```python
import numpy as np

# --- Parameters ---
F = 1.0
V = 1.0
k0 = 36 * 10**6
neg_dH = 6500.0
E = 12000.0
rhoCp = 500.0
Tf = 298.0
CAf = 10.0
UA = 150.0
Tj0 = 298.0
rhojCj = 600.0
Fj = 1.25
Vj = 0.25
R = 1.987

def equations(x):
   CA, T, Tj = x

   r = CA * k0 * np.exp(-E / (R * T))


   eq1 = F * (CAf - CA) - r * V

   eq2 = F * rhoCp * (Tf - T) + (neg_dH) * V * r - UA * (T - Tj)


   eq3 = Fj * rhojCj * (Tj0 - Tj) + UA * (T - Tj)

   return [eq1, eq2, eq3]

def jacobian(x):
   CA, T, Tj = x

   # Pre-calculate terms
   exp_term = np.exp(-E / (R * T))
   r = CA * k0 * exp_term


   dr_dCA = k0 * exp_term

   dr_dT = r * (E / (R * T**2))
   J = np.zeros((3, 3))


   J[0, 0] = -F - V * dr_dCA      # d/dCA
   J[0, 1] = -V * dr_dT           # d/dT
   J[0, 2] = 0                    # d/dTj


   J[1, 0] = neg_dH * V * dr_dCA                          # d/dCA
   J[1, 1] = -F * rhoCp + neg_dH * V * dr_dT - UA         # d/dT
   J[1, 2] = UA                                           # d/dTj


   J[2, 0] = 0                                            # d/dCA
   J[2, 1] = UA                                           # d/dT
   J[2, 2] = -Fj * rhojCj - UA                            # d/dTj

   return J

def newton_raphson(equations_func, jacobian_func, initial_guess, tolerance=1e-6,
max_iterations=100):
   x_current = np.array(initial_guess, dtype=float)

   for i in range(max_iterations):
       F_x = np.array(equations_func(x_current))
       if np.linalg.norm(F_x) < tolerance:
           return x_current

       J_x = np.array(jacobian_func(x_current))

       try:
           delta_x = np.linalg.solve(J_x, F_x)
       except np.linalg.LinAlgError:
           print("Singular Matrix")
           return None

       x_current = x_current - delta_x

       if np.linalg.norm(delta_x) < tolerance:
           return x_current

   print("Did not converge")
   return None

# --- DRIVER CODE TO FIND 3 STEADY STATES ---
```

```
guess1 = [15.0,300.0,300.0]
sol1 = newton_raphson(equations, jacobian, guess1)


guess2 = [20.0, 300.0, 300.0]

sol2 = newton_raphson(equations, jacobian, guess2)


guess3 =[0.1,400.0,310.0]
sol3 = newton_raphson(equations, jacobian, guess3)

print("--- Solutions [CA, T, Tj] ---")
print(f"State 1: {sol1}")
print(f"State 2: {sol2}")
print(f"State 3: {sol3}")
```

```
def diff_equations(t,x):
   CA, T, Tj = x

   r = CA * k0 * np.exp(-E / (R * T))


   eq1 = (F * (CAf - CA) - r * V)/V

   eq2 = (F * rhoCp * (Tf - T) + (neg_dH) * V * r - UA * (T - Tj))/(rhoCp*V)


   eq3 = (Fj * rhojCj * (Tj0 - Tj) + UA * (T - Tj))/(rhojCj*Vj)

   return np.array([eq1, eq2, eq3])

def RK4_general(t, x, h, equations_func):
   x = np.array(x, dtype=float)
```

```
    k1 = h * equations_func(t, x)


    k2 = h * equations_func(t + 0.5*h, x + 0.5*k1)


    k3 = h * equations_func(t + 0.5*h, x + 0.5*k2)


    k4 = h * equations_func(t + h, x + k3)

    x_new = x + (k1 + 2*k2 + 2*k3 + k4) / 6
    return x_new
```

```
def diff_equations(t,x):
    CA, T, Tj = x

    r = CA * k0 * np.exp(-E / (R * T))


    eq1 = (F * (CAf - CA) - r * V)/V

    eq2 = (F * rhoCp * (Tf - T) + (neg_dH) * V * r - UA * (T - Tj))/(rhoCp*V)


    eq3 = (Fj * rhojCj * (Tj0 - Tj) + UA * (T - Tj))/(rhojCj*Vj)

    return np.array([eq1, eq2, eq3])

def RK4_general(t, x, h, equations_func):
    x = np.array(x, dtype=float)


    k1 = h * equations_func(t, x)
```

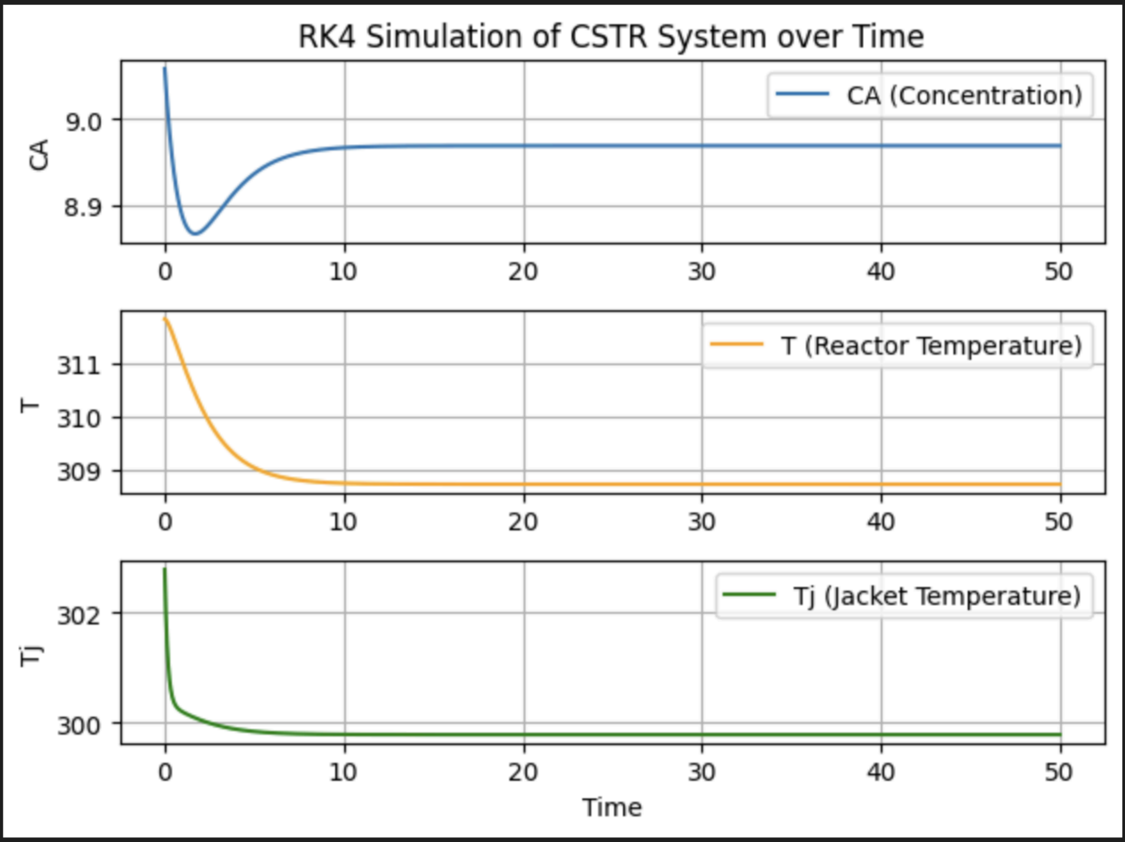
```
   k2 = h * equations_func(t + 0.5*h, x + 0.5*k1)


   k3 = h * equations_func(t + 0.5*h, x + 0.5*k2)


   k4 = h * equations_func(t + h, x + k3)

   x_new = x + (k1 + 2*k2 + 2*k3 + k4) / 6
   return x_new
```

```
Initial Guess : [  9.05824706 311.81423058 302.7857051 ]
Converged Values [  8.96856145 308.72696097 299.78782683]
```



```
import matplotlib.pyplot as plt
```

```
# Initial conditions (using a slightly perturbed sol1 to see evolution)
initial_state = np.array([6.16496225*1.01, 337.8843926*1.01,  304.64739877*1.01])
print("Initial Guess :", initial_state)
```

```python
# Time parameters
t_start = 0.0
t_end = 50.0  # Simulate up to t = 50
h = 0.0001      # Time step
```

```python
# Calculate number of iterations needed
num_iterations = int((t_end - t_start) / h)
```

```python
t_values = [t_start]
CA_values = [initial_state[0]]
T_values = [initial_state[1]]
Tj_values = [initial_state[2]]
```

```python
current_state = initial_state
current_time = t_start
```

```python
# Perform RK4 integration and store values
for i in range(num_iterations):
   current_state = RK4_general(current_time, current_state, h, diff_equations)
   current_time += h
```

```python
   t_values.append(current_time)
   CA_values.append(current_state[0])
   T_values.append(current_state[1])
   Tj_values.append(current_state[2])
```

```python
print("Converged Values", current_state)
```

```python
# Plot the results
```

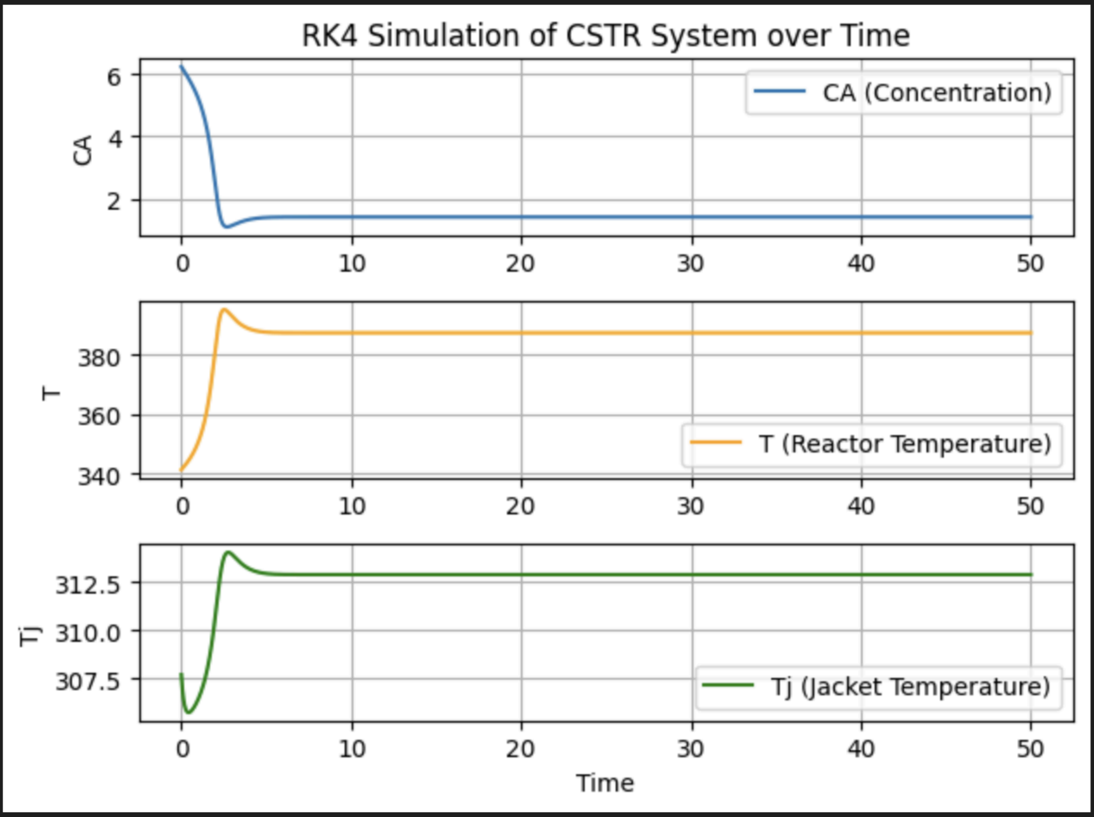
```python
plt.subplot(3, 1, 1)
plt.plot(t_values, CA_values, label='CA (Concentration)')
plt.ylabel('CA')
plt.title('RK4 Simulation of CSTR System over Time')
plt.grid(True)
plt.legend()
```

```python
plt.subplot(3, 1, 2)
plt.plot(t_values, T_values, label='T (Reactor Temperature)', color='orange')
plt.ylabel('T')
plt.grid(True)
plt.legend()
```

```python
plt.subplot(3, 1, 3)
plt.plot(t_values, Tj_values, label='Tj (Jacket Temperature)', color='green')
plt.xlabel('Time')
plt.ylabel('Tj')
plt.grid(True)
plt.legend()
```

```python
plt.tight_layout()
plt.show()
```

```
Initial Guess : [  6.22661187 341.26323653 307.69387276]
Converged Values [  1.40939329 387.34230978 312.89038496]
```



```python
from scipy.optimize import fsolve, root_scalar
from scipy.integrate import solve_ivp
```

```python
# --- Parameters ---
T0 = 300
TL = 400
T_inf = 200
L = 10
alpha = 0.05
beta = 2.7e-9
```

```python
# ==========================================
# 1. Finite Difference Method (FDM)
# ==========================================
N = 50  # Number of nodes
x_fdm = np.linspace(0, L, N)
dx = x_fdm[1] - x_fdm[0]
```

```python
def fdm_residuals(T_inner):
   # Construct full T array: [T0, ...inner..., TL]
   T = np.concatenate(([T0], T_inner, [TL]))
   res = []
   for i in range(1, N-1):
       # Central difference for 2nd derivative
       d2T = (T[i+1] - 2*T[i] + T[i-1]) / dx**2
       # Residual = ODE at node i
       eq = d2T - alpha * (T[i] - T_inf) - beta * (T[i]**4 - T_inf**4)
       res.append(eq)
   return res
```

```python
# Initial guess (Linear profile)
T_guess = np.linspace(T0, TL, N)[1:-1]
# Solve
T_inner_sol = fsolve(fdm_residuals, T_guess)
T_fdm = np.concatenate(([T0], T_inner_sol, [TL]))
```

```python
# ==========================================
# 2. Shooting Method
# ==========================================
def ode_system(x, y):
   T, dT = y
   d2T = alpha * (T - T_inf) + beta * (T**4 - T_inf**4)
   return [dT, d2T]
```

```python
def shooting_objective(slope_guess):
   # Integrate without enforcing t_eval=[L] to avoid crash if it fails early
   sol = solve_ivp(ode_system, [0, L], [T0, slope_guess])
```

```python
   # Check if integration reached L
   if sol.t[-1] < L:
       return 1e5 # Return large error if solver failed early
```

```python
   T_L_calculated = sol.y[0][-1]
   return T_L_calculated - TL
```

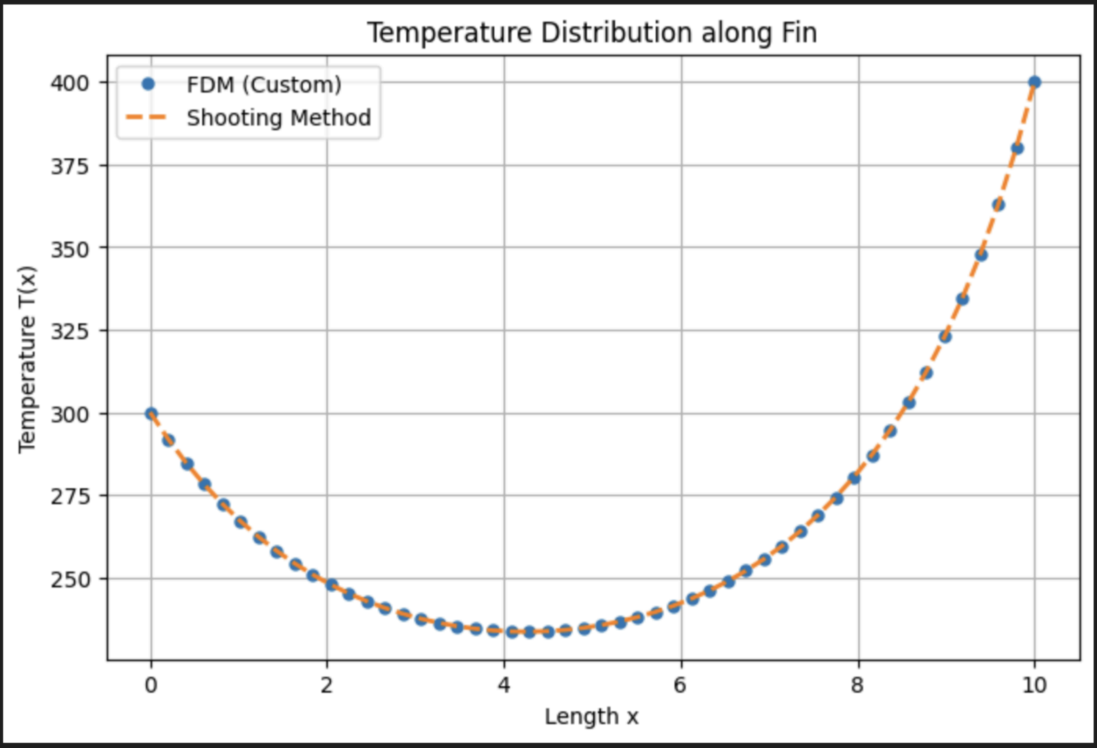
```python
# Find correct slope.
# Note: Reduced bracket range [-50, 50] helps avoid divergence
try:
   root = root_scalar(shooting_objective, bracket=[-50, 50], method='brentq')
   correct_slope = root.root
   print(f"Shooting Method: Found initial slope dT/dx = {correct_slope:.4f}")
except ValueError as e:
   print(f"Shooting Method Error: {e}")
   correct_slope = 0 # Fallback
```

```python
# Final integration for plotting
sol_shoot = solve_ivp(ode_system, [0, L], [T0, correct_slope],
t_eval=np.linspace(0, L, 50))
x_shoot = sol_shoot.t
T_shoot = sol_shoot.y[0]
```

```python
# ==========================================
# Plotting
# ==========================================
plt.figure(figsize=(8, 5))
plt.plot(x_fdm, T_fdm, 'o', label='FDM (Custom)', markersize=5)
plt.plot(x_shoot, T_shoot, '--', label='Shooting Method', linewidth=2)
```

```
plt.title('Temperature Distribution along Fin')
plt.xlabel('Length x')
plt.ylabel('Temperature T(x)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Shooting Method: Found initial slope dT/dx = -41.7351
```


Temperature Distribution along Fin
FDM (Custom)
Shooting Method
Temperature T(x)
Length x