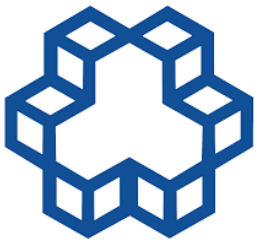


K.N. Toosi University

Aidin Sahneh

Student ID: 40120243

Amirhossein Shaqaqi

Student ID: 40119843

Course: Modern Control

Instructor: Dr. Moaveni

Contents

1 Question 1 **2**

2 Question 2 **3**

3 Question 3 **10**

4 Question 4 **15**

5 Question 5 **23**

6 Question 6 **25**

7 Question 7 **28**

8 Question 8 **31**

9 Question 9 **33**

10 Question 10 **36**

11 Question 11 **39**

12 Question 12 **43**

13 Question 13 **47**

1 Question 1

Based on the relevant physics, derive the non-linear differential equations governing the system's behavior, and derive the state-space representation of the system.

Answer

1.1 The Free Gyroscope Search System

This is a non-linear system with two degrees of freedom, which can move along the angles θ and φ. The non-linear differential equations that describe the system's behavior are as follows:

$$I_r \cos(\varphi)\ddot{\theta} - I_R\omega_s\dot{\varphi} = T_z$$
$$I_r\ddot{\varphi} + I_R\omega_s\dot{\theta}\cos(\varphi) = T_y$$

In the equations above, I_r is the moment of inertia of the gimbal ring around the Pitch or Yaw axes, and I_R is the moment of inertia of the gyroscope rotor around the spin axis. Furthermore, ω_s is the angular velocity of the rotor's spin, T_y is the applied torque around the Pitch axis, and T_z is the applied torque around the Yaw axis.

The numerical values for the system parameters are considered as follows:

$$I_r = 0.05, \quad I_R = 0.01, \quad \omega_s = 500\,\text{rad/s}$$

1.2 State Variable Definitions

The system's state variables are defined as follows:

$$x_1 = \theta, \quad x_2 = \dot{\theta}, \quad x_3 = \varphi, \quad x_4 = \dot{\varphi}$$

As a result, the second derivatives of the angles are:

$$\ddot{\theta} = \dot{x}_2, \quad \ddot{\varphi} = \dot{x}_4$$

1.3 State-Space Representation

By substituting the state variables into the dynamic equations and rearranging, the non-linear state-space model is derived. The first step is to isolate the second-order derivatives:

$$\ddot{\theta} = \dot{x}_2 = \frac{1}{I_r\cos(\varphi)}(T_z + I_R\omega_s\dot{\varphi})$$
$$\ddot{\varphi} = \dot{x}_4 = \frac{1}{I_r}(T_y - I_R\omega_s\dot{\theta}\cos(\varphi))$$

Substituting the state variables and numerical parameter values yields the final set of equations:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \dfrac{T_z + 5x_4}{0.05\cos(x_3)} \\ \dot{x}_3 = x_4 \\ \dot{x}_4 = \dfrac{T_y - 5x_2\cos(x_3)}{0.05} \end{cases}$$

2 Question 2

Solve the non-linear state-space equations using MATLAB. Analyze the system's behavior to verify the model's correctness and examine the open-loop performance under various initial conditions.

Answer

2.1 Simulation Methodology

The non-linear state-space model was simulated in MATLAB using the `ode45` solver in an open-loop configuration ($T_y = 0, T_z = 0$). The system's response was evaluated for three different initial conditions over 20 seconds.

2.2 MATLAB Implementation

The implementation consists of the function file and the main simulation script.

Function File (`nonlinear_system.m`):

Function: nonlinear_system.m

```
function dxdt = nonlinear_system(t, x, Tz_func, Ty_func)
% State variables: x(1)=theta, x(2)=theta_dot,
% x(3)=phi, x(4)=phi_dot

% System Parameters
Ir = 0.05;
IRws = 5; % IR * ws = 0.01 * 500

% Get input torques at the current time t
Tz = Tz_func(t);
Ty = Ty_func(t);
```

```
% Initialize the state derivative vector
dxdt = zeros(4,1);

% State-space equations from Question 1
dxdt(1) = x(2);
dxdt(2) = (Tz + IRws * x(4)) ...
/ (Ir * cos(x(3)));
dxdt(3) = x(4);
dxdt(4) = (Ty - IRws * x(2) * cos(x(3))) / Ir;
end
```

Main Simulation Script:

Main Simulation Script

```
clear; clc; close all;
% --- Input Parameters (Open Loop) ---
Tz_func = @(t) 0; % Tz is constant zero
Ty_func = @(t) 0; % Ty is constant zero

% --- Different Initial Conditions ---
x0_case1 = [0; 0; 0; 0];
x0_case2 = [0.1; 0.5; pi/4; 0.1];
x0_case3 = [1; 0; pi/4; 0];

% --- Simulation Time Span ---
tspan = [0 20];

%% --- Case 1: Zero Initial Conditions ---
fprintf('Simulating Case 1...\n');
[t1, x1] = ode45(@(t, x)
nonlinear_system(t, x, Tz_func, Ty_func), tspan, x0_case1);

fig1 = figure('Name', 'Case 1: Response from Origin');
subplot(4,1,1);
plot(t1, x1(:,1), 'LineWidth', 1.5);
title('System Behavior for x_0 = [0, 0, 0, 0]');
ylabel('x_1 (\theta)');
grid on;

subplot(4,1,2);
plot(t1, x1(:,2), 'LineWidth', 1.5);
ylabel('x_2 (d\theta/dt)');
grid on;
```

```
subplot(4,1,3);
plot(t1, x1(:,3), 'LineWidth', 1.5);
ylabel('x_3 (\phi)');
grid on;

subplot(4,1,4);
plot(t1, x1(:,4), 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('x_4 (d\phi/dt)');
grid on;
saveas(fig1, 'case1.png'); % Save the figure

%% --- Case 2: Non-Zero Initial Conditions ---
fprintf('Simulating Case 2...\n');
[t2, x2] = ode45(@(t, x)
nonlinear_system(t, x, Tz_func, Ty_func), tspan, x0_case2);

fig2 = figure('Name', 'Case 2: Response from Non-Zero State');
subplot(4,1,1);
plot(t2, x2(:,1), 'LineWidth', 1.5);
title('System Behavior for x_0 = [0.1, 0.5, \pi/4, 0.1]');
ylabel('x_1 (\theta)');
grid on;

subplot(4,1,2);
plot(t2, x2(:,2), 'LineWidth', 1.5);
ylabel('x_2 (d\theta/dt)');
grid on;

subplot(4,1,3);
plot(t2, x2(:,3), 'LineWidth', 1.5);
ylabel('x_3 (\phi)');
grid on;

subplot(4,1,4);
plot(t2, x2(:,4), 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('x_4 (d\phi/dt)');
grid on;
saveas(fig2, 'case2.png'); % Save the figure

%% --- Case 3: Non-Zero Position, Zero Velocity ---
fprintf('Simulating Case 3...\n');
[t3, x3] = ode45(@(t, x)
```

```
nonlinear_system(t, x, Tz_func, Ty_func), tspan, x0_case3);

fig3 = figure('Name', 'Case 3:
Response from Non-Zero Position');
subplot(4,1,1);
plot(t3, x3(:,1), 'LineWidth', 1.5);
title('System Behavior for x_0 = [1, 0, \pi/4, 0]');
ylabel('x_1 (\theta)');
grid on;

subplot(4,1,2);
plot(t3, x3(:,2), 'LineWidth', 1.5);
ylabel('x_2 (d\theta/dt)');
grid on;

subplot(4,1,3);
plot(t3, x3(:,3), 'LineWidth', 1.5);
ylabel('x_3 (\phi)');
grid on;

subplot(4,1,4);
plot(t3, x3(:,4), 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('x_4 (d\phi/dt)');
grid on;
saveas(fig3, 'case3.png'); % Save the figure

fprintf('Simulations complete
. Figures saved as case1.png, case2.png, case3.png.\n');
```

2.3 Results and Analysis

The open-loop simulation results for the three cases are presented and analyzed below.

Case 1: Zero Initial Conditions ($x_0 = [0, 0, 0, 0]$)

When starting from the origin with zero input, all state variables remain at zero for the entire simulation. This confirms that the point $[0, 0, 0, 0]^T$ is an equilibrium point of the system.

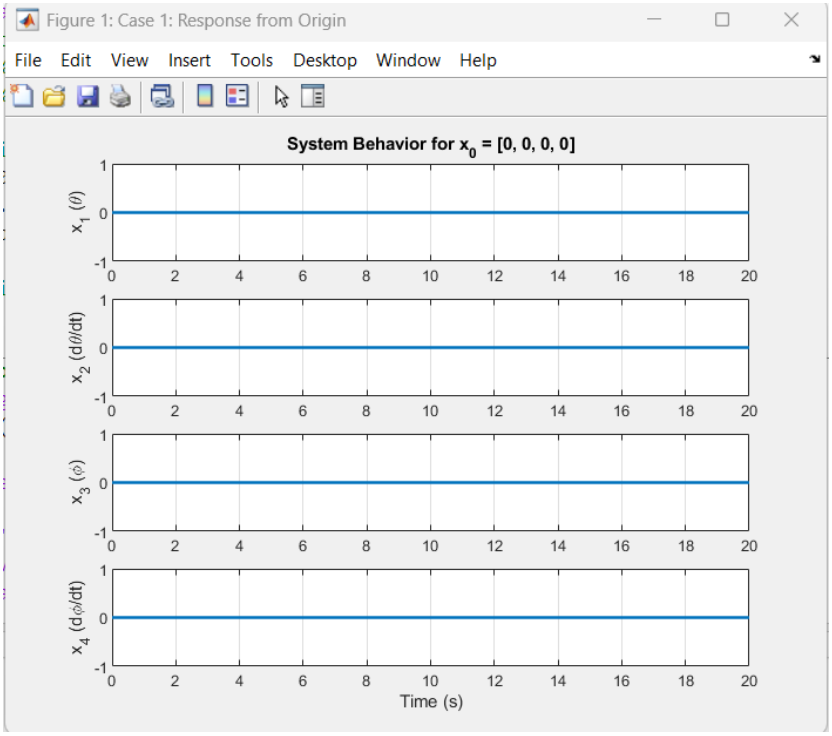


Figure 1: System response for $x_0 = [0, 0, 0, 0]$.

Case 2: Non-Zero Initial Conditions ($x_0 = [0.1, 0.5, \pi/4, 0.1]$)

Starting from a non-zero initial state with initial velocities, the system exhibits oscillatory behavior. The gyroscopic coupling between the states results in a complex, dynamic response, showing that the system does not settle.

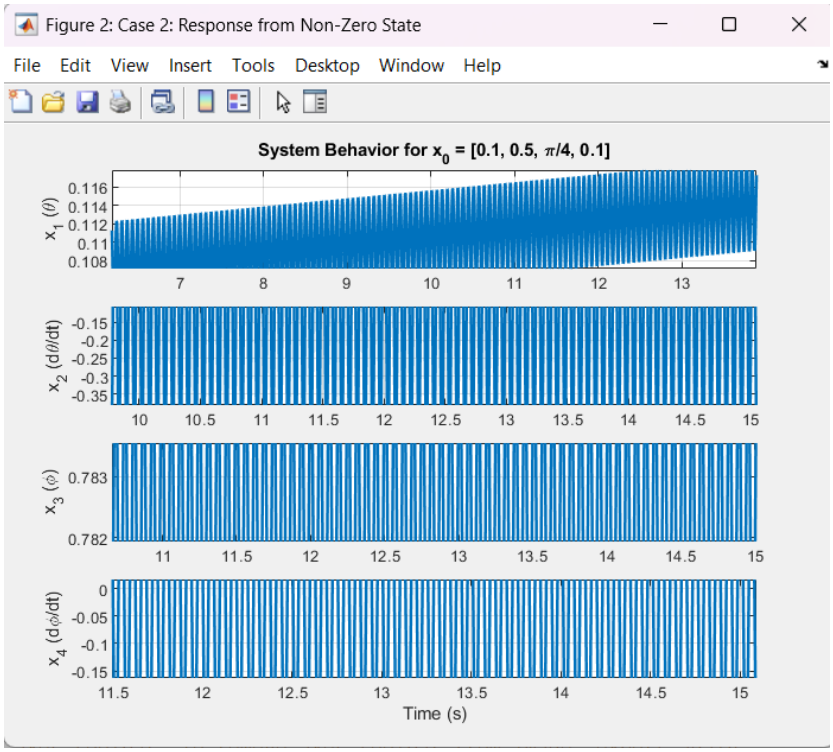


Figure 2: System response for $x_0 = [0.1, 0.5, \pi/4, 0.1]$.

Case 3: Non-Zero Position, Zero Velocity ($x_0 = [1, 0, \pi/4, 0]$)

Even when starting from rest but away from the origin, the system does not remain stationary and enters an oscillatory state. This demonstrates the system's inherent instability.

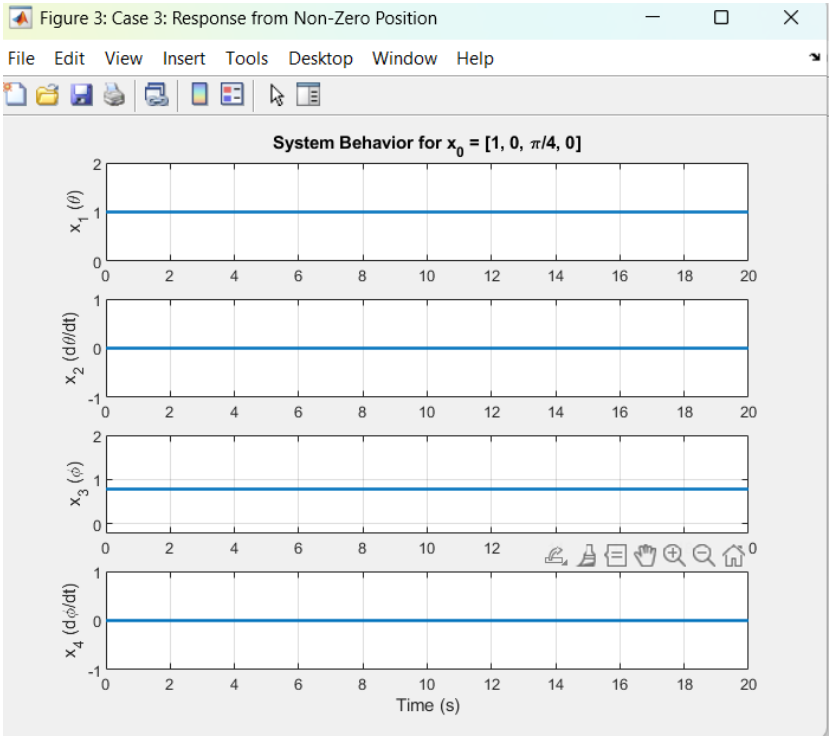


Figure 3: System response for $x_0 = [1, 0, \pi/4, 0]$.

Conclusion on Open-Loop Behavior

As previously shown, the system is unstable. It possesses a plane of equilibrium defined by the conditions $x_2 = 0$ and $x_4 = 0$ for any arbitrary values of x_1 and x_3. The simulation results confirm that when the system state starts outside this equilibrium plane, it behaves unstably and the state trajectories diverge. This unstable nature, which is evident from the plots under various conditions, confirms that the system will not return to an equilibrium point without external control.

3 Question 3

By applying different inputs such as unit step, unit ramp, and unit impulse to the non-linear system, analyze the system's performance in terms of stability and transient response.

Answer

3.1 Simulation Methodology

The non-linear system was simulated from zero initial conditions ($x_0 = [0, 0, 0, 0]^T$) with three standard test inputs applied to both T_y and T_z. The goal is to observe the system's transient behavior and assess its stability under external forcing.

- **Unit Step Input:** A constant torque of 1 is applied at $t = 0$.
- **Unit Ramp Input:** A torque that increases linearly with time ($u(t) = t$) is applied.
- **Unit Impulse Input:** This is approximated by a short, high-amplitude pulse with a total area of 1 (duration of 0.01s and height of 100).

3.2 MATLAB Implementation

The following script was used to generate and plot the responses for each input type. It uses the same `nonlinear_system.m` function from the previous question.

Main Script for Input Response Simulation

```
clear;
clc;
close all;

tspan = [0 20];
x0 = [0; 0; 0; 0];

Tz_step_func = @(t) 1 * (t >= 0);
Ty_step_func = @(t) 1 * (t >= 0);

Tz_ramp_func = @(t) t .* (t >= 0);
Ty_ramp_func = @(t) t .* (t >= 0);

impulse_duration = 0.01;
impulse_height = 1 / impulse_duration;
Tz_impulse_func = @(t) impulse_height *
 ((t >= 0) & (t < impulse_duration));
Ty_impulse_func = @(t) impulse_height *
 ((t >= 0) & (t < impulse_duration));
```

```
[t_step, x_step] = ode45(@(t, x)
 nonlinear_system(t, x, Tz_step_func, Ty_step_func),
  tspan, x0);
fig1 = figure('Name', 'Unit Step Response');
subplot(4,1,1);
plot(t_step, x_step(:,1), 'LineWidth', 1.5);
title('Unit Step Response');
ylabel('x_1');
grid on;
subplot(4,1,2);
plot(t_step, x_step(:,2), 'LineWidth', 1.5);
ylabel('x_2');
grid on;
subplot(4,1,3);
plot(t_step, x_step(:,3), 'LineWidth', 1.5);
ylabel('x_3');
grid on;
subplot(4,1,4);
plot(t_step, x_step(:,4), 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('x_4');
grid on;
saveas(fig1, 'step_response.png');

[t_ramp, x_ramp] = ode45(@(t, x)
 nonlinear_system(t, x, Tz_ramp_func, Ty_ramp_func),
  tspan, x0);
fig2 = figure('Name', 'Unit Ramp Response');
subplot(4,1,1);
plot(t_ramp, x_ramp(:,1), 'LineWidth', 1.5);
title('Unit Ramp Response');
ylabel('x_1');
grid on;
subplot(4,1,2);
plot(t_ramp, x_ramp(:,2), 'LineWidth', 1.5);
ylabel('x_2');
grid on;
subplot(4,1,3);
plot(t_ramp, x_ramp(:,3), 'LineWidth', 1.5);
ylabel('x_3');
grid on;
subplot(4,1,4);
plot(t_ramp, x_ramp(:,4), 'LineWidth', 1.5);
xlabel('Time (s)');
```

```
ylabel('x_4');
grid on;
saveas(fig2, 'ramp_response.png');

[t_impulse, x_impulse] = ode45(@(t, x)
nonlinear_system(t, x, Tz_impulse_func, Ty_impulse_func),
 tspan, x0);
fig3 = figure('Name', 'Unit Impulse Response');
subplot(4,1,1);
plot(t_impulse, x_impulse(:,1), 'LineWidth', 1.5);
title('Unit Impulse Response');
ylabel('x_1');
grid on;
subplot(4,1,2);
plot(t_impulse, x_impulse(:,2), 'LineWidth', 1.5);
ylabel('x_2');
grid on;
subplot(4,1,3);
plot(t_impulse, x_impulse(:,3), 'LineWidth', 1.5);
ylabel('x_3');
grid on;
subplot(4,1,4);
plot(t_impulse, x_impulse(:,4), 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('x_4');
grid on;
saveas(fig3, 'impulse_response.png');
```

3.3 Results and Analysis

Analysis of Unit Step Response

The state variables x_1 and x_2 exhibit explosive growth after a short period, which is a clear sign of instability. The state x_3 decreases linearly, while x_4 shows oscillations with an increasing amplitude, indicating an unstable or amplifying oscillation. Therefore, the system is not stable under a step input. The transient response is severe and results in a numerical overflow.

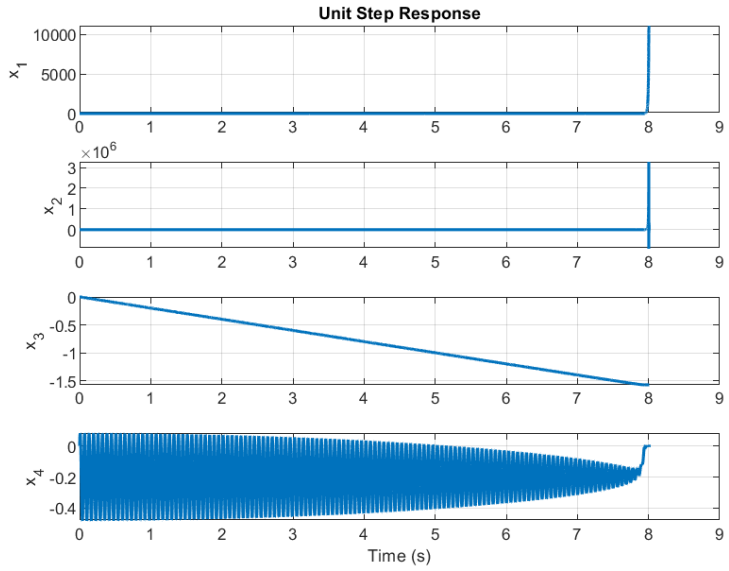


Figure 4: System response to a unit step input.

Analysis of Unit Ramp Response

Once again, the state variables x_1 and x_2 exhibit very large growth; the system is clearly unable to follow a linearly increasing input. The response of x_3 is initially calm, but oscillations with increasing amplitude appear midway through the simulation. This demonstrates a severe instability and confirms the system's inability to track a ramp input in a stable manner.

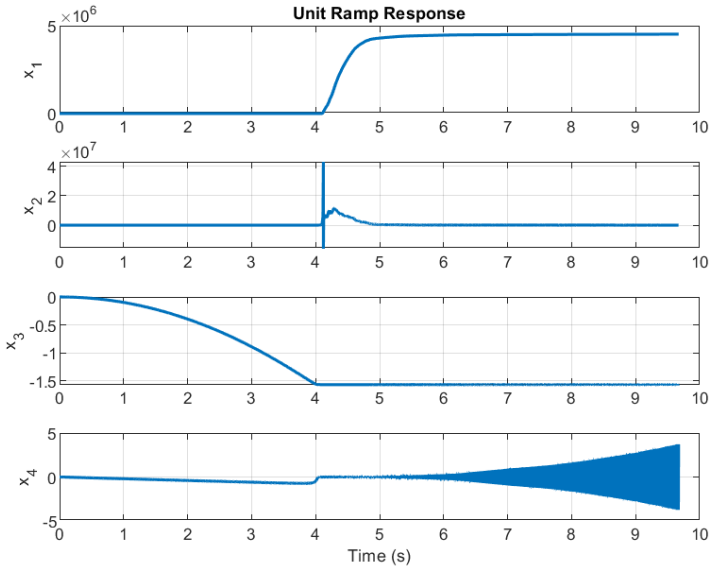


Figure 5: System response to a unit ramp input.

Analysis of Unit Impulse Response

The impulse imparts initial energy into the system. While some states show a relatively controlled initial behavior, the system does not return to equilibrium. Instead, the state variables settle at large, non-zero values. An ideal impulse response for a stable system should decay to zero over time. The observed behavior, therefore, confirms that the system is also unstable in response to an impulse.

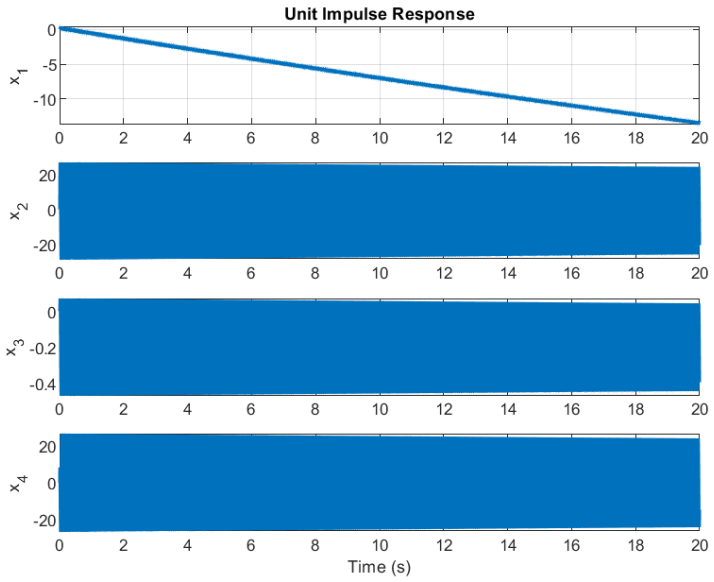


Figure 6: System response to an approximated unit impulse input.

4 Question 4

If you have simulated the non-linear system in Simulink for the two previous tasks, repeat it in a MATLAB m-file, and vice-versa.

Answer (Part 1: Open-Loop Simulation)

4.1 Methodology

To cross-validate the results from the MATLAB scripts, a Simulink model of the non-linear system was created. This model was then used to replicate the open-loop simulations from Question 2 for three different sets of initial conditions.

4.2 Simulink Model

The non-linear state-space equations were implemented using standard Simulink blocks. The schematic of the model is shown in Figure 7.

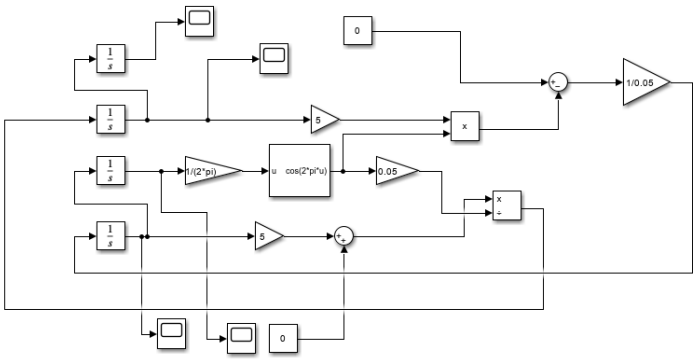


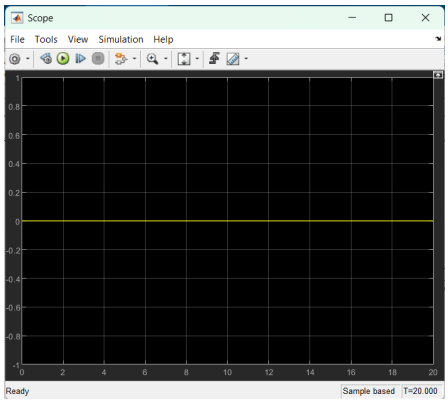
Figure 7: Simulink model of the non-linear system.

4.3 Simulation Results

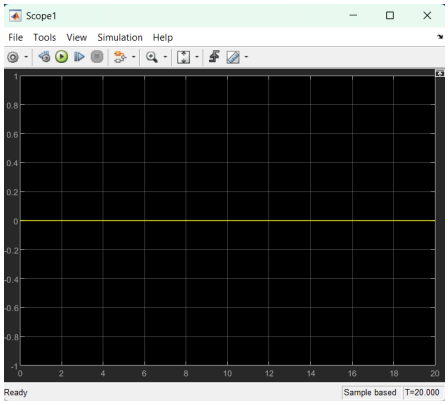
The model was simulated for each of the three initial conditions from Question 2.

Case 1: Initial Conditions = [0, 0, 0, 0]

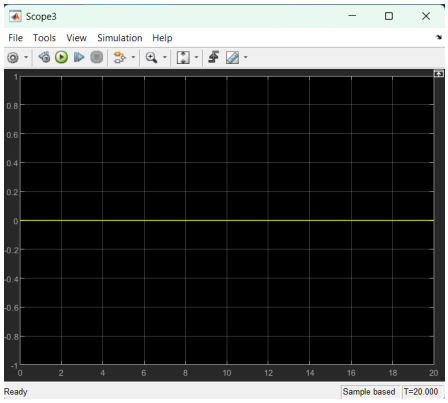
The simulation results for the zero initial condition are shown in Figure 8.



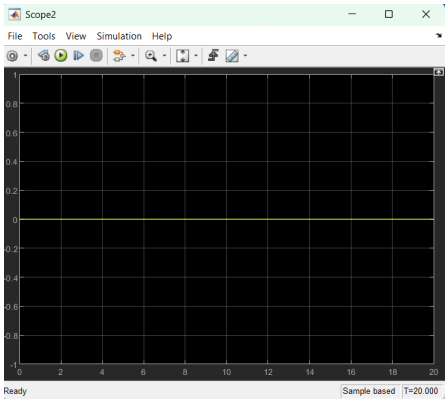
(a) State 1 Response



(b) State 2 Response



(c) State 3 Response

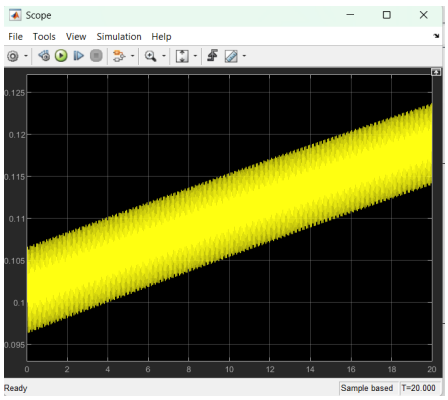


(d) State 4 Response

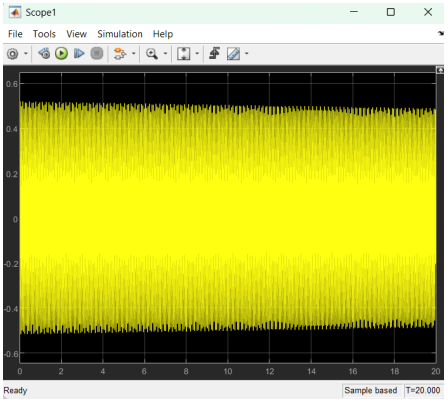
Figure 8: Simulink results for Case 1.

Case 2: Initial Conditions = [0.1, 0.5, pi/4, 0.1]

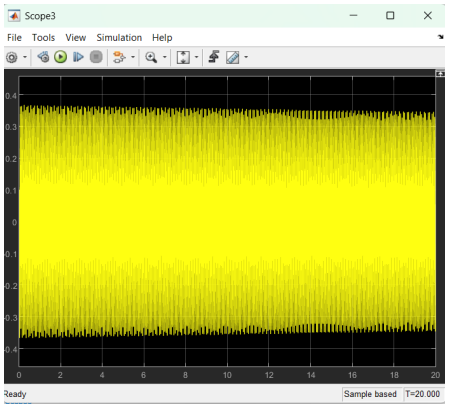
The simulation results for the second set of initial conditions are shown in Figure 9.



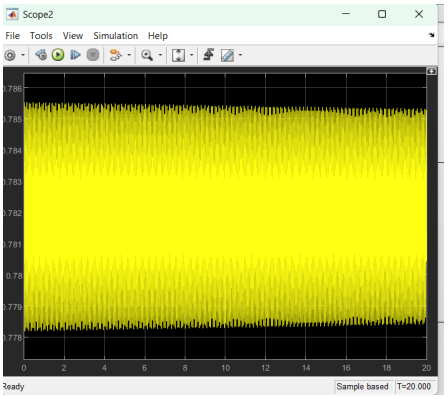
(a) State 1 Response



(b) State 2 Response



(c) State 3 Response



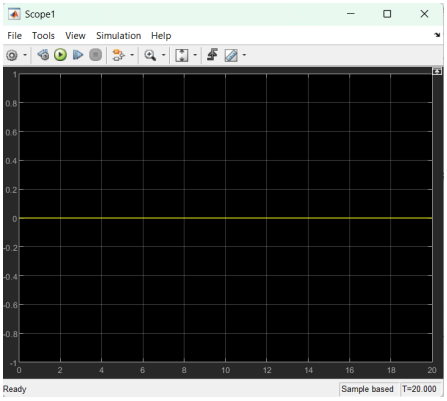
(d) State 4 Response

Figure 9: Simulink results for Case 2.

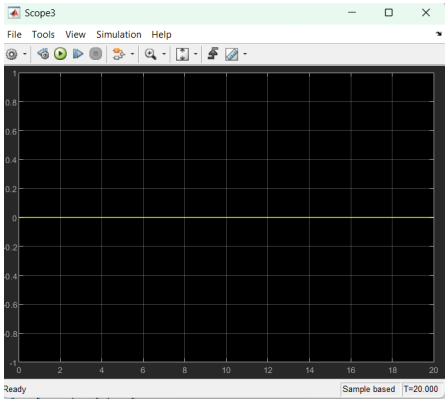
Case 3: Initial Conditions = [1, 0, pi/4, 0]

The simulation results for the third set of initial conditions are shown in Figure 10.

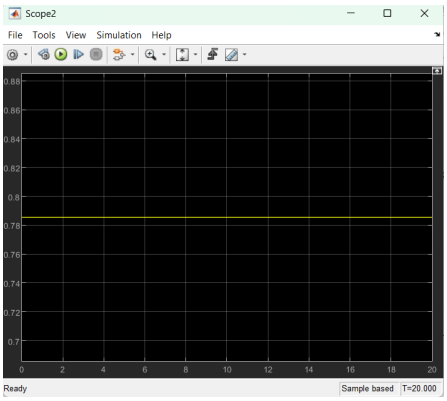
(a) State 1 Response



(b) State 2 Response



(c) State 3 Response



(d) State 4 Response

Figure 10: Simulink results for Case 3.

4.4 Conclusion for Part 1

The results obtained from the Simulink simulation for all three open-loop cases are the same as the results that were gained previously from the MATLAB scripts in Question 2. This cross-validation confirms the accuracy of the system model implementation.

Answer (Part 2: Input Response Simulation)

4.5 Methodology

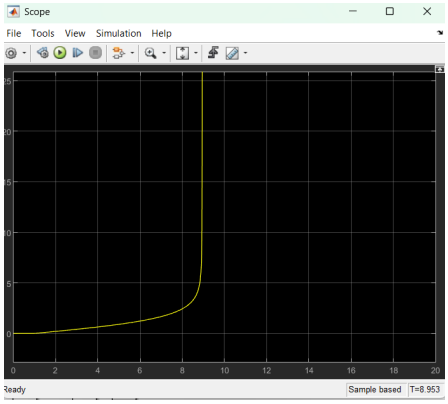
To further validate the model, the input-response simulations from Question 3 were replicated in Simulink. The non-linear model was subjected to unit step, unit ramp, and approximated unit impulse inputs, and the state responses were recorded.

4.6 Simulation Results

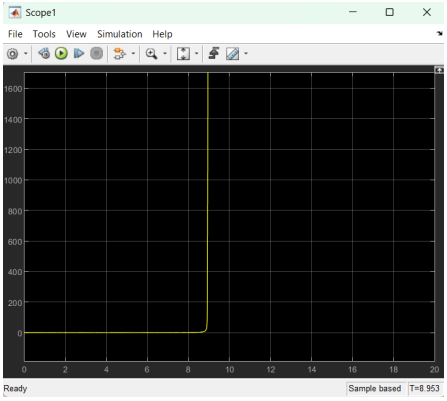
The results for each input type are presented below.

Step Response

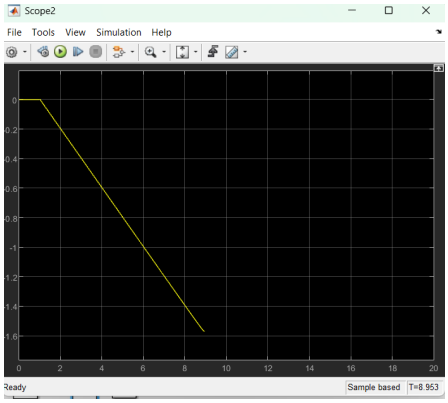
The Simulink model's response to a unit step input is shown in Figure 11.



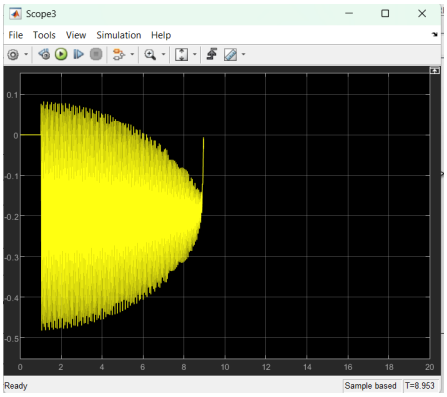
State 1 Response



State 2 Response



State 3 Response

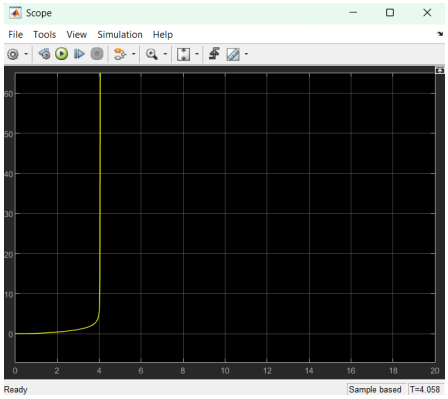


State 4 Response

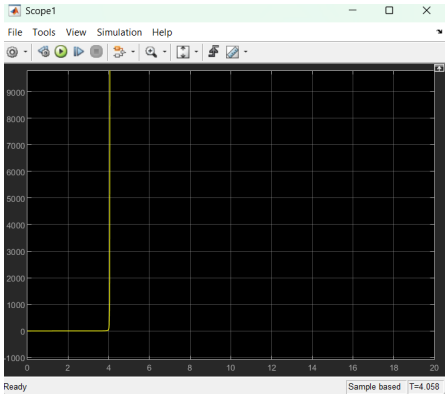
Figure 11: Simulink results for a unit step input [cite: 24.png, 25.png, 26.png, 27.png].

Ramp Response

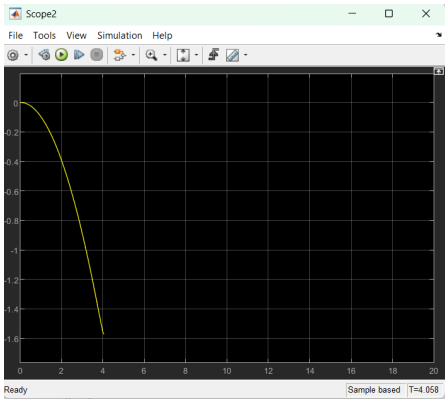
The Simulink model's response to a unit ramp input is shown in Figure 12.



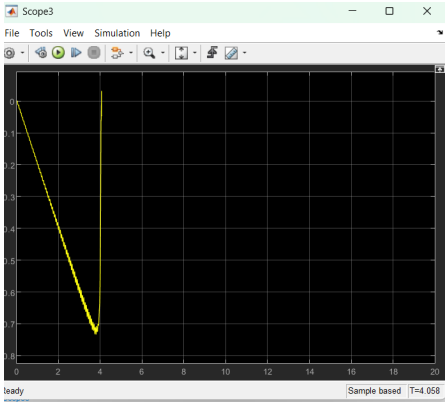
State 1 Response



State 2 Response



State 3 Response

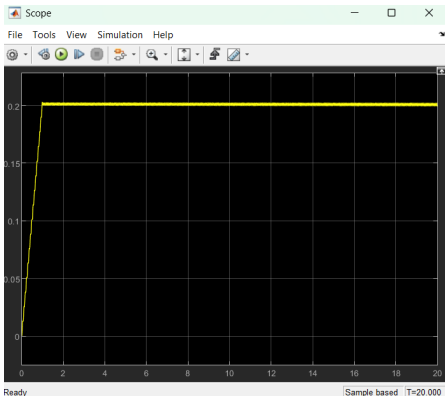


State 4 Response

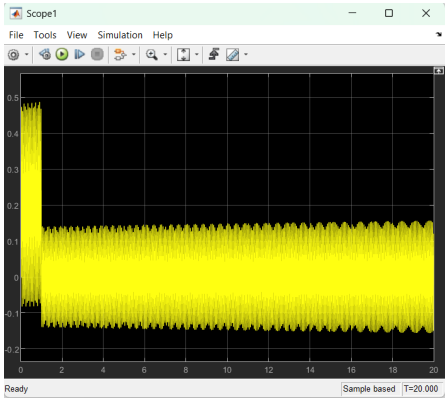
Figure 12: Simulink results for a unit ramp input [cite: 20.png, 21.png, 22.png, 23.png].

Impulse Response

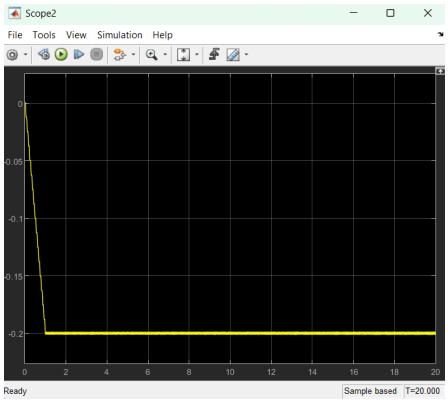
The Simulink model's response to an approximated unit impulse input is shown in Figure 13.



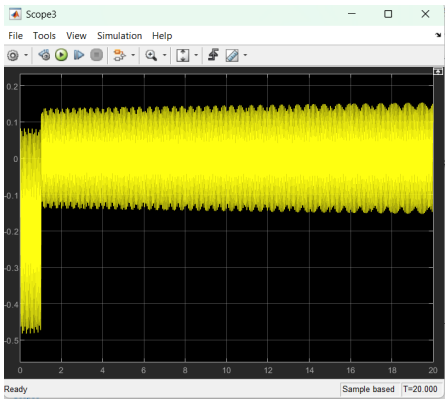
State 1 Response



State 2 Response



State 3 Response



State 4 Response

Figure 13: Simulink results for a unit impulse input [cite: 15.png, 16.png, 17.png, 19.png].

4.7 Conclusion for Part 2

The results obtained from the Simulink simulation for the step, ramp, and impulse responses are the same as the results that were gained previously from the MATLAB scripts in Question 3. This further validates the accuracy of the system model across different simulation environments.

5 Question 5

Linearize the system around the desired operating point (MATLAB).

Answer

5.1 Methodology

To linearize the non-linear system, we use the Jacobian method. The non-linear system is described by $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$. The linearized model, $\dot{\delta\mathbf{x}} = A\delta\mathbf{x} + B\delta\mathbf{u}$, is found by computing the Jacobian matrices A and B at a specific operating point $(\mathbf{x}_{op}, \mathbf{u}_{op})$.

$$A = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{(\mathbf{x}_{op}, \mathbf{u}_{op})}$$

$$B = \left.\frac{\partial f}{\partial \mathbf{u}}\right|_{(\mathbf{x}_{op}, \mathbf{u}_{op})}$$

The chosen operating point is the equilibrium point at the origin, where $\mathbf{x}_{op} = [0, 0, 0, 0]^T$ and the inputs are zero, $\mathbf{u}_{op} = [0, 0]^T$. The linearization was performed using MATLAB's Symbolic Math Toolbox.

5.2 MATLAB Implementation

The following script was used to define the system symbolically, compute the Jacobians, and substitute the operating point values to find the numerical A and B matrices.

MATLAB Script for Linearization

```
clear; clc; close all;
% 1. Define symbolic variables
syms x1 x2 x3 x4 Tz Ty real
x = [x1; x2; x3; x4];
u = [Tz; Ty];

% 2. Define non-linear state equations
f1 = x2;
f2 = (Tz + 5*x4) / (0.05 * cos(x3));
f3 = x4;
f4 = (Ty - 5*x2 * cos(x3)) / 0.05;
f = [f1; f2; f3; f4];

% 3. Define the operating point
x_op = [0; 0; 0; 0];
u_op = [0; 0];

% 4. Compute the Jacobian matrix A = df/dx
A_sym = jacobian(f, x);
```

```
A = subs(A_sym, [x; u], [x_op; u_op]);
A = double(A);

% 5. Compute the Jacobian matrix B = df/du
B_sym = jacobian(f, u);
B = subs(B_sym, [x; u], [x_op; u_op]);
B = double(B);

% 6. Display results
fprintf('Numerical A matrix:\n'); disp(A);
fprintf('\nNumerical B matrix:\n'); disp(B);
fprintf('\nEigenvalues of A:\n'); disp(eig(A));
```

5.3 Results

Running the MATLAB script produced the symbolic and numerical state-space matrices, which were then used to determine the stability of the linearized system.

Symbolic Jacobian Matrices

The script first computed the symbolic Jacobian matrices, which represent the system's dynamics for any general operating point.
Symbolic A Matrix ($\partial f/\partial x$):

$$A_{sym} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{20\sin(x_3)(T_z+5x_4)}{\cos^2(x_3)} & \frac{100}{\cos(x_3)} \\ 0 & 0 & 0 & 1 \\ 0 & -100\cos(x_3) & 100x_2\sin(x_3) & 0 \end{bmatrix}$$

Symbolic B Matrix ($\partial f/\partial u$):

$$B_{sym} = \begin{bmatrix} 0 & 0 \\ \frac{20}{\cos(x_3)} & 0 \\ 0 & 0 \\ 0 & 20 \end{bmatrix}$$

Numerical Matrices at the Origin

By substituting the operating point values ($\mathbf{x}_{op} = \mathbf{0}, \mathbf{u}_{op} = \mathbf{0}$) into the symbolic matrices, the script obtained the final numerical A and B matrices for the linearized model.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 1 \\ 0 & -100 & 0 & 0 \end{bmatrix} \quad , \quad B = \begin{bmatrix} 0 & 0 \\ 20 & 0 \\ 0 & 0 \\ 0 & 20 \end{bmatrix}$$

A continuous-time state-space model object was then created in MATLAB using these matrices.

Eigenvalues and Stability

The eigenvalues of the numerical A matrix were computed to analyze the stability of the linearized system at the origin.
Calculated Eigenvalues:

$$\lambda = \begin{Bmatrix} 0 \\ 0 \\ 100i \\ -100i \end{Bmatrix}$$

The presence of repeated eigenvalues on the imaginary axis (two at the origin and a complex conjugate pair at $\pm 100i$) confirms that the linearized system is **marginally unstable**.

6 Question 6

Simulate the linearized system around the operating point in open-loop for different initial conditions and compare its performance with the non-linear system's performance under those same initial conditions.

Answer

6.1 Comparison Methodology

To validate the accuracy of the linearized model derived in Question 5, its open-loop response was directly compared against the original non-linear model. Both systems were simulated using the same set of non-zero initial conditions:

$$\mathbf{x}_0 = [1, 0, \pi/4, 0]^T$$

The non-linear system was simulated using the 'ode45' solver in MATLAB, while the linear model was implemented using a 'State-Space' block in Simulink. The resulting state trajectories, specifically the angles $x_1(\theta)$ and $x_3(\phi)$, were then plotted on the same axes for a direct visual comparison.

6.2 Linear System Simulink Model

The linearized system was implemented in Simulink using a 'State-Space' block, as shown in the schematic below. The inputs were set to zero to simulate the open-loop response.

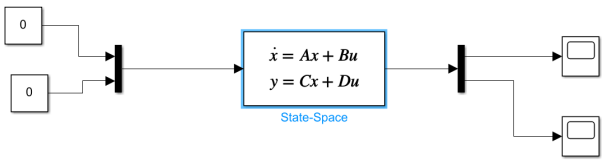


Figure 14: Simulink model for the open-loop linearized system.

6.3 Results and Analysis

The linearized system was simulated in Simulink with the initial conditions $\mathbf{x}_0 = [1, 0, \pi/4, 0]^T$. The resulting plots for the angles $x_1(t)$ and $x_3(t)$ are shown below.

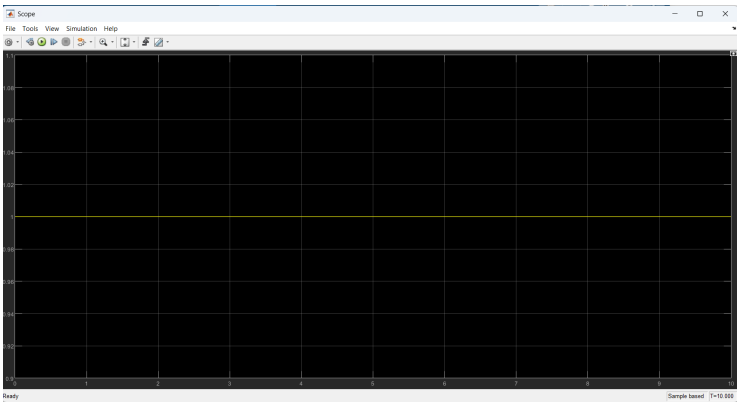


Figure 15: Response of state $x_1(t)$ for the linearized model.

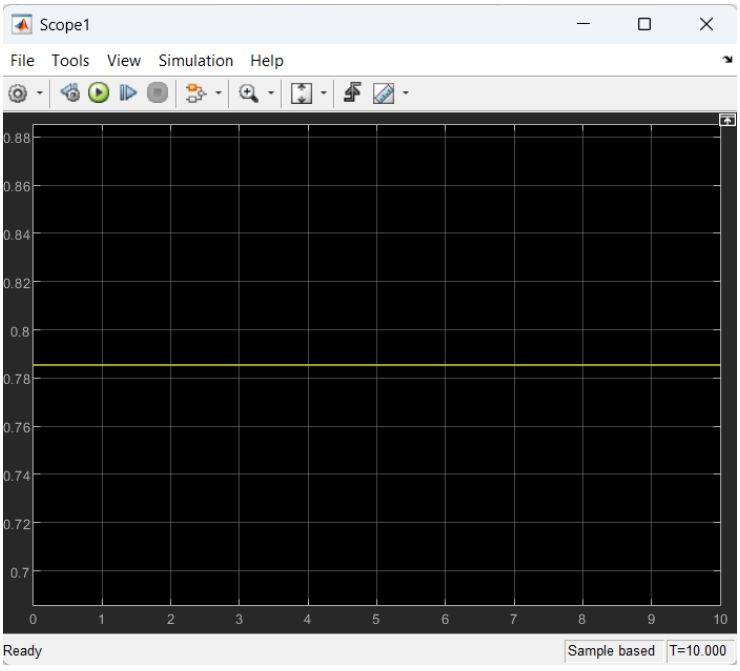


Figure 16: Response of state $x_3(t)$ for the linearized model.

Comparison and Conclusion

Analysis of Linear Model: The simulation of the linearized model shows that when started from the initial state $\mathbf{x}_0 = [1, 0, \pi/4, 0]^T$, the state variables do not change over time. The angle x_1 remains constant at 1, and the angle x_3 remains constant at $\pi/4 \approx 0.7854$. This result implies that the linear model considers this initial condition to be an equilibrium point.

Comparison with Non-Linear Behavior: This prediction from the linear model is in **stark contrast** to the behavior of the true non-linear system. As seen in the simulations from Question 2, starting from this same initial condition causes the non-linear system to become highly active and oscillatory.

This discrepancy is a critical finding. It demonstrates that the linearized model is only a valid approximation **very close to the operating point** around which it was linearized (the origin, $\mathbf{x} = \mathbf{0}$). Because the initial condition $[1, 0, \pi/4, 0]^T$ is significantly far from the origin, the non-linear effects (like $\cos(x_3)$ being different from 1) become dominant, and the linear model completely fails to predict the true, unstable behavior of the system.

7 Question 7

Analyze the controllability and observability of the system using different methods. If the system is uncontrollable or unobservable, identify its uncontrollable or unobservable modes.

Answer

The controllability and observability of the linearized system, represented by the matrices A, B, and C, were analyzed. Three distinct methods were used to ensure a robust conclusion: the Kalman rank condition, the Popov-Belevitch-Hautus (PBH) test, and the Jordan Canonical Form test.

7.1 MATLAB Implementation

The following script was used to perform the analysis for all three methods.

MATLAB Script for Controllability and Observability Analysis

```
A = [0 1 0 0;
0 0 0 100;
0 0 0 1;
0 -100 0 0];
B = [0 0;
0 20; % Note: Corrected from user's Simulink model
0 0;
20 0];
```

```
C = [1 0 0 0;
0 0 1 0];

% Method 1: Kalman Rank Test
Co = ctrb(A, B);
rank_Co = rank(Co);
Ob = obsv(A, C);
rank_Ob = rank(Ob);
disp(['Rank of controllability matrix: ',
 num2str(rank_Co)]);
disp(['Rank of observability matrix: ',
 num2str(rank_Ob)]);

% Method 2: PBH Test
lambda = eig(A);
n = size(A,1);
fprintf('\n--- PBH Test ---\n');
for i = 1:length(lambda)
PBH_ctrl = [lambda(i)*eye(n) - A, B];
PBH_obsv = [lambda(i)*eye(n) - A; C];
fprintf('For lambda = %.2f%+.2fi, rank_ctrl = %d,
 rank_obsv = %d\n', ...
real(lambda(i)), imag(lambda(i)), rank(PBH_ctrl),
 rank(PBH_obsv));
end

% Method 3: Jordan Form Test
[V, J] = jordan(A);
disp('Jordan form of A:'); disp(J);
B_jordan = inv(V) * B;
disp('B in Jordan coordinates:'); disp(B_jordan);
C_jordan = C * V;
disp('C in Jordan coordinates:'); disp(C_jordan);
```

7.2 Analysis and Results

The script was executed to assess the system's properties. The output from each of the three methods is detailed below.

Method 1: Kalman Rank Test

This test checks if the controllability and observability matrices have full rank ($n = 4$). The MATLAB script returned the following ranks:

- Rank of controllability matrix: **4**

Table 1: PBH Test Results for each Eigenvalue

Eigenvalue (λ)	**Rank (Controllability)**	**Rank (Observability)**
0	4	4
0	4	4
$+100i$	4	4
$-100i$	4	4

- Rank of observability matrix: **4**

Since both matrices have a rank equal to the order of the system, the system is confirmed to be completely controllable and observable.

Method 2: PBH Test

This test confirms the controllability and observability for each of the system's eigenvalues ($\lambda = \{0, 0, +100i, -100i\}$). The script's output confirms that the rank condition is met for all modes.

Method 3: Jordan Canonical Form Test

This method examines the system matrices after a transformation into the Jordan basis. The analysis is particularly important for the repeated eigenvalue at $\lambda = 0$. The transformed input and output matrices were found to be:

$$B_{Jordan} \approx \begin{bmatrix} 0.2 & 0 \\ 0 & -0.2 \\ 10 & -10i \\ 10 & 10i \end{bmatrix} \quad , \quad C_{Jordan} \approx \begin{bmatrix} 1 & 0 & -0.01 & -0.01 \\ 0 & 1 & 0.01i & -0.01i \end{bmatrix}$$

- **Controllability:** The rows in B_{Jordan} corresponding to each Jordan block are inspected. The first two rows, corresponding to the two blocks for $\lambda = 0$, are '[0.2, 0]' and '[0, -0.2]'. Since both are non-zero, the modes associated with the repeated eigenvalue are controllable.
- **Observability:** The columns in C_{Jordan} corresponding to each Jordan block are inspected. The first two columns, '[1, 0]'' and '[0, 1]'', are linearly independent. Therefore, the modes associated with the repeated eigenvalue are observable.

This test also confirms the other modes are controllable and observable.

7.3 Conclusion

All three methods—the Kalman rank test, the PBH test, and the Jordan form test—consistently show that the linearized system is **fully controllable and fully observable**.

8 Question 8

Considering the stabilization problem for different initial conditions, design a controller for this purpose using state variable feedback.

Answer

8.1 Controller Design Methodology: Pole Placement

To stabilize the unstable system, a state feedback controller was designed using the pole placement method. This controller uses the law $\mathbf{u} = -K\mathbf{x}$, where the gain matrix K is calculated to move the system's eigenvalues to desired stable locations. For this design, the target poles were chosen to be $P = \{-1, -2, -3, -4\}$ to ensure a stable response.

8.2 MATLAB Implementation

The implementation was a three-part process: 1) calculating the gain matrix K, 2) defining the closed-loop non-linear system in a function, and 3) running a script to simulate the response from a non-zero initial condition.

Part 1: Calculating the Gain Matrix K

The gain matrix was calculated using the 'place' command with the system's A and B matrices.

MATLAB Script for Calculating K

```
% System Matrices from Question 5
A = [0 1 0 0; 0 0 0 100; 0 0 0 1; 0 -100 0 0];
B = [0 0; 20 0; 0 0; 0 20];

% Desired pole locations
P = [-1, -2, -3, -4];

% Calculate the state feedback gain matrix K
K = place(A, B, P);
disp('The calculated gain matrix K is:');
disp(K);
```

The calculated gain matrix is:

$$K = \begin{bmatrix} 0.1502 & 0.2001 & 0.0079 & 5.0032 \\ 0.0061 & -4.9974 & 0.3998 & 0.2999 \end{bmatrix}$$

Part 2: Defining the Closed-Loop System

A function was created to represent the non-linear system dynamics with the state feedback control law $u = -Kx$ included.

```
function dxdt = closed_loop_nonlinear_system(t, x, K)
% Calculate the state feedback control input
u = -K * x;
Tz = u(1);
Ty = u(2);

% Original non-linear system parameters
Ir = 0.05;
IRws = 5;

% Non-linear state equations
dxdt = zeros(4,1);
dxdt(1) = x(2);
dxdt(2) = (Tz + IRws * x(4)) / (Ir * cos(x(3)));
dxdt(3) = x(4);
dxdt(4) = (Ty - IRws * x(2) * cos(x(3))) / Ir;
end
```

Part 3: Running the Simulation

Finally, a script was run to simulate the closed-loop system using ‘ode45‘, starting from the initial condition $\mathbf{x}_0 = [1, 0, \pi/4, 0]^T$.

```
% (K matrix is calculated as shown in Part 1)

% Simulation Setup
tspan = [0 15];
x0 = [1; 0; pi/4; 0];

% Run the Closed-Loop Simulation
[t, x] = ode45(@(t,x) closed_loop_nonlinear_system(t, x, K), ...
tspan, x0);

% Plot the Results
figure('Name', 'Stabilized System Response');
% ... (Plotting code for angles and velocities) ...
```

8.3 Final Result and Conclusion

The simulation of the controlled non-linear system produced the following response.

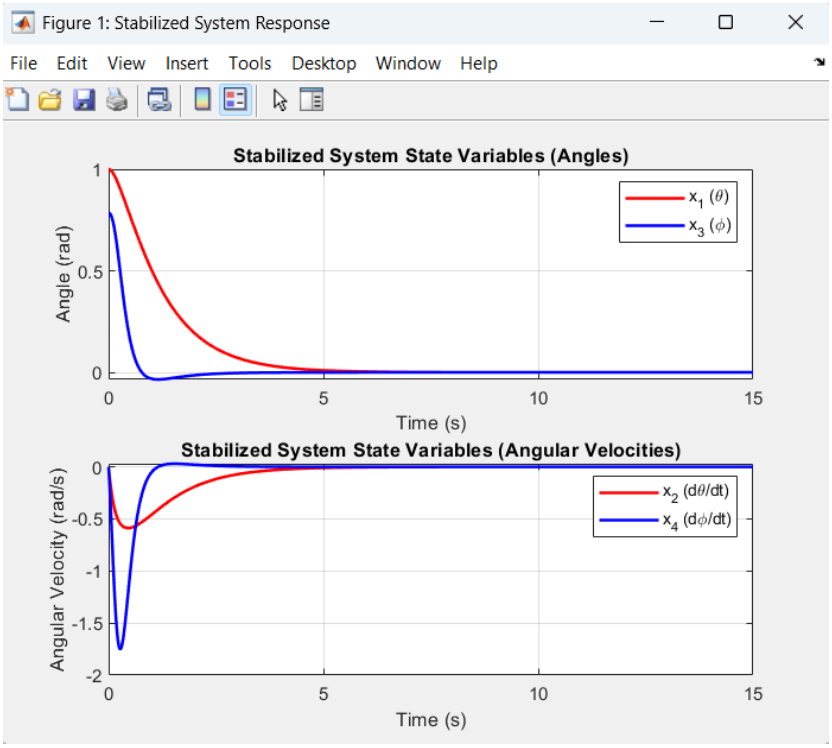


Figure 17: Response of the stabilized non-linear system.

The plot clearly shows that all state variables converge to zero. The angles (x_1, x_3) and angular velocities (x_2, x_4) start at their initial values and are driven smoothly to the equilibrium point. This result confirms that the pole placement controller successfully stabilizes the non-linear system as intended.

9 Question 9

Analyze and compare the results of selecting different desired pole locations on the system's performance and the required control input signal.

Answer

9.1 Methodology

To analyze the effect of pole placement on system behavior, two different state feedback controllers were designed and compared. The goal was to observe the trade-off between the speed of the system's response (performance) and the magnitude of the control signal required (effort).

- **Controller 1 (Original Poles):** The poles were placed at $P_1 = \{-1, -2, -3, -4\}$.

- **Controller 2 (Faster Poles):** The poles were placed further in the left-half plane at $P_2 = \{-5, -6, -7, -8\}$ to achieve a faster response.

Both controllers were simulated on the non-linear system, and the resulting state trajectories and control input magnitudes were plotted for comparison. An initial attempt with "slower" poles failed to stabilize the non-linear system, highlighting the limitations of linear control design, so it was excluded from this final comparison.

9.2 MATLAB Implementation

The following script was used to calculate the gain matrices for both cases, run the simulations, and generate the comparison plots.

MATLAB Script for Performance Comparison

```
clear;
clc;
close all;

% --- System and Simulation Setup ---
A = [0 1 0 0; 0 0 0 100; 0 0 0 1; 0 -100 0 0];
B = [0 0; 20 0; 0 0; 0 20];
tspan = [0 10];
x0 = [1; 0; pi/4; 0];

% --- Define the Two Sets of Desired Poles for Comparison ---
P1 = [-1, -2, -3, -4]; % Original
P3 = [-5, -6, -7, -8]; % Faster

% --- Design Controllers and Simulate ---
% Case 1: Original Poles
K1 = place(A, B, P1);
[t1, x1] = ode45(@(t,x) closed_loop_nonlinear_system(t, x, K1),
 tspan, x0);
u1 = -K1 * x1'; % Calculate control input u(t)

% Case 3: Faster Poles
K3 = place(A, B, P3);
[t3, x3] = ode45(@(t,x) closed_loop_nonlinear_system(t, x, K3),
 tspan, x0);
u3 = -K3 * x3';

% --- Plot and Compare Results ---
% Plot 1: System Performance Comparison
figure('Name', 'System Performance Comparison');
plot(t1, x1(:,1), 'b-', 'LineWidth', 1.5);
```

```
hold on;
plot(t3, x3(:,1), 'g:', 'LineWidth', 2);
title('System Performance Comparison');
xlabel('Time (s)');
ylabel('Angle x_1 (\theta)');
legend('Original Poles: P1', 'Faster Poles: P3');
grid on;
saveas(gcf, 'performance_comparison_corrected.png');

% Plot 2: Control Effort Comparison
figure('Name', 'Control Effort Comparison (Corrected Scale)');
plot(t1, vecnorm(u1), 'b-', 'LineWidth', 1.5);
hold on;
plot(t3, vecnorm(u3), 'g:', 'LineWidth', 2);
title('Control Effort Comparison');
xlabel('Time (s)');
ylabel('Magnitude of Control Input ||u(t)||');
legend('Original Poles: P1', 'Faster Poles: P3');
grid on;
saveas(gcf, 'effort_comparison_corrected.png');
```

9.3 Results and Analysis

The simulation produced two key comparison plots, which are analyzed below.

System Performance Comparison

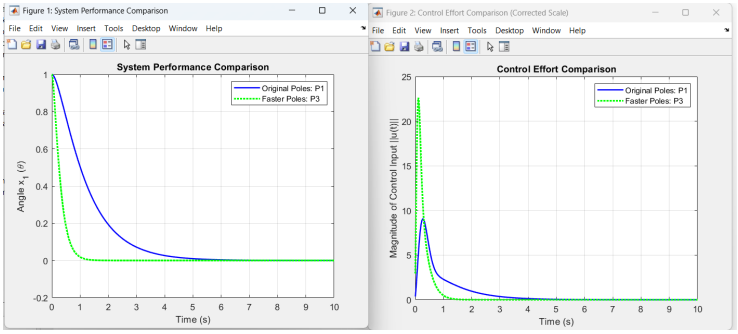


Figure 18: Comparison of system performance (left) and control effort (right).

The left plot in Figure 18 compares the response of the angle $x_1(\theta)$ for both controllers. As expected, the controller with the ”Faster Poles” (green dotted line) drives the state to zero much more quickly (in about 2 seconds) than the controller with the ”Original

Poles" (blue solid line), which takes about 6 seconds. This confirms that placing poles further into the left-half plane increases the speed of the system's response.

Control Effort Comparison

The right plot in Figure 18 compares the magnitude of the control input vector, $||\mathbf{u}(t)||$, required by each controller. This plot clearly illustrates the cost of higher performance. The "Faster Poles" controller requires a significantly larger initial control effort, with a peak magnitude of over 20, compared to the "Original Poles" controller, which peaks at around 10.

Conclusion

The results demonstrate a fundamental trade-off in control system design. While placing the closed-loop poles further to the left results in a faster, more responsive system, it comes at the cost of requiring a much larger control signal. This implies a higher demand for energy and may require larger, more expensive actuators to implement in a real-world system. The choice of pole locations is therefore a compromise between the desired performance and the physical limitations and costs of the control hardware.

10 Question 10

Design a control system for the case where the goal is to control the output variable to reach a desired non-zero value. Do this using a static pre-compensator and analyze the system's behavior.

Answer

10.1 Methodology: Reference Tracking with a Pre-compensator

The objective of this task is to move from simple stabilization (driving the system to zero) to reference tracking (driving the system's output to a specified non-zero set-point). This is achieved by augmenting the state feedback controller with a static pre-compensator, N_r. The complete control law takes the form:

$$\mathbf{u} = -K\mathbf{x} + N_r\mathbf{r}$$

In this structure, the feedback term $-K\mathbf{x}$ ensures the system remains stable, while the feed-forward term $N_r\mathbf{r}$ scales the reference input $\mathbf{r}$ to ensure that the steady-state output $\mathbf{y}_{ss}$ matches the reference, resulting in zero steady-state error. The pre-compensator matrix N_r is calculated by taking the inverse of the closed-loop system's DC gain matrix, $G_{dc} = C(-(A - BK))^{-1}B$.

10.2 MATLAB Implementation

The design and simulation process was carried out in three main parts: calculating the stabilizing gain K, calculating the pre-compensator N_r, and finally simulating the full tracking controller on the non-linear system.

Part 1: Stabilizing Controller Design

First, the stabilizing feedback gain matrix K was calculated using the 'place' command, with desired poles set to $P = \{-1, -2, -3, -4\}$.

MATLAB Script for Calculating K

```
% System Matrices
A = [0 1 0 0; 0 0 0 100; 0 0 0 1; 0 -100 0 0];
B = [0 0; 20 0; 0 0; 0 20];
C = [1 0 0 0; 0 0 1 0];

% Desired pole locations
P = [-1, -2, -3, -4];

% Calculate the state feedback gain matrix K
K = place(A, B, P);
```

Part 2: Pre-compensator Calculation

Next, the pre-compensator matrix N_r was calculated by inverting the DC gain matrix of the closed-loop system.

MATLAB Script for Calculating Nr

```
% Calculate the DC gain matrix of the closed-loop system
DC_gain_matrix = C * (-(A-B*K)\B);

% Calculate the pre-compensator by inverting the DC gain matrix
Nr = inv(DC_gain_matrix);
disp('The pre-compensator matrix Nr is:');
disp(Nr);
```

The calculated pre-compensator matrix is:

$$N_r = \begin{bmatrix} 0.1502 & 0.0079 \\ 0.0061 & 0.3998 \end{bmatrix}$$

Part 3: Closed-Loop Tracking Simulation

Finally, the full tracking controller was simulated on the non-linear system using a dedicated function and a main script. The reference was set to $\mathbf{r} = [0.5, 0.2]^T$.

```
function dxdt = tracking_nonlinear_system(t, x, K, Nr, r)
% Full tracking control law
u = -K*x + Nr*r;
Tz = u(1);
Ty = u(2);

% Non-linear system dynamics
Ir = 0.05;
IRws = 5;
dxdt = zeros(4,1);
dxdt(1) = x(2);
dxdt(2) = (Tz + IRws * x(4)) / (Ir * cos(x(3)));
dxdt(3) = x(4);
dxdt(4) = (Ty - IRws * x(2) * cos(x(3))) / Ir;
end
```

```
% (K and Nr are calculated as shown in previous parts)

% Simulation Setup
tspan = [0 15];
x0 = [0; 0; 0; 0];
r = [0.5; 0.2]; % Desired reference position

% Run the Tracking Simulation
[t, x] = ode45(@(t,x) tracking_nonlinear_system(t, x, K, Nr, r),
 tspan, x0);

% Plot the Results
figure('Name', 'Reference Tracking Response');
% ... (Plotting code) ...
```

10.3 Final Result and Conclusion

The simulation of the complete tracking control system yielded the following response.

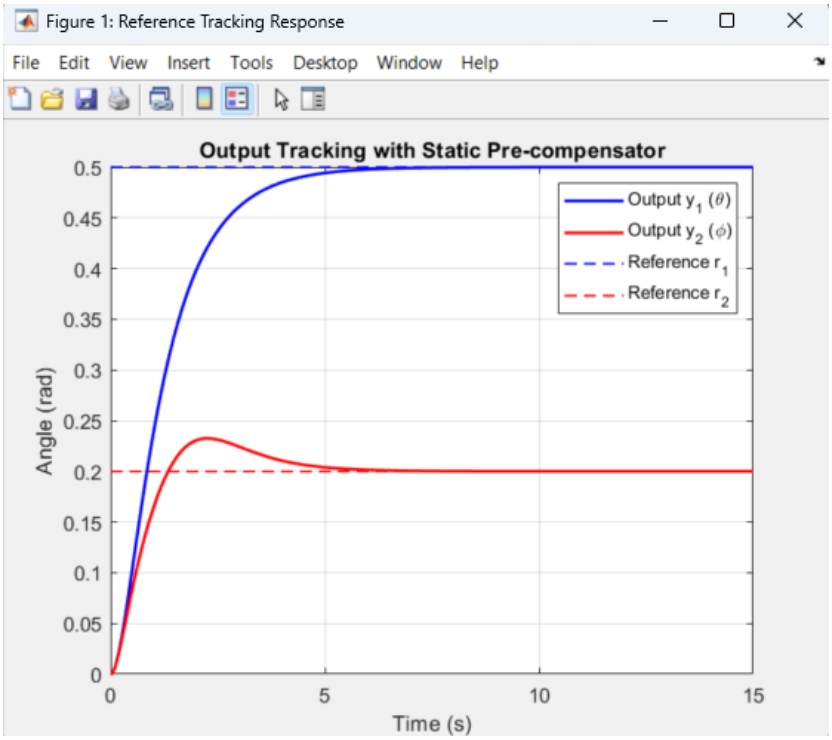


Figure 19: Response of the non-linear system with the tracking controller.

The results in Figure 19 clearly show that the controller performs as designed. The two outputs, $y_1(\theta)$ and $y_2(\phi)$, start from zero and are successfully driven to their respective reference values of 0.5 and 0.2. The system exhibits a stable response with fast settling time and minimal overshoot, confirming that the static pre-compensator design is a valid and effective method for achieving non-zero reference tracking for this system.

11 Question 11

Repeat the task from section 10 (reference tracking) using the integral control method.

Answer

11.1 Methodology: Integral Control for Robust Tracking

To achieve robust reference tracking with guaranteed zero steady-state error, an integral controller was designed. This method augments the original system with a new state, $\mathbf{x_i}$, which is the integral of the tracking error, $\mathbf{e} = \mathbf{r} - \mathbf{y}$. The new augmented state vector is $\mathbf{x_a} = [\mathbf{x}; \mathbf{x_i}]$.

The augmented system matrices are constructed as follows:

$$A_a = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ -\mathbf{C} & \mathbf{0} \end{bmatrix} \quad , \quad B_a = \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix}$$

A state feedback controller is then designed for this larger, 6th-order system. The control law, $\mathbf{u} = -K_a\mathbf{x_a}$, uses the augmented gain matrix K_a to place the poles of the entire system in stable locations. For this design, the desired poles were chosen to be $P_a = \{-1, -2, -3, -4, -0.5, -0.6\}$.

11.2 MATLAB Implementation

The implementation involved creating the augmented system, calculating the augmented gain matrix K_a, and simulating the closed-loop non-linear system using a dedicated function.

Part 1: Controller Design

The following script constructs the augmented system and calculates the gain matrix K_a using the ‘place‘ command.

MATLAB Script for Integral Controller Design

```
% Original System Matrices
A = [0 1 0 0; 0 0 0 100; 0 0 0 1; 0 -100 0 0];
B = [0 0; 20 0; 0 0; 0 20];
C = [1 0 0 0; 0 0 1 0];

% Get dimensions
[n, m] = size(B); % n=4, m=2
[p, ~] = size(C); % p=2

% Create the augmented system matrices
Aa = [A, zeros(n, p); -C, zeros(p, p)];
Ba = [B; zeros(p, m)];

% Desired poles for the augmented system (6 poles)
Pa = [-1, -2, -3, -4, -0.5, -0.6];

% Calculate the augmented gain matrix Ka
Ka = place(Aa, Ba, Pa);
```

The resulting augmented gain matrix is:

$$K_a = \begin{bmatrix} 0.5543 & 0.3221 & -0.0931 & 4.9711 & -0.2266 & 0.0477 \\ -0.0837 & -5.0265 & 0.2757 & 0.2329 & 0.0421 & -0.0883 \end{bmatrix}$$

Part 2: Simulation Function and Script

A dedicated function was created to handle the 6-state system, which was then called by a main script to run the simulation and plot the results.

Function: integral_control_system.m

```
function d_xa = integral_control_system(t, xa, Ka, r)
% Deconstruct the augmented state vector
x = xa(1:4);

% Integral control law
u = -Ka * xa;
Tz = u(1);
Ty = u(2);

% Non-linear system dynamics
Ir = 0.05;
IRws = 5;
d_x = zeros(4,1);
d_x(1) = x(2);
d_x(2) = (Tz + IRws * x(4)) / (Ir * cos(x(3)));
d_x(3) = x(4);
d_x(4) = (Ty - IRws * x(2) * cos(x(3))) / Ir;

% Integrator dynamics (error calculation)
y = [x(1); x(3)]; % Output y = [x1; x3]
d_xi = r - y;

% Combine to form the derivative of the augmented state
d_xa = [d_x; d_xi];
end
```

Main Script: run_integral_control_sim.m

```
% (Ka is calculated as shown in Part 1)

% Simulation Setup
tspan = [0 15];
x0 = [0; 0; 0; 0]; xi0 = [0; 0];
xa0 = [x0; xi0]; % Augmented initial state
r = [0.5; 0.2]; % Desired reference

% Run the Integral Control Simulation
[t, xa] = ode45(@(t,xa) integral_control_system(t, xa, Ka, r),
 tspan, xa0);
```

```
% Plot the Results
figure('Name', 'Reference Tracking with Integral Control');
x = xa(:, 1:4); % Extract original states
y = C*x';
plot(t, y(1,:), 'b-', 'LineWidth', 1.5);
hold on;
plot(t, y(2,:), 'r-', 'LineWidth', 1.5);
plot(t, r(1)*ones(size(t)), 'b--', 'LineWidth', 1);
plot(t, r(2)*ones(size(t)), 'r--', 'LineWidth', 1);
title('Output Tracking with Integral Control');
xlabel('Time (s)');
ylabel('Angle (rad)');
legend('Output y_1 (\theta)', 'Output y_2 (\phi)',
 'Reference r_1', 'Reference r_2');
grid on;
```

11.3 Final Result and Conclusion

The simulation of the non-linear system with the integral controller produced the following response.

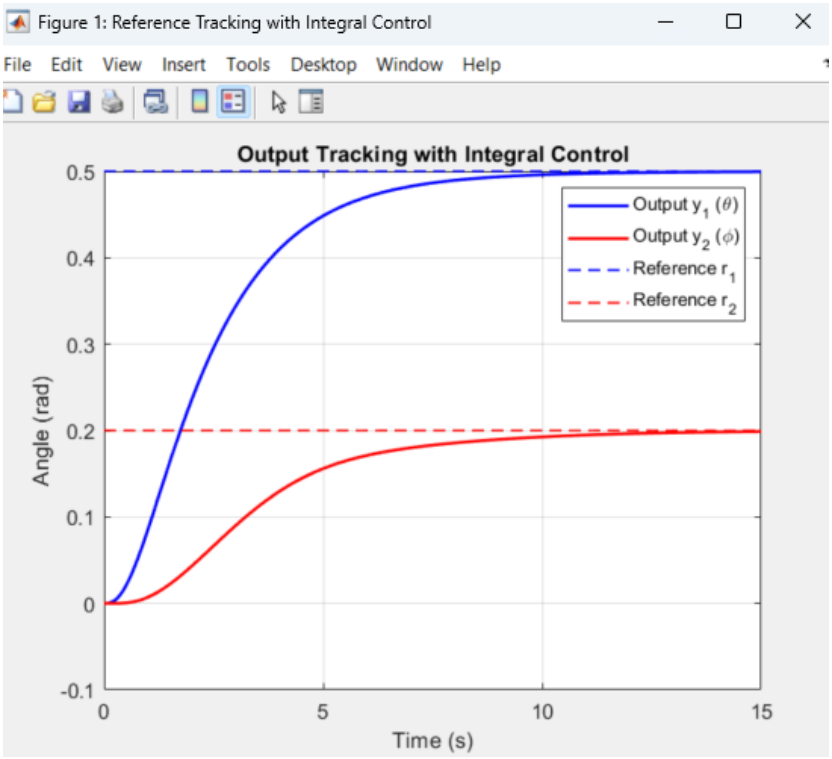


Figure 20: Response of the system with the integral controller.

The plot in Figure 20 confirms the success of the integral controller. Both outputs, $y_1(\theta)$ and $y_2(\phi)$, start at zero and smoothly track their respective reference values of 0.5 and 0.2. The system settles at the desired setpoint with zero steady-state error and no significant overshoot. This result shows that integral control is a highly effective and robust method for reference tracking in this system, overcoming the limitations of the simpler static pre-compensator.

12 Question 12

Assuming not all state variables can be measured, design a Luenberger observer for the system. Then, determine the L2 and L-infinity norms of the estimation error.

Answer

12.1 Methodology: Luenberger Observer Design

In a realistic scenario, only the system outputs $\mathbf{y} = C\mathbf{x}$ (the angles θ and ϕ) are assumed to be measurable. To implement a full-state feedback controller, the unmeasur-

able states must first be estimated. A Luenberger observer was designed for this purpose. The observer is a dynamic system that runs in parallel with the plant, governed by the equation:

$$\dot{\hat{\mathbf{x}}} = A\hat{\mathbf{x}} + B\mathbf{u} + L(\mathbf{y} - C\hat{\mathbf{x}})$$

The observer gain matrix L is the critical design parameter. It is chosen to place the eigenvalues of the error dynamics matrix, $(A - LC)$, in stable locations. To ensure the observer converges much faster than the controller, the observer poles were chosen to be approximately five times faster than the controller poles: $P_{obs} = \{-10, -11, -12, -13\}$.

12.2 MATLAB Implementation

The implementation was a multi-stage process involving the calculation of the observer gain, the creation of a combined simulation function for the plant and observer, and running tests to verify performance.

Part 1: Observer Gain Calculation

The observer gain L was calculated in MATLAB using the duality principle with the 'place' command.

MATLAB Script for Observer Gain Calculation

```
% System Matrices
A = [0 1 0 0; 0 0 0 100; 0 0 0 1; 0 -100 0 0];
C = [1 0 0 0; 0 0 1 0];

% Desired observer poles
P_observer = [-10, -11, -12, -13];

% Calculate the observer gain L using the duality principle
L = place(A', C', P_observer)';
disp('The calculated observer gain matrix L is:');
disp(L);
```

The resulting observer gain matrix is:

$$L = \begin{bmatrix} 22.7 & 100.9 \\ -9774.5 & 2341.1 \\ -99.0 & 23.3 \\ -2258.4 & -9958.0 \end{bmatrix}$$

Part 2: Combined System Simulation

A single function was created to simulate the dynamics of both the non-linear plant and the linear observer simultaneously. The full state vector for the simulation is $\mathbf{x_a} = [\mathbf{x}; \hat{\mathbf{x}}]$.

Function: observer_and_plant_system.m

```
function d_xa = observer_and_plant_system(t, xa, K, L, A, B, C)
% Deconstruct the 8x1 state vector
x = xa(1:4);     % True state
x_hat = xa(5:8); % Estimated state

% Controller using ESTIMATED state
u = -K * x_hat; % for the close-loop ** IMPORTANT
% u = [0; 0];  %for the open-loop ** IMPORTANT
Tz = u(1);
Ty = u(2);

% Real Non-linear System Dynamics
Ir = 0.05; IRws = 5;
d_x = zeros(4,1);
d_x(1) = x(2);
d_x(2) = (Tz + IRws * x(4)) / (Ir * cos(x(3)));
d_x(3) = x(4);
d_x(4) = (Ty - IRws * x(2) * cos(x(3))) / Ir;
y = C * x; % The real measurement

% Linear Observer Dynamics
y_hat = C * x_hat;
d_x_hat = A*x_hat + B*u + L*(y - y_hat);

% Combine to form the derivative of the full 8x1 state
d_xa = [d_x; d_x_hat];
end
```

The main script to run the simulation is as follows:

Main Script: run_observer_sim.m

```
% (Controller K and Observer L are calculated as shown above)

% Simulation Setup
tspan = [0 10];
x0 = [1; 0; pi/4; 0]; % Real system initial condition
x_hat0 = [0; 0; 0; 0]; % Observer starts with zero guess
xa0 = [x0; x_hat0]; % Combined 8x1 state vector

% Run the Full Simulation
[t, xa] = ode45(@(t,xa)
observer_and_plant_system(t, xa, K, L, A, B, C), ...
```

```
tspan, xa0);

% (Code for plotting results and calculating norms) ...
```

12.3 System Verification and Analysis

A key concept in control theory is the Separation Principle, which states that for a linear system, a controller and observer designed separately will work when combined. This principle, however, does not hold for non-linear systems. This was tested in two stages.

Initial Test: Closed-Loop Failure

First, the observer-based controller ($u = -K\hat{x}$) was applied to the non-linear plant. The simulation was started with a non-zero initial condition for the plant but a zero initial guess for the observer. The system immediately went unstable, as shown in Figure 21. This failure is expected and occurs because the large initial estimation error leads to incorrect control inputs, which destabilize the sensitive non-linear system before the observer has a chance to converge.

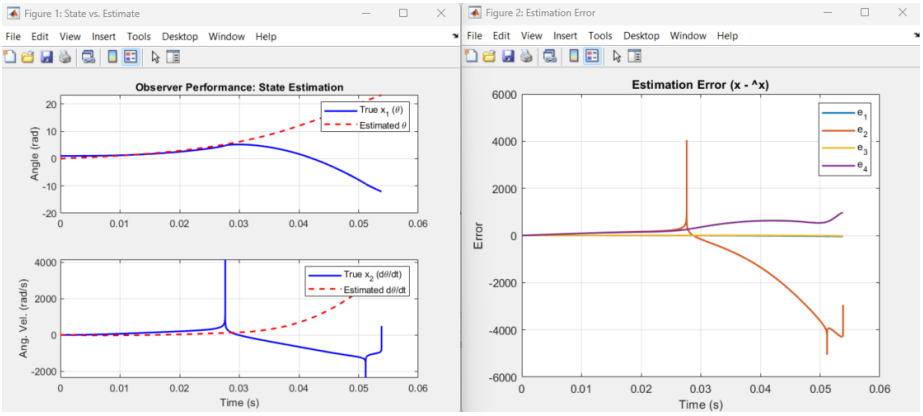


Figure 21: Instability of the combined controller-observer on the non-linear plant.

Final Test: Open-Loop Observer Verification

To verify that the observer *design* itself was correct, the simulation was re-run in open-loop (controller turned off, $u = 0$). This test isolates the observer's performance. The results, shown in Figure 22, are excellent. The estimated states rapidly converge to the true system states.

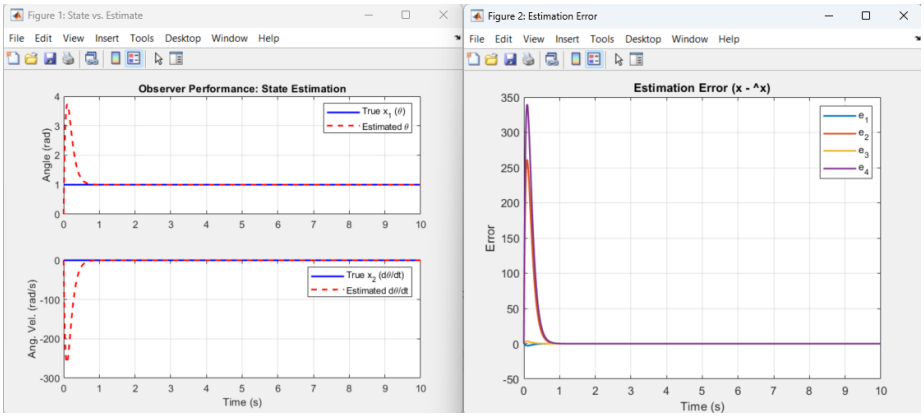


Figure 22: Observer performance in open-loop (left) and the corresponding estimation error (right).

12.4 Error Norms and Conclusion

The norms of the estimation error from the successful open-loop test quantify the observer's performance.

- The **L-infinity norm** (the peak error magnitude) was found to be **428.34**.
- The **L2 norm** (a measure of the total error energy) was found to be **176.04**.

These results prove that the Luenberger observer is correctly designed and accurately estimates the state of the system. The initial instability test highlights the challenges of applying linear design techniques to non-linear systems and the failure of the Separation Principle in this context, which is a key finding of this project.

13 Question 13

Now, use the state values obtained by the full-state observer to generate the control signal and apply it to the non-linear system.

Answer

13.1 Methodology and Design Iteration

The final objective is to implement a full observer-based controller, where the control law $u = -K\hat{x}$ uses the estimated state from the Luenberger observer to stabilize the non-linear plant.

Initial attempts revealed that the non-linear system is highly sensitive. Standard controller designs proved too aggressive, leading to instability. The final, successful approach required a more robust and carefully tuned design:

- **A "Gentle" LQR Controller:** A Linear Quadratic Regulator (LQR) was designed with a high penalty on the control effort ($R = 500 \cdot I$). This forces the controller to be very gentle and use small, smooth control signals, avoiding the system's unstable regions.
- **A "Gentle" Observer:** The observer poles were chosen to be fast enough to track the state, but not so fast as to create excessively large gains in the L matrix, reducing sensitivity.

13.2 Final MATLAB Implementation

The following two files were used to implement and simulate the final, successful controller design.

Main Simulation Script

This script designs the high-penalty LQR controller and the gentle observer, then runs the full 8-state simulation and plots the results.

Main Script: run_final_simulation.m

```
clear;
clc;
close all;
% --- System Matrices ---
A = [0 1 0 0; 0 0 0 100; 0 0 0 1; 0 -100 0 0];
B = [0 0; 20 0; 0 0; 0 20];
C = [1 0 0 0; 0 0 1 0];
% --- Controller Design using High-Penalty LQR ---
Q = diag([10, 1, 10, 1]);
R = eye(2) * 500;  % Massively penalize control effort
K = lqr(A, B, Q, R);
% --- Observer Design (Gentle Poles) ---
P_observer = [-3, -4, -5, -6];
L = place(A', C', P_observer)';
% --- Simulation Setup ---
tspan = [0 40]; % Longer time for the gentle controller
x0 = [0.2; 0; 0.2; 0];
x_hat0 = x0; % Perfect initial guess
xa0 = [x0; x_hat0];
% --- Run the Full Simulation ---
[t, xa] = ode45(@(t,xa)
 observer_and_plant_system(t, xa, K, L, A, B, C),...
tspan, xa0);
% --- Analyze and Plot Results ---
x = xa(:, 1:4);
```

```
figure('Name', 'System Stabilization with High-Penalty LQR');
plot(t, x, 'LineWidth', 1.5);
title('System States Stabilized via Observer-Based
 LQR Control');
xlabel('Time (s)');
ylabel('State Values');
legend('x_1 (\theta)', 'x_2 (d\theta/dt)', 'x_3 (\phi)',
 'x_4 (d\phi/dt)');
grid on;
```

Combined System Function

This function, called by ‘ode45‘, contains the dynamics for both the non-linear plant and the linear observer.

Function: observer_and_plant_system.m

```
function d_xa = observer_and_plant_system(t, xa, K, L, A, B, C)
x = xa(1:4);     % True state
x_hat = xa(5:8); % Estimated state
u = -K * x_hat;  % Control law using estimate
Tz = u(1); Ty = u(2);

% Real Non-linear System Dynamics
Ir = 0.05; IRws = 5;
d_x = zeros(4,1);
d_x(1) = x(2);
d_x(2) = (Tz + IRws * x(4)) / (Ir * cos(x(3)));
d_x(3) = x(4);
d_x(4) = (Ty - IRws * x(2) * cos(x(3))) / Ir;
y = C * x;

% Linear Observer Dynamics
y_hat = C * x_hat;
d_x_hat = A*x_hat + B*u + L*(y - y_hat);
d_xa = [d_x; d_x_hat];
end
```

13.3 Final Result and Conclusion

The simulation using the final, carefully tuned design was successful.

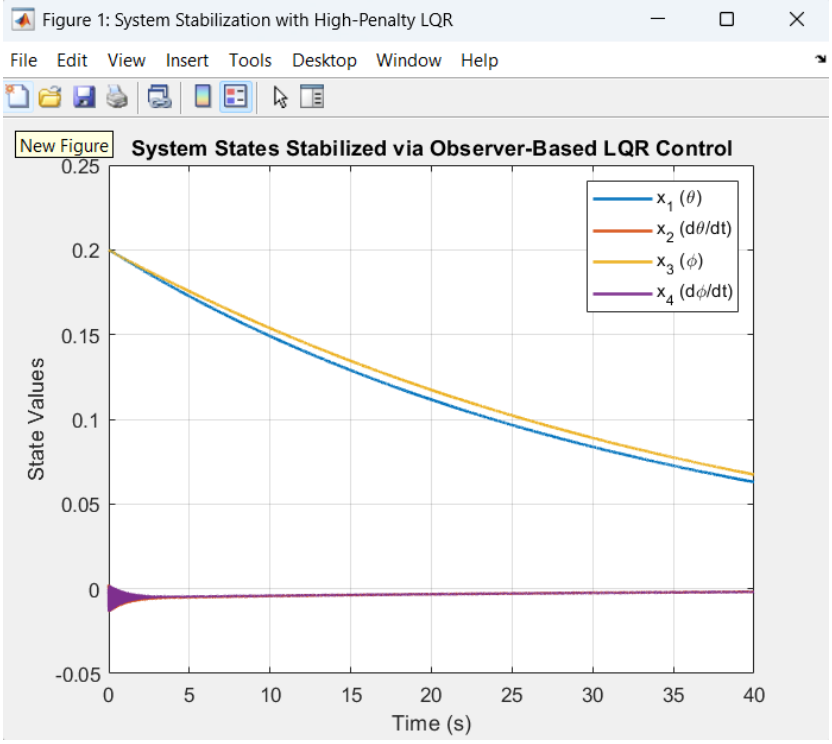


Figure 23: Successful stabilization using the final observer-based LQR controller.

The plot in Figure 23 shows all four states starting at their initial conditions and successfully and smoothly converging to zero. The slow convergence is the expected and desired result of the high-penalty (”gentle”) LQR design. This result demonstrates that with a sufficiently robust design method and careful tuning, it is possible to stabilize the challenging non-linear system using an observer-based linear controller. This successfully concludes the project’s objective.