LAB 7

1.Laplace equation using Gauss-Seidal iteration

Working principle

The Laplace equation is a second-order partial differential equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

It describes steady-state heat conduction, electrostatic potentials, and other equilibrium phenomena.

Finite Difference Approach

1. Discretize the rectangular domain into a grid with spacing hx and hy along the x- and y-axes.
2. Approximate the partial derivatives with finite difference formulas:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2}$$

3. The Laplace equation becomes:

$u_{i,j} = \frac{u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}}{4}$ (assuming uniform grid spacing, hx= hy).

4. Use Gauss-Seidel Iteration:
   - Update each grid point using the above formula.
   - Iteratively update until the solution converges (error between consecutive iterations is below a threshold).

Pseudocode

1. Input:
   - Domain size [xmin,xmax][ymin,ymax].
   - Boundary conditions u(x,y)at the edges.
   - Grid resolution (nx ,ny).
   - Convergence criterion ϵ.
2. Discretize the domain:
   - Create a grid with spacing hx, hy.
   - Initialize grid values u(x,y), satisfying boundary conditions.
3. Gauss-Seidel Iteration:
   - For each interior grid point (i,j), update:

$$u_{i,j}{}^{k+1} = \frac{u_{i+1,j}{}^{k} + u_{i-1,j}{}^{k+1} + u_{i,j+1}{}^{k} + u_{i,j-1}{}^{k+1}}{4}$$

- Check convergence: Stop when: $\max(|u^{k+1}-u^{k}|)<\epsilon$

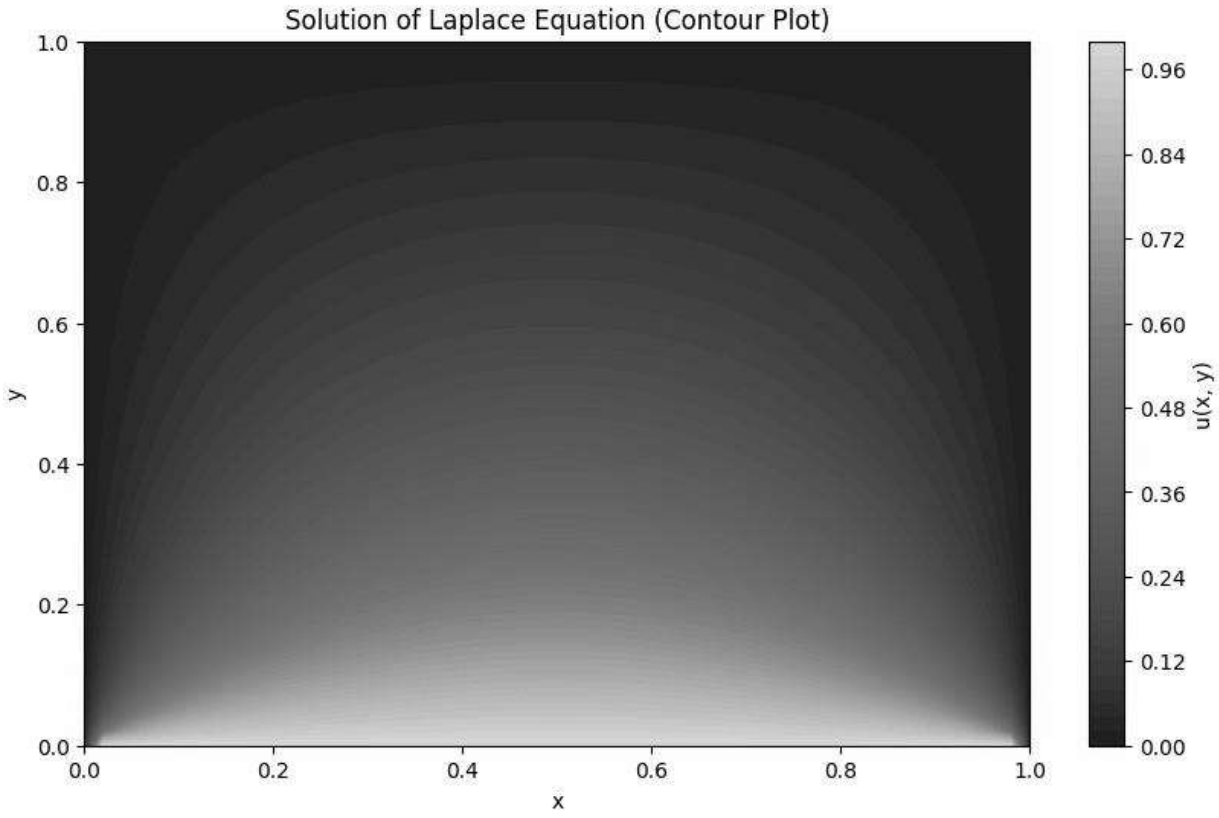
4. Output:
   - Final grid values u(x,y).
   - Visualize the results using a 3D surface plot or heatmap.

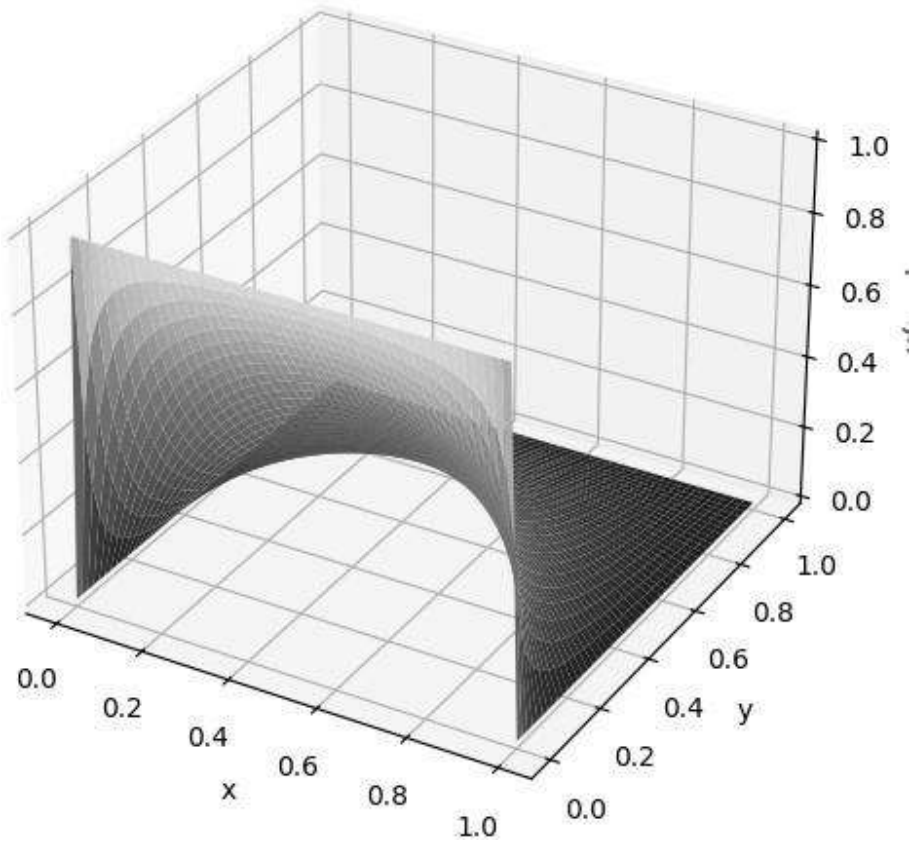
5.

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

# Define the Laplace solver using Gauss-Seidel iteration
def solve_laplace_gauss_seidel(domain, boundary_conditions, nx, ny,
tol):
    """
    Solve the Laplace equation using Gauss-Seidel iteration.

    Parameters:
    domain: tuple (xmin, xmax, ymin, ymax) defining the problem domain
    boundary_conditions: dictionary with keys "top", "bottom", "left",
"right" specifying boundary values
    nx, ny: number of grid points along x and y axes
    tol: convergence tolerance

    Returns:
    u: 2D array of solution values
    x, y: grid points
    """
    # Unpack domain
```

```
    xmin, xmax, ymin, ymax = domain
    hx = (xmax - xmin) / (nx - 1)
    hy = (ymax - ymin) / (ny - 1)

    # Create grid
    x = np.linspace(xmin, xmax, nx)
    y = np.linspace(ymin, ymax, ny)
    u = np.zeros((ny, nx))

    # Apply boundary conditions
    u[0, :] = boundary_conditions["top"]  # Top boundary
    u[-1, :] = boundary_conditions["bottom"]  # Bottom boundary
    u[:, 0] = boundary_conditions["left"]  # Left boundary
    u[:, -1] = boundary_conditions["right"]  # Right boundary

    # Iterative solution using Gauss-Seidel
    max_iter = 10000
    for iteration in range(max_iter):
        u_old = u.copy()

        # Update interior points
        for i in range(1, ny-1):
            for j in range(1, nx-1):
                u[i, j] = 0.25 * (u[i+1, j] + u[i-1, j] + u[i, j+1] +
u[i, j-1])

        # Check convergence
        error = np.max(np.abs(u - u_old))
        if error < tol:
            print(f"Converged after {iteration} iterations with error
{error}")
            break

    return x, y, u

# Define problem parameters
domain = (0, 1, 0, 1)  # [xmin, xmax, ymin, ymax]
boundary_conditions = {
    "top": 1,     # u(x, y=1) = 1
    "bottom": 0, # u(x, y=0) = 0
    "left": 0,    # u(x=0, y) = 0
    "right": 0    # u(x=1, y) = 0
}
nx, ny = 50, 50  # Number of grid points
tol = 1e-6  # Convergence tolerance

# Solve the Laplace equation
x, y, u = solve_laplace_gauss_seidel(domain, boundary_conditions, nx,
ny, tol)
```

```
# Visualization
X, Y = np.meshgrid(x, y)

plt.figure(figsize=(10, 6))
plt.contourf(X, Y, u, levels=50, cmap="viridis")
plt.colorbar(label="u(x, y)")
plt.title("Solution of Laplace Equation (Contour Plot)")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, u, cmap="viridis")
ax.set_title("Solution of Laplace Equation (3D Surface Plot)")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("u(x, y)")
plt.show()

Converged after 1792 iterations with error 9.987875817518699e-07
```




Solution of Laplace Equation (3D Surface Plot)
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
x
0.0
0.2
0.4
0.6
0.8
1.0
y

2.Poisson's equation using Gauss-Seidal iteration

Working Principle

Working principle

The Poisson equation is a second-order partial differential equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

where u(x,y) is the unknown function, and f(x,y) is the source term that represents the distribution of sources or sinks..

Finite Difference Approach

5. Discretize the rectangular domain into a grid with uniform spacing hx=hy =h..
6. Approximate the partial derivatives with finite difference formulas:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}$$

7. The Poisson's equation becomes:

$$u_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - h^2 f_{i,j}}{4}$$

8. Use Gauss-Seidel Iteration:

   Use the above formula iteratively to update the value of $u_{i,j}$at each grid point until the solution converges.

   Convergence is achieved when the maximum difference between successive iterations is less than a specified tolerance (ϵ\epsilonϵ).

Pseudocode

6. Input:
   - Domain size [xmin,xmax][ymin,ymax].
   - Boundary conditions u(x,y)at the edges.
   - Source term f(x,y)
   - Grid resolution (nx ,ny).
   - Convergence criterion ϵ.
7. Discretize the domain:
   - Create a grid with spacing hx, hy.
   - Initialize grid values u(x,y), satisfying boundary conditions.
8. Gauss-Seidel Iteration:
   - For each interior grid point (i,j), update:

$$u_{i,j}{}^{k+1} = \frac{u_{i+1,j}{}^{k} + u_{i-1,j}{}^{k+1} + u_{i,j+1}{}^{k} + u_{i,j-1}{}^{k+1} - h^2 f_{i,j}}{4}$$

- Check convergence: Stop when: $\max(|u^{k+1} - u^{k}|) < \epsilon$

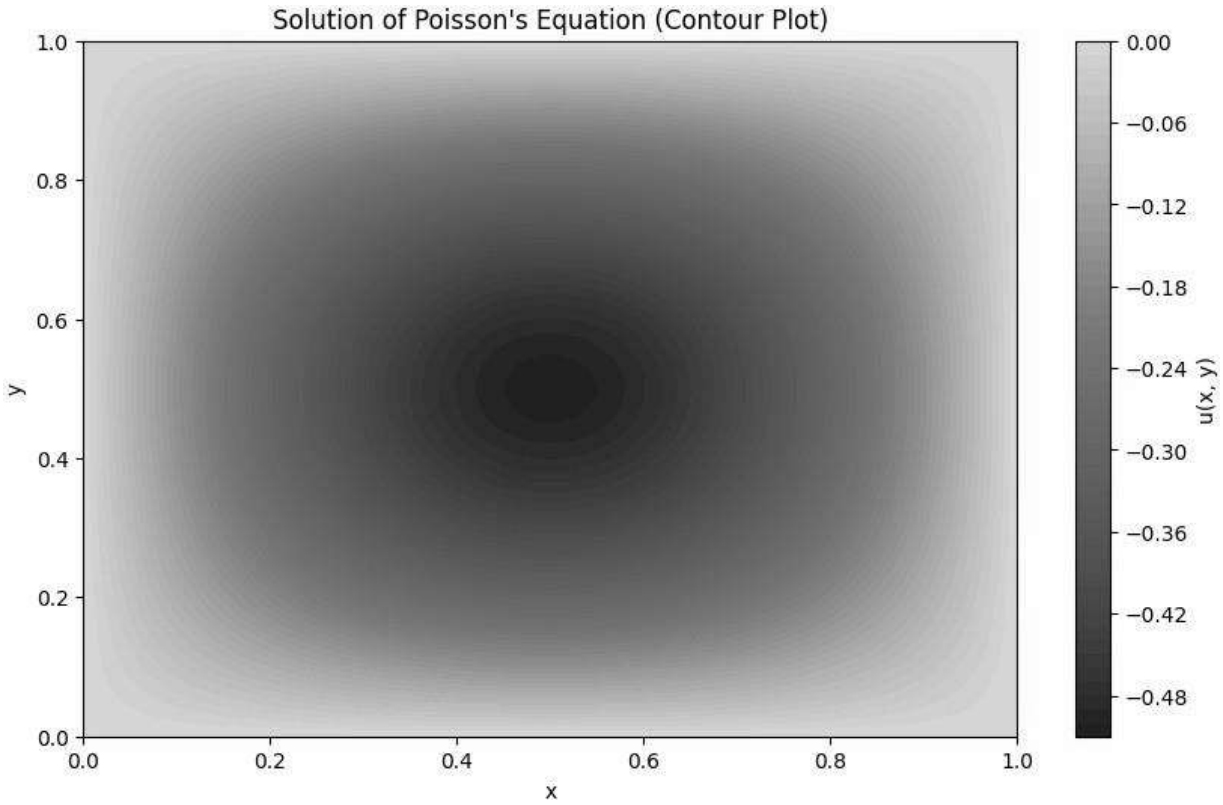
9. Output:
   - Final grid values u(x,y).
   - Visualize the results using contour plots and 3D surface plot.

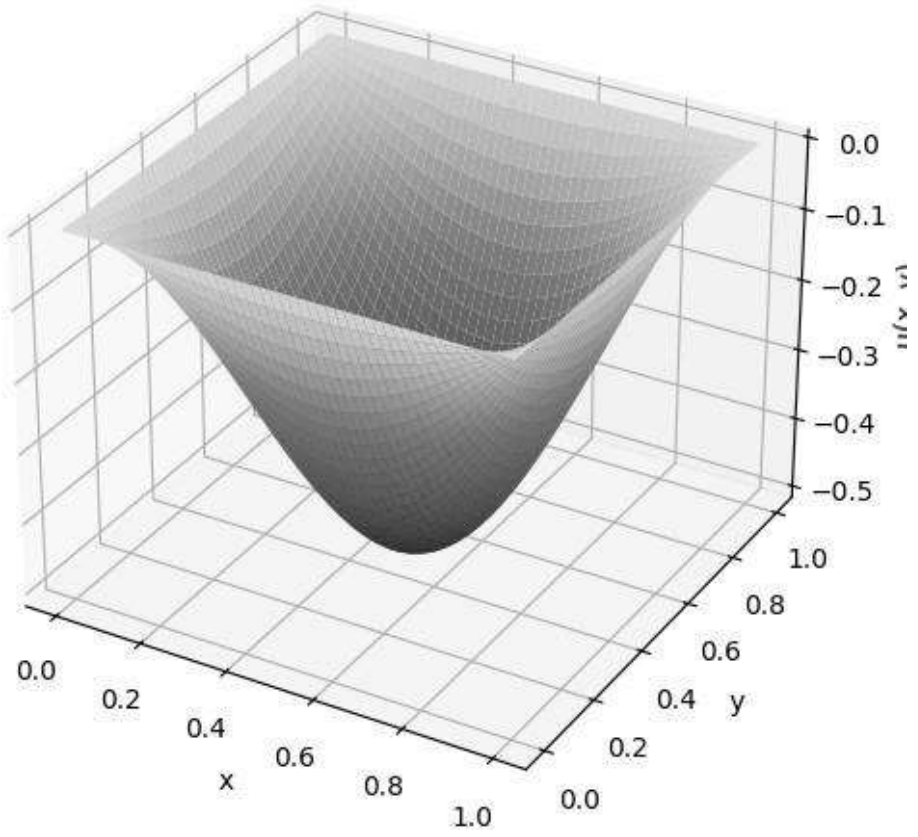
Final grid values u(x,y). Visualize the results using contour plots and 3D surface plots.

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

# Define the Poisson solver using Gauss-Seidel iteration
def solve_poisson_gauss_seidel(domain, boundary_conditions, f, nx, ny,
tol):
    """
    Solve the Poisson equation using Gauss-Seidel iteration.

    Parameters:
    domain: tuple (xmin, xmax, ymin, ymax) defining the problem domain
    boundary_conditions: dictionary with keys "top", "bottom", "left",
"right" specifying boundary values
    f: function representing the source term f(x, y)
    nx, ny: number of grid points along x and y axes
    tol: convergence tolerance

    Returns:
    u: 2D array of solution values
    x, y: grid points
```

```
    """
    # Unpack domain
    xmin, xmax, ymin, ymax = domain
    hx = (xmax - xmin) / (nx - 1)
    hy = (ymax - ymin) / (ny - 1)
    h = hx  # Assuming uniform grid spacing

    # Create grid
    x = np.linspace(xmin, xmax, nx)
    y = np.linspace(ymin, ymax, ny)
    u = np.zeros((ny, nx))

    # Apply boundary conditions
    u[0, :] = boundary_conditions["top"]  # Top boundary
    u[-1, :] = boundary_conditions["bottom"]  # Bottom boundary
    u[:, 0] = boundary_conditions["left"]  # Left boundary
    u[:, -1] = boundary_conditions["right"]  # Right boundary

    # Compute source term on the grid
    F = np.zeros((ny, nx))
    for i in range(ny):
        for j in range(nx):
            F[i, j] = f(x[j], y[i])

    # Iterative solution using Gauss-Seidel
    max_iter = 10000
    for iteration in range(max_iter):
        u_old = u.copy()

        # Update interior points
        for i in range(1, ny-1):
            for j in range(1, nx-1):
                u[i, j] = 0.25 * (u[i+1, j] + u[i-1, j] + u[i, j+1] +
u[i, j-1] - h**2 * F[i, j])

        # Check convergence
        error = np.max(np.abs(u - u_old))
        if error < tol:
            print(f"Converged after {iteration} iterations with error
{error}")
            break

    return x, y, u

# Define problem parameters
domain = (0, 1, 0, 1)  # [xmin, xmax, ymin, ymax]
boundary_conditions = {
    "top": 0,    # u(x, y=1) = 0
    "bottom": 0, # u(x, y=0) = 0
    "left": 0,   # u(x=0, y) = 0
```

```
    "right": 0    # u(x=1, y) = 0
}
nx, ny = 50, 50  # Number of grid points
tol = 1e-6  # Convergence tolerance

# Define source term f(x, y)
def source_term(x, y):
    return 10 * np.sin(np.pi * x) * np.sin(np.pi * y)

# Solve the Poisson equation
x, y, u = solve_poisson_gauss_seidel(domain, boundary_conditions,
source_term, nx, ny, tol)

# Visualization
X, Y = np.meshgrid(x, y)

plt.figure(figsize=(10, 6))
plt.contourf(X, Y, u, levels=50, cmap="viridis")
plt.colorbar(label="u(x, y)")
plt.title("Solution of Poisson's Equation (Contour Plot)")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, u, cmap="viridis")
ax.set_title("Solution of Poisson's Equation (3D Surface Plot)")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("u(x, y)")
plt.show()

Converged after 1858 iterations with error 9.998938609312447e-07
```


Solution of Poisson's Equation (Contour Plot)
1.0
0.8
0.6
0.4
0.2
0.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x
0.00
−0.06
−0.12
−0.18
−0.24
−0.30
−0.36
−0.42
−0.48
u(x, y)


Solution of Poisson's Equation (3D Surface Plot)
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
u(x, y)
0.0
0.2
0.4
0.6
0.8
1.0
x
0.0
0.2
0.4
0.6
0.8
1.0
y

3.One-dimensional heat equation using Bendre- Schmidt method

Working Principle

The one-dimensional heat equation is a parabolic partial differential equation that describes the distribution of heat (or temperature) in a given region over time. The general form of the heat equation is:

$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial y^2}$where:

- u(x,t)is the temperature at position x and time t,
- α is the thermal diffusivity constant,
- $\frac{\partial^2 u}{\partial y^2}$ is the spatial second derivative of the temperature.

Finite Difference Approach

We discretize both the time and space domains to solve the heat equation using the finite difference method. The spatial domain is divided into grid points, and the time domain is divided into time steps.

The Bendre-Schmidt method is a specific finite difference approach used to solve the heat equation. It involves discretizing both the time and space domains using explicit schemes.

- For space discretization, we use a grid with spacing Δx.
- For time discretization, we use a time step Δt.

The method leads to an update formula for the temperature at the next time step ${u_i}^{n+1}$:

${u_i}^{n+1} = {u_i}^n + \lambda({u_{i-1}}^n - 2{u_i}^n + {u_{i+1}}^n)$where:

- $\lambda = \frac{\alpha \Delta t}{\Delta x^2}$
- ${u_i}^n$ is the temperature at grid point i and time step n.

Pseudocode:

Input:

- Length of the rod, L,
- Time duration, T,
- Number of grid points nx for space and time steps nt,
- Thermal diffusivity constant α,
- Initial temperature distribution u(x,0)
- Boundary conditions u(0,t)and u(L,t).

Discretization:

- Define spatial grid points x0,x1,…,xn, and time steps t0,t1,…,tm .
- Choose time step size Δt and spatial step size Δx.

Initialization:

- Set the initial temperature profile u(x,0)at all spatial points.
- Apply boundary conditions at x=0and x =L.

Time-stepping:

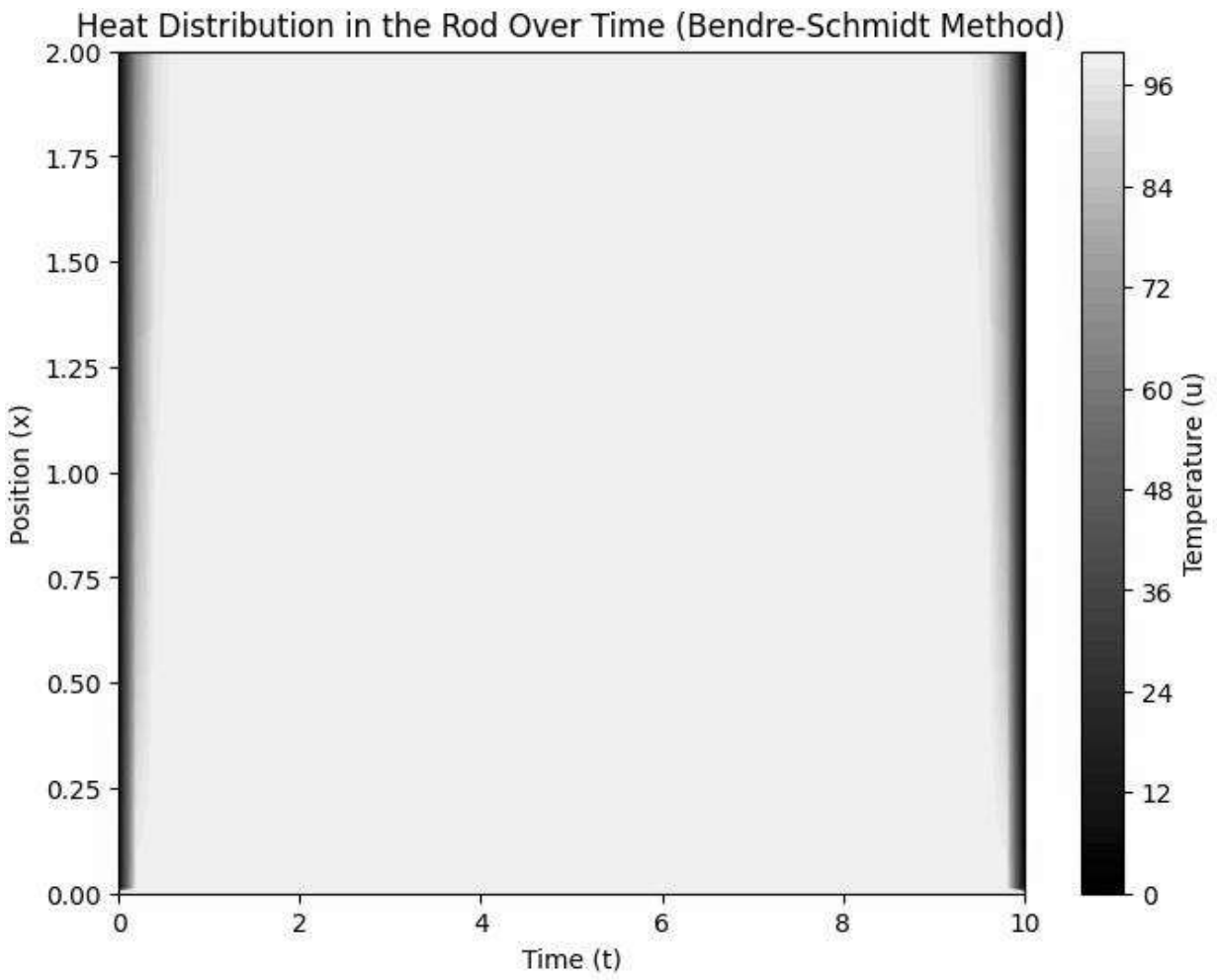
- For each time step nnn, update the temperature profile using the Bendre-Schmidt update rule: $u_i{}^{n+1} = u_i{}^n + \lambda(u_{i-1}{}^n - 2u_i{}^n + u_{i+1}{}^n)$

Output:

- Plot the temperature distribution at different time steps.
- Display the results using a graphical representation (line plot, 3D plot).

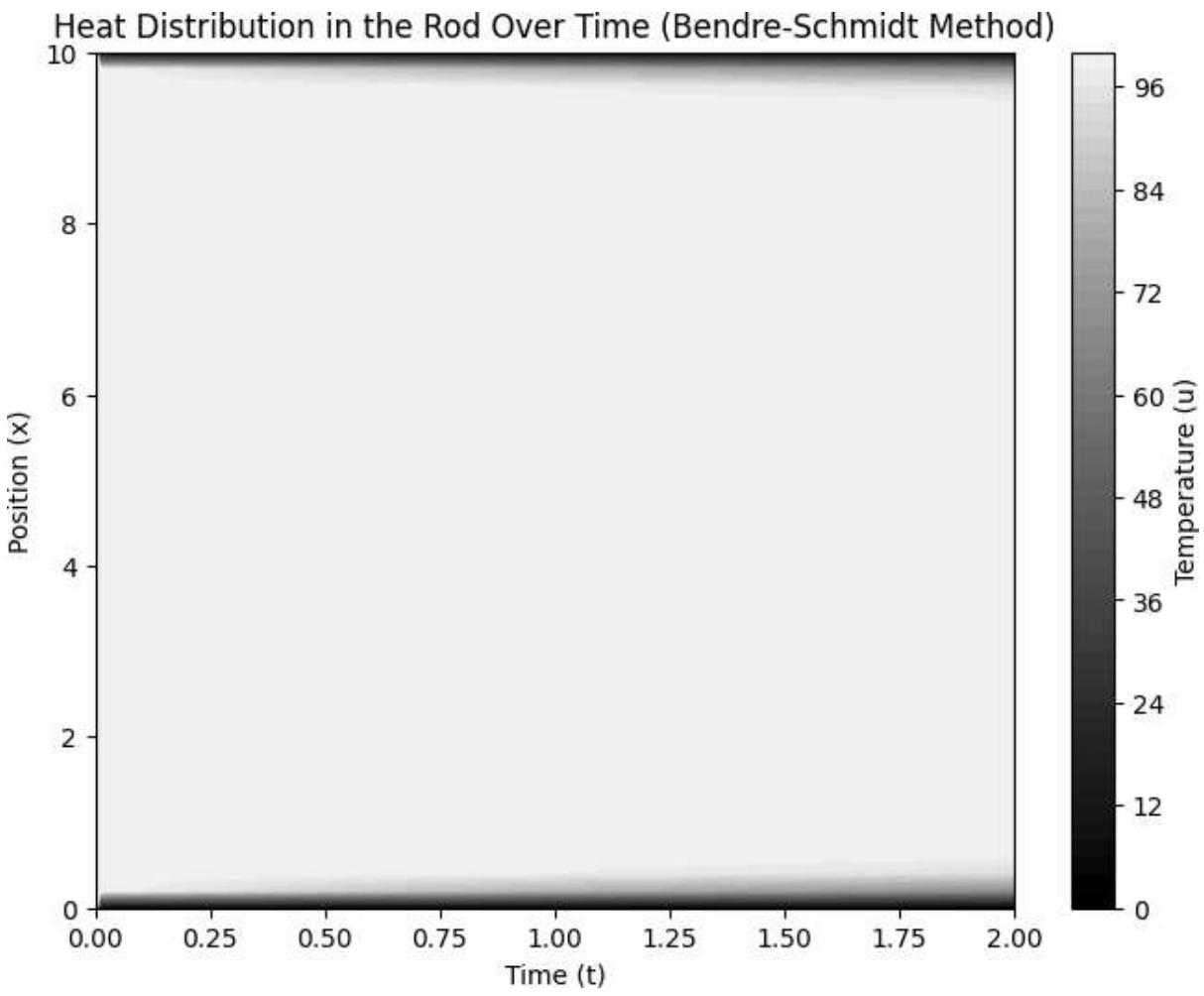
```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
L = 10      # Length of the rod
Tmax = 2    # Total time
```

```
Nx = 50      # Number of spatial steps
Nt = 200     # Number of time steps
alpha = 0.01  # Thermal diffusivity

# Discretization
dx = L / (Nx - 1)
dt = Tmax / (Nt - 1)

# Stability condition for the explicit method
if alpha * dt / dx**2 > 0.5:
    raise ValueError("The stability condition is not satisfied.
Decrease dt or increase dx.")

# Initial and boundary conditions
u = np.zeros((Nt, Nx))  # Temperature distribution
u[:, 0] = 0  # Left boundary
u[:, -1] = 0  # Right boundary
u[0, :] = 100  # Initial temperature distribution

# Bendre-Schmidt method
for n in range(0, Nt-1):
    for i in range(1, Nx-1):
        u[n+1, i] = u[n, i] + alpha * dt / dx**2 * (u[n, i+1] - 2*u[n,
i] + u[n, i-1])

# Visualization: Contour plot
plt.figure(figsize=(8, 6))
t = np.linspace(0, Tmax, Nt)
x = np.linspace(0, L, Nx)

# Create meshgrid for plotting using x and t
X, T = np.meshgrid(x, t)

# Corrected call to contourf with X and T as the axes
plt.contourf(X, T, u, 50, cmap='inferno')  # Contour plot

plt.colorbar(label='Temperature (u)')
plt.title("Heat Distribution in the Rod Over Time (Bendre-Schmidt
Method)")
plt.xlabel("Time (t)")
plt.ylabel("Position (x)")
plt.show()
```

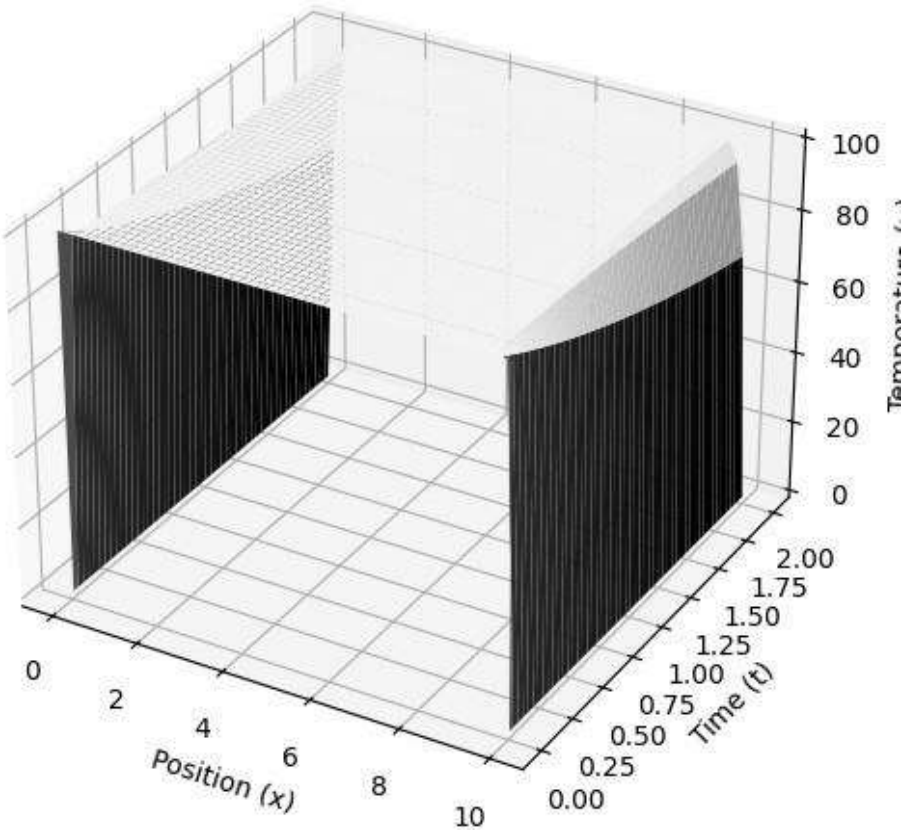



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Parameters
L = 10       # Length of the rod
Tmax = 2     # Total time
Nx = 50      # Number of spatial steps
Nt = 200     # Number of time steps
alpha = 0.01  # Thermal diffusivity

# Discretization
dx = L / (Nx - 1)
dt = Tmax / (Nt - 1)

# Stability condition for the explicit method
if alpha * dt / dx**2 > 0.5:
    raise ValueError("The stability condition is not satisfied.
Decrease dt or increase dx.")
```

```
# Initial and boundary conditions
u = np.zeros((Nt, Nx))  # Temperature distribution
u[:, 0] = 0  # Left boundary
u[:, -1] = 0  # Right boundary
u[0, :] = 100  # Initial temperature distribution

# Bendre-Schmidt method (Explicit)
for n in range(0, Nt-1):
    for i in range(1, Nx-1):
        u[n+1, i] = u[n, i] + alpha * dt / dx**2 * (u[n, i+1] - 2*u[n,
i] + u[n, i-1])

# Visualization: Contour plot
plt.figure(figsize=(8, 6))
t = np.linspace(0, Tmax, Nt)
x = np.linspace(0, L, Nx)
plt.contourf(t, x, u.T, 50, cmap='inferno')  # Transpose u for correct
dimensions
plt.colorbar(label='Temperature (u)')
plt.title("Heat Distribution in the Rod Over Time (Bendre-Schmidt
Method)")
plt.xlabel("Time (t)")
plt.ylabel("Position (x)")
plt.show()

# 3D Surface Plot
X, T = np.meshgrid(x, t)  # Create meshgrid for plotting
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u, cmap='inferno')  # 3D Surface plot
ax.set_title("Heat Distribution in the Rod Over Time (3D Surface
Plot)")
ax.set_xlabel("Position (x)")
ax.set_ylabel("Time (t)")
ax.set_zlabel("Temperature (u)")
plt.show()
```


Heat Distribution in the Rod Over Time (Bendre-Schmidt Method)
Position (x)
Time (t)
Temperature (u)
0
2
4
6
8
10
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
12
24
36
48
60
72
84
96

Heat Distribution in the Rod Over Time (3D Surface Plot)



4.One-dimensional heat equation using Crank- Nicholson method

Working principle

The one-dimensional heat equation is given as:

$\frac{\partial u(x,t)}{\partial t} = \alpha \frac{\partial^2 u(x,t)}{\partial y^2}$where:

- u(x,t) is the temperature distribution function over space and time.
- α is the thermal diffusivity constant.
- x represents spatial coordinates, and t represents time.

Crank-Nicholson Method

The Crank-Nicholson method is an implicit finite difference method that is numerically stable and accurate. It is a combination of the forward Euler method (in time) and the central difference method (in space), making it second-order accurate both in space and time.

The Crank-Nicholson method discretizes the heat equation as follows:

$$\frac{1}{\Delta t}\left(u_i^{n+1} - u_i^n\right) = \frac{\alpha}{2}\left(\frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2} + \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}\right)$$

Rearranging the terms results in the following tridiagonal system:

$$-\frac{\alpha\Delta t}{2\Delta x^2}u_{i-1}^{n+1} + \left(1 + \frac{\alpha\Delta t}{\Delta x^2}\right)u_i^{n+1} - \frac{\alpha\Delta t}{2\Delta x^2}u_{i+1}^{n+1} = \frac{\alpha\Delta t}{2\Delta x^2}u_{i-1}^n + \left(1 - \frac{\alpha\Delta t}{\Delta x^2}\right)u_i^n + \frac{\alpha\Delta t}{2\Delta x^2}u_{i+1}^n$$

Where:

- u_i^n is the value of the solution at spatial grid point iii and time step nnn.
- Δt is the time step, and Δx is the spatial grid spacing.

The method involves iterating over time steps and solving the system of equations for each time step to evolve the temperature distribution over time.

Pseudocode:

1. Input:
   - Initial conditions u(x, 0)
   - Boundary conditions u(0, t) and u(L, t)
   - Time step Δt
   - Spatial step Δx
   - Thermal diffusivity constant α

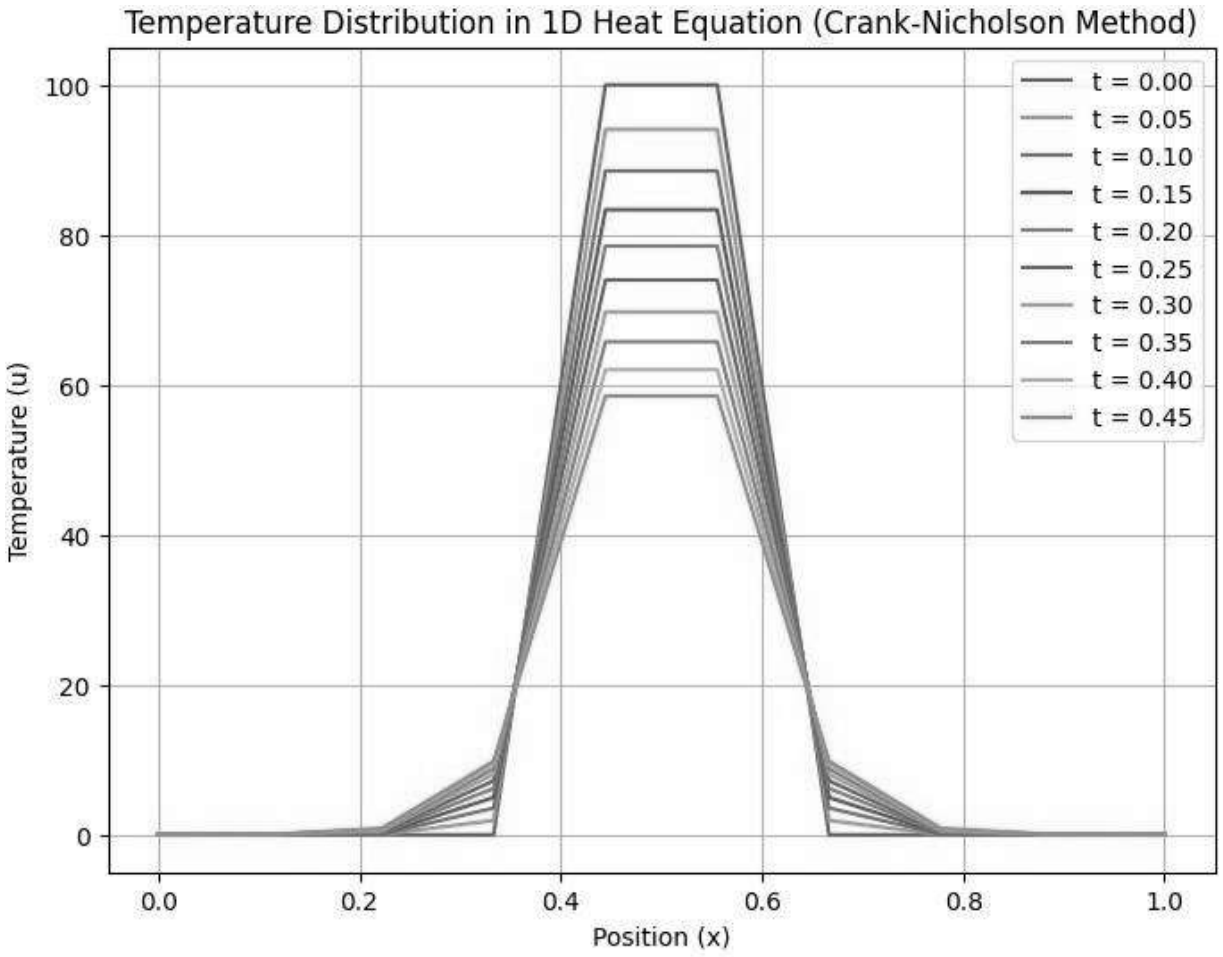
- Total time T

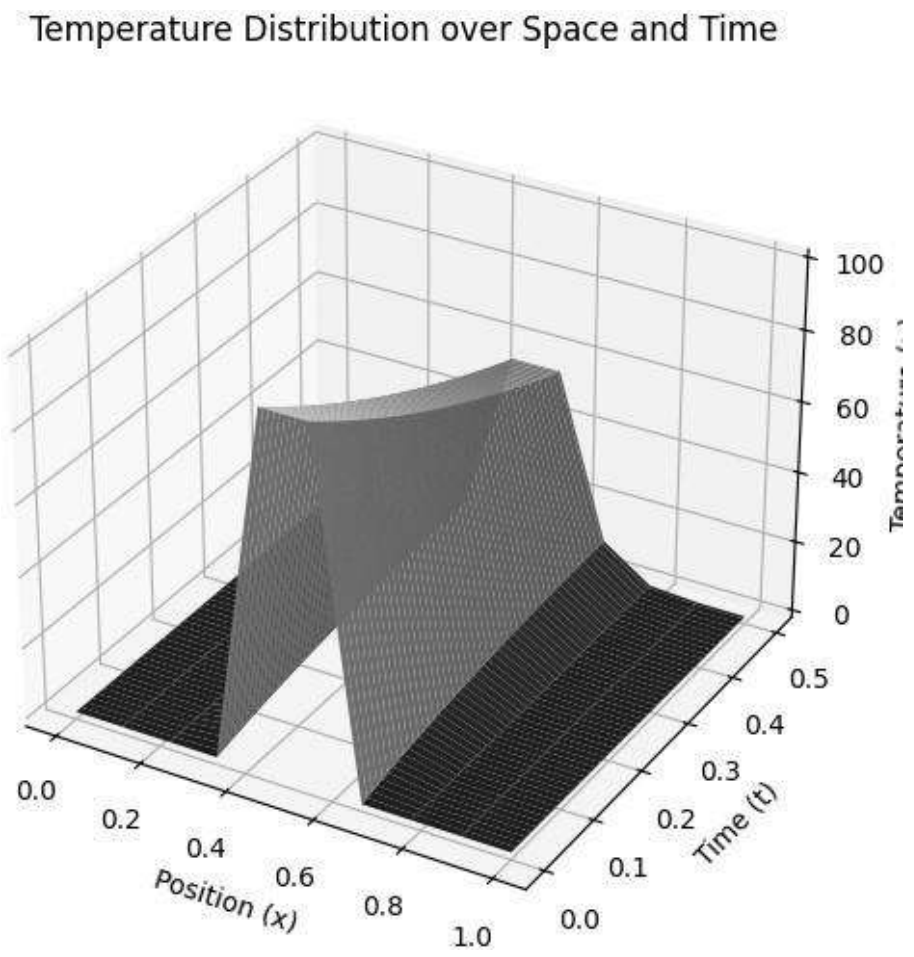
2. Discretize the domain:
   - Define spatial grid with points: x0, x1, ..., xN
   - Define time grid with points: t0, t1, ..., tN
3. Set initial condition for the temperature distribution at t=0.
4. For each time step from t=0 to T:
   - Compute the Crank-Nicholson finite difference scheme for all interior points.
   - Solve the resulting tridiagonal system using Gaussian elimination or other methods.
5. Output: The temperature distribution at each time step.
6. Visualization:
   - Plot the temperature distribution at various time steps using matplotlib.
   - Generate 2D surface plot of the temperature distribution over space and time.
7. Stop when the final time step is reached or maximum iterations are completed.

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

# Crank-Nicholson method for 1D heat equation
def crank_nicholson_heat_eq(L, T, alpha, nx, nt, u_initial,
boundary_conditions):
    """
    Solves the 1D heat equation using the Crank-Nicholson method.

    Parameters:
    - L: length of the rod
    - T: total time
    - alpha: thermal diffusivity constant
    - nx: number of spatial grid points
    - nt: number of time steps
    - u_initial: initial temperature distribution
    - boundary_conditions: boundary conditions at x=0 and x=L

    Returns:
    - x: spatial grid points
    - t: time points
    - u: solution array (temperature distribution over time and space)
    """
    # Discretize the space and time
    dx = L / (nx - 1)  # space step
    dt = T / (nt - 1)  # time step
    r = alpha * dt / (2 * dx**2)  # Crank-Nicholson parameter

    # Initialize the solution array
```

```
    u = np.zeros((nt, nx))

    # Set initial condition
    u[0, :] = u_initial

    # Apply boundary conditions
    u[:, 0] = boundary_conditions[0]  # u(0, t)
    u[:, -1] = boundary_conditions[1]  # u(L, t)

    # Coefficients for the tridiagonal matrix
    A = np.diag((1 + 2 * r) * np.ones(nx - 2))  # Main diagonal
    B = np.diag(-r * np.ones(nx - 3), 1)  # Upper diagonal
    C = np.diag(-r * np.ones(nx - 3), -1)  # Lower diagonal

    for n in range(0, nt - 1):
        # Construct the right-hand side vector
        b = np.zeros(nx - 2)
        for i in range(1, nx - 1):
            b[i - 1] = r * (u[n, i + 1] + u[n, i - 1]) + (1 - 2 * r) *
u[n, i]

        # Solve the system of equations (Ax = b)
        # Using np.linalg.solve to solve the tridiagonal system
        u_next = np.linalg.solve(A, b)

        # Update the solution for the next time step
        u[n + 1, 1:-1] = u_next

    # Return the grid and solution
    x = np.linspace(0, L, nx)
    t = np.linspace(0, T, nt)
    return x, t, u

# Problem parameters
L = 1.0      # length of the rod
T = 0.5      # total time
alpha = 0.01 # thermal diffusivity
nx = 10      # number of spatial points
nt = 100     # number of time steps

# Initial condition: Temperature distribution at t = 0
u_initial = np.zeros(nx)
u_initial[4:6] = 100  # Hot spot in the middle

# Boundary conditions: u(0, t) and u(L, t)
boundary_conditions = [0, 0]  # u(0, t) = 0, u(L, t) = 0

# Solve the heat equation
x, t, u = crank_nicholson_heat_eq(L, T, alpha, nx, nt, u_initial,
boundary_conditions)
```

```
# Visualization: Plot temperature distribution at various time steps
plt.figure(figsize=(8, 6))
for n in range(0, nt, int(nt/10)):  # Plot every 10th time step
    plt.plot(x, u[n, :], label=f"t = {t[n]:.2f}")

plt.title("Temperature Distribution in 1D Heat Equation (Crank-
Nicholson Method)")
plt.xlabel("Position (x)")
plt.ylabel("Temperature (u)")
plt.legend()
plt.grid(True)
plt.show()

# 3D Surface plot of the temperature distribution
X, T = np.meshgrid(x, t)
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u, cmap='viridis')
ax.set_title("Temperature Distribution over Space and Time")
ax.set_xlabel("Position (x)")
ax.set_ylabel("Time (t)")
ax.set_zlabel("Temperature (u)")
plt.show()
```


Temperature Distribution in 1D Heat Equation (Crank-Nicholson Method)
Temperature (u)
Position (x)
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0
t = 0.00
t = 0.05
t = 0.10
t = 0.15
t = 0.20
t = 0.25
t = 0.30
t = 0.35
t = 0.40
t = 0.45


Temperature Distribution over Space and Time
100
80
60
40
20
0
0.0
0.2
0.4
0.6
0.8
1.0
Position (x)
0.0
0.1
0.2
0.3
0.4
0.5
Time (t)

Test Case

Test Case 1:

- Problem Setup:
  - Length of rod L=1.0
  - Total time T=0.5
  - Thermal diffusivity α=0.01
  - Grid size nx=10, nt=100
  - Initial temperature distribution: A hot spot in the middle of the rod.
  - Boundary conditions: u(0,t)=0, u(L,t)=0
- Expected Results:
  - The temperature in the middle of the rod will decrease over time.
  - Boundary points will remain at zero temperature.

Test Case 2:

- Problem Setup:
  - Length of rod L=2.0
  - Total time T=1.0
  - Thermal diffusivity α=0.02
  - Grid size nx=20, nt=200
  - Initial temperature distribution: Heat distributed across the rod.
  - Boundary conditions: u(0,t)=0, u(L,t)=0
- Expected Results:
  - Temperature distribution will evolve, and heat will dissipate over time.