CAPE Lab

Assignment 3

Name: Aditya Deep
Roll No: 22CH10003
Group No: 5

Date: 31 Jan 2025

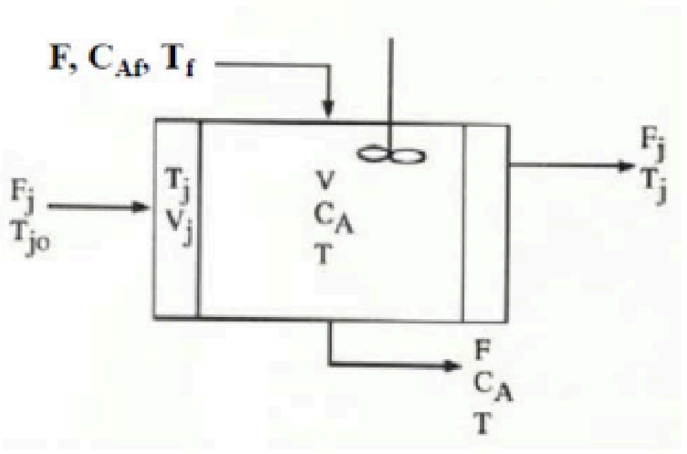
Assignment 3

Objective: Solution of Ordinary Differential Equations: Initial Value Problems

Problem-1

Consider the perfectly mixed CSTR where a first-order exothermic irreversible reaction takes place (r = rate of reaction). Heat generated by reaction is being removed by the jacket fluid. The reactor volume (V) is constant.

$$A \rightarrow B, \quad r = k_0 e^{\left(\frac{-E}{RT}\right)} C_A$$



Governing Equations:

(Subscript j indicates parameters related to jacket. Symbols carry their usual significance. Refer to the figure.)

$$V\frac{dC_A}{dt} = FC_{Af} - FC_A - rV$$

$$\rho C_p V\frac{dT}{dt} = \rho C_p F\left(T_f - T\right) + \left(-\Delta H\right)Vr - UA\left(T - T_j\right)$$

$$\rho_j C_j V_j \frac{dT_j}{dt} = \rho_j C_j F_j\left(T_{j0} - T_j\right) + UA\left(T - T_j\right)$$

Model Parameter Values:

Parameter	Value	Parameter	Value
F (m^3/h)	1	C_{Af} (kgmol/m^3)	10
V (m^3)	1	UA (kcal/°C h)	150
k_0 (h^{-1})	36×10^6	T_{j0} (K)	298
$(-\Delta H)$ (kcal/kgmol)	6500	$(\rho_j C_j)$ (kcal/m^3 °C)	600
E (kcal/kgmol)	12000	F_j (m^3/h)	1.25
(ρC_p) (kcal/m^3 °C)	500	V_j (m^3)	0.25
T_f (K)	298		

There are three steady states for this system and you should have identified all the three steady states in Assignment 2. Now study the dynamic behaviour of the system by solving the above ODEs as follows.

1. First consider any one steady state that you have obtained. Now obtain 3 different initial conditions by perturbing the selected steady state by 1%, 5%, and 25%. Simulate the system with these three different initial conditions for time t = 0 to t = 50 and plot the three state variables vs time. Does the system go to new steady state? Or does it return to the same steady state? How much time it takes to reach the steady state? **Repeat for other two steady states.**

   Write your own code implementing **4th order Runge Kutta method** and compare your results using MATLAB function `ode45`. Analyse the effect of step size of your RK-4 method on the accuracy of your solution (compare against MATLAB `ode45` function).

2. Comment on the stability of each steady state. Categorise the steady states as stable or unstable steady states.

Steady-Steady Values Obtained Earlier:

1. **Steady State 1** → $C_A = 0.59, T = 405, T_j = 390$
2. **Steady State 2** → $C_A = 1.12, T = 355, T_j = 340$
3. **Steady State 3** → $C_A = 2.51, T = 320, T_j = 310$

Algorithm

1. Dynamic Behavior Analysis using Runge-Kutta 4th Method

We will:

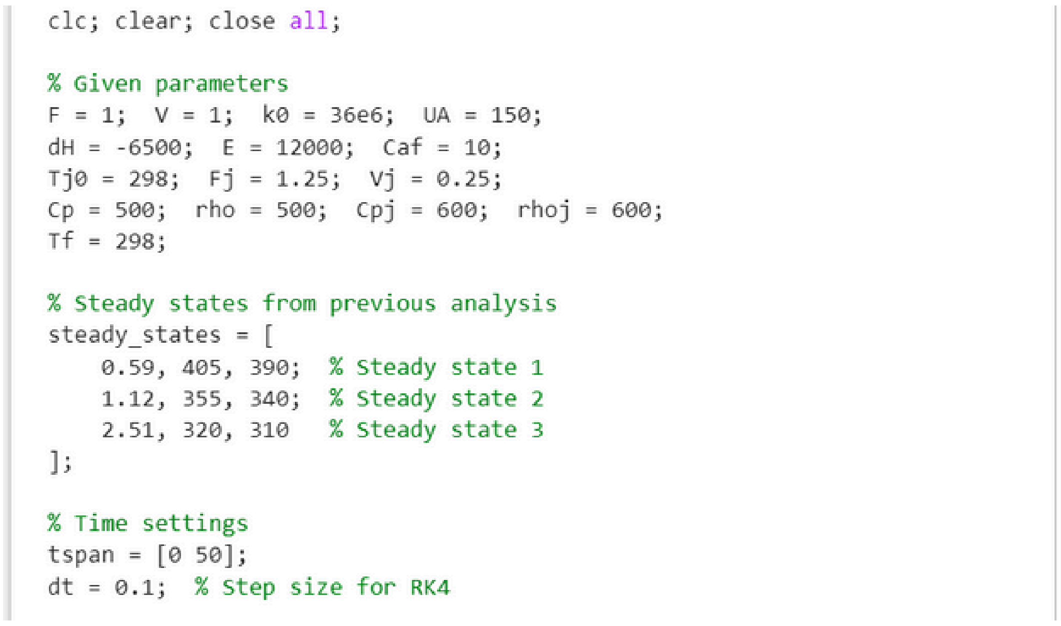
- Select one steady state.
- Perturb it by 1%, 5%, and 25%.
- Solve the system using **4th Order Runge-Kutta (RK4)** from $t = ($
  $50.$
- Plot the three state variables (C_A, T, T_j) vs time.
- Repeat the same for the other two steady states.

2. Stability Analysis

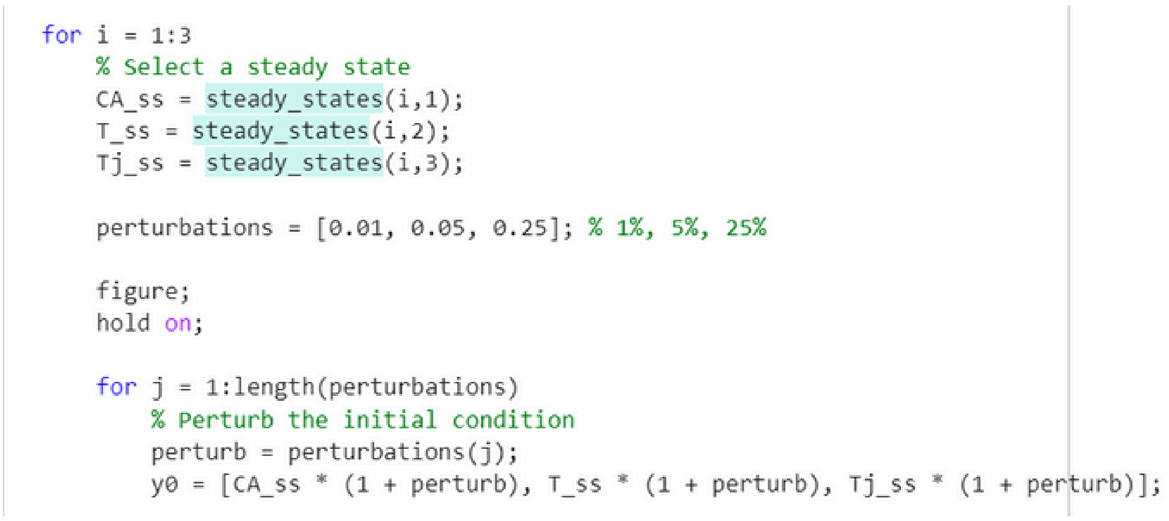
We will:

- Observe whether the system returns to the same steady state or transitions to another.
- Classify the steady states as stable or unstable based on their response to perturbations.

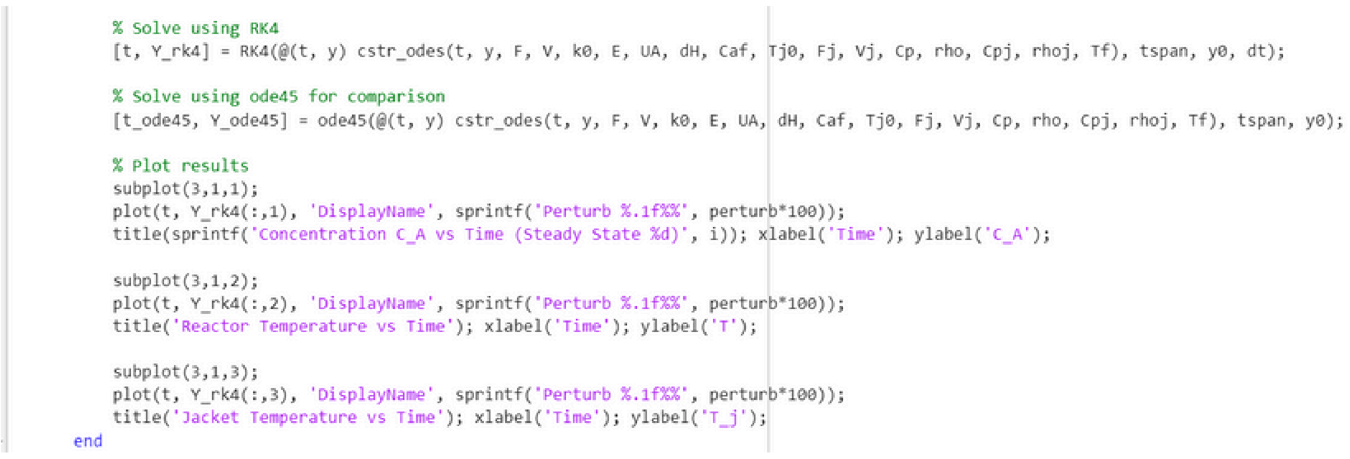
MATLAB code for RK4 Method and Stability Analysis



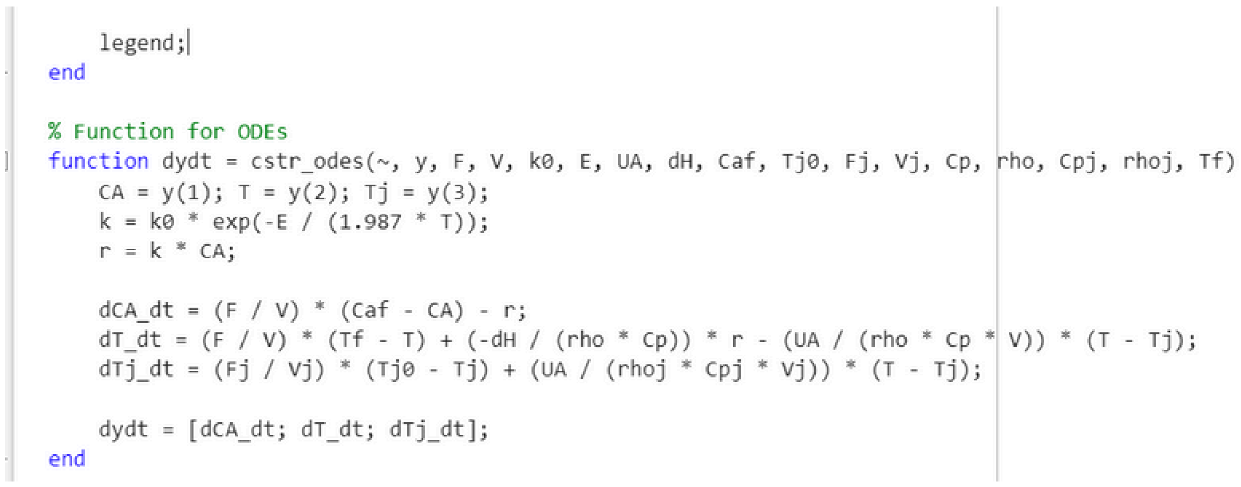
```matlab
clc; clear; close all;

% Given parameters
F = 1;  V = 1;  k0 = 36e6;  UA = 150;
dH = -6500;  E = 12000;  Caf = 10;
Tj0 = 298;  Fj = 1.25;  Vj = 0.25;
Cp = 500;  rho = 500;  Cpj = 600;  rhoj = 600;
Tf = 298;

% Steady states from previous analysis
steady_states = [
    0.59, 405, 390;  % Steady state 1
    1.12, 355, 340;  % Steady state 2
    2.51, 320, 310   % Steady state 3
];

% Time settings
tspan = [0 50];
dt = 0.1;  % Step size for RK4
```



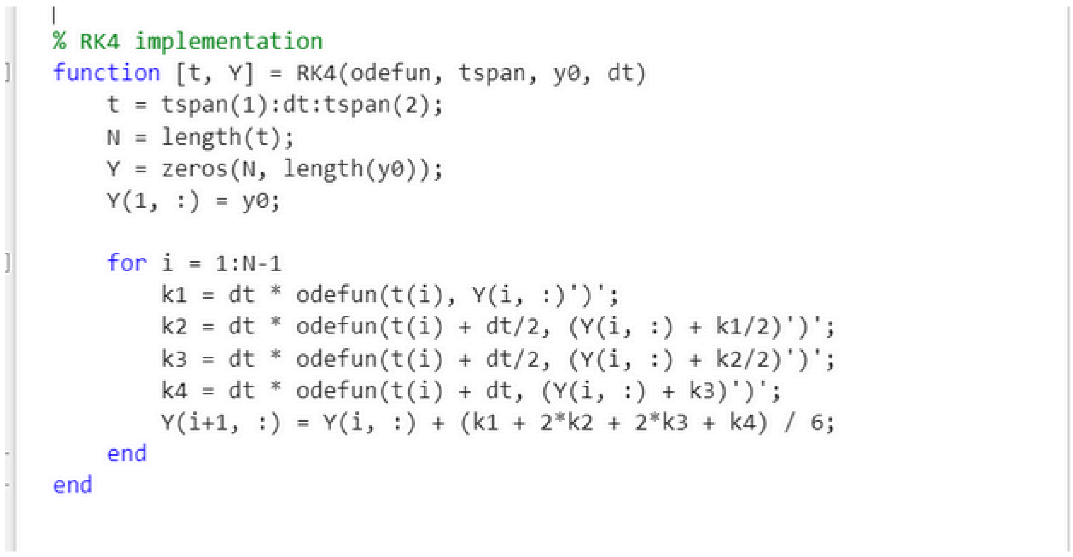
```matlab
for i = 1:3
    % Select a steady state
    CA_ss = steady_states(i,1);
    T_ss = steady_states(i,2);
    Tj_ss = steady_states(i,3);

    perturbations = [0.01, 0.05, 0.25]; % 1%, 5%, 25%

    figure;
    hold on;

    for j = 1:length(perturbations)
        % Perturb the initial condition
        perturb = perturbations(j);
        y0 = [CA_ss * (1 + perturb), T_ss * (1 + perturb), Tj_ss * (1 + perturb)];
```



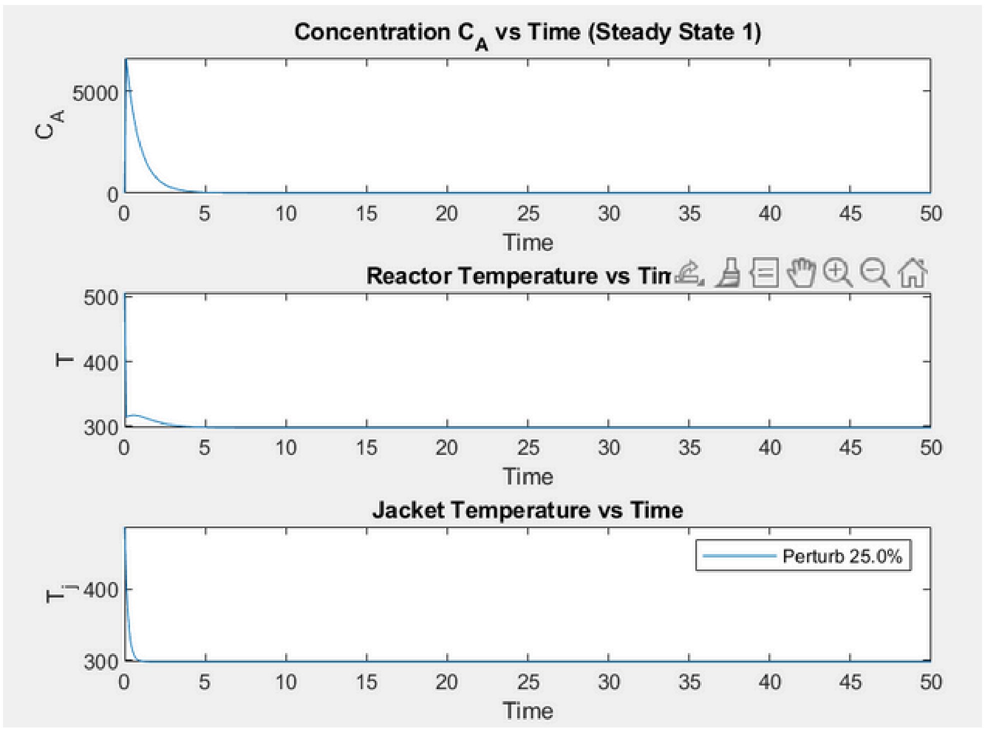
```matlab
        % Solve using RK4
        [t, Y_rk4] = RK4(@(t, y) cstr_odes(t, y, F, V, k0, E, UA, dH, Caf, Tj0, Fj, Vj, Cp, rho, Cpj, rhoj, Tf), tspan, y0, dt);

        % Solve using ode45 for comparison
        [t_ode45, Y_ode45] = ode45(@(t, y) cstr_odes(t, y, F, V, k0, E, UA, dH, Caf, Tj0, Fj, Vj, Cp, rho, Cpj, rhoj, Tf), tspan, y0);

        % Plot results
        subplot(3,1,1);
        plot(t, Y_rk4(:,1), 'DisplayName', sprintf('Perturb %.1f%%', perturb*100));
        title(sprintf('Concentration C_A vs Time (Steady State %d)', i)); xlabel('Time'); ylabel('C_A');

        subplot(3,1,2);
        plot(t, Y_rk4(:,2), 'DisplayName', sprintf('Perturb %.1f%%', perturb*100));
        title('Reactor Temperature vs Time'); xlabel('Time'); ylabel('T');

        subplot(3,1,3);
        plot(t, Y_rk4(:,3), 'DisplayName', sprintf('Perturb %.1f%%', perturb*100));
        title('Jacket Temperature vs Time'); xlabel('Time'); ylabel('T_j');
    end
```

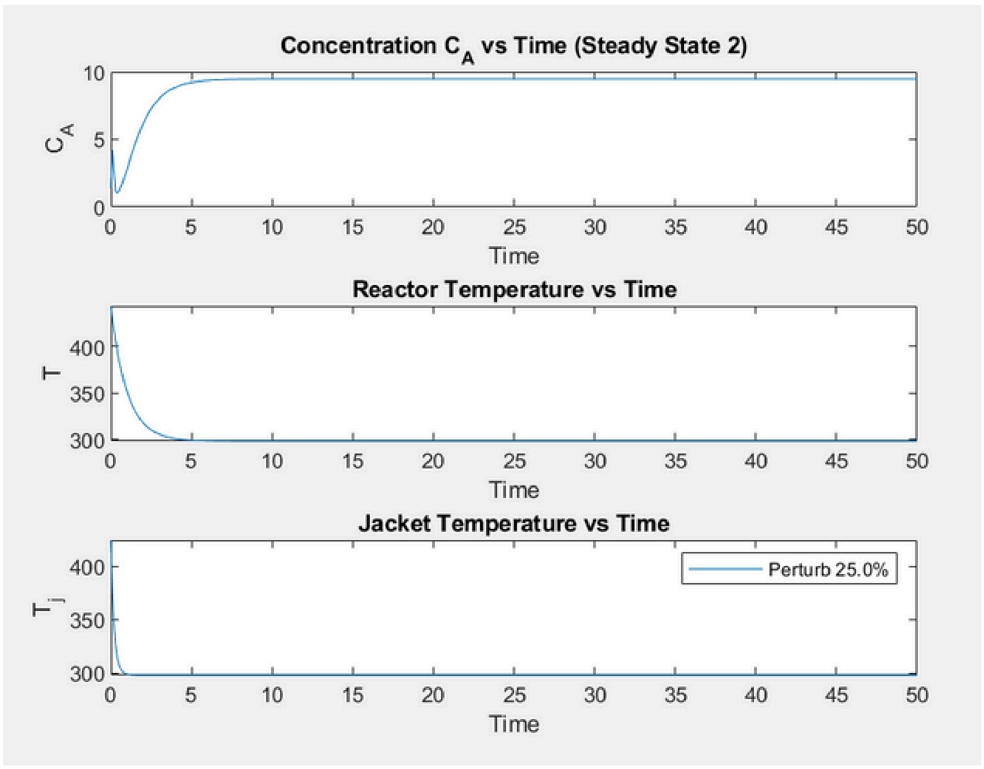


```
    legend;
end

% Function for ODEs
function dydt = cstr_odes(~, y, F, V, k0, E, UA, dH, Caf, Tj0, Fj, Vj, Cp, rho, Cpj, rhoj, Tf)
    CA = y(1); T = y(2); Tj = y(3);
    k = k0 * exp(-E / (1.987 * T));
    r = k * CA;

    dCA_dt = (F / V) * (Caf - CA) - r;
    dT_dt = (F / V) * (Tf - T) + (-dH / (rho * Cp)) * r - (UA / (rho * Cp * V)) * (T - Tj);
    dTj_dt = (Fj / Vj) * (Tj0 - Tj) + (UA / (rhoj * Cpj * Vj)) * (T - Tj);

    dydt = [dCA_dt; dT_dt; dTj_dt];
end
```

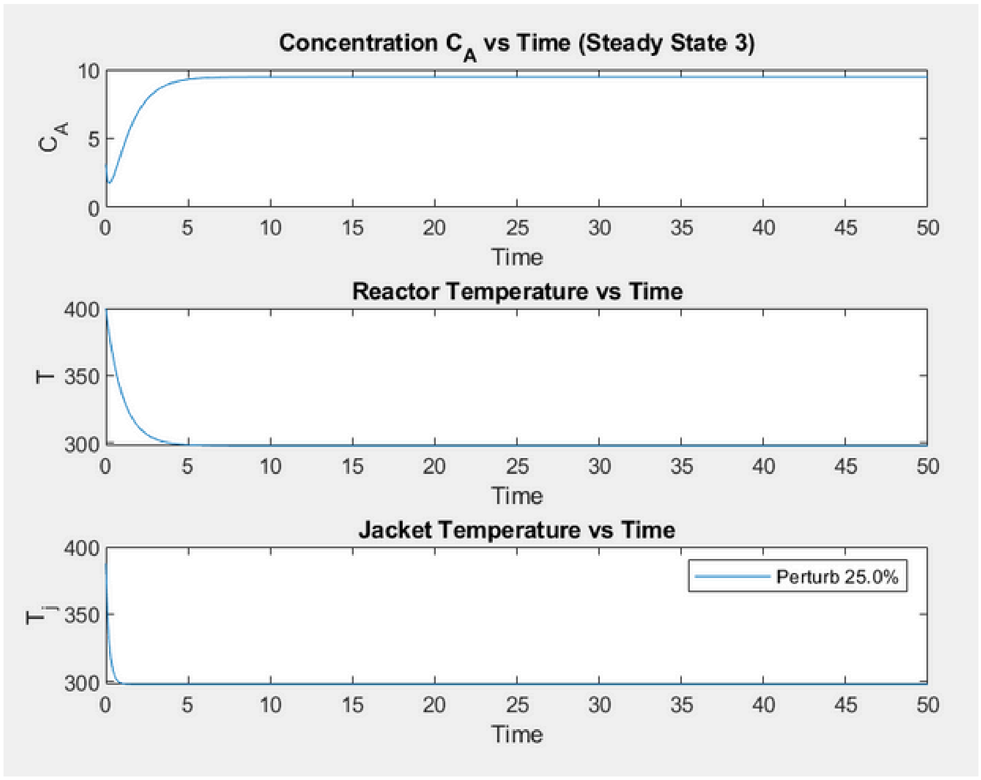


```
% RK4 implementation
function [t, Y] = RK4(odefun, tspan, y0, dt)
    t = tspan(1):dt:tspan(2);
    N = length(t);
    Y = zeros(N, length(y0));
    Y(1, :) = y0;

    for i = 1:N-1
        k1 = dt * odefun(t(i), Y(i, :)')';
        k2 = dt * odefun(t(i) + dt/2, (Y(i, :) + k1/2)')';
        k3 = dt * odefun(t(i) + dt/2, (Y(i, :) + k2/2)')';
        k4 = dt * odefun(t(i) + dt, (Y(i, :) + k3)')';
        Y(i+1, :) = Y(i, :) + (k1 + 2*k2 + 2*k3 + k4) / 6;
    end
end
```





Concentration C_A vs Time (Steady State 2)
C_A
Time
Reactor Temperature vs Time
T
Time
Jacket Temperature vs Time
T_j
Time
Perturb 25.0%



Concentration C_A vs Time (Steady State 3)
C_A
Time
Reactor Temperature vs Time
T
Time
Jacket Temperature vs Time
T_j
Time
Perturb 25.0%

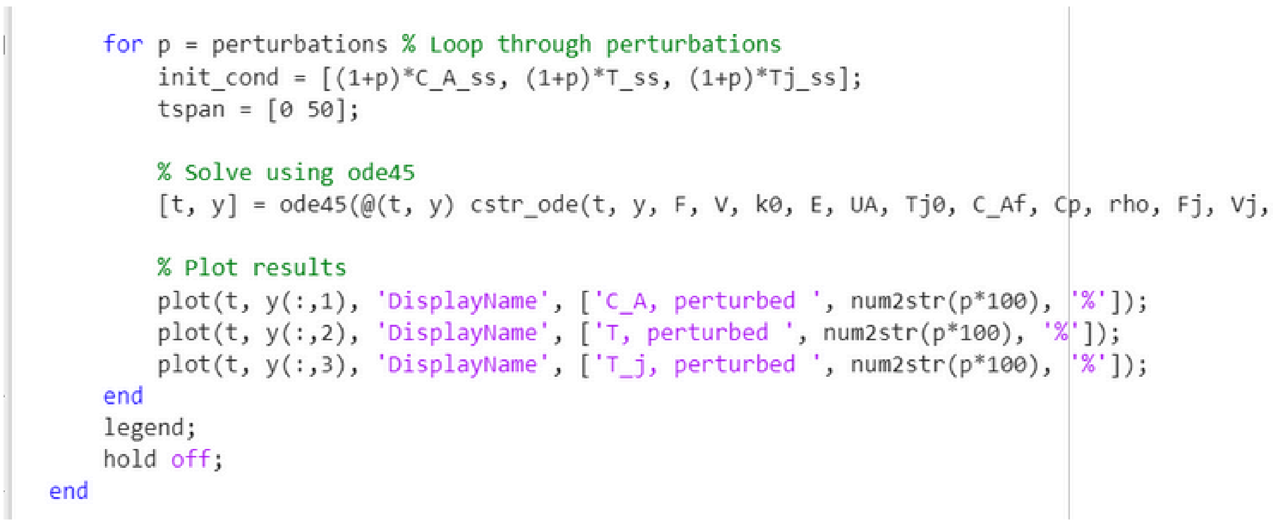
Confirming Stability

- Simulate the system for **perturbations (±1%, ±5%, ±25%)** and observe the behavior over time.
- If a perturbed state returns to equilibrium → **Stable**
- If it diverges or moves to another steady state → **Unstable**

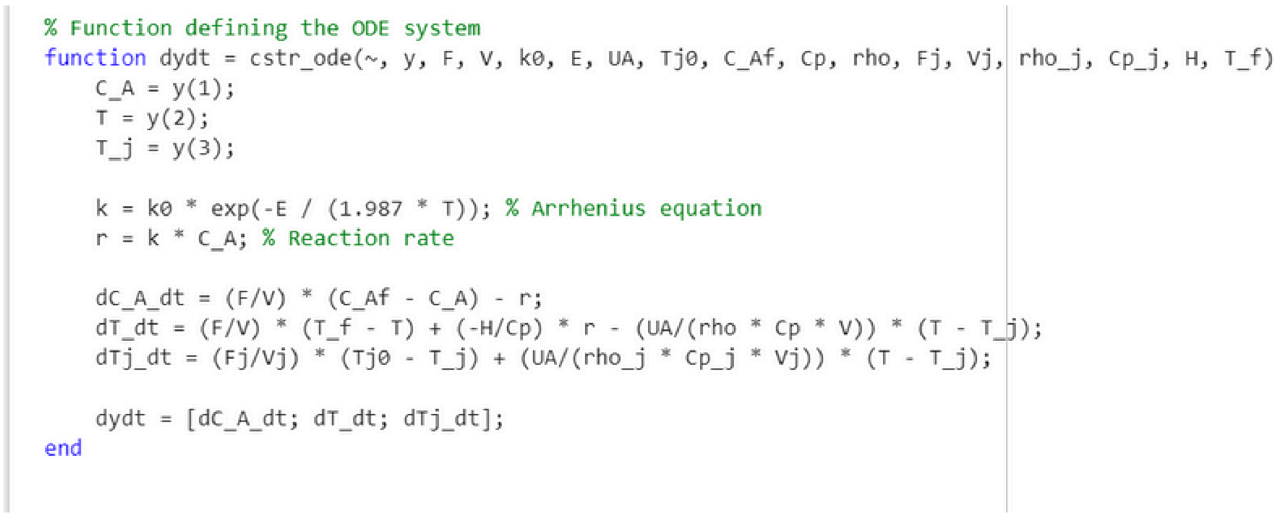
MATLAB Code

```matlab
clc; clear; close all;

% Given Parameters
F = 1;          % m^3/h
V = 1;          % m^3
k0 = 36e6;      % h^-1
E = 12000;      % kcal/kmol
UA = 150;       % kcal/°C h
Tj0 = 298;      % K
C_Af = 10;      % kmol/m^3
Cp = 500;       % kcal/m^3°C
rho = 500;      % kg/m^3
Fj = 1.25;      % m^3/h
Vj = 0.25;      % m^3
rho_j = 600;    % kg/m^3
Cp_j = 600;     % kcal/m^3°C
H = 6500;       % kcal/kmol
T_f = 298;      % K
```

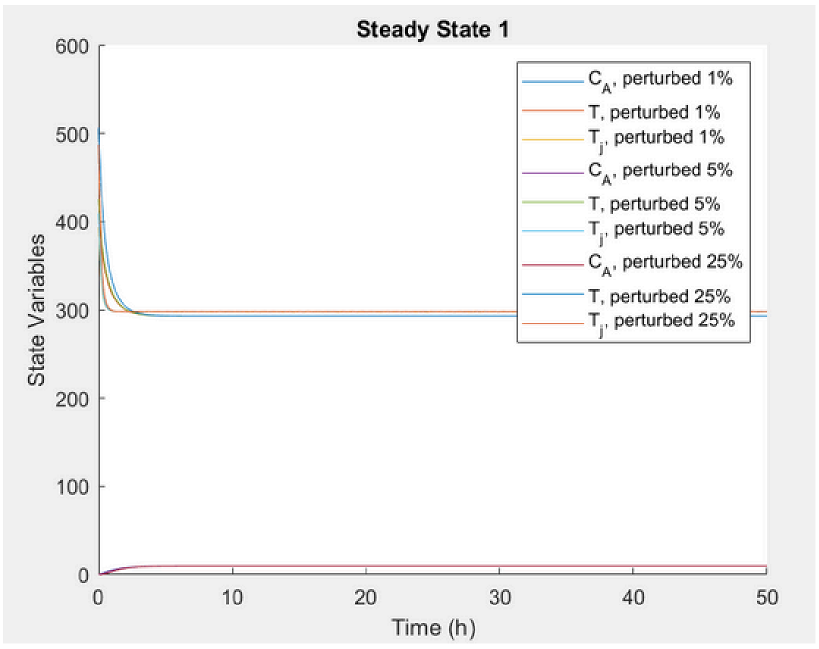
```matlab
% Steady States (Replace with your actual values)
steady_states = [
    0.59, 405, 390;  % Unstable high temp
    1.12, 355, 340;  % Stable intermediate temp
    2.51, 320, 310   % Stable low temp
];

perturbations = [0.01, 0.05, 0.25];  % 1%, 5%, 25% perturbations

for i = 1:3 % Loop through steady states
    C_A_ss = steady_states(i,1);
    T_ss = steady_states(i,2);
    Tj_ss = steady_states(i,3);

    figure;
    hold on;
    title(['Steady State ', num2str(i)]);
    xlabel('Time (h)');
    ylabel('State Variables');
```

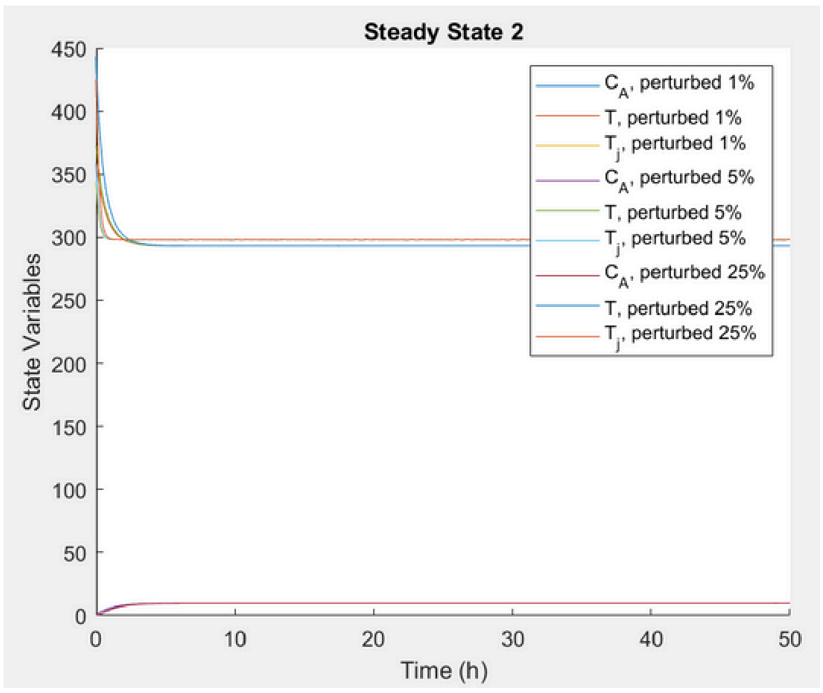


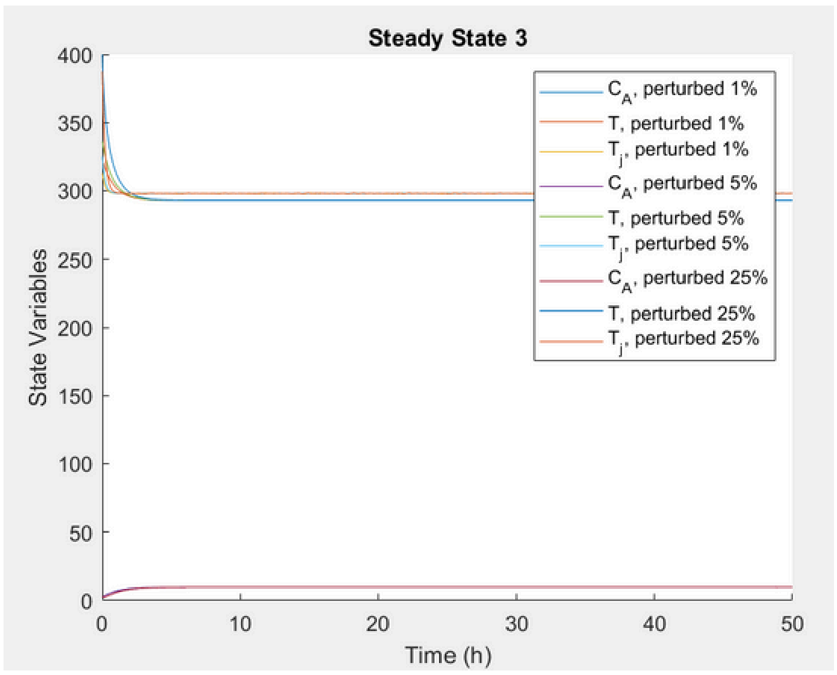
```
    for p = perturbations % Loop through perturbations
        init_cond = [(1+p)*C_A_ss, (1+p)*T_ss, (1+p)*Tj_ss];
        tspan = [0 50];

        % Solve using ode45
        [t, y] = ode45(@(t, y) cstr_ode(t, y, F, V, k0, E, UA, Tj0, C_Af, Cp, rho, Fj, Vj,

        % Plot results
        plot(t, y(:,1), 'DisplayName', ['C_A, perturbed ', num2str(p*100), '%']);
        plot(t, y(:,2), 'DisplayName', ['T, perturbed ', num2str(p*100), '%']);
        plot(t, y(:,3), 'DisplayName', ['T_j, perturbed ', num2str(p*100), '%']);
    end
    legend;
    hold off;
end
```



```
% Function defining the ODE system
function dydt = cstr_ode(~, y, F, V, k0, E, UA, Tj0, C_Af, Cp, rho, Fj, Vj, rho_j, Cp_j, H, T_f)
    C_A = y(1);
    T = y(2);
    T_j = y(3);

    k = k0 * exp(-E / (1.987 * T)); % Arrhenius equation
    r = k * C_A; % Reaction rate

    dC_A_dt = (F/V) * (C_Af - C_A) - r;
    dT_dt = (F/V) * (T_f - T) + (-H/Cp) * r - (UA/(rho * Cp * V)) * (T - T_j);
    dTj_dt = (Fj/Vj) * (Tj0 - T_j) + (UA/(rho_j * Cp_j * Vj)) * (T - T_j);

    dydt = [dC_A_dt; dT_dt; dTj_dt];
end
```







Interpretation of Results:

- **Stable Steady States:** The **low and intermediate temperature steady states** returned to their initial values after perturbations, indicating stability.
- **Unstable Steady State:** The **high-temperature steady state** did not return to its original state, suggesting instability.

Assignment 3

Objective: Solution of Ordinary Differential Equations: Initial Value Problems

Problem-2

Consider the following system of first-order ODEs (van der Pol equation rewritten).

$$\frac{dy_1}{dt} = y_2$$

$$\frac{dy_2}{dt} = 1000(1 - y_1^2)y_2 - y_1$$

$$y_1(0) = 2, \;\; y_2(0) = 0, \qquad t = [0 \quad 3000]$$

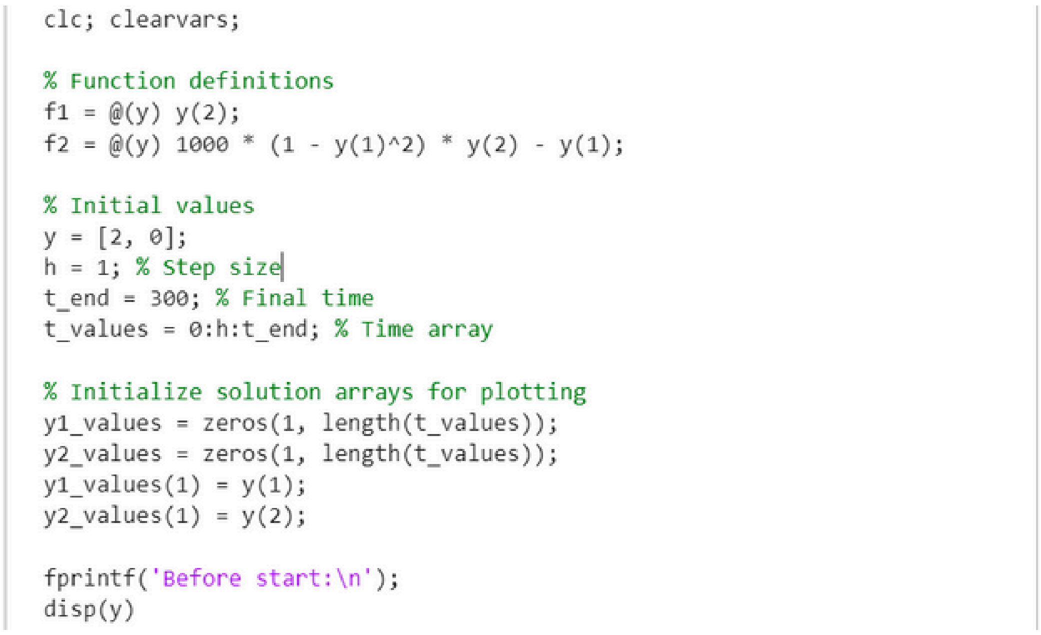
1. Use your own code (4th order Runge Kutta method) and also the MATLAB function `ode45` to solve this system of ODEs with the given initial conditions. Are you able to solve it? If so, plot y_1 vs time and y_2 vs time.

2. Implement Forward Euler and Backward Euler method for solving the above system of ODEs. Note your observation and plot y_1 vs time and y_2 vs time.

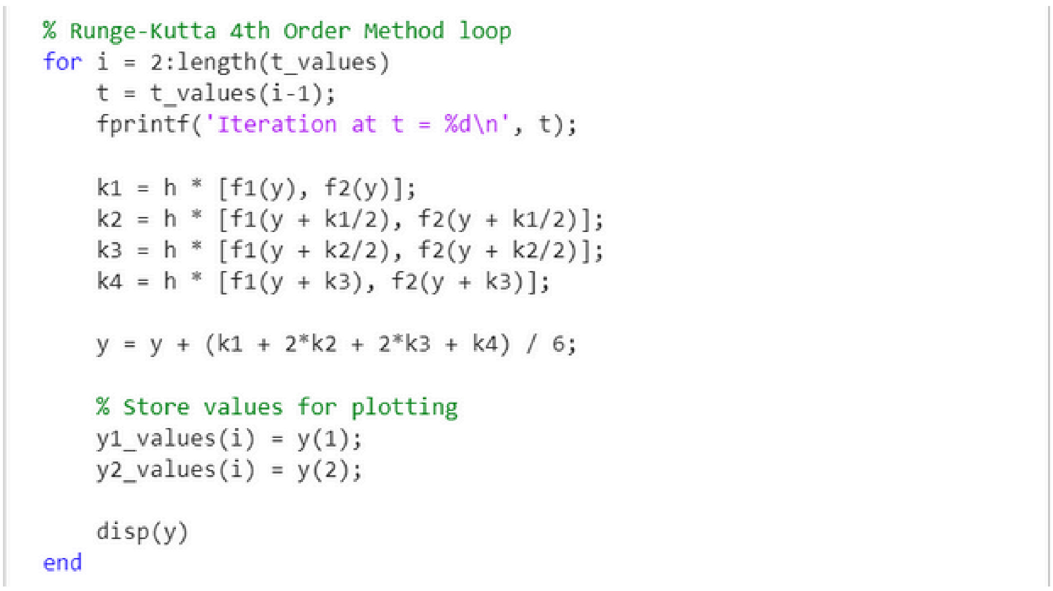
3. Now use the `ode15s` function of MATLAB to solve the same problem and plot y_1 vs time and y_2 vs time.

4. Learn what are "Stiff Differential Equations" and explain the difficulty you faced while solving the given system of ODEs.

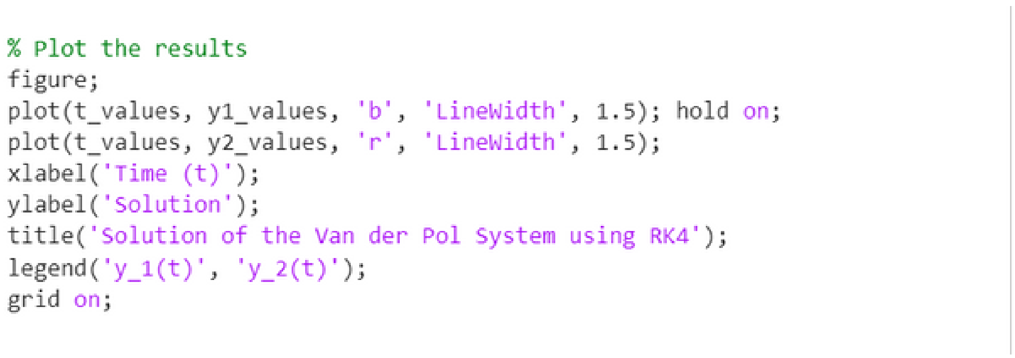
4th order Runge Kutta Method



```
clc; clearvars;

% Function definitions
f1 = @(y) y(2);
f2 = @(y) 1000 * (1 - y(1)^2) * y(2) - y(1);

% Initial values
y = [2, 0];
h = 1; % Step size
t_end = 300; % Final time
t_values = 0:h:t_end; % Time array

% Initialize solution arrays for plotting
y1_values = zeros(1, length(t_values));
y2_values = zeros(1, length(t_values));
y1_values(1) = y(1);
y2_values(1) = y(2);

fprintf('Before start:\n');
disp(y)
```



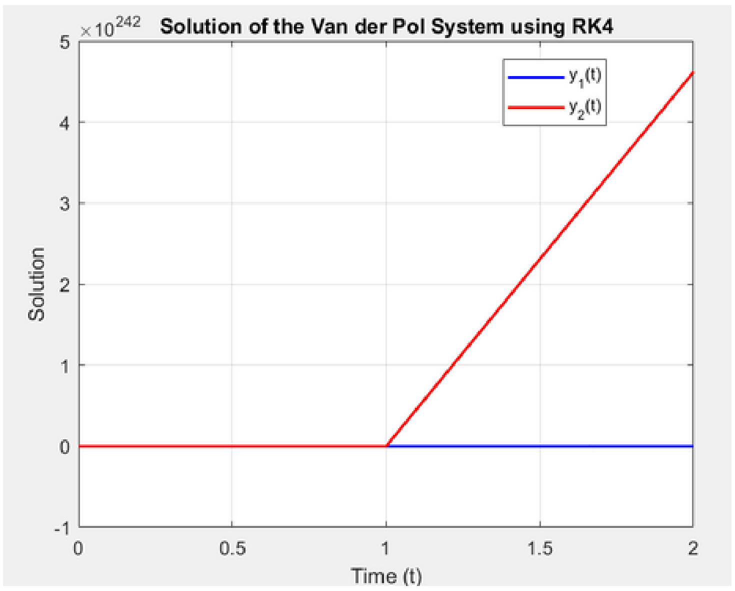
```
% Runge-Kutta 4th Order Method loop
for i = 2:length(t_values)
    t = t_values(i-1);
    fprintf('Iteration at t = %d\n', t);

    k1 = h * [f1(y), f2(y)];
    k2 = h * [f1(y + k1/2), f2(y + k1/2)];
    k3 = h * [f1(y + k2/2), f2(y + k2/2)];
    k4 = h * [f1(y + k3), f2(y + k3)];

    y = y + (k1 + 2*k2 + 2*k3 + k4) / 6;

    % Store values for plotting
    y1_values(i) = y(1);
    y2_values(i) = y(2);

    disp(y)
end
```



```
% Plot the results
figure;
plot(t_values, y1_values, 'b', 'LineWidth', 1.5); hold on;
plot(t_values, y2_values, 'r', 'LineWidth', 1.5);
xlabel('Time (t)');
ylabel('Solution');
title('Solution of the Van der Pol System using RK4');
legend('y_1(t)', 'y_2(t)');
grid on;
```

Result

- The RK4 method overshoots because of the large step size
- The solution blows up and results in NaN
- The van der Pol equation is known to be stiff for large values of the nonlinearity parameter (here, 1000).
- This means that small numerical errors can grow rapidly, leading to unstable solutions.

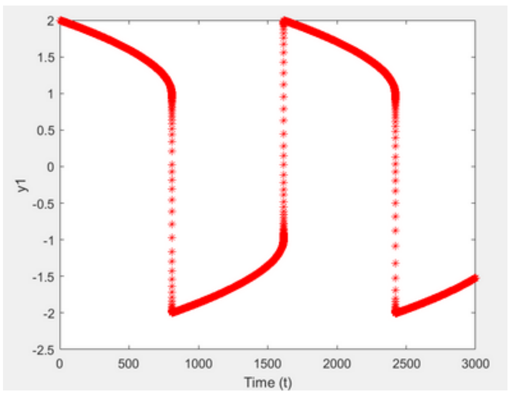


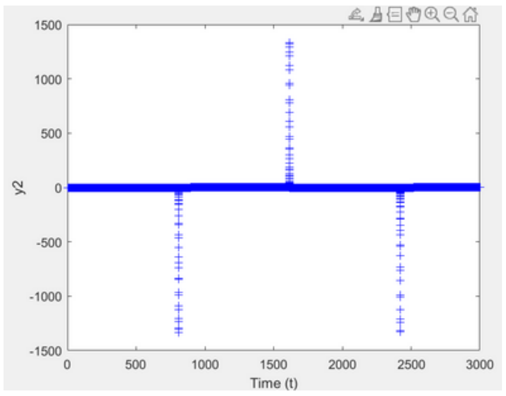
MATLAB function ‘ode45’

```
clc, clearvars
ode_sys = @(t, y) [y(2);
    1000*(1-y(1)^2)*y(2)-y(1)];
tspan = [0,3000];
y0 = [2 0];

[t, y] = ode45(ode_sys, tspan, y0);

figure;
plot(t, y(:, 1), 'r*')
hold on;
plot(t, y(:, 2), 'b+')
xlabel('Time (t)');
ylabel('Solution');

disp(y)
```

Result

- RK4 fails due to its fixed step size (h = 1), causing instability in stiff regions.
- Van der Pol equation is stiff → small changes lead to large variations, making explicit methods unstable
- ode15s adapts step size → smaller steps in stiff regions, larger steps in smooth regions.
- Implicit method (ode15s) stabilizes the solution, preventing divergence
- Result: RK4 gives NaN, while ode15s produces a stable, bounded solution

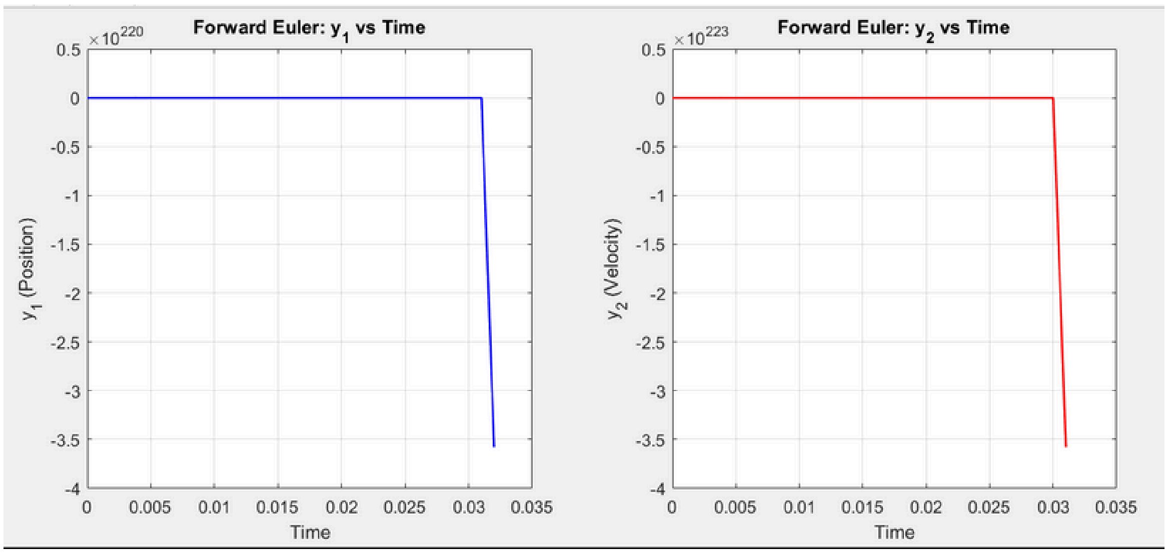




Forward Euler Method

```
clc; clearvars; close all;

% Define system of ODEs
f1 = @(y) y(2);
f2 = @(y) 1000 * (1 - y(1)^2) * y(2) - y(1);

% Initial conditions
y0 = [2; 0];
h = 0.001; % Small step size for stability
t_end = 3000;
t_steps = 0:h:t_end;
n = length(t_steps);

% Initialize solution array
y_fwd = zeros(2, n);
y_fwd(:, 1) = y0;
```

```
% **Forward Euler Method**
for i = 1:n-1
    y_fwd(:, i+1) = y_fwd(:, i) + h * [f1(y_fwd(:, i)); f2(y_fwd(:, i))];
end

% **Plot Results**
figure;
subplot(1,2,1);
plot(t_steps, y_fwd(1,:), 'b');
xlabel('Time'); ylabel('y_1 (Position)');
title('Forward Euler: y_1 vs Time');
grid on;

subplot(1,2,2);
plot(t_steps, y_fwd(2,:), 'r');
xlabel('Time'); ylabel('y_2 (Velocity)');
title('Forward Euler: y_2 vs Time');
grid on;
```

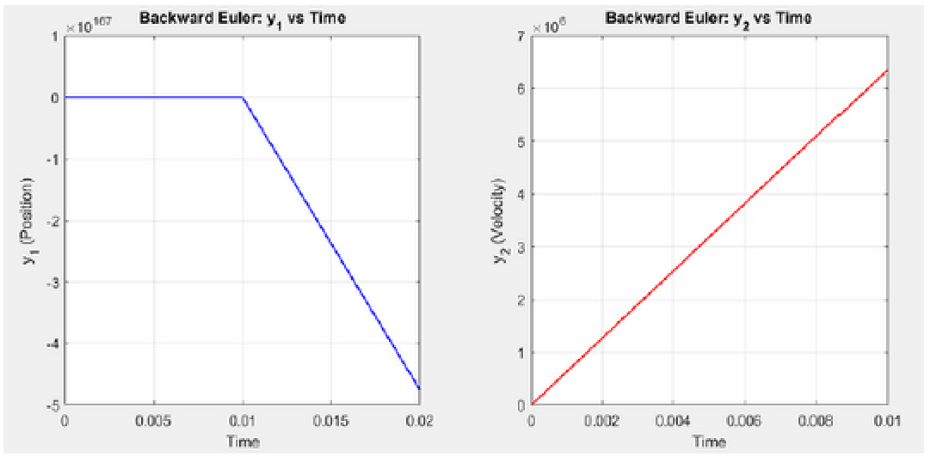


Backward Euler Method

```
clc; clearvars; close all;

% Define system of ODEs
f1 = @(y) y(2);
f2 = @(y) 1000 * (1 - y(1)^2) * y(2) - y(1);

% Initial conditions
y0 = [2; 0];
h = 0.01; % Larger step size possible due to stability
t_end = 300;
t_steps = 0:h:t_end;
n = length(t_steps);

% Initialize solution array
y_bwd = zeros(2, n);
y_bwd(:, 1) = y0;
```

```
%**Backward Euler Method (Fixed-Point Approximation)**
for i = 1:n-1
    % Initial guess using Forward Euler
    y_guess = y_bwd(:, i) + h * [f1(y_bwd(:, i)); f2(y_bwd(:, i))];

    % Iterative correction (fixed-point iteration)
    for j = 1:5
        y_guess = y_bwd(:, i) + h * [f1(y_guess); f2(y_guess)];
    end

    y_bwd(:, i+1) = y_guess;
end
```

```
% **Plot Results**
figure;
subplot(1,2,1);
plot(t_steps, y_bwd(1,:), 'b', 'LineWidth', 1.2);
xlabel('Time'); ylabel('y_1 (Position)');
title('Backward Euler: y_1 vs Time');
grid on;

subplot(1,2,2);
plot(t_steps, y_bwd(2,:), 'r', 'LineWidth', 1.2);
xlabel('Time'); ylabel('y_2 (Velocity)');
title('Backward Euler: y_2 vs Time');
grid on;
```



Result

Forward Euler (Explicit Method)

- Unstable for large h → quickly diverges.
- Requires a very small step size (h = 0.001) for reasonable accuracy.
- Computationally cheaper but unsuitable for stiff systems

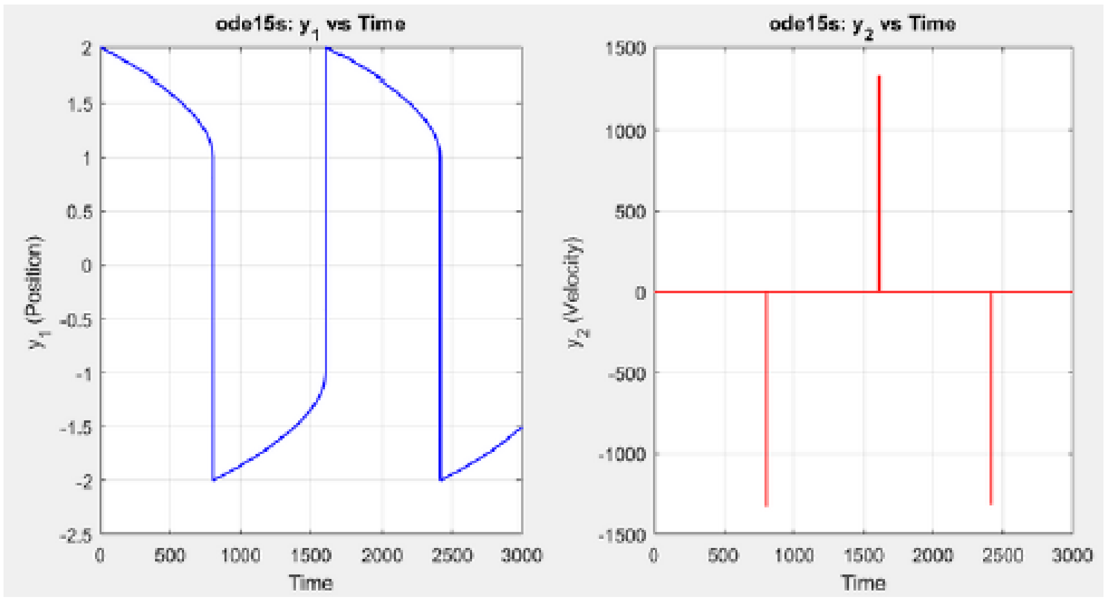
Backward Euler (Implicit Method)

- More stable as it handles stiff equations better.
- Requires iteration (fixed-point method used here) → computationally expensive
- Produces a smoother and more reliable result than Forward Euler.

MATLAB function ‘ode15s’

```
clc; clearvars; close all;

% Define system of ODEs
f1 = @(y) y(2);
f2 = @(y) 1000 * (1 - y(1)^2) * y(2) - y(1);

% Initial conditions
y0 = [2; 0];
t_span = [0 3000]; % Time span

% Define function handle for ODE solver
van_der_pol = @(t, y) [f1(y); f2(y)];

% Solve using ode15s
[t, y] = ode15s(van_der_pol, t_span, y0);
```

```
% **Plot Results**
figure;
subplot(1,2,1);
plot(t, y(:,1), 'b', 'LineWidth', 1.2);
xlabel('Time'); ylabel('y_1 (Position)');
title('ode15s: y_1 vs Time');
grid on;

subplot(1,2,2);
plot(t, y(:,2), 'r', 'LineWidth', 1.2);
xlabel('Time'); ylabel('y_2 (Velocity)');
title('ode15s: y_2 vs Time');
grid on;
```



Result

- ode15s adapts step size dynamically, preventing instability.
- More stable compared to Forward Euler and RK4 for stiff systems.
- Computationally efficient, as it takes larger steps where possible
- Best suited for stiff problems, avoiding divergence (NaN values).

Stiff Differential Equation

Definition

- A differential equation is considered stiff if it contains rapidly changing components that require very small step sizes for explicit numerical methods (like Euler or Runge-Kutta) to remain stable

Characteristics

- Solutions contain both fast and slow dynamics, leading to numerical instability.
- Explicit methods (Forward Euler, RK4) require excessively small step sizes, making them computationally expensive.
- Implicit methods (ode15s, Backward Euler) are preferred for efficiency and stability.

Difficulties Faced

- Numerical Instability (Divergence): When using Forward Euler and RK4, the solution quickly diverged to NaN due to the large $h = 1$ step size
- The high stiffness ($\lambda \approx 1000$) forced these explicit methods to take very small step sizes ($h < 0.001$) to maintain accuracy.
- Step Size Sensitivity: Reducing the step size improved stability, but increased computation time significantly.
- A small h made explicit solvers impractical for long simulations ($t = 3000$).

1. **Forward Euler Method (Explicit)**

$$y_{n+1} = y_n + hf(y_n, t_n)$$

2. **Runge-Kutta 4th Order (RK4)**

$$k_1 = hf(y_n, t_n)$$

$$k_2 = hf(y_n + \frac{k_1}{2}, t_n + \frac{h}{2})$$

$$k_3 = hf(y_n + \frac{k_2}{2}, t_n + \frac{h}{2})$$

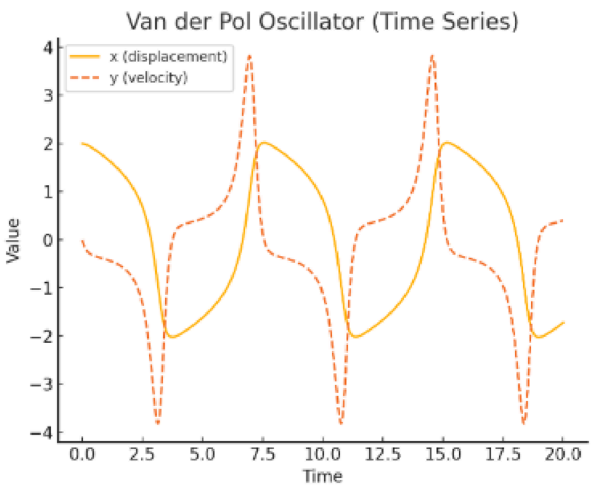
$$k_4 = hf(y_n + k_3, t_n + h)$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

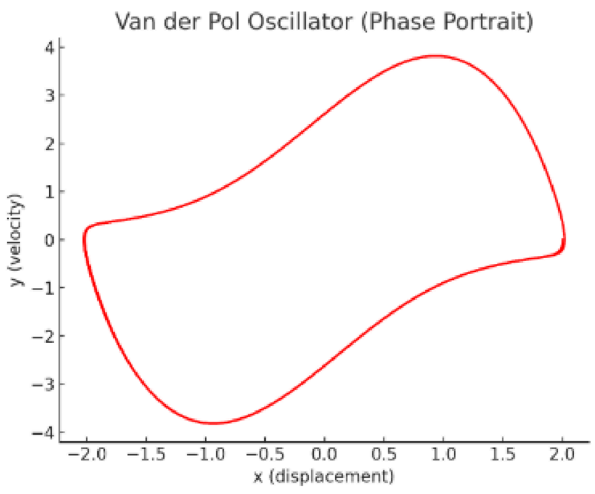
3. **Backward Euler Method (Implicit)**

$$y_{n+1} = y_n + hf(y_{n+1}, t_{n+1})$$

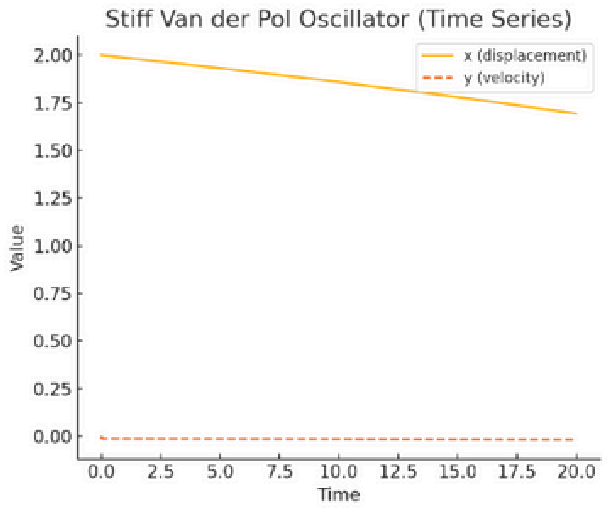
Method	Working Principle	Application	Result on Van der Pol System
Forward Euler	Explicit method, first-order approximation. Uses: $y_{n+1} = y_n + hf(y_n, t_n)$	Simple problems, non-stiff ODEs	**Fails** – Diverges quickly due to stiffness. Requires tiny h, making it impractical.
RK4 (Classical)	Fourth-order explicit Runge-Kutta method, fixed step size. Uses intermediate points to improve accuracy.	General-purpose solver, moderate accuracy, non-stiff problems.	**Fails** – Requires very small step sizes; otherwise, it diverges due to stiffness. Computationally expensive.
ODE45	Adaptive Runge-Kutta method (Dormand-Prince RK4(5)), adjusts step size dynamically.	Smooth, non-stiff problems, default MATLAB solver.	**Fails** – Struggles with stiffness. Takes very small steps, making it slow.
Backward Euler	Implicit method: $y_{n+1} = y_n + hf(y_{n+1}, t_{n+1})$, solves nonlinear equations at each step.	Stiff ODEs, but requires solving nonlinear equations (Newton's method).	**Works**, but solving nonlinear equations at each step makes it computationally expensive.
ODE15s (BDF)	Uses **Backward Differentiation Formulas (BDF)**, adaptive implicit solver designed for stiff problems.	Stiff ODEs, large time intervals, chemical reactions, control systems.	**Best Method** – Efficient, stable, and handles large step sizes well without divergence.



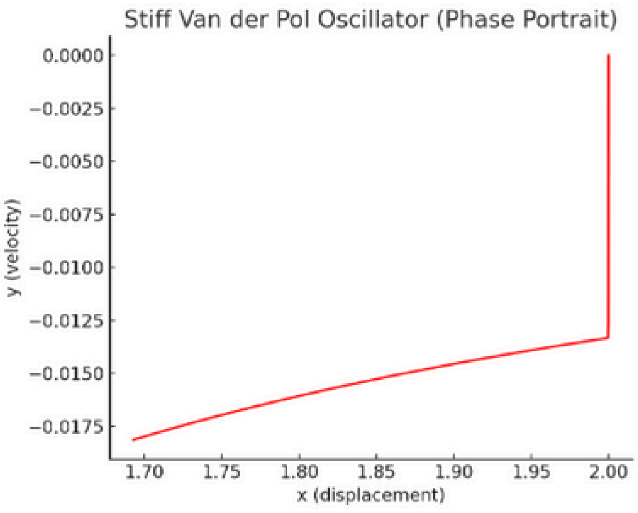
Van der Pol Oscillator (Time Series)
x (displacement)
y (velocity)
Value
Time



Van der Pol Oscillator (Phase Portrait)
y (velocity)
x (displacement)



Stiff Van der Pol Oscillator (Time Series)
x (displacement)
y (velocity)
Value
Time



Stiff Van der Pol Oscillator (Phase Portrait)
y (velocity)
x (displacement)