Assignment-5: Numerical Solution of a Partial Differential Equation

Name:- Bajarang Mishra

Roll no:- 22CH10016

Group:- 2

Problem Statement:-

Problem

Consider the following unsteady-state heat conduction problem in a one-dimensional slab of 1 m thickness.

$$\frac{1}{\alpha}\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2}, \qquad T(x, t=0) = 350\ K,\ T(x=0, t) = 300\ K,\ T(x = 1\ \text{m}, t) = 400\ K$$

Determine the unsteady temperature distribution $T(x,t)$ in the slab at different times (t = 1, 5, 10, 50, 100 s) for three different values of thermal diffusivity (α = 1, 10, 100 m^2/s).

Use: Explicit discretization, Implicit discretization, and Crank-Nicholson discretization.

In the following description of discretization, n stands for time and i stands for space.

Explicit discretization:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \alpha \frac{T_{i+1}^n + T_{i-1}^n - 2T_i^n}{(\Delta x)^2}$$

Implicit discretization:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \alpha \frac{T_{i+1}^{n+1} + T_{i-1}^{n+1} - 2T_i^{n+1}}{(\Delta x)^2}$$

Crank-Nicholson discretization:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \frac{\alpha}{2}\left[\frac{T_{i+1}^{n+1} + T_{i-1}^{n+1} - 2T_i^{n+1}}{(\Delta x)^2} + \frac{T_{i+1}^n + T_{i-1}^n - 2T_i^n}{(\Delta x)^2}\right]$$

NOTE: Solve the above problem using MATLAB function pdepe also and compare your results with the output of pdepe.

Introduction:-

Unsteady-state heat conduction in a 1D slab is solved using numerical methods. The heat equation is given by:

1/α(dT/dt) = d^2(T)/d(x^2), T=350K initially, $T = 300\ K$ at $x = 0$, $T = 400\ K$ T=400K at $x = L$)

where:

- T(x,t) is the temperature distribution,
- α is the thermal diffusivity,
- x is the spatial coordinate,
- t is time.

To solve this equation, four methods are implemented:

1. Explicit Finite Difference Method
2. Implicit Finite Difference Method
3. Crank-Nicolson Method
4. **MATLAB's `pdepe` function**

Numerical Methods Overview/Algorithm:-

1. **Explicit Method:** Uses forward time and centered space (FTCS) discretization. It is conditionally stable, requiring a small time step for convergence. Simple implementation but requires very small time steps for stability ($\Delta t \leq \Delta x^2/(2\alpha)$). Computationally expensive for large α.

   Ti,n+1-Ti,n/Δt=α(Ti+1,n−2Ti,n+Ti-1,n)/Δx^2

2. **Implicit Method:** Employs backward time and centered space (BTCS) discretization, which is unconditionally stable but requires solving a system of linear equations.Unconditionally stable. Solves a tridiagonal system at each step. Allows larger Δt, improving efficiency.

   Ti,n+1-Ti,n/Δt=α(Ti+1,n+1−2Ti,n+1+Ti-1,n+1)/Δx^2

3. **Crank-Nicolson Method:** A combination of explicit and implicit method for second-order accuracy, as it takes the average of both of these methods. Also unconditionally stable and more accurate than implicit for the same Δt.

   $T_{i,n+1}-T_{i,n}/\Delta t=\alpha/2[(T_{i+1,n}-2T_{i,n}+T_{i-1,n})/\Delta x^2+(T_{i+1,n+1}-2T_{i,n+1}+T_{i-1,n+1})/\Delta x^2]$

4. **pdepe Solution:** MATLAB's built-in solver used for benchmarking the numerical results. It is highly accurate, easy to use, but has very little control over the numerical scheme.

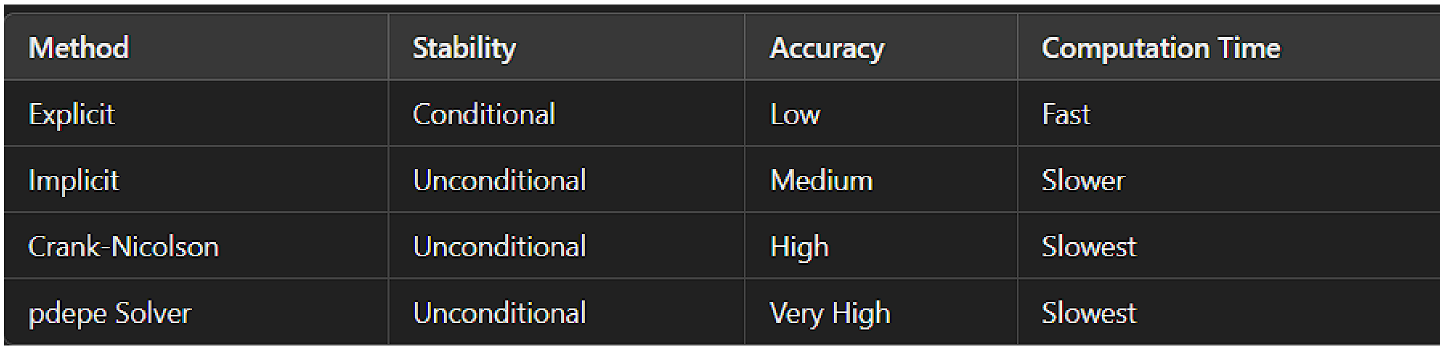
Results and Analysis:-

Temperature distributions were computed at different time intervals for varying thermal diffusivity values. The key observations include:

- The Explicit Method exhibited instability for large time steps due to its conditional stability constraint.
- The Implicit Method remained stable but introduced slight numerical diffusion.
- The Crank-Nicolson Method provided accurate results with minimal diffusion.
- The `pdepe` function served as a reliable reference solution for validation.

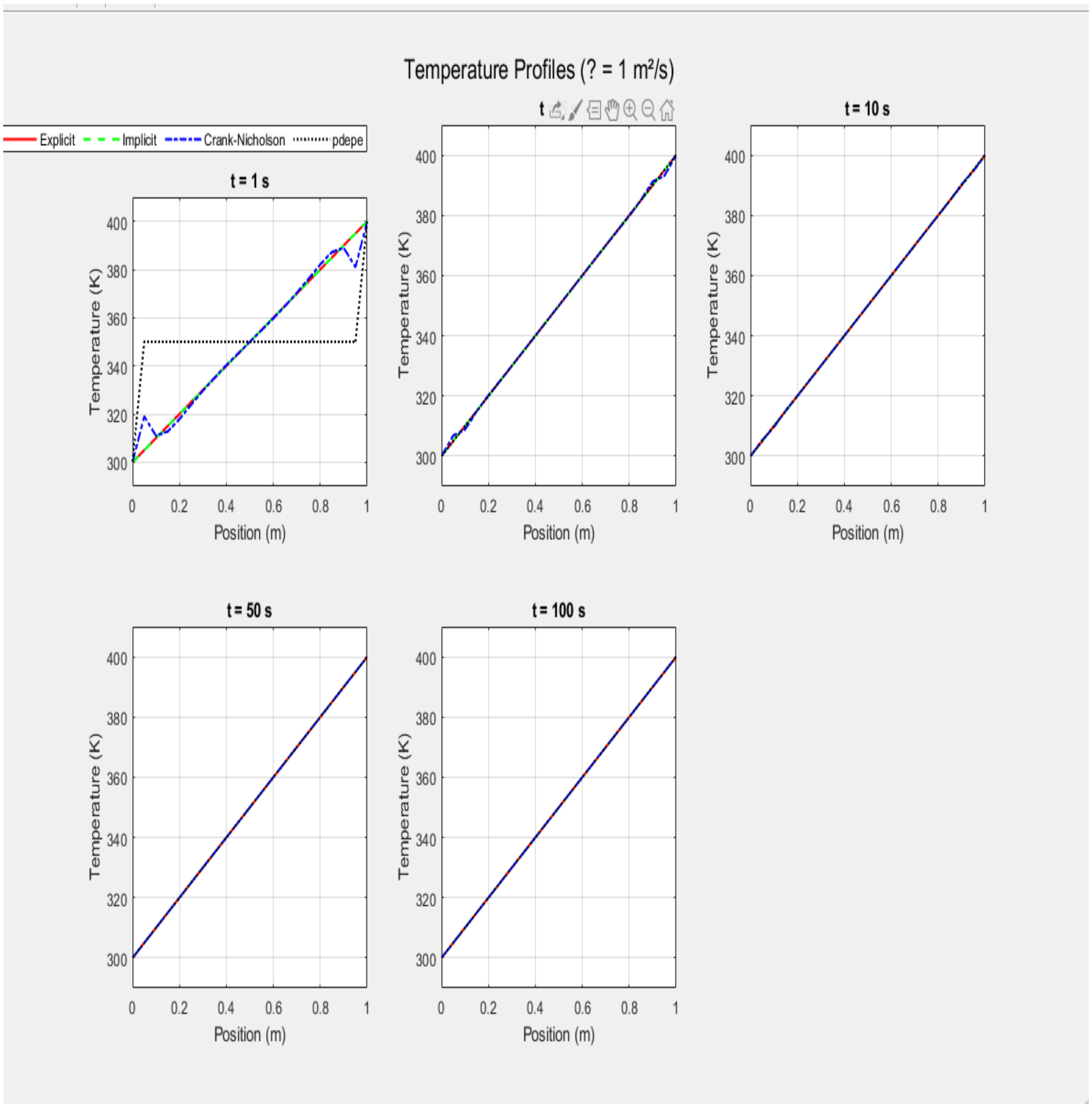
Plots illustrate the temperature distribution at selected time steps, highlighting differences in solution accuracy and stability across the methods.

Table:-

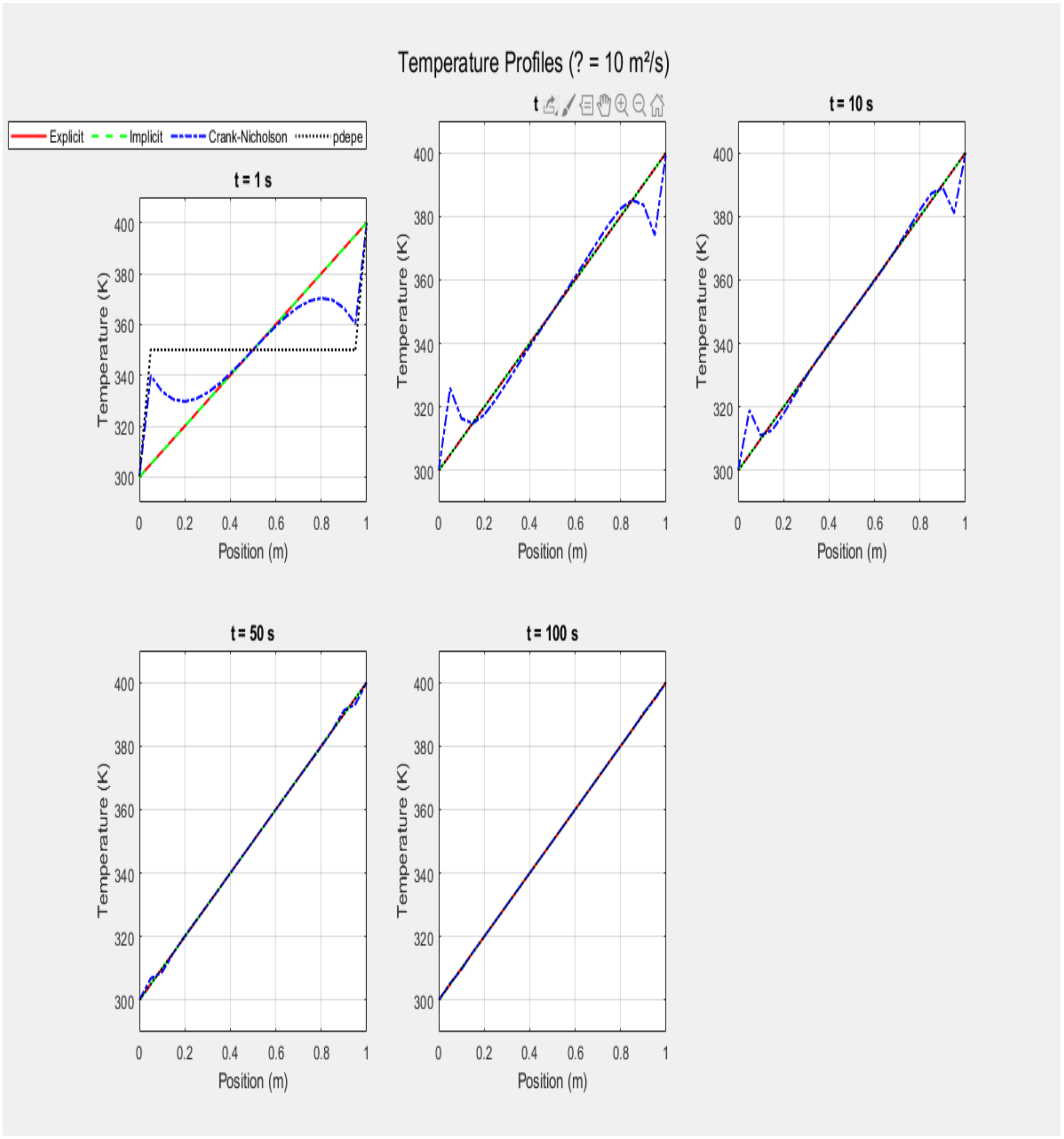


Method	Stability	Accuracy	Computation Time
Explicit	Conditional	Low	Fast
Implicit	Unconditional	Medium	Slower
Crank-Nicolson	Unconditional	High	Slowest
pdepe Solver	Unconditional	Very High	Slowest

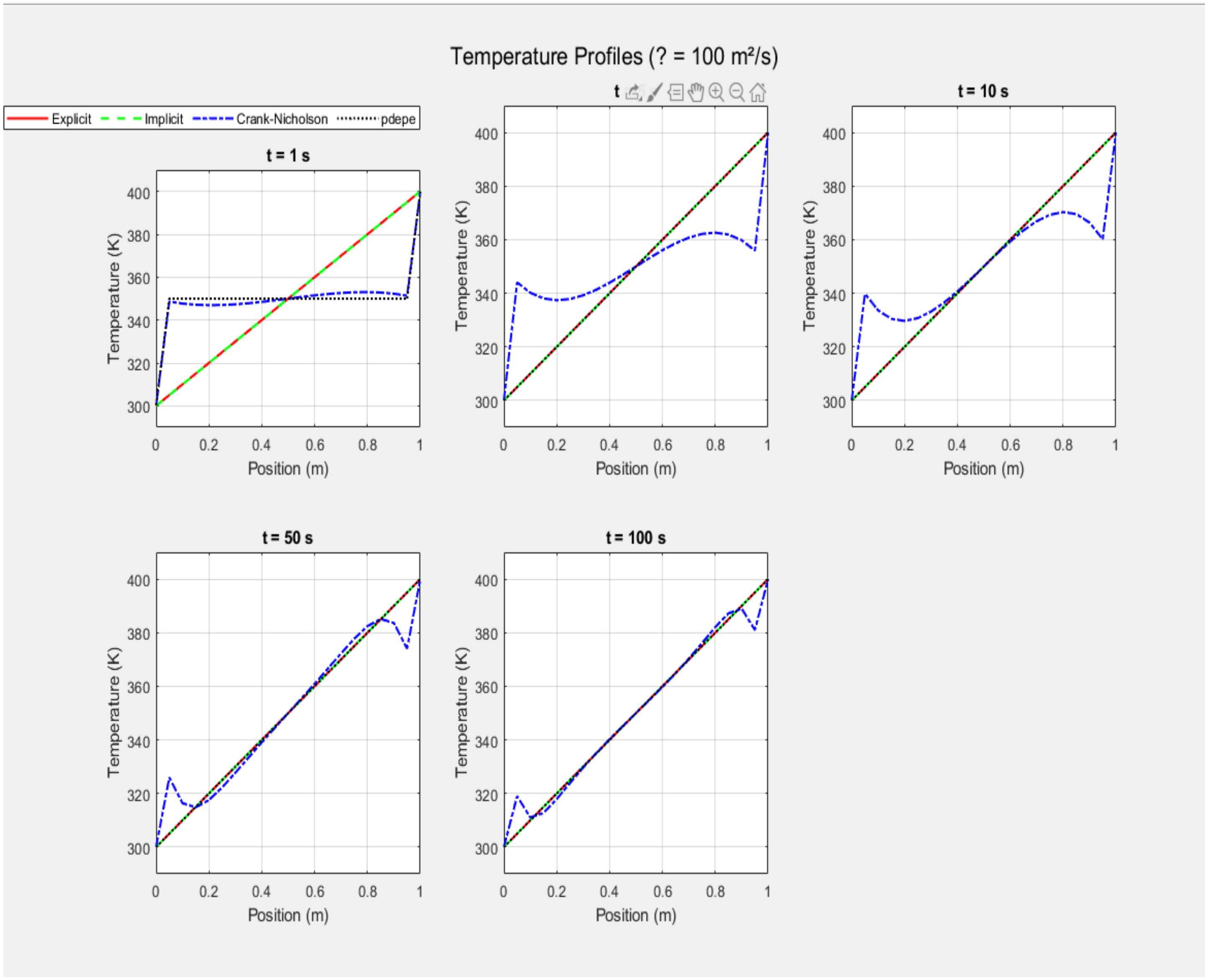
Plots:-



For alpha = 1 m^2/s, Temperature Distribution over the entire length in different timestamps



For alpha = 10 m^2/s, Temperature Distribution over the entire length in different timestamps



For alpha = 100 m^2/s, Temperature Distribution over the entire length in different timestamps

Conclusion:-

Among the numerical schemes, the Crank-Nicolson method offers the best trade-off between accuracy and computational efficiency. The explicit method, while simple, requires careful time step selection. The implicit method, though stable, introduces some numerical smoothing. The MATLAB `pdepe` function proves to be a robust tool for solving such PDEs. These insights are valuable for selecting the appropriate method based on stability and accuracy requirements in practical applications.

Final Takeaway:

- If high precision is needed → **Use `pdepe` or Crank-Nicolson**.
- If speed is the priority → **Use Explicit for small Δt\Delta tΔt, Implicit otherwise**.
- Crank-Nicolson offers a balance between stability and accuracy but is computationally expensive.

Code:-

```
% CAPE2025 Assignment 5 - Numerical Solution of PDE
% Unsteady-state heat conduction in a 1D slab

slab_length = 1; % Length of the slab
num_points = 21; % Number of spatial points
space_grid = linspace(0, slab_length, num_points); % Spatial grid
target_times = [1, 5, 10, 50, 100]; % Target times
thermal_diffusivities = [1, 10, 100]; % Thermal diffusivities

% Solve for each thermal diffusivity
for alpha = thermal_diffusivities
    % --- Explicit Method ---
    T_explicit = compute_explicit(alpha, space_grid, target_times);

    % --- Implicit Method ---
    T_implicit = compute_implicit(alpha, space_grid, target_times);

    % --- Crank-Nicholson Method ---
    T_cn = compute_cn(alpha, space_grid, target_times);
    % --- pdepe Solution ---
    T_pdepe = solve_using_pdepe(alpha, space_grid, target_times);

    % --- Create figure with subplots for all times ---
    figure('Position', [100, 100, 1200, 800]);
    sgtitle(['Temperature Profiles (α = ', num2str(alpha), ' m²/s)']);

    % Plot each time in a subplot
    for i = 1:length(target_times)
        subplot(2, 3, i); % 2x3 grid (5th plot uses position 5)
        plot(space_grid, T_explicit(:,i), 'r-', 'LineWidth', 1.5); hold on;
        plot(space_grid, T_implicit(:,i), 'g--', 'LineWidth', 1.5);
        plot(space_grid, T_cn(:,i), 'b-.', 'LineWidth', 1.5);
        plot(space_grid, T_pdepe(:,i), 'k:', 'LineWidth', 1.5);

        title(['t = ', num2str(target_times(i)), ' s']);
        xlabel('Position (m)'); ylabel('Temperature (K)');
        ylim([290, 410]); % Consistent scale for comparison
        grid on;

        if i == 1 % Add legend to first subplot
            legend('Explicit', 'Implicit', 'Crank-Nicholson', 'pdepe', ...
                   'Location', 'northoutside', 'NumColumns', 4);
        end
    end
end

% Explicit Method Solver
function T = compute_explicit(alpha, space_grid, target_times)
    delta_x = space_grid(2) - space_grid(1); % Spatial step size
    num_points = length(space_grid);
    max_dt = 0.5 * delta_x^2 / alpha; % Stability condition for time step

    % Initial condition (uniform temperature)
    T_initial = 350 * ones(num_points, 1);
    T_initial(1) = 300; % Boundary condition at x=0
    T_initial(end) = 400; % Boundary condition at x=slab_length

    num_times = length(target_times);
    T = zeros(num_points, num_times); % Matrix to store temperature at each
time
    current_time = 0;
    T_current = T_initial;

    for i = 1:num_times
        target_time = target_times(i);
        while current_time < target_time
            dt = min(max_dt, target_time - current_time);
            Q = alpha * dt / delta_x^2;
            T_next = T_current;
            % Update internal points (vectorized)
            T_next(2:end-1) = T_current(2:end-1) + Q * (T_current(3:end) -
2*T_current(2:end-1) + T_current(1:end-2));
            % Enforce boundary conditions
            T_next(1) = 300;
            T_next(end) = 400;
            T_current = T_next;
            current_time = current_time + dt;
        end
        T(:,i) = T_current;
    end
end

% Implicit Method Solver
function T = compute_implicit(alpha, space_grid, target_times)
    delta_x = space_grid(2) - space_grid(1); % Spatial step size
    num_points = length(space_grid);
    dt = 0.1; % Time step size

    % Initial condition (uniform temperature)
    T_initial = 350 * ones(num_points, 1);
    T_initial(1) = 300; % Boundary condition at x=0
    T_initial(end) = 400; % Boundary condition at x=slab_length

    num_times = length(target_times);
    T = zeros(num_points, num_times); % Matrix to store temperature at each
time
    current_time = 0;
    T_current = T_initial;

    for i = 1:num_times
        target_time = target_times(i);
        while current_time < target_time
            dt_step = min(dt, target_time - current_time);
            Q = alpha * dt_step / delta_x^2;
            internal_points = num_points - 2; % Number of internal points
(excluding boundaries)
            main_diag = (1 + 2*Q) * ones(internal_points, 1);
            lower_diag = -Q * ones(internal_points-1, 1);
            upper_diag = -Q * ones(internal_points-1, 1);
            rhs = T_current(2:end-1);
            % Adjust RHS for boundary conditions
            rhs(1) = rhs(1) + Q * T_current(1);
            rhs(end) = rhs(end) + Q * T_current(end);
            % Solve tridiagonal system
            sol = thomas(lower_diag, main_diag, upper_diag, rhs);
            T_next = T_current;
            T_next(2:end-1) = sol;
            T_current = T_next;
            current_time = current_time + dt_step;
        end
        T(:,i) = T_current;
    end
end

% Crank-Nicholson Method Solver
function T = compute_cn(alpha, space_grid, target_times)
    delta_x = space_grid(2) - space_grid(1); % Spatial step size
    num_points = length(space_grid);
    dt = 0.1; % Time step size

    % Initial condition (uniform temperature)
    T_initial = 350 * ones(num_points, 1);
    T_initial(1) = 300; % Boundary condition at x=0
    T_initial(end) = 400; % Boundary condition at x=slab_length

    num_times = length(target_times);
    T = zeros(num_points, num_times); % Matrix to store temperature at each
time
    current_time = 0;
    T_current = T_initial;

    for i = 1:num_times
        target_time = target_times(i);
        while current_time < target_time
            dt_step = min(dt, target_time - current_time);
            Q = alpha * dt_step / delta_x^2;
            internal_points = num_points - 2;
            main_diag = (1 + Q) * ones(internal_points, 1);
            lower_diag = -Q/2 * ones(internal_points-1, 1);
            upper_diag = -Q/2 * ones(internal_points-1, 1);
            % RHS construction
            rhs = T_current(2:end-1) + (Q/2) * (T_current(3:end) -
2*T_current(2:end-1) + T_current(1:end-2));
            % Adjust RHS for boundary conditions
            rhs(1) = rhs(1) + (Q/2) * T_current(1);
            rhs(end) = rhs(end) + (Q/2) * T_current(end);
            % Solve tridiagonal system
            sol = thomas(lower_diag, main_diag, upper_diag, rhs);
            T_next = T_current;
            T_next(2:end-1) = sol;
            T_current = T_next;
            current_time = current_time + dt_step;
        end
        T(:,i) = T_current;
    end
```

```
end

% Corrected Thomas Algorithm for Tridiagonal Systems
function x = thomas(a, b, c, d)
    n = length(b);
    cp = zeros(n, 1);
    dp = zeros(n, 1);

    if n >= 1
        cp(1) = c(1)/b(1);
        dp(1) = d(1)/b(1);
    end

    for i = 2:n-1
        denom = b(i) - a(i-1) * cp(i-1);
        cp(i) = c(i) / denom;
        dp(i) = (d(i) - a(i-1) * dp(i-1)) / denom;
    end

    % Handle last row
    if n > 1
        denom = b(n) - a(n-1) * cp(n-1);
        dp(n) = (d(n) - a(n-1) * dp(n-1)) / denom;
    end

    % Back substitution
    x = zeros(n, 1);
    if n >= 1
        x(n) = dp(n);
        for i = n-1:-1:1
            x(i) = dp(i) - cp(i) * x(i+1);
        end
    end
end

% pdepe Solver
function T_pdepe = solve_using_pdepe(alpha, space_grid, target_times)
    m = 0;
    solution = pdepe(m, @(x,t,u,DuDx) pdefun(x,t,u,DuDx,alpha), @(x) icfun(x),
...
          @(xl,ul,xr,ur,t) bcfun(xl,ul,xr,ur,t,alpha), space_grid,
target_times);
    T_pdepe = squeeze(solution(:,:,1))';
end

function [c,f,s] = pdefun(x, t, u, DuDx, alpha)
    c = 1 / alpha;
    f = DuDx;
    s = 0;
```

```
end

function u0 = icfun(x)
    u0 = 350; % Initial condition (uniform temperature)
end

function [pl,ql,pr,qr] = bcfun(xl, ul, xr, ur, t, alpha)
    pl = ul - 300; % Left boundary condition (x=0)
    ql = 0;
    pr = ur - 400; % Right boundary condition (x=slab_length)
    qr = 0;
end
```