Assignment-4: Numerical Solution of Ordinary Differential Equation: Boundary Value Problem

Name:- Bajarang Mishra

Roll no:- 22CH10016

Group:- 3

Problem Statement:-

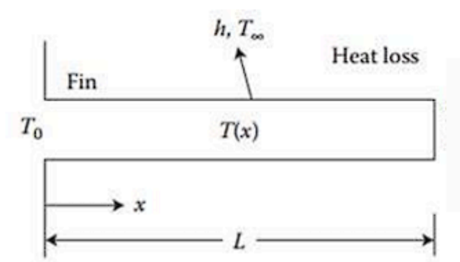
Objective: Numerical solution of Ordinary Differential Equation: Boundary Value Problem

Consider the steady-state heat transfer in a fin of uniform cross-section as shown below. The thermophysical properties of the fin material are constant. Find the temperature along the length of the fin $T(x)$ using

(a) Finite Difference Method (write your own code)

(b) Shooting Method (write your own code)

(c) MATLAB function `bvp4c`



The following BVP represents the governing equation for the fin.

$$\frac{d^2T}{dx^2} - \beta(T - T_\infty) = 0, \quad T(x=0) = T_0, T(x=L) = T_L$$

Given: $T_0 = 100$, $T_L = 30$, $T_\infty = 30$, $L = 2$, $\beta = 1.5$ (in appropriate units)

Method-1:- Finite Difference Method

The finite difference method discretizes the domain into N+1 grid points and approximates derivatives using finite differences.

Algorithm

1. **Discretize the domain into N+1 points.**
2. **Construct the coefficient matrix and right-hand side vector.**
3. **Apply boundary conditions.**
4. **Solve the resulting linear system.**

For this problem:-

It will discretize x in to n points, x0,x1..,xn with step size as h=L/n
Then it will approximate the second derivative as

d/dx(dT/dx) ~= (Ti+1-2Ti+Ti-1) / h^2

After substituting in the given equation we will get

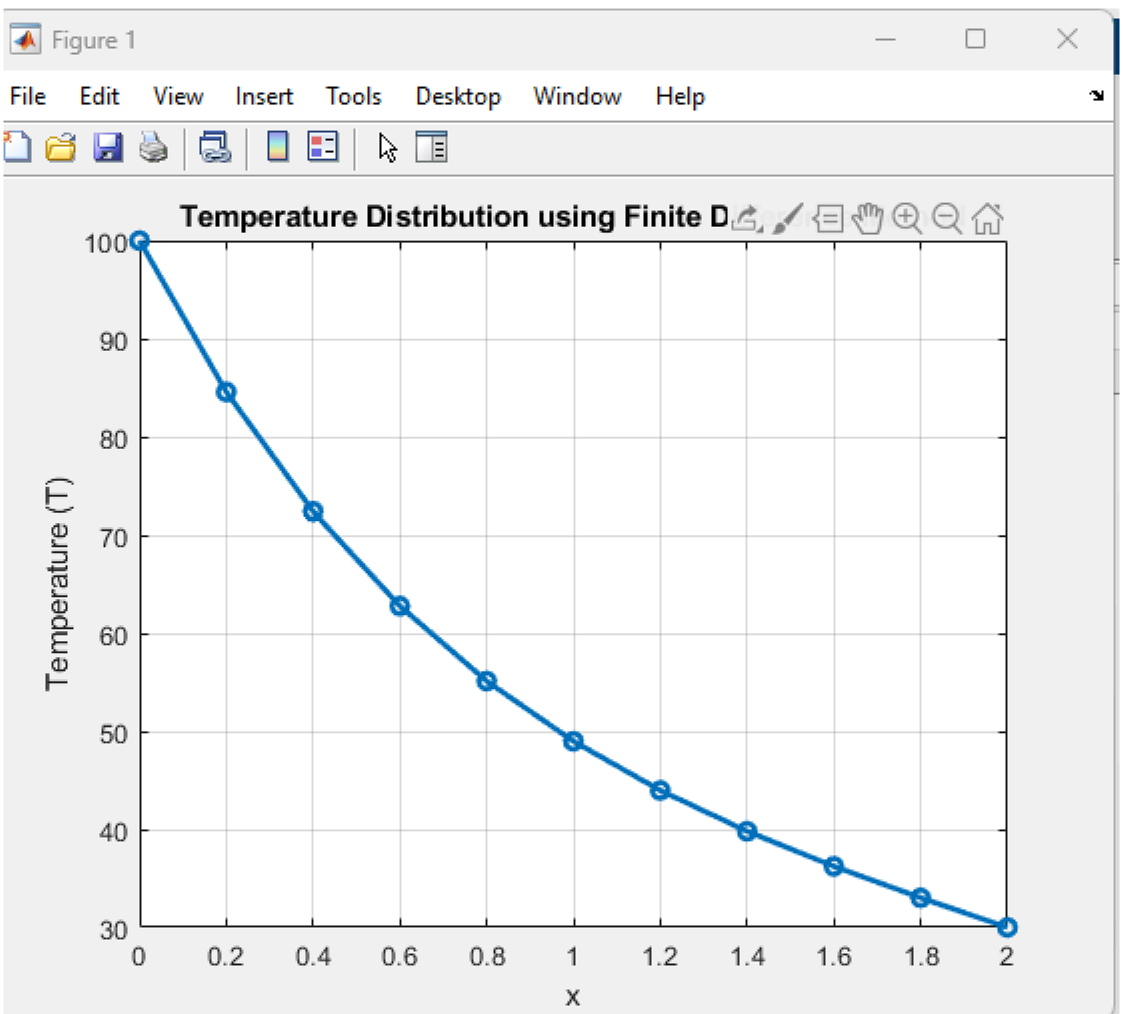
-Ti-1+(2+h^2*beta)Ti-Ti+1 = h^2*beta*T(inf)

And then solving these systems of equations using boundary values and matrix methods

Code:-

```
% Finite Difference Method
clc; clear;

% Parameters
L = 2; beta = 1.5; T0 = 100; TL = 30; T_inf = 30;
N = 10; % Number of intervals
h = L / N;

% Coefficient matrix and RHS vector
A = zeros(N+1, N+1);
b = zeros(N+1, 1);

% Fill interior points
for i = 2:N
   A(i, i-1) = -1;
   A(i, i)   = 2 + h^2 * beta;
   A(i, i+1) = -1;
   b(i)      = h^2 * beta * T_inf;
end
```

```
% Apply boundary conditions
A(1, 1) = 1; b(1) = T0; % At x=0
A(N+1, N+1) = 1; b(N+1) = TL; % At x=L

% Solve for temperature distribution
T_fd = A\b;

% Plot results
x_fd = linspace(0, L, N+1);
plot(x_fd, T_fd, 'o-', 'LineWidth', 2);
xlabel('x'); ylabel('Temperature (T)');
title('Temperature Distribution using Finite Difference Method');
grid on;
```



“Finite Difference Method”

Method-2:- Shooting Method

The shooting method converts the BVP into an initial value problem (IVP). We guess an initial slope $T'(0)$, and then we solve the IVP using numerical methods and adjust the guess iteratively until the boundary condition at $x=L$ is satisfied.

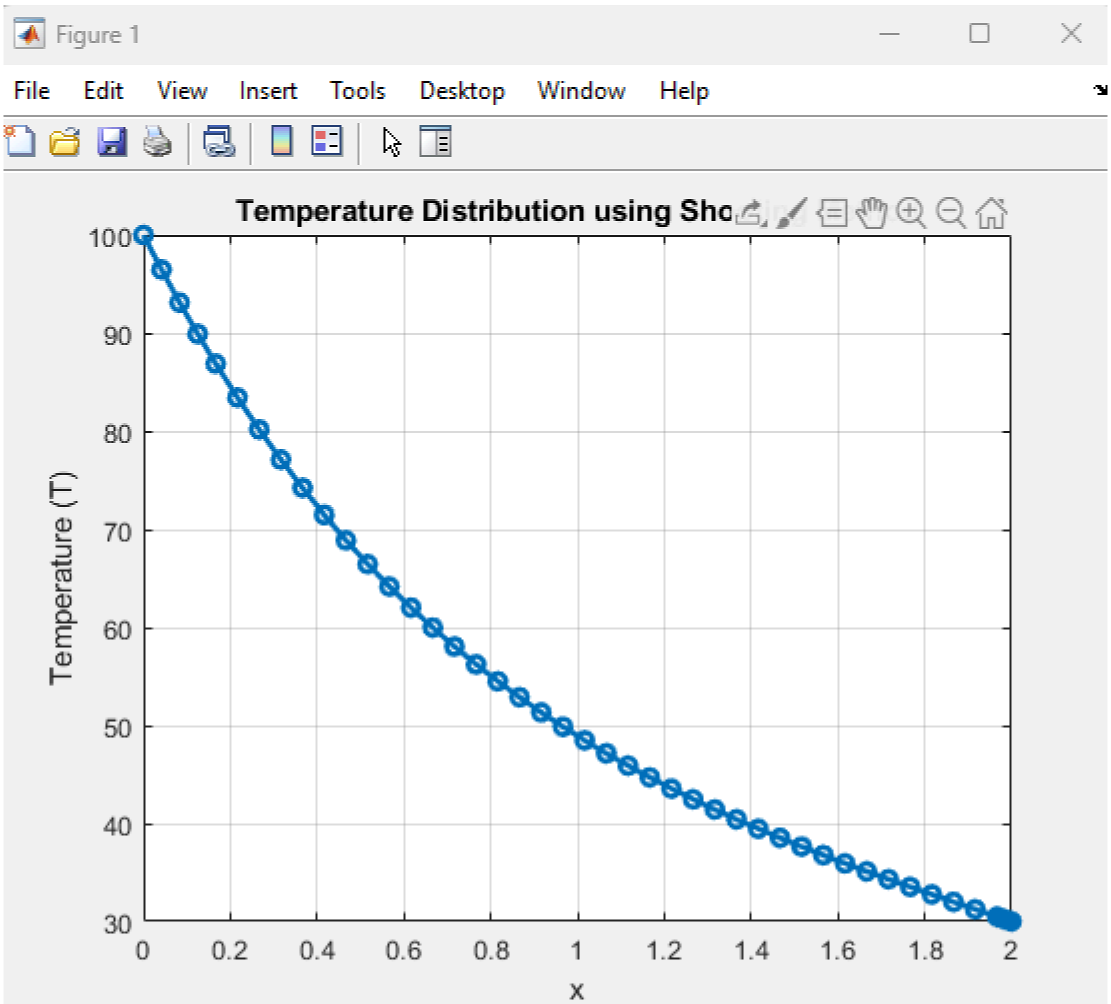
Algorithm

1. **Guess an initial value for $T'(0)$**
2. **Solve the IVP using ODE solvers.**
3. **Compare the computed $T(L)$ with the given boundary condition.**
4. **Adjust the guess for $T'(0)$ using a root-finding method (e.g., secant or bisection).**

Code:-

```
% Shooting method
clc; clear;

% Parameters
L = 2; beta = 1.5; T0 = 100; TL = 30; T_inf = 30;

% Define ODE as a system of first-order equations
odefun = @(x, y) [y(2); beta * (y(1) - T_inf)];

% Shooting method iteration
shooting_func = @(guess_T_prime0) ...
   ode45(odefun, [0 L], [T0 guess_T_prime0]).y(1,end) - TL;

% Adjusted initial guesses for root-finding (closer range for better sign change)
```

```
guess_low = -100;
guess_high = -50;

% Check the function values at the initial guesses
disp('Function value at guess_low:');
low_value = shooting_func(guess_low);
disp(low_value); % Evaluate at guess_low

disp('Function value at guess_high:');
high_value = shooting_func(guess_high);
disp(high_value); % Evaluate at guess_high

% Ensure the function values have opposite signs for fzero to work
if low_value * high_value > 0
    error('Function values at guess_low and guess_high do not have opposite signs. Adjust the guesses.');
end

% Use fzero to find the correct initial slope
T_prime0_solution = fzero(shooting_func, [guess_low guess_high]);

% Solve with correct initial slope
[x_shoot, y_shoot] = ode45(odefun, [0 L], [T0 T_prime0_solution]);

% Plot results
plot(x_shoot, y_shoot(:,1), 'o-', 'LineWidth', 2);
xlabel('x'); ylabel('Temperature (T)');
title('Temperature Distribution using Shooting Method');
grid on;
```



“Shooting Method”

Method-3:- MATLAB function “bvp4c”

Mathematical Working

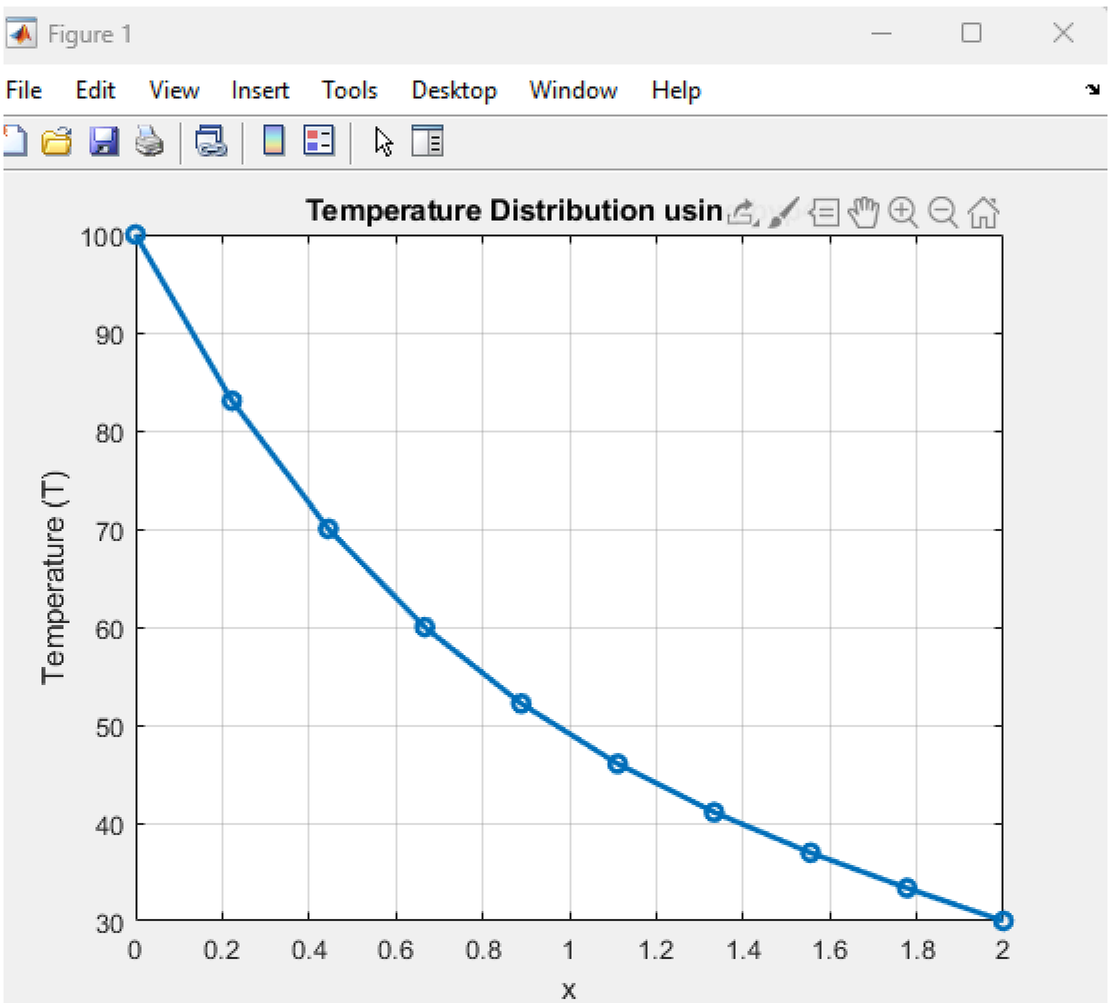
The `bvp4c` function solves BVPs using collocation methods. It requires:

- A function defining the differential equation.
- Boundary condition functions.
- An initial guess for the solution.

Code:-

```
% bvp4c Method
clc; clear;

% Parameters
L = 2; beta = 1.5; T0 = 100; TL = 30; T_inf = 30;

% Define ODE function
odefun_bvp4c = @(x, y) [y(2); beta * (y(1) - T_inf)];

% Define boundary conditions
bcfun_bvp4c = @(ya, yb) [ya(1)-T0; yb(1)-TL];

% Initial guess structure
solinit_bvp4c = bvpinit(linspace(0, L, 10), [T0 TL]);

% Solve BVP
sol_bvp4c = bvp4c(odefun_bvp4c, bcfun_bvp4c, solinit_bvp4c);

% Extract solution and plot results
```

```
x_bvp4c = sol_bvp4c.x;
T_bvp4c = sol_bvp4c.y(1,:);
plot(x_bvp4c, T_bvp4c, 'o-', 'LineWidth', 2);
xlabel('x'); ylabel('Temperature (T)');
title('Temperature Distribution using bvp4c');
grid on;
```



"bvp4c" method

Key Findings and Analysis:-

Comparison of Methods:-

Method	**Accuracy**	**Complexity**	**Ease of Implementation**
Finite Difference	**High**	**Moderate**	**Easy**
Shooting Method	**Moderate**	**High**	**Requires iterations**
`bvp4c`	**High**	**Low**	**Easy**

Performance Analysis:-

- The finite difference method is straightforward but requires careful handling of discretization.
- The shooting method requires iterative guessing but is useful for IVP solvers.
- `bvp4c` is robust and efficient for this type of problem.

Summary and Conclusion:-

All three methods successfully solve the BVP for temperature distribution in a fin. The finite difference method provides a simple matrix-based approach but may suffer from discretization errors if not finely resolved. The shooting method transforms the BVP into an IVP but requires iterative adjustments to meet boundary conditions. Finally, `bvp4c` is highly accurate and user-friendly for solving BVPs in MATLAB.
For practical use in engineering applications where accuracy and ease are critical, `bvp4c` is recommended as it automates much of the complexity while providing reliable results.