    dy = L / Ny
    x = np.linspace(0, L, Nx + 1)
    y = np.linspace(0, L, Ny + 1)

    # Temporal discretization
    dt = t_end / Nt
    t = np.linspace(0, t_end, Nt + 1)

    # Stability condition check
    if alpha * dt / max(dx**2, dy**2) > 0.5:
        raise ValueError("Stability condition violated")

    # Initialize temperature array
    T = np.zeros((Nx + 1, Ny + 1, Nt + 1))

    # Initial condition
    T[:, 0, 0] = x * (L - x)  # Bottom edge

    # Time stepping
    for n in range(0, Nt):
        # Internal grid points
        for i in range(1, Nx):
            for j in range(1, Ny):
                T[i, j, n + 1] = T[i, j, n] + alpha * dt * (
                    (T[i + 1, j, n] - 2 * T[i, j, n] + T[i - 1, j, n]) / dx**2
                    + (T[i, j + 1, n] - 2 * T[i, j, n] + T[i, j - 1, n]) / dy**2
                )

        # Boundary conditions
        T[0, :, n + 1] = 0  # Left edge
        T[:, Ny, n + 1] = 0  # Top edge
        T[Nx, 1:Ny, n + 1] = T[Nx - 1, 1:Ny, n + 1]  # Right edge (Neumann)
        T[:, 0, n + 1] = x * (L - x)  # Bottom edge

    return x, y, t, T
```

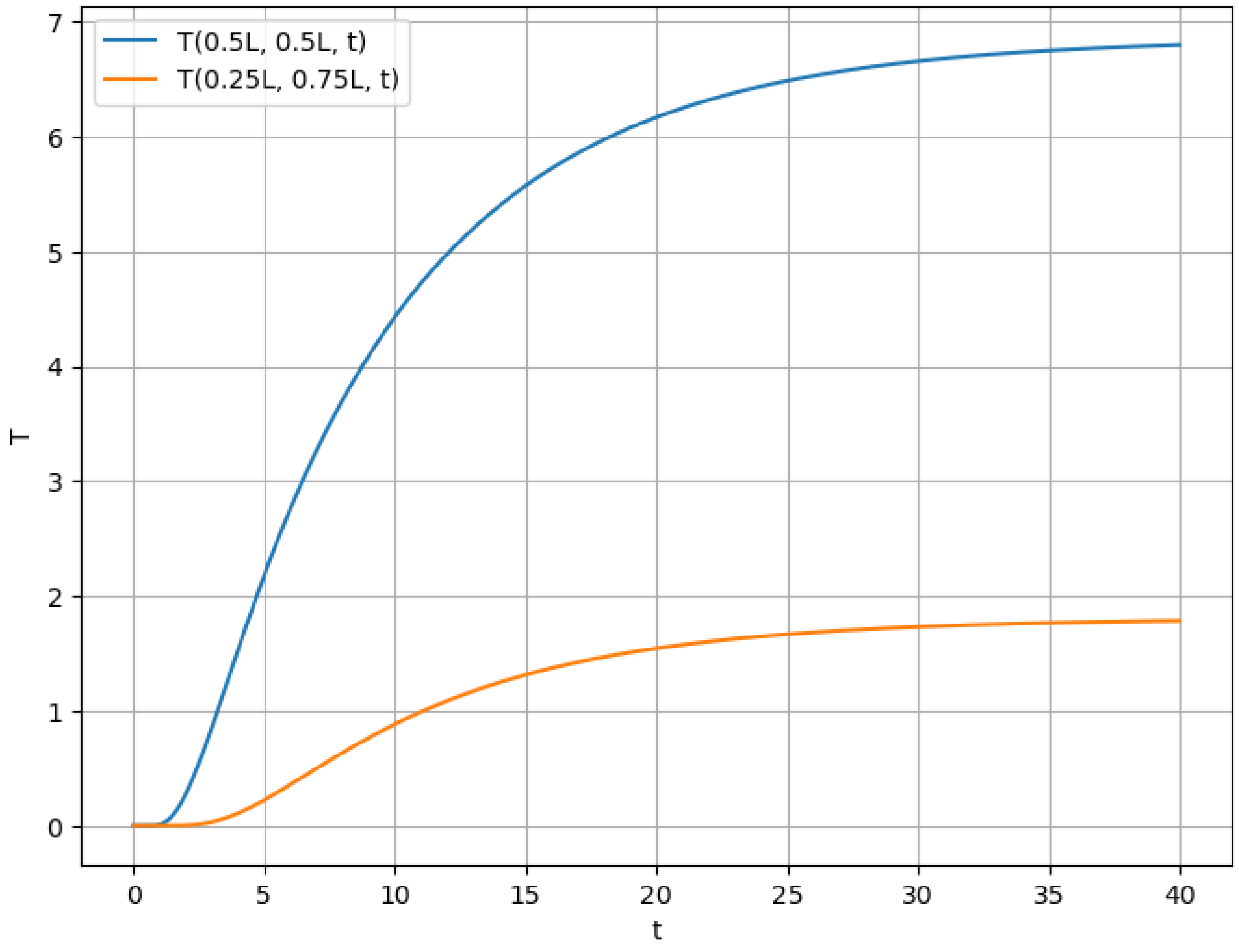
2.1

```
L = 10
Nx = 40
Ny = 40
tend = 40

alpha = 1
dx = L / Nx
dt = dx**2 / (4 * alpha)
Nt = int(tend / dt)

print(f"Maximum stable time step: {dt:.3f} (Nt = {Nt})")

# Compute solution
x, y, t, T = heat2d_explicit(L, tend, Nx, Ny, Nt, alpha)
```

```
Maximum stable time step: 0.016 (Nt = 2560)
```

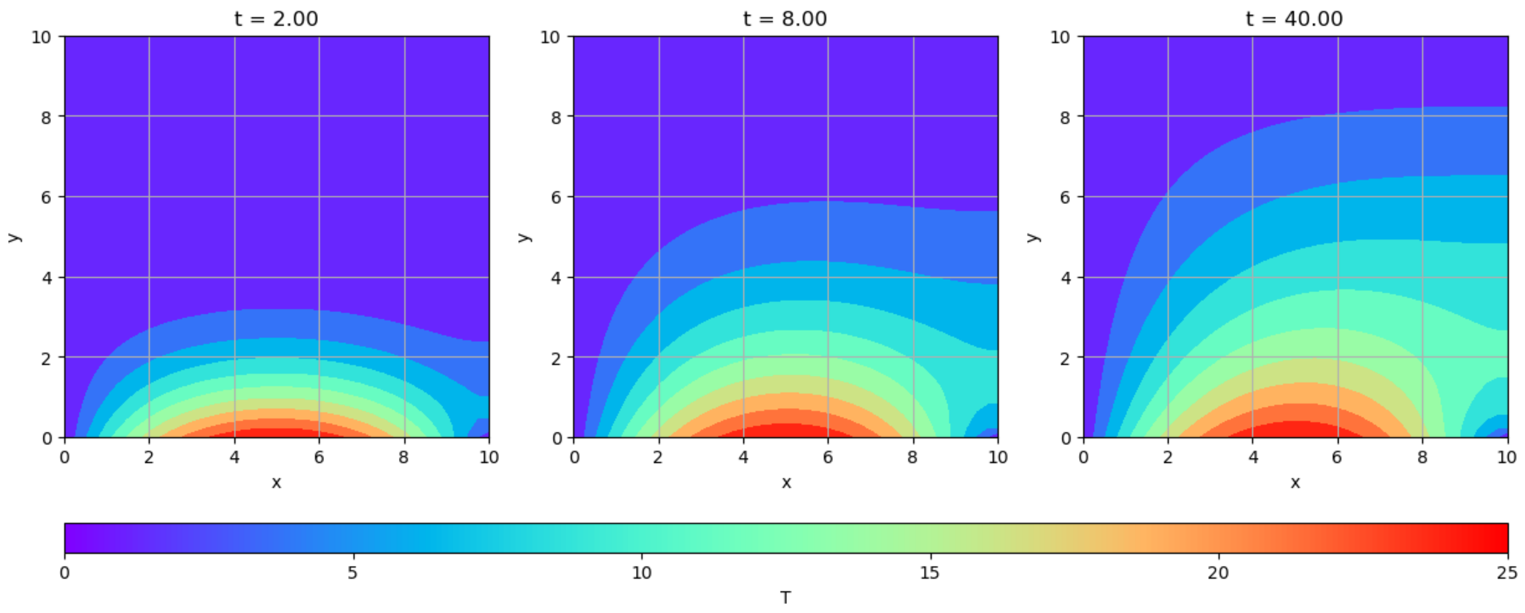
2.2

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))
plt.plot(t, T[20, 20, :], label="T(0.5L, 0.5L, t)")
plt.plot(t, T[10, 30, :], label="T(0.25L, 0.75L, t)")
plt.xlabel("t")
plt.ylabel("T")
plt.legend()
plt.grid()
plt.show()
```



2.3

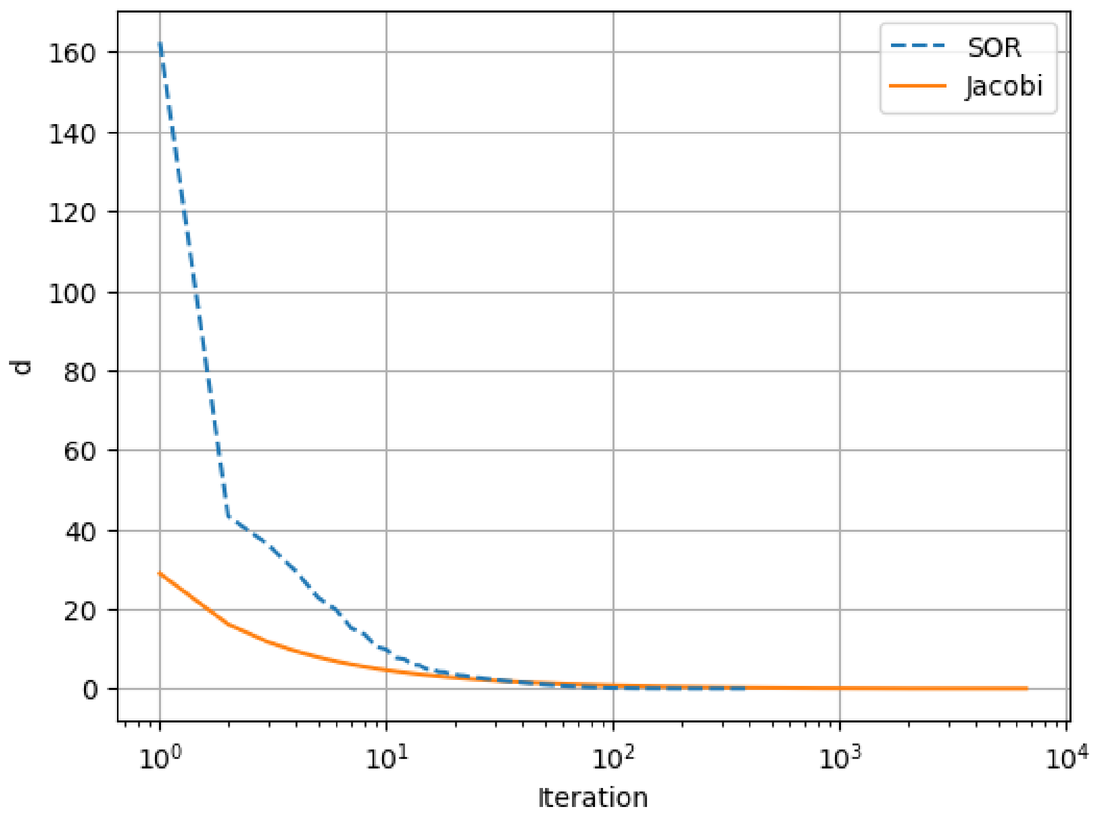
```
import matplotlib.cm as cm

X, Y = np.meshgrid(x, y)
contour_regions = np.linspace(0, np.max(T), 11)
n1 = int(2 / dt)
n2 = int(8 / dt)
n3 = int(40 / dt)

fig, ax = plt.subplots(1, 3, figsize=(15, 6))
for i, n in enumerate([n1, n2, n3]):
    ax[i].contourf(X, Y, T[:, :, n].T, levels=contour_regions, cmap="rainbow")
    ax[i].set_xlabel("x")
    ax[i].set_ylabel("y")
    ax[i].set_title(f"t = {t[n]:.2f}")
    ax[i].grid()

norm = cm.colors.Normalize(vmin=0, vmax=np.max(T))
cbar = fig.colorbar(
    cm.ScalarMappable(norm=norm, cmap="rainbow"),
    ax=ax,
    orientation="horizontal",
    pad=0.15,
    aspect=50,
)
cbar.set_label("T")
plt.show()
```



3.

```
import numpy as np


def SORitersolve(L, N, omega, tol, maxiter, print_iters=False):
    """
    Solve the 2D steady state heat equation with the point SOR method
    and return the solution as a 2D array.

    Parameters:
    -----------
    L: float
        Length of the square domain
    N: int
        Number of grid points in each direction
    omega: float
        Relaxation parameter
    tol: float
        Convergence tolerance
    maxiter: int
        Maximum number of iterations
    print_iters: bool
        Print convergence behavior

    Returns:
    --------
    x: array, shape (N+1,)
        Grid points in the x direction
    y: array, shape (N+1,)
        Grid points in the y direction
    T: array, shape (N+1, N+1)
        Temperature at each point in space
    iter: int
        Number of iterations
    dvals: array, shape (iter,)
        Convergence behavior
    """

    # Discretization
    x = np.linspace(0, L, N + 1)
    y = np.linspace(0, L, N + 1)

    # Initialize
    T = np.zeros((N + 1, N + 1))
    T[:, 0] = x * (L - x)

    # Track convergence
    dvals = np.zeros(maxiter)

    # Point Jacobi
    converged = False
    iter = 0
    while not converged:
        # The 'old' guess (iteration k) is saved as the result from the previous guess
        Told = np.copy(T)

        # Iteration counter
        iter += 1

        # Loop over all nodes
        for i in range(N + 1):
            for j in range(N + 1):
                # Dirichlet BCs (left, bottom, top edges)
                if i == 0 or j == 0 or j == N:
                    continue

                # Neumann on right edge: dT/dx = 0
                elif i == N:
                    That = Told[i - 1, j]
                    T[i, j] = Told[i, j] + omega * (That - Told[i, j])

                # Interior nodes affected by the Neumann condition: i=N-1
                elif i == N - 1:
                    That = (1 / 3) * (T[i - 1, j] + Told[i, j + 1] + T[i, j - 1])

                    T[i, j] = Told[i, j] + omega * (That - Told[i, j])

                # All other interior nodes
                else:
                    That = (1 / 4) * (
                        Told[i + 1, j] + T[i - 1, j] + Told[i, j + 1] + T[i, j - 1]
                    )

                    # SOR update
                    T[i, j] = Told[i, j] + omega * (That - Told[i, j])

        # Check if converged
        d = np.linalg.norm(T - Told)
        dvals[iter - 1] = d

        # Print convergence behavior (helpful for debugging)
        if print_iters:
            print(iter, d)

        # Accept or reject current solution
        if d < tol:
            converged = True
        elif iter == maxiter:
            print("no convergence")
            break

    return x, y, T, iter - 1, dvals


def Jacobitersolve(L, N, omega, tol, maxiter, print_iters=False):
    # Discretization
    x = np.linspace(0, L, N + 1)
    y = np.linspace(0, L, N + 1)

    # Initialize
    T = np.zeros((N + 1, N + 1))
    T[:, 0] = x * (L - x)

    # Track convergence
    dvals = np.zeros(maxiter)

    # Point Jacobi
    converged = False
    iter = 0
    while not converged:
        # The 'old' guess (iteration k) is saved as the result from the previous guess
        Told = np.copy(T)

        # Iteration counter
        iter += 1

        # Loop over all nodes
        for i in range(N + 1):
            for j in range(N + 1):
                # Dirichlet BCs (left, bottom, top edges)
                if i == 0 or j == 0 or j == N:
                    continue

                # Neumann on right edge: dT/dx = 0
                elif i == N:
                    T[i, j] = Told[i - 1, j]

                # Interior nodes affected by the Neumann condition: i=N-1
                elif i == N - 1:
                    T[i, j] = (1 / 3) * (
                        Told[i - 1, j] + Told[i, j + 1] + Told[i, j - 1]
                    )

                # All other interior nodes
                else:
                    T[i, j] = (1 / 4) * (
                        Told[i + 1, j]
                        + Told[i - 1, j]
                        + Told[i, j + 1]
                        + Told[i, j - 1]
                    )

        # Check if converged
        d = np.linalg.norm(T - Told)
        dvals[iter - 1] = d

        # Print convergence behavior (helpful for debugging)
        if print_iters:
            print(iter, d)

        # Accept or reject current solution
        if d < tol:
            converged = True
        elif iter == maxiter:
            print("no convergence")
            break

    return x, y, T, iter, dvals
```

```
L = 10
N = 40
tol = 1e-6
maxiter = 10000
omega = 1.8

xSOR, ySOR, TSOR, iterSOR, dvalsSOR = SORitersolve(L, N, omega, tol, maxiter)
xJac, yJac, TJac, iterJac, dvalsJac = Jacobitersolve(L, N, omega, tol, maxiter)

plt.semilogx(
    np.arange(1, iterSOR + 1),
    dvalsSOR[:iterSOR],
    label="SOR",
    linestyle="--",
    zorder=10,
)
plt.semilogx(np.arange(1, iterJac + 1), dvalsJac[:iterJac], label="Jacobi")

plt.xlabel("Iteration")
plt.ylabel("d")
plt.legend()
plt.grid()
plt.show()
```



4

LM = [[1, 2, 4], [2, 3, 4], [3, 5, 4], [5, 6, 4]]

$$K_e = \frac{A_e}{4}\begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$

Where $A_e = 0.125$

$$K_e == \begin{bmatrix} 0.0625 & -0.03125 & -0.03125 \\ -0.03125 & 0.0625 & -0.03125 \\ -0.03125 & -0.03125 & 0.0625 \end{bmatrix}$$

$$K_e = \begin{bmatrix} 1.0 & -0.5 & -0.5 \\ -0.5 & 1 & -0.5 \\ -0.5 & -0.5 & 1 \end{bmatrix}$$

$$F_e = \frac{f \times A_e}{3}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$F_e = \begin{bmatrix} 0.04167 \\ 0.04167 \\ 0.04167 \end{bmatrix}$$

$$K = \begin{bmatrix} 0.0625 & -0.03125 & 0 & -0.03125 & 0 & 0 \\ -0.03125 & 0.125 & -0.03125 & -0.0625 & 0 & 0 \\ 0 & -0.03125 & 0.125 & -0.0625 & -0.03125 & 0 \\ -0.03125 & -0.0625 & -0.0625 & 0.25 & -0.0625 & -0.03125 \\ 0 & 0 & -0.03125 & -0.0625 & 0.125 & -0.03125 \\ 0 & 0 & 0 & -0.03125 & -0.03125 & 0.0625 \end{bmatrix}$$

$$F = \begin{bmatrix} 0.04167 \\ 0.08333 \\ 0.08333 \\ 0.16667 \\ 0.08333 \\ 0.04167 \end{bmatrix}$$

```python
A_e = 0.125
f = 1
LM = [[1, 2, 4], [2, 3, 4], [3, 5, 4], [5, 6, 4]]
size = 6
K_e = (A_e / 4) * np.array([[2, -1, -1], [-1, 2, -1], [-1, -1, 2]])

F_e = (f * A_e / 3) * np.array([1, 1, 1])

K_global = np.zeros((size, size))
F_global = np.zeros(size)

for element_nodes in LM:
    for i in range(3):
        for j in range(3):
            K_global[element_nodes[i] - 1, element_nodes[j] - 1] += K_e[i, j]
        F_global[element_nodes[i] - 1] += F_e[i]

dirichlet_nodes = [1, 2, 3]

# Apply  boundary conditions
for node in dirichlet_nodes:
    K_global[node - 1, :] = 0
    K_global[:, node - 1] = 0
    K_global[node - 1, node - 1] = 1
    F_global[node - 1] = 0


U = np.linalg.solve(K_global, F_global)
U
```

```
array([0.        , 0.        , 0.        , 1.46666667, 2.        ,
       2.4       ])
```