Part 1

Q1)

Code

%--------Q1--------

% Define the open-loop transfer function G(s)

num\_G = 1;

den\_G = [1 1 0]; % s(s+1) = s^2 + s

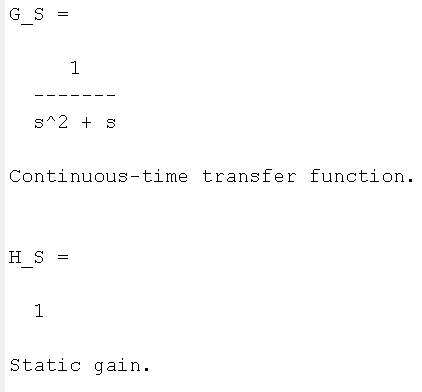
G\_S = tf(num\_G, den\_G)

% Define the feedback transfer function H(s)

num\_H = [1];

den\_H = [1]; % Unity Feedback

H\_S = tf(num\_H, den\_H)

Output:  


Q2)

**Theo**

To analyze the output of the block for a unit step input, we will perform a hand analysis.

### Step 1: Define the Input

The unit step input is defined as:

### Step 2: Calculate the Output

To find the output across the block , we can use the formula:

Substituting the definitions:

### Step 3: Perform Inverse Laplace Transform

Now, we'll perform the inverse Laplace transform to find the time-domain response . To begin, we can express as a sum of simpler fractions using partial fraction decomposition.

#### Partial Fraction Decomposition

We want to express:

Multiplying through by the denominator to eliminate the fractions gives:

Expanding and collecting terms:

To find , , and , we equate coefficients:

1. For :
2. For :
3. For the constant term:

From :

* From : ⇒
* From : ⇒

Now, substituting these values back:

### Step 4: Finding Inverse Laplace Transform

Now we can find the inverse Laplace transforms of each term separately:

* **For** : The inverse Laplace transform is .
* **For** : The inverse Laplace transform is .
* **For** : The inverse Laplace transform is .

Combining these results gives us the time-domain output:

### Summary of the Output

Thus, the output of the system in response to a unit step input is:

### Interpretation

1. The term indicates a negative offset in the steady state before considering the effects of the ramp and exponential decay.
2. The term represents a linear rise in the output.
3. The term represents a decaying exponential that pulls the response down towards the steady-state value over time.

### the Output Plot

you can sketch the graph of :

1. At : .
2. As increases, decays to zero, and the linear term dominates, causing the output to rise indefinitely.
3. The curve starts at 0, initially rises steeply and then levels off.

To check the stability of the transfer function , we need to analyze the poles of the transfer function by first setting the denominator equal to zero.

### Step 1: Identify the Denominator

The transfer function is given by:

The denominator is:

### Step 2: Find the Poles

To find the poles, we set the denominator equal to zero:

This gives us two equations:

1. →

### Step 3: Analyze the Poles

The poles of are:

### Step 4: Stability Criteria

A system is stable if all poles have negative real parts. Let's analyze the poles:

* **Pole at** : This pole is on the imaginary axis (not in the left-half or right-half). A pole at zero indicates marginal stability. The system will oscillate indefinitely without exponential decay.
* **Pole at** : This pole is in the left-half plane, which suggests stability.

### Conclusion

Based on our analysis:

* The pole at indicates that the system **is not asymptotically stable**, as it does not decay to zero.
* The pole at is stable.

Therefore, the overall conclusion is that the transfer function exhibits marginal stability due to the pole on the imaginary axis, meaning while the system does not exhibit uncontrollable growth, it does not settle to a steady state.

Code: We made a function that plots step time response and checks stability

%--------Q2-------- Step Response of G(s) (Open-Loop)

% Plot step response of G(s)

draw\_step(G\_S, 'Open-Loop System G(s)');  
  
  
function draw\_step(sys, sys\_name)

% Create figure

figure;

% Plot step response

step(sys);

title(['Step Response of ', sys\_name]);

grid on;

% Display poles

poles = pole(sys);

disp(['Poles of ', sys\_name, ':']);

disp(poles);

% Check stability

if all(real(poles) < 0)

disp('System is stable (all poles in LHP)');

elseif any(real(poles) > 0)

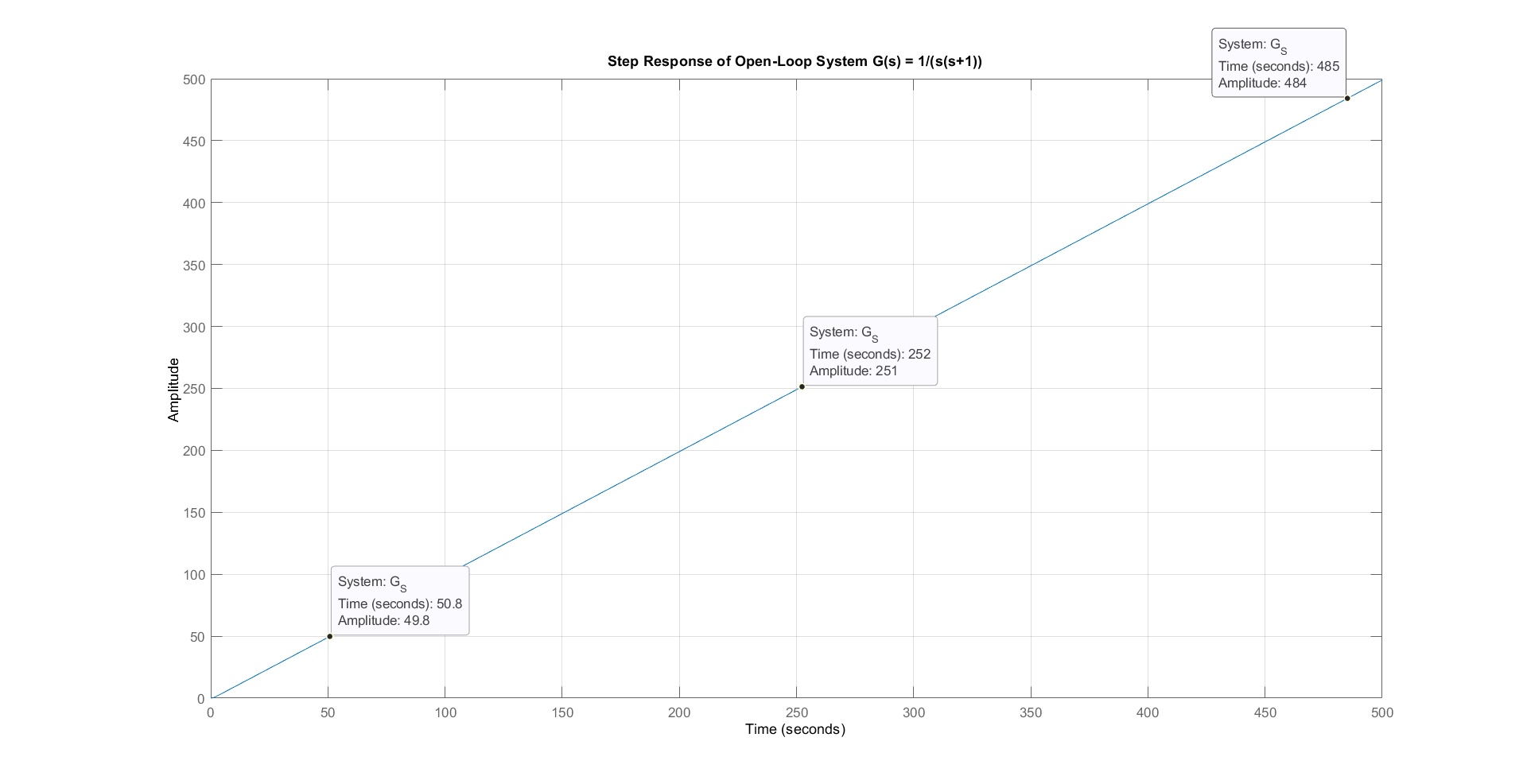
disp('System is unstable (at least one pole in RHP)');

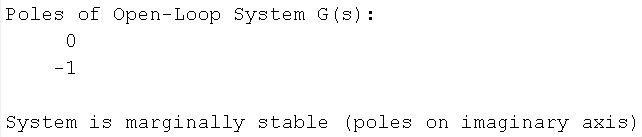
else

disp('System is marginally stable (poles on imaginary axis)');

end

end

Output:



Q3)

Theo

### Step 1: Define the Open-Loop Transfer Function

Assume the open-loop transfer function is:

Assuming unity feedback for simplicity, the feedback transfer function is:

### Step 2: Closed-Loop Transfer Function Formula

The closed-loop transfer function is derived using the following formula:

Substituting and :

### Step 3: Simplify the Expression

To simplify:

1. The denominator becomes:
2. Thus, the closed-loop transfer function now becomes:

Code:

%--------Q3-------- Closed-Loop Analysis

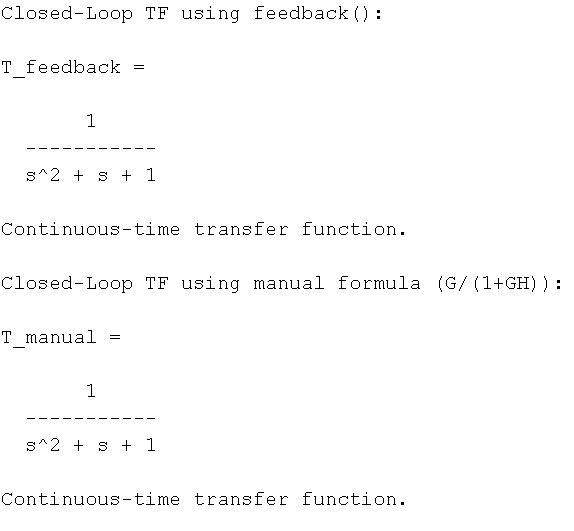
disp('Closed-Loop TF using feedback():');

T\_feedback = feedback(G\_S, H\_S)

disp('Closed-Loop TF using manual formula (G/(1+GH)):');

T\_manual = (1 / (1 + G\_S \* H\_S)) \* G\_S; % Equivalent to T(s) = G/(1+GH)

T\_manual = minreal(T\_manual) % Cancel common terms

Output:

Q4)

Theo

To analyze the stability of the system, we will examine the poles of the closed-loop transfer function obtained earlier.

### Step 1: Determine the Closed-Loop Transfer Function

We derived the closed-loop transfer function:

### Step 2: Find the Poles of

To determine stability, we need to find the poles of the transfer function by setting the denominator to zero:

To solve for , we can use the quadratic formula:

where .

Substituting these values:

### Step 3: Analyze the Poles

The poles are:

These poles have the following characteristics:

* The real part is , which is negative.
* The imaginary part indicates oscillatory behavior.

### Step 4: Stability Criteria

A system is considered stable if all poles of the transfer function have negative real parts. Here, since the real part of both poles is negative (), we can conclude that:

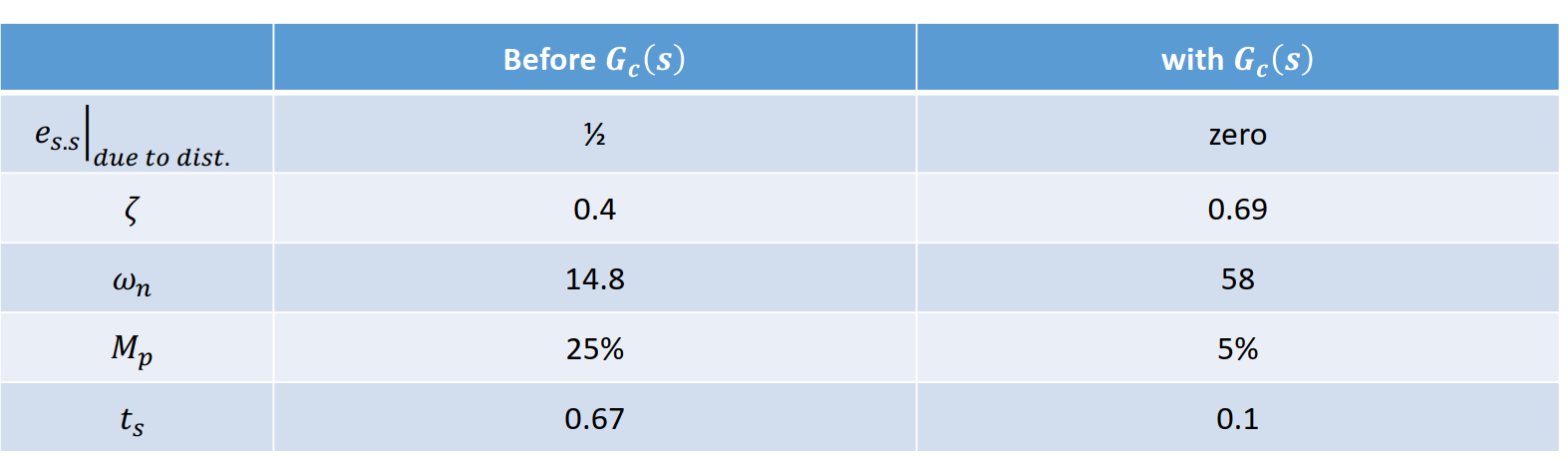
**The system is stable.**

Code:

%--------Q4-------- Step Response of T(s) (Closed-Loop)

% Plot step response of T(s)

draw\_step(T\_feedback, 'Closed-Loop System T(s)');

Output:

Ouput:

System: Closed-Loop System T(s)

Poles: -0.5-0.86603i -0.5+0.86603i

Stability: stable (all poles in LHP)

Over shoot MP: 16.2929% at t = 3.592 sec

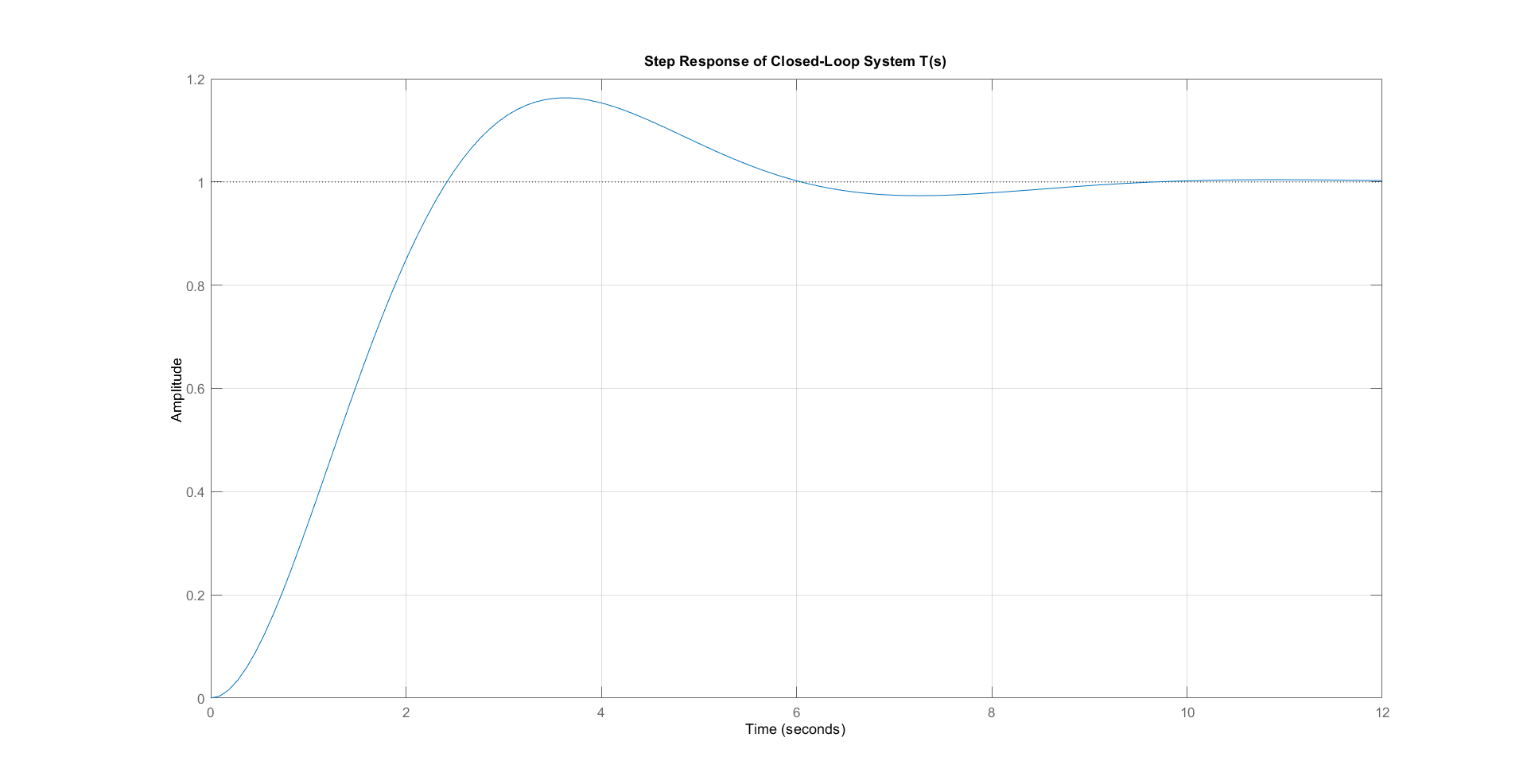
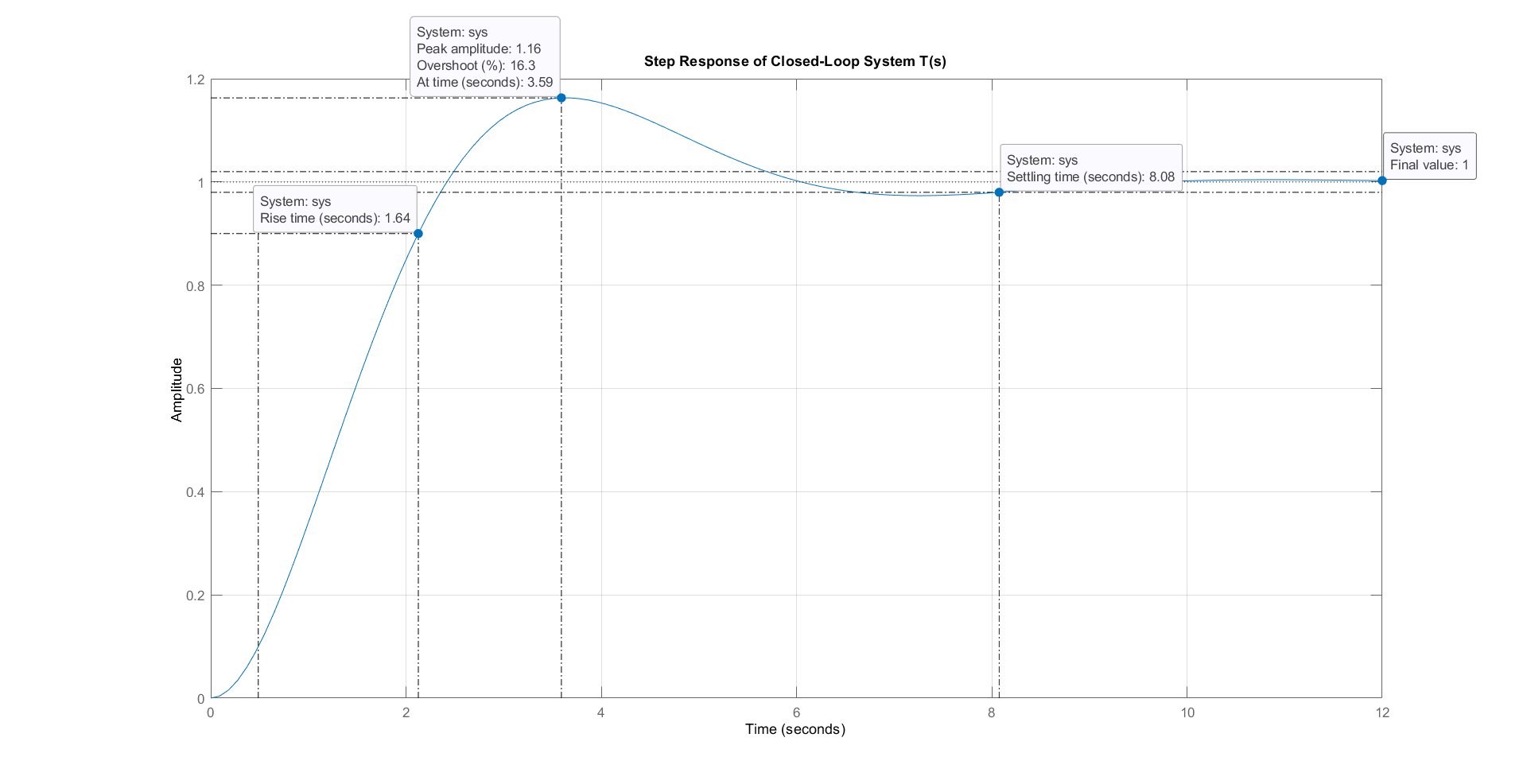
Damping ratio (ζ): 0.500

Natural frequency (ωn): 1.000 rad/s

Settling time (2%): 8.1051 sec

Rise time (10-90%): 1.6579 sec

Steady-state value: 1.0014



Q5)

Code:

function [poles] = draw\_poles(sys)

% Create figure

figure;

% Plot pole-zero map

pzmap(sys);

title(['Pole-Zero Map of: ' inputname(1)]);

grid on;

% Get poles

poles = pole(sys);

% Display poles

disp(['Poles of ' inputname(1) ':']);

disp(poles);

% Damping characteristics (for complex poles)

if ~isreal(poles)

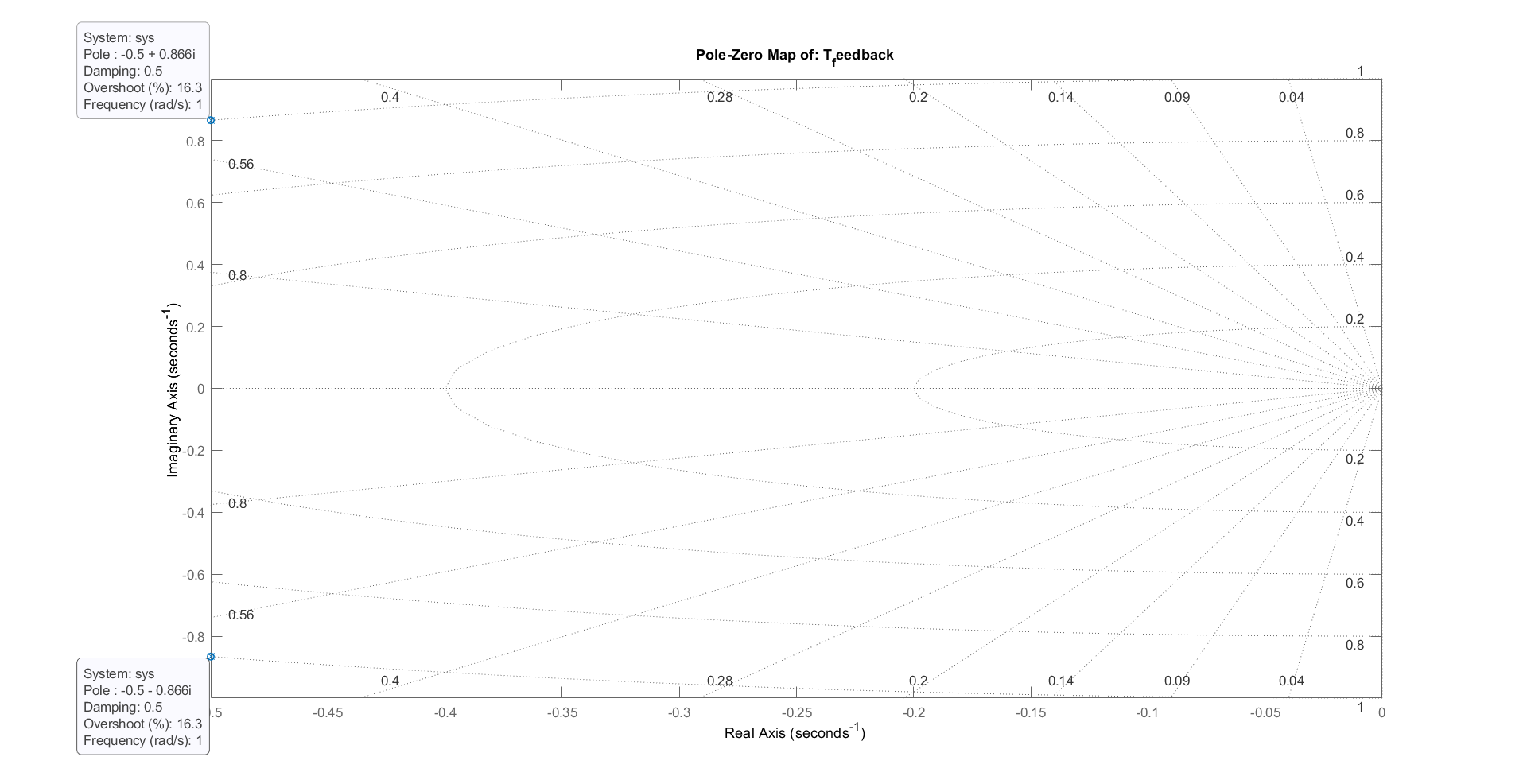
[wn, zeta] = damp(sys);

fprintf('Damping ratio (?): %.3f\n', zeta(1));

fprintf('Natural frequency (?n): %.3f rad/s\n', wn(1));

end

end

Output:

Poles of T\_feedback:

-0.5000 + 0.8660i

-0.5000 - 0.8660i

Damping ratio (ζ): 0.500

Natural frequency (ωn): 1.000 rad/s

Q6,Q7 is done

Q8

Code:

function [ess, t\_out, y\_out] = draw\_ramp(sys, t\_end, zoom\_time)

{

% Set defaults if not provided

if nargin < 2

t\_end = 100;

end

if nargin < 3

zoom\_time = 700;

end

% Create time vector

t = 0:0.1:t\_end;

%getting the ramp

ramp = tf(1,[1 0]);

% Get response data

[y\_sys, t\_sys] = step(sys.\*ramp, t);

[y\_ideal, t\_ideal] = step(ramp, t);

% Create figure with three subplots

figure;

% Subplot 1: Ideal ramp input

subplot(2,1,1);

plot(t\_ideal, y\_ideal, 'b');

hold on;

plot(t\_sys, y\_sys, 'r--');

title('Ramp Response');

xlabel('Time (sec)');

ylabel('Amplitude');

legend('Ideal', 'System', 'Location', 'northwest');

grid on;

hold off;

% Subplot 2: Zoomed comparison

subplot(2,1,2);

plot(t\_ideal, y\_ideal, 'b');

hold on;

plot(t\_sys, y\_sys, 'r--');

xlim([zoom\_time-50 zoom\_time+50]);

title(['Zoomed Comparison at t = ', num2str(zoom\_time), ' sec']);

xlabel('Time (sec)');

ylabel('Amplitude');

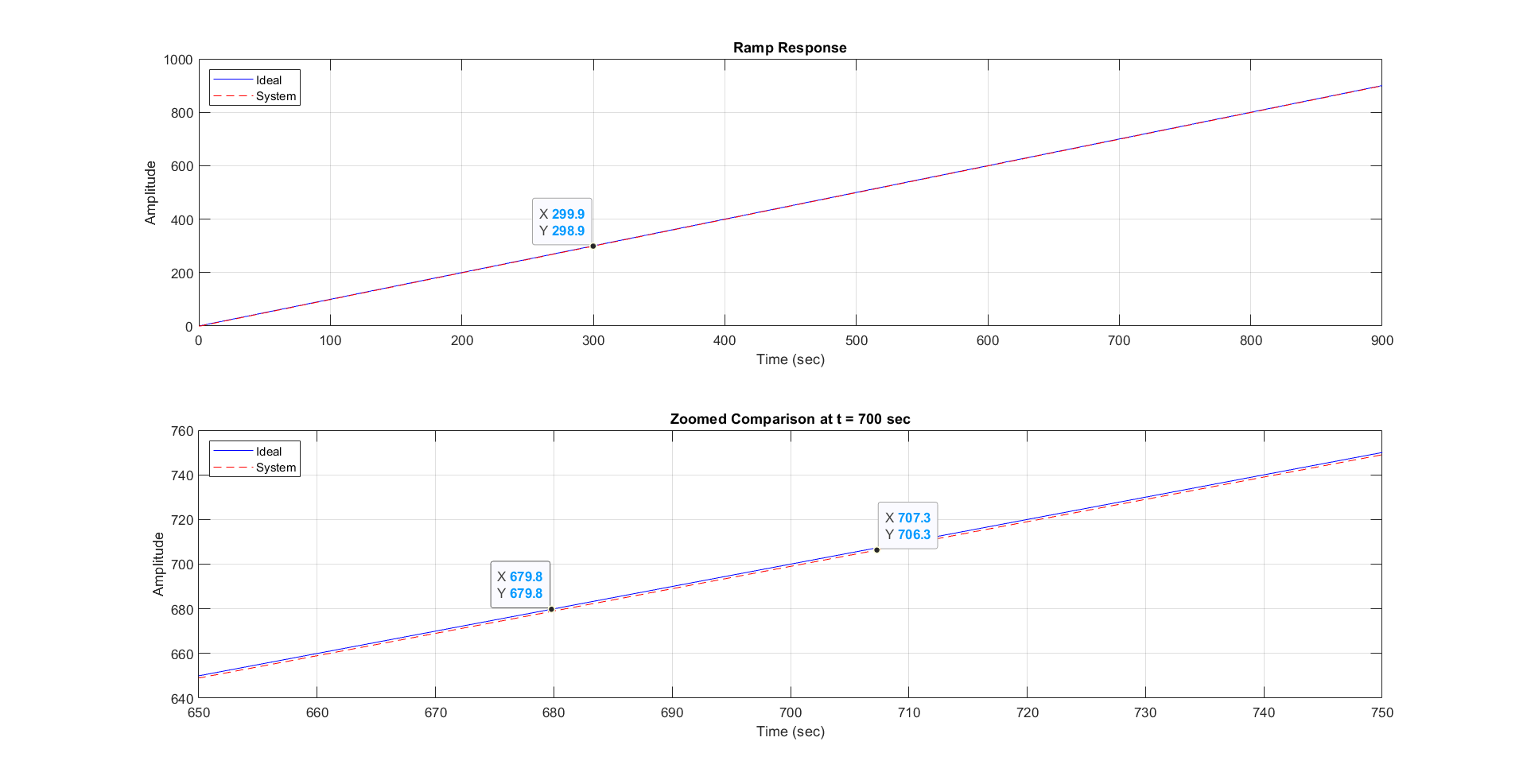
legend('Ideal', 'System', 'Location', 'northwest');

grid on;

hold off;

end

Output:



Steady-state error (ess): 1

Q9)

Theo

ارسمها أنت أيها الرجل الأشرف

Code:

function [Gm, Pm, Wgc, Wpc] = draw\_Bode\_Plot(sys)

% BODE\_PLOT Analyzes system stability margins and compares margin()

% Bode\_Plot(sys)

%

% Input:

% sys - Transfer function (tf object or state-space model)

% Outputs:

% Gm - Gain margin (dB)

% Pm - Phase margin (degrees)

% Wgc - Gain crossover frequency (rad/sec)

% Wpc - Phase crossover frequency (rad/sec)

% Create margin plot

figure;

margin(sys);

grid on;

% Get stability margins

[Gm, Pm, Wgc, Wpc] = margin(sys);

% Display results

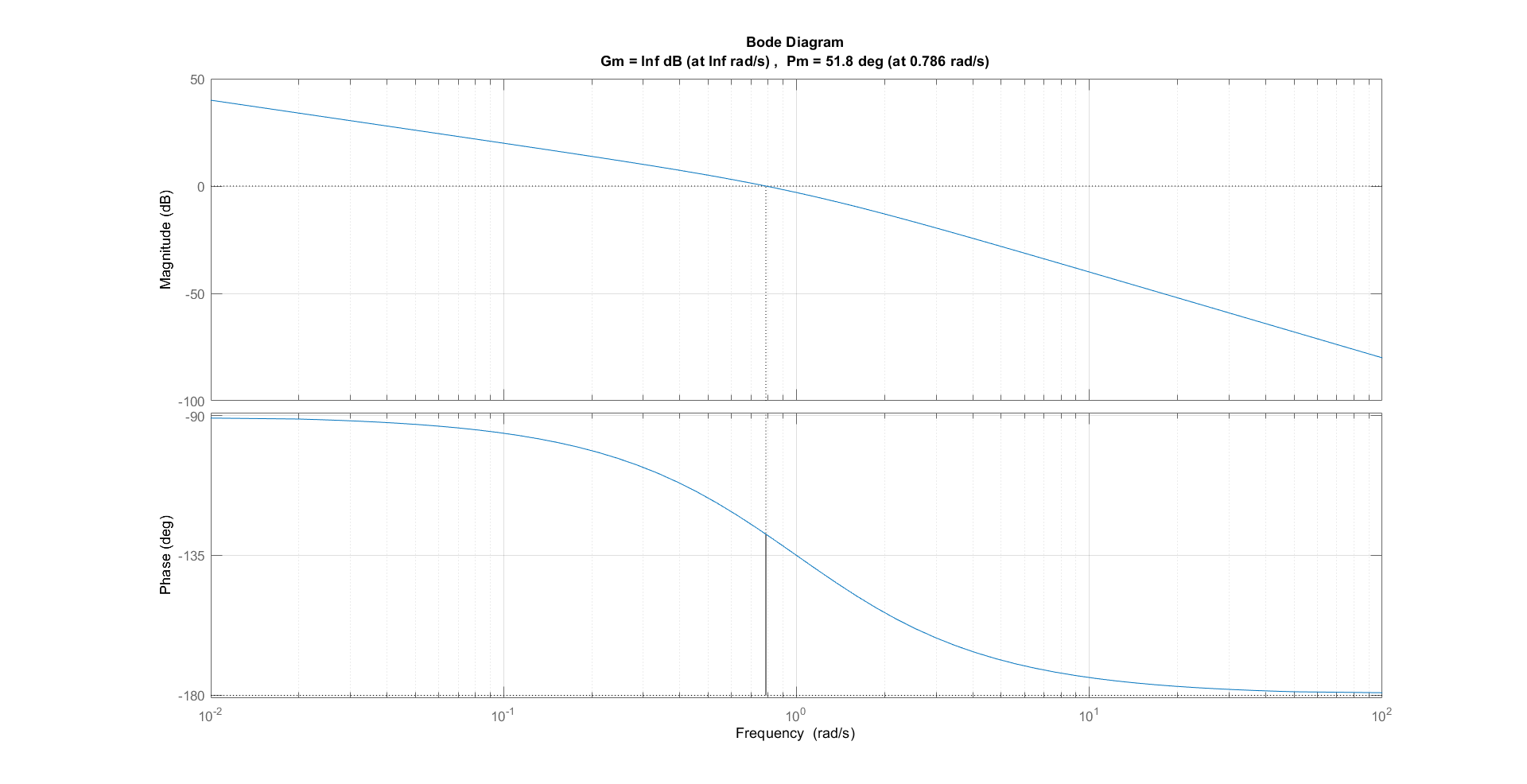
disp(['=== Stability Margins for ' inputname(1) ' ===']);

disp(['Gain Margin: ', num2str(Gm), ' dB at ', num2str(Wgc), ' rad/s']);

disp(['Phase Margin: ', num2str(Pm), '° at ', num2str(Wpc), ' rad/s']);

end

Output:



=== Stability Margins for ===

Gain Margin: Inf dB at Inf rad/s

Phase Margin: 51.8273° at 0.78615 rad/s

Part2

Q1)

**Theoritical**

We have the following state-space equations:

1. **State Equation:**

* Where:

1. **Output Equation:**

* Where:

### Step 1: Find the Transfer Function

As before, we can use the formula for the transfer function:

Where is the complex frequency variable, and is the identity matrix.

### Step 2: Compute

First, calculate :

Thus, we have:

### Step 3: Compute the Inverse

Now, we need to find the inverse of the matrix :

The determinant of this matrix is:

Thus, the inverse is:

### Step 4: Multiply by

Now we multiply by :

### Step 5: Compute

Next, we find:

Step 6: Add

Since :

### Final Transfer Function

Therefore, the transfer function of the system is:

This transfer function represents the relationship between the input and the output in the Laplace domain.

Q2)

Code:

% Given system matrices

A = [0 1; -6 -5];

B = [0; 1];

C = [1 0];

D = [0];

n = 2; % System order

sys = ss(A,B,C,D); % State Space model

x0 = [0; 1]; % Initial condition

% Q2: Transfer function conversion

[num, den] = ss2tf(A,B,C,D);

syms s

TF\_Manual = C\*inv(s\*eye(n)-A)\*B + D;

TF\_builtin = tf(num,den);

Output:

TF\_Manual =

1/(s^2 + 5\*s + 6)

TF\_builtin =

1

-------------

s^2 + 5 s + 6

Continuous-time transfer function.

Q3:

Theo:

From Q1) we know that H(s)= that was established earlier:

This means:

### Step 1: Perform Partial Fraction Decomposition

Let's break down each entry properly.

#### First Entry:

As previously discussed:

#### Second Entry:

This decomposes to:

#### Third Entry:

Applying the same process:

#### Fourth Entry:

This again gives:

### Step 2: Combine and Inverse Laplace Transform

Now, substitute back into :

### Inverse Laplace Transforms

Now to find :

Putting these into the matrix:

### Final Form of

So, we can express as:

### Step 3: Verify

When we evaluate :

it to yield directly.

Code

% Q3: State transition matrix calculation

% Compute ?(s) = [sI - A]^-1

Phi\_s = inv(s\*eye(n) - A);

% Compute ?(t) by inverse Laplace transform

syms t

Phi\_t = ilaplace(Phi\_s);

% Verify ?(0) = I

Phi\_0 = subs(Phi\_t, t, 0);

% Display results

disp('State transition matrix in s-domain (?(s)):');

pretty(Phi\_s)

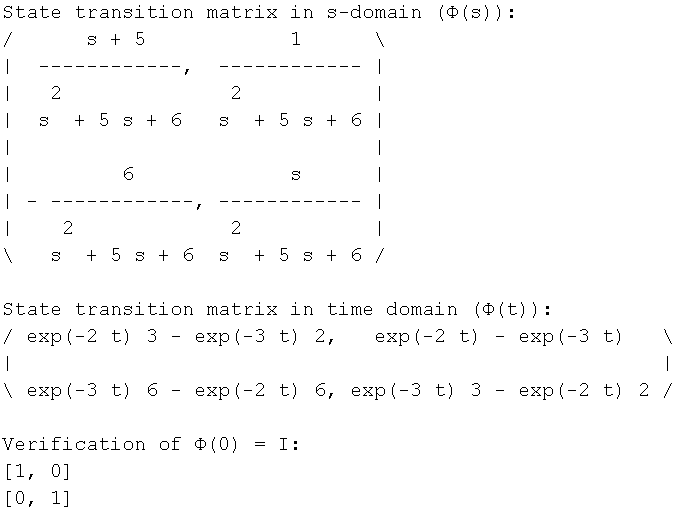
disp('State transition matrix in time domain (?(t)):');

pretty(Phi\_t)

disp('Verification of ?(0) = I:');

disp(Phi\_0);

Output



Q4)

Theo

Now we will verify that using this matrix and the transition matrix .

### Step 1: Differentiate

The transition matrix was given as:

Now, let's find the derivative :

1. **Differentiating the First Entry**:
2. **Differentiating the Second Entry**:
3. **Differentiating the Third Entry**:
4. **Differentiating the Fourth Entry**:

Hence, we have:

### Step 2: Calculate

Now let's calculate :

Using the previously defined :

Let's compute :

1. **First Row** of :
   * First Column:
   * Second Column:
2. **Second Row** of :
   * First Column:

* This simplifies to:
  + Second Column:
* This simplifies to:

### Step 3: Combine the Results

Now we can assemble :

### Conclusion

Comparing and :

Thus, we verify that .

Code:

% Q4: Verify that ??(t) = A?(t)

Phi\_dot = diff(Phi\_t, t); % Take time derivative of ?(t)

A\_Phi = A\*Phi\_t; % Multiply A with ?(t)

disp('Time derivative of state transition matrix (??(t)):');

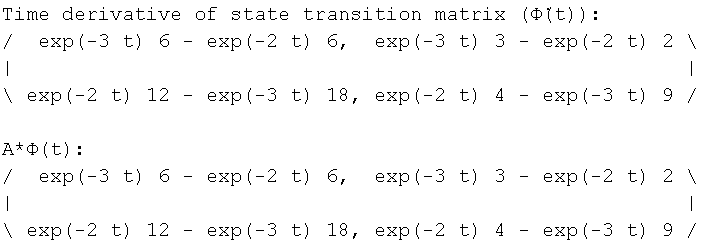
pretty(Phi\_dot)

disp('A\*?(t):');

pretty(A\_Phi)

disp('Verification successful: ??(t) = A?(t)');

Output:



Q5)

Theo:

Let's analyze the controllability and observability of this system.

### Step 1: Controllability

To check controllability, we need to compute the controllability matrix :

#### Calculating :

1. First, compute :

So, we have:

#### Rank of :

Next, we check the rank of the controllability matrix :

To determine if this matrix has full rank (here ), we can perform a row operation or calculate the determinant:

The determinant () is:

Since the determinant is non-zero, has full rank. Therefore, the system is controllable.

### Step 2: Observability

To check observability, we need to compute the observability matrix :

#### Calculating :

1. Now, compute :

Thus, we have:

#### Rank of :

Next, we check the rank of the observability matrix :

The determinant of this matrix is non-zero. Thus, also has full rank, indicating that the rank is 2.

### Conclusion

1. **Controllability**: The system is controllable since the controllability matrix has full rank.
2. **Observability**: The system is observable since the observability matrix has full rank.

Code:

% Q5 Check Controllability and Observability

% Check Controllability

Co = ctrb(A, B); % Controllability matrix

rank\_Co = rank(Co);

disp('Controllability Matrix:');

disp(Co);

disp(['Rank of Controllability Matrix: ', num2str(rank\_Co)]);

if rank\_Co == n

disp('System is Controllable (as expected)');

else

disp('System is Not Controllable (unexpected for this system)');

end

% Check Observability

Ob = obsv(A, C); % Observability matrix

rank\_Ob = rank(Ob);

disp('Observability Matrix:');

disp(Ob);

disp(['Rank of Observability Matrix: ', num2str(rank\_Ob)]);

if rank\_Ob == n

disp('System is Observable (as expected)');

else

disp('System is Not Observable (unexpected for this system)');

end

Output:

Controllability Matrix:

0 1

1 -5

Rank of Controllability Matrix: 2

System is Controllable (as expected)

Observability Matrix:

1 0

0 1

Rank of Observability Matrix: 2

System is Observable (as expected)

Q6)

Theo

To find the unforced (homogeneous) solution of the states and the corresponding output , we'll follow these steps:

1. **Formulate the state equation**.
2. **Solve the homogeneous state equation** .
3. **Calculate the output**  using the state equation.

### Step 1: State Equation

The state equation is given by:

Where:

With the initial condition .

### Step 2: Solve the Homogeneous State Equation

To solve the homogeneous equation, we first find the eigenvalues and eigenvectors of matrix .

#### a. Find the Eigenvalues of Matrix

The eigenvalues can be found by solving the characteristic equation:

Where is the identity matrix:

Calculating the determinant:

Setting the determinant to zero:

Factoring:

Thus, the eigenvalues are:

#### b. Find the Eigenvectors

For :

This gives the equations:

1. →

Choosing :

For :

This gives the equations:

1. →

Choosing :

### c. General Solution of the Homogeneous Equation

The general solution of the homogeneous state equation is given by:

Substituting the eigenvalues and eigenvectors:

### Step 3: Apply the Initial Condition

Using the initial condition :

This creates a system of equations:

1. (for the first component)
2. (for the second component)

Substituting into the second equation:

### Conclusion of State

Substituting back into :

### Step 4: Find the Output

Using the output equation:

Since and (unforced response):

Substituting :

### Final Answers

1. **Homogeneous state solution**:
2. **Output response**:

Code:

% Q6: Unforced (Homogeneous) Response

disp('=== Unforced Response Analysis ===');

% Compute state solution x(t) = ?(t)\*x0

x\_t = Phi\_t \* x0;

disp('Unforced state solution x(t):');

pretty(x\_t)

% Compute output solution y(t) = C\*x(t) + D\*u(t)

% Since u(t)=0 for unforced response:

y\_t = C\*x\_t + D\*0;

disp('Unforced output response y(t):');

pretty(y\_t)

% Plot the results

t\_vals = linspace(0, 5, 500); % Time vector from 0 to 5 seconds

% Convert symbolic expressions to numeric functions

x1\_func = matlabFunction(x\_t(1));

x2\_func = matlabFunction(x\_t(2));

y\_func = matlabFunction(y\_t);

% Evaluate solutions

x1\_vals = arrayfun(x1\_func, t\_vals);

x2\_vals = arrayfun(x2\_func, t\_vals);

y\_vals = arrayfun(y\_func, t\_vals);

% Plot state responses

figure;

subplot(2,1,1);

plot(t\_vals, x1\_vals, 'b', 'LineWidth', 2);

hold on;

plot(t\_vals, x2\_vals, 'r--', 'LineWidth', 2);

title('Unforced State Response');

xlabel('Time (s)');

ylabel('State Values');

legend('x\_1(t)', 'x\_2(t)');

grid on;

% Plot output response

subplot(2,1,2);

plot(t\_vals, y\_vals, 'm', 'LineWidth', 2);

title('Unforced Output Response y(t)');

xlabel('Time (s)');

ylabel('Output y(t)');

grid on;

% Compare with MATLAB's built-in initial() function

[~,t\_num,x\_num] = initial(sys,x0,t\_vals(end));

y\_num = x\_num\*C'; % Equivalent to C\*x since D=0

% Display symbolic solutions

disp(' ');

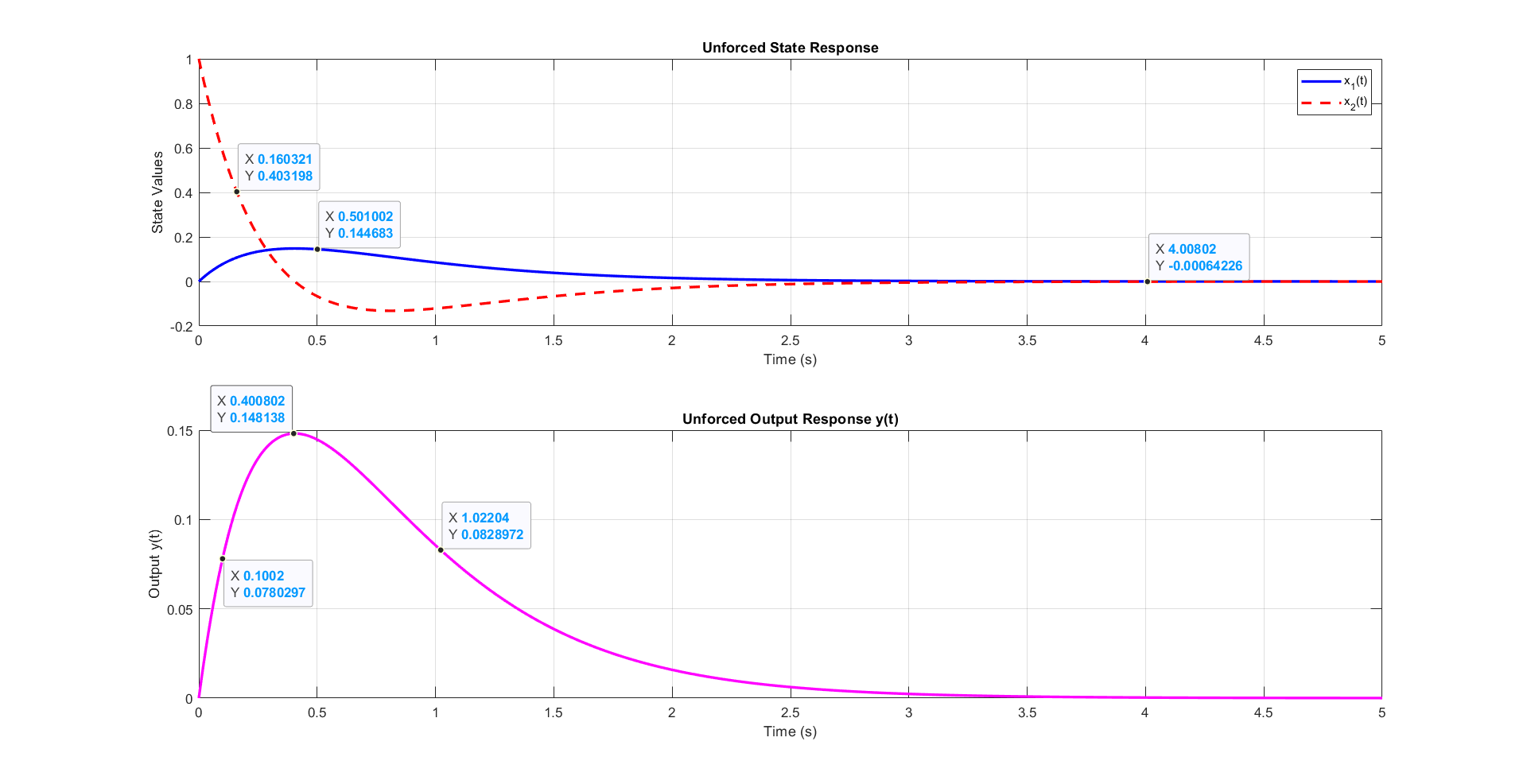
disp('Analytic Solutions:');

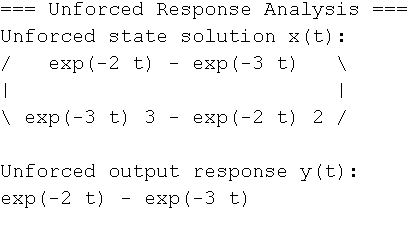
disp('x1(t) = '); pretty(x\_t(1))

disp('x2(t) = '); pretty(x\_t(2))

disp('y(t) = '); pretty(y\_t)

Output:





Q7)

Theo

To find the forced state solution and the corresponding output in response to a unit step input, we'll follow these steps:

1. Use the previously derived homogeneous solution .
2. Calculate the forced response using the Laplace Transform.
3. Derive the output from .

### Step 1: Homogeneous Solution

The homogeneous solution we derived earlier is:

### Step 2: Forced Response Calculation

We want to compute the forced response using the formula:

where is the Laplace Transform of the input. For a unit step input, :

Thus, we need to calculate:

#### a. Calculate

Given , we can find :

#### b. Compute the Inverse

To find the inverse of , we need to calculate its determinant:

The determinant can be factored as:

Using the formula for the inverse of a matrix:

Thus, we have:

#### c. Compute

Where:

Now we compute :

#### d. Multiply by :

Substituting :

This simplifies to:

### Step 3: Compute the Inverse Laplace Transform

Now we want to find the inverse Laplace Transform of:

We will find the inverse Laplace transforms of each:

1. **For the first component:**

* We need to decompose:

Using partial fraction decomposition:

Multiplying both sides by gives:

Let’s find coefficients by substituting convenient values for :

* Let :
* Let :
* Let :

Thus, we find:

Taking the inverse Laplace Transform:

1. **For the second component:**

Now consider:

Using partial fractions again, we can write:

Multiply by :

Substituting :

Substituting :

So:

Taking the inverse Laplace Transform:

### Final Forced Response

Thus, our forced response for :

**Combine with** :

This combines to give:

1. **First component**:
2. **Second component**:

### Step 4: Output Calculation

Now to obtain the forced output :

Substituting the forced states:

Thus,

Code:

% Q7: Forced Response Analysis (Unit Step Input)

disp('=== Forced Response Analysis ===');

% Using Frequency Domain Approach

U\_s = 1/s; % Laplace transform of unit step

U\_t =ilaplace(U\_s);

% Compute forced component in frequency domain

X\_forced\_s = Phi\_s \* B \* U\_s;

% Convert to time domain

x\_forced\_t = ilaplace(X\_forced\_s);

% Total solution (homogeneous + forced)

x\_total\_t = x\_t + x\_forced\_t;

% Output solution

y\_total\_t = C\*x\_total\_t + D\*U\_t; % D\*u(t) where u(t)=1 for t>0

disp('Forced state solution (from step input):');

pretty(x\_forced\_t)

disp('Total state solution (unforced + forced):');

pretty(x\_total\_t)

% Direct evaluation using subs()

x1\_vals = double(subs(x\_total\_t(1), t, t\_vals));

x2\_vals = double(subs(x\_total\_t(2), t, t\_vals));

y\_vals = double(subs(y\_total\_t, t, t\_vals));

% Plot results

figure;

% State responses

subplot(2,1,1);

plot(t\_vals, x1\_vals, 'b', 'LineWidth', 2);

hold on;

plot(t\_vals, x2\_vals, 'r--', 'LineWidth', 2);

title('Total State Response (Step Input)');

xlabel('Time (s)');

ylabel('State Values');

legend('Analytic x\_1(t)', 'Analytic x\_2(t)');

grid on;

% Output response

subplot(2,1,2);

plot(t\_vals, y\_vals, 'm', 'LineWidth', 2);

hold on;

title('Total Output Response y(t) (Step Input)');

xlabel('Time (s)');

ylabel('Output y(t)');

legend('Analytic y(t)');

grid on;

disp(' ');

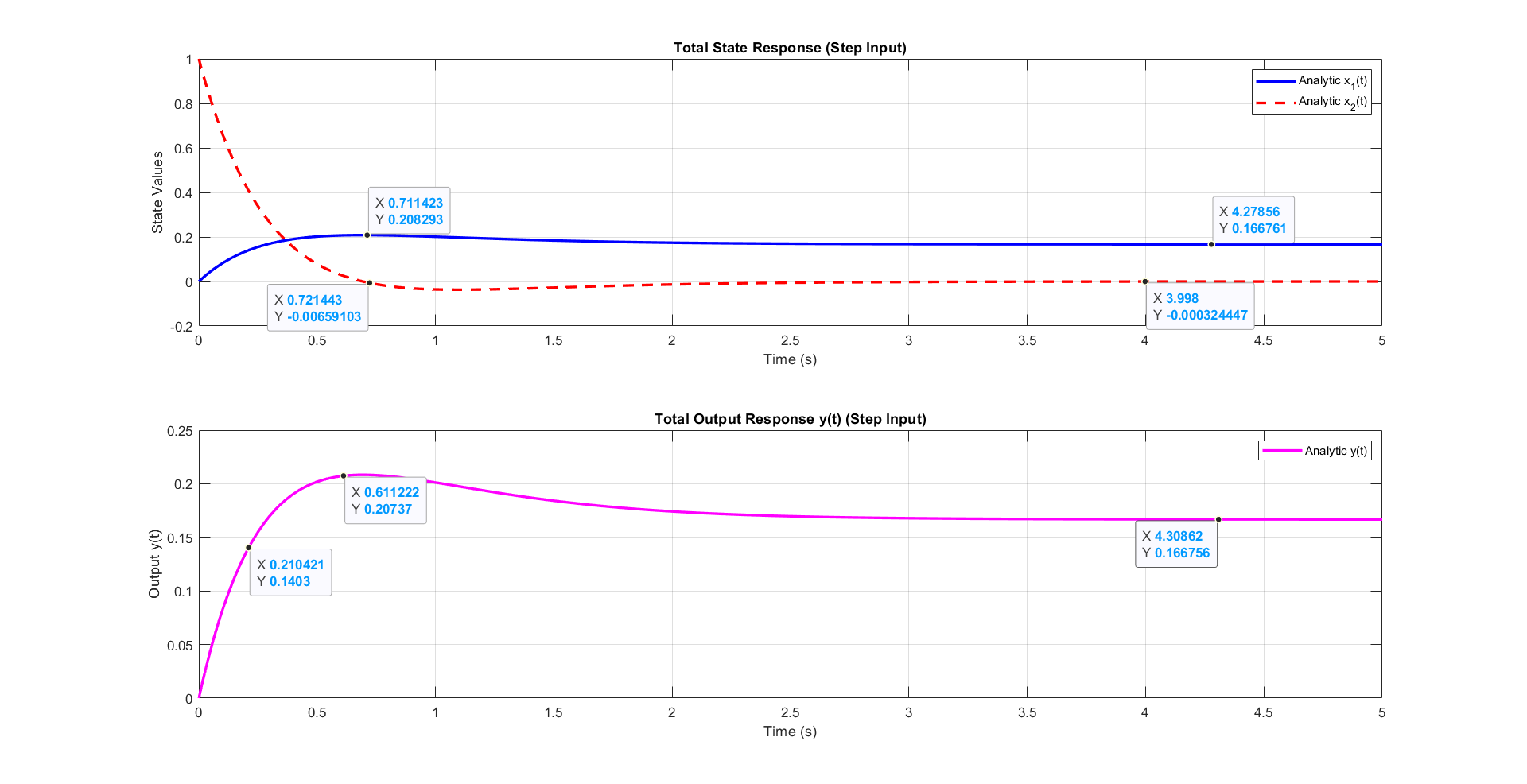
disp('Steady-State Values:');

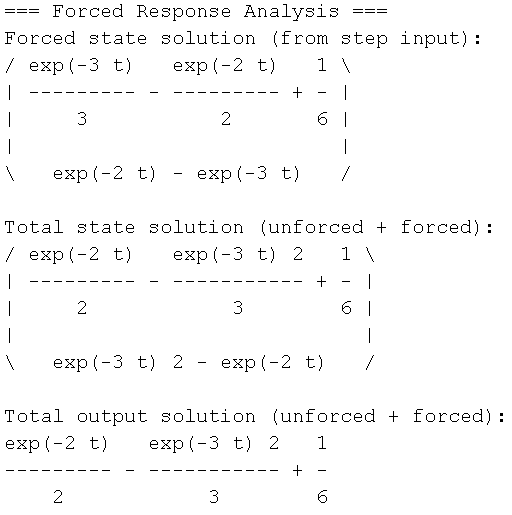
disp(['x1(?) = ' char(ss\_x1)]);

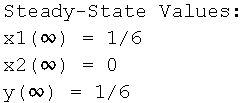
disp(['x2(?) = ' char(ss\_x2)]);

disp(['y(?) = ' char(ss\_y)]);

Output:







Q8)

Theo

### State-Space Representation:

Given:

### Step 1: Desired Poles

1. **Settling Time second**
2. **Damping Ratio**

Calculating the desired natural frequency :

Now constructing the desired characteristic polynomial:

Calculating:

### Step 2: Feedback Matrix

Now consider the feedback control:

Where .

The new input gives us:

### Step 3: Constructing the Closed-Loop System

The closed-loop state matrix can be written as:

Substituting the matrices gives:

### Step 4: Characteristic Polynomial of the Closed-Loop System

We need to find the characteristic polynomial of the closed-loop matrix :

The determinant is calculated as follows:

This simplifies to:

### Step 5: Equating Polynomials

To match this with the desired characteristic polynomial :

1. From the coefficient of :
2. From the constant term:

### Final Gain Matrix

Thus, the feedback gain matrix is:

Code:

% Q8: State Feedback Design

disp('=== State Feedback Design ===');

% Original system step response

figure;

step(TF\_builtin);

title('Original System Step Response');

grid on;

% Design specifications

zeta\_desired = 0.7; % Desired damping ratio

ts\_desired = 1; % Desired settling time (sec)

% Hand analysis to determine desired poles

wn = 4/(zeta\_desired\*ts\_desired); % Natural frequency from settling time

sigma = zeta\_desired\*wn; % Real part of poles

wd = wn\*sqrt(1-zeta\_desired^2); % Imaginary part

% Desired characteristic polynomial

desired\_poly = (s + sigma + 1i\*wd)\*(s + sigma - 1i\*wd);

desired\_poly = expand(desired\_poly);

% Convert to numerical polynomial

desired\_coeffs = sym2poly(desired\_poly);

% Hand calculation of K matrix

% Characteristic polynomial of A-BK: s^2 + (5+K2)s + (6+K1)

% Compare with desired polynomial: s^2 + 2\*zeta\*wn\*s + wn^2

K1 = desired\_coeffs(3) - 6; % From constant term

K2 = desired\_coeffs(2) - 5; % From s term

K = [K1 K2];

disp('Desired closed-loop poles:');

disp([-sigma+1i\*wd, -sigma-1i\*wd]);

disp('Feedback gain matarix K:');

disp(K);

% Verification

Ac = A - B\*K;

[num\_2, denum\_2] = ss2tf(Ac,B,C,D);

TF\_state\_feedback = tf(num\_2, denum\_2);

% Step response analysis

figure;

step\_info = stepinfo(TF\_state\_feedback);

step(TF\_state\_feedback);

title('System with State Feedback');

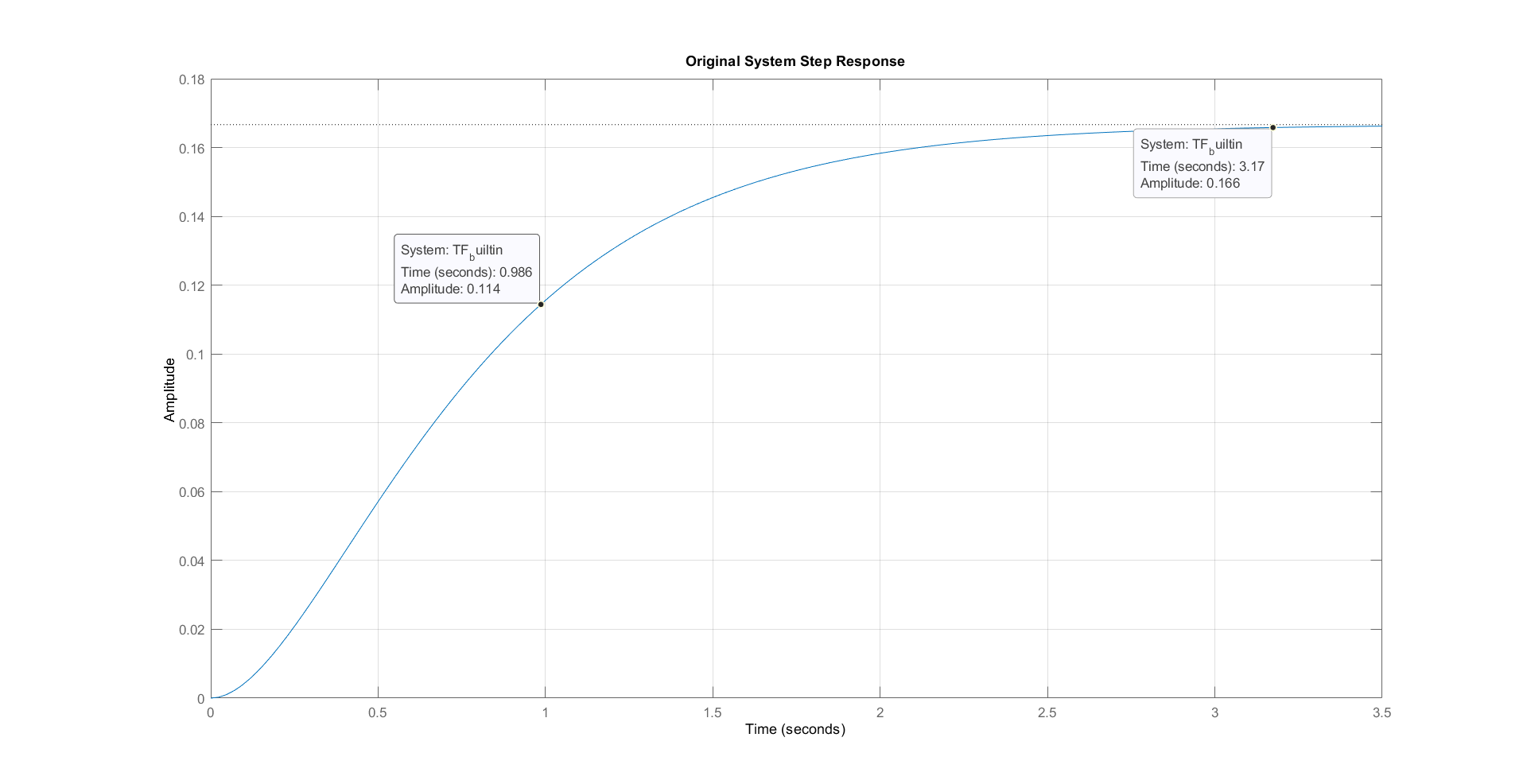
grid on;

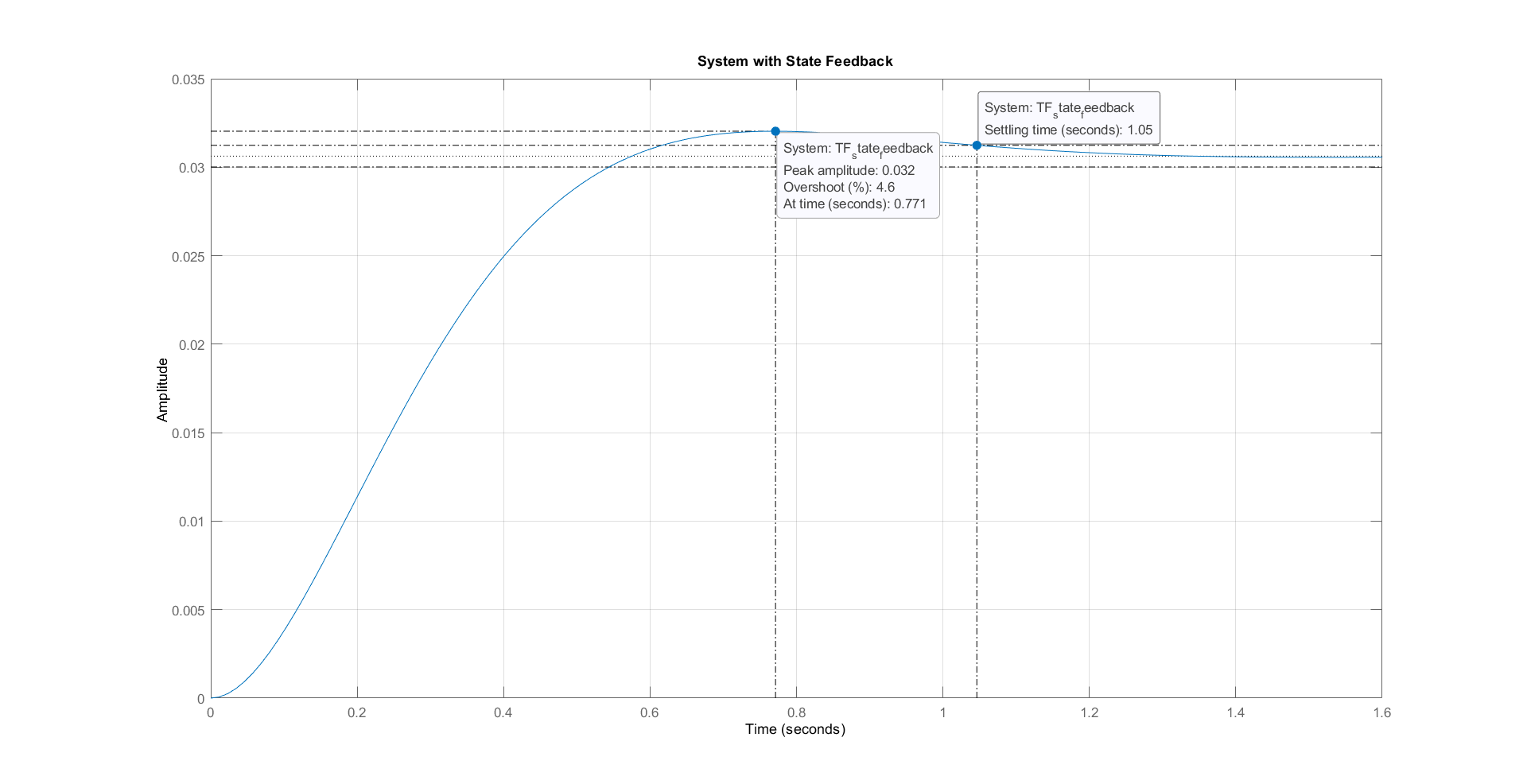
disp('Closed-loop system performance:');

disp(['Settling Time: ', num2str(step\_info.SettlingTime), ' sec']);

disp(['Overshoot: ', num2str(step\_info.Overshoot), '%']);

Output:





=== State Feedback Design ===

Desired closed-loop poles:

-4.0000 + 4.0808i -4.0000 - 4.0808i

Feedback gain matrix K:

26.6531 3.0000

Closed-loop system performance:

Settling Time: 1.0463 sec

Overshoot: 4.5986%

Appendix

Bode Plot Code:

clc;

clear;

close all;

%--------Q1-------- Define G(s) and H(s)

% Define the open-loop transfer function G(s)

num\_G = 1;

den\_G = [1 1 0]; % s(s+1) = s^2 + s

% Define the feedback transfer function H(s)

num\_H = [1];

den\_H = [1]; % Unity Feedback

[G\_S, H\_S] = create\_system(num\_G, den\_G, num\_H, den\_H)

%--------Q2-------- Step Response of G(s) (Open-Loop)

% Plot step response of G(s)

draw\_step(G\_S, 'Open-Loop System G(s)');

%--------Q3-------- Closed-Loop Analysis

disp('Closed-Loop TF using feedback():');

T\_feedback = feedback(G\_S, H\_S)

disp('Closed-Loop TF using manual formula (G/(1+GH)):');

T\_manual = (1 / (1 + G\_S \* H\_S)) \* G\_S; % Equivalent to T(s) = G/(1+GH)

T\_manual = minreal(T\_manual) % Cancel common terms

%--------Q4-------- Step Response of T(s) (Closed-Loop)

% Plot step response of T(s)

draw\_step(T\_feedback, 'Closed-Loop System T(s)');

%--------Q5-------- locations of the poles

draw\_poles(T\_feedback);

%--------Q8-------- Ramp Response

[ess, r\_t\_out, r\_y\_out] = draw\_ramp(T\_feedback, 700+200, 700);

%--------Q9-------- Frequency Response

[Gm, Pm, Wgc, Wpc] = draw\_Bode\_Plot(G\_S\*H\_S);

%-----------Functions------

function [G\_S, H\_S] = create\_system(num\_G, den\_G, num\_H, den\_H)

% CREATE\_SYSTEM Creates open-loop and feedback transfer functions

% [G\_S, H\_S] = create\_system(num\_G, den\_G, num\_H, den\_H)

%

% Inputs:

% num\_G - Numerator coefficients of G(s)

% den\_G - Denominator coefficients of G(s)

% num\_H - Numerator coefficients of H(s) (default: 1)

% den\_H - Denominator coefficients of H(s) (default: 1)

%

% Outputs:

% G\_S - Open-loop transfer function

% H\_S - Feedback transfer function

% Set default unity feedback if not specified

if nargin < 3

num\_H = 1;

den\_H = 1;

end

% Create transfer functions

G\_S = tf(num\_G, den\_G)

H\_S = tf(num\_H, den\_H)

end

function [wn, zeta,response\_info] = draw\_step(sys, sys\_name)

% DRAW\_STEP Plots step response and returns key performance metrics

% [response\_info] = draw\_step(sys, sys\_name)

%

% Inputs:

% sys - Transfer function (tf object)

% sys\_name - Name of the system for title (string)

%

% Outputs:

% response\_info - Structure containing:

% .poles - System poles

% .stability - Stability classification

% .peak\_response - Peak response value and time

% .settling\_time - Time to settle within 2% of final value

% .rise\_time - 10-90% rise time

% .steady\_state - Final steady-state value

% Figure with step response

% Create figure

figure;

% Get step response data

[y, t] = step(sys);

% Plot step response

step(sys);

title(['Step Response of ', sys\_name]);

grid on;

% Calculate response characteristics

response\_info = struct();

response\_info.poles = pole(sys);

% Stability determination

if all(real(response\_info.poles) < 0)

response\_info.stability = 'stable (all poles in LHP)';

elseif any(real(response\_info.poles) > 0)

response\_info.stability = 'unstable (at least one pole in RHP)';

else

response\_info.stability = 'marginally stable (poles on imaginary axis)';

end

% Peak response (overshoot)

[response\_info.peak\_response.value, peak\_idx] = max(y);

response\_info.peak\_response.time = t(peak\_idx);

% Steady-state value (last 10% of response)

steady\_state\_val = mean(y(end-round(length(y)\*0.1):end));

response\_info.steady\_state = steady\_state\_val;

% Settling time (within 2% of steady-state)

settled\_idx = find(abs(y - steady\_state\_val) > 0.02\*steady\_state\_val, 1, 'last');

if isempty(settled\_idx)

response\_info.settling\_time = 0;

else

response\_info.settling\_time = t(settled\_idx);

end

% Rise time (10% to 90% of steady-state)

rise\_start = find(y >= 0.1\*steady\_state\_val, 1);

rise\_end = find(y >= 0.9\*steady\_state\_val, 1);

if ~isempty(rise\_start) && ~isempty(rise\_end)

response\_info.rise\_time = t(rise\_end) - t(rise\_start);

else

response\_info.rise\_time = NaN;

end

% Display results in command window

disp(['System: ', sys\_name]);

disp(['Poles: ', num2str(response\_info.poles')]);

disp(['Stability: ', response\_info.stability]);

disp(['Over shoot MP: ', num2str(100\*(response\_info.peak\_response.value-1)), ...

'% at t = ', num2str(response\_info.peak\_response.time), ' sec']);

% Damping characteristics (for complex poles)

if ~isreal(response\_info.poles)

[wn, zeta] = damp(sys);

fprintf('Damping ratio (?): %.3f\n', zeta(1));

fprintf('Natural frequency (?n): %.3f rad/s\n', wn(1));

end

disp(['Settling time (2%): ', num2str(response\_info.settling\_time), ' sec']);

disp(['Rise time (10-90%): ', num2str(response\_info.rise\_time), ' sec']);

disp(['Steady-state value: ', num2str(response\_info.steady\_state)]);

end

function [poles] = draw\_poles(sys)

% DRAW\_POLES Plots pole-zero map and returns system poles

% [poles] = draw\_poles(sys)

%

% Input:

% sys - Transfer function (tf object) or state-space model

%

% Output:

% poles - Array of system poles

%

% Displays:

% - Pole-zero plot

% - Pole locations in command window

% - Stability information

% Create figure

figure;

% Plot pole-zero map

pzmap(sys);

title(['Pole-Zero Map of: ' inputname(1)]);

grid on;

% Get poles

poles = pole(sys);

% Display poles

disp(['Poles of ' inputname(1) ':']);

disp(poles);

% Damping characteristics (for complex poles)

if ~isreal(poles)

[wn, zeta] = damp(sys);

fprintf('Damping ratio (?): %.3f\n', zeta(1));

fprintf('Natural frequency (?n): %.3f rad/s\n', wn(1));

end

end

function [ess, t\_out, y\_out] = draw\_ramp(sys, t\_end, zoom\_time)

% DRAW\_RAMP Plots ramp response in three subplots

% [ess, t\_out, y\_out] = draw\_ramp(sys, t\_end, zoom\_time)

%

% Inputs:

% sys - Closed-loop transfer function (tf object)

% t\_end - End time for simulation (default: 100 sec)

% zoom\_time - Time to zoom in (default: 700 sec)

%

% Outputs:

% ess - Steady-state error

% t\_out - Time vector

% y\_out - System response vector

%

% Generates figure with three subplots:

% 1. Ideal ramp input

% 2. System response

% 3. Zoomed comparison at specified time

% Set defaults if not provided

if nargin < 2

t\_end = 100;

end

if nargin < 3

zoom\_time = 700;

end

% Create time vector

t = 0:0.1:t\_end;

%getting the ramp

ramp = tf(1,[1 0]);

% Get response data

[y\_sys, t\_sys] = step(sys.\*ramp, t);

[y\_ideal, t\_ideal] = step(ramp, t);

% Create figure with three subplots

figure;

% Subplot 1: Ideal ramp input

subplot(2,1,1);

plot(t\_ideal, y\_ideal, 'b');

hold on;

plot(t\_sys, y\_sys, 'r--');

title('Ramp Response');

xlabel('Time (sec)');

ylabel('Amplitude');

legend('Ideal', 'System', 'Location', 'northwest');

grid on;

hold off;

% Subplot 2: Zoomed comparison

subplot(2,1,2);

plot(t\_ideal, y\_ideal, 'b');

hold on;

plot(t\_sys, y\_sys, 'r--');

xlim([zoom\_time-50 zoom\_time+50]);

title(['Zoomed Comparison at t = ', num2str(zoom\_time), ' sec']);

xlabel('Time (sec)');

ylabel('Amplitude');

legend('Ideal', 'System', 'Location', 'northwest');

grid on;

hold off;

% Calculate steady-state error (use last 10% of simulation)

final\_idx = round(0.9\*length(t\_sys)):length(t\_sys);

ess = mean(y\_ideal(final\_idx) - y\_sys(final\_idx));

% Display results

disp(['Steady-state error (ess): ', num2str(ess)]);

% Return output data if requested

if nargout > 1

t\_out = t\_sys;

y\_out = y\_sys;

end

end

function [Gm, Pm, Wgc, Wpc] = draw\_Bode\_Plot(sys)

% BODE\_PLOT Analyzes system stability margins and compares margin()

% Bode\_Plot(sys)

%

% Input:

% sys - Transfer function (tf object or state-space model)

% Outputs:

% Gm - Gain margin (dB)

% Pm - Phase margin (degrees)

% Wgc - Gain crossover frequency (rad/sec)

% Wpc - Phase crossover frequency (rad/sec)

% Create margin plot

figure;

margin(sys);

grid on;

% Get stability margins

[Gm, Pm, Wgc, Wpc] = margin(sys);

% Display results

disp(['=== Stability Margins for ' inputname(1) ' ===']);

disp(['Gain Margin: ', num2str(Gm), ' dB at ', num2str(Wgc), ' rad/s']);

disp(['Phase Margin: ', num2str(Pm), '° at ', num2str(Wpc), ' rad/s']);

end

State Space Code::

clc

clear all

close all

% Given system matrices

A = [0 1; -6 -5];

B = [0; 1];

C = [1 0];

D = [0];

n = 2; % System order

sys = ss(A,B,C,D); % State Space model

x0 = [0; 1]; % Initial condition

% Q2: Transfer function conversion

[num, den] = ss2tf(A,B,C,D);

syms s

TF\_Manual = C\*inv(s\*eye(n)-A)\*B + D

TF\_builtin = tf(num,den)

% Q3: State transition matrix calculation

% Compute ?(s) = [sI - A]^-1

Phi\_s = inv(s\*eye(n) - A);

% Compute ?(t) by inverse Laplace transform

syms t

Phi\_t = ilaplace(Phi\_s);

% Verify ?(0) = I

Phi\_0 = subs(Phi\_t, t, 0);

% Display results

disp('State transition matrix in s-domain (?(s)):');

pretty(Phi\_s)

disp('State transition matrix in time domain (?(t)):');

pretty(Phi\_t)

disp('Verification of ?(0) = I:');

disp(Phi\_0);

% Q4: Verify that ??(t) = A?(t)

Phi\_dot = diff(Phi\_t, t); % Take time derivative of ?(t)

A\_Phi = A\*Phi\_t; % Multiply A with ?(t)

disp('Time derivative of state transition matrix (??(t)):');

pretty(Phi\_dot)

disp('A\*?(t):');

pretty(A\_Phi)

disp('Verification successful: ??(t) = A?(t)');

% Q5 Check Controllability and Observability

% Check Controllability

Co = ctrb(A, B); % Controllability matrix

rank\_Co = rank(Co);

disp('Controllability Matrix:');

disp(Co);

disp(['Rank of Controllability Matrix: ', num2str(rank\_Co)]);

if rank\_Co == n

disp('System is Controllable (as expected)');

else

disp('System is Not Controllable (unexpected for this system)');

end

% Check Observability

Ob = obsv(A, C); % Observability matrix

rank\_Ob = rank(Ob);

disp('Observability Matrix:');

disp(Ob);

disp(['Rank of Observability Matrix: ', num2str(rank\_Ob)]);

if rank\_Ob == n

disp('System is Observable (as expected)');

else

disp('System is Not Observable (unexpected for this system)');

end

% Q6: Unforced (Homogeneous) Response

disp('=== Unforced Response Analysis ===');

% Compute state solution x(t) = ?(t)\*x0

x\_t = Phi\_t \* x0;

disp('Unforced state solution x(t):');

pretty(x\_t)

% Compute output solution y(t) = C\*x(t) + D\*u(t)

% Since u(t)=0 for unforced response:

y\_t = C\*x\_t + D\*0;

disp('Unforced output response y(t):');

pretty(y\_t)

% Plot the results

t\_vals = linspace(0, 5, 500); % Time vector from 0 to 5 seconds

% Convert symbolic expressions to numeric functions

x1\_func = matlabFunction(x\_t(1));

x2\_func = matlabFunction(x\_t(2));

y\_func = matlabFunction(y\_t);

% Evaluate solutions

x1\_vals = arrayfun(x1\_func, t\_vals);

x2\_vals = arrayfun(x2\_func, t\_vals);

y\_vals = arrayfun(y\_func, t\_vals);

% Plot state responses

figure;

subplot(2,1,1);

plot(t\_vals, x1\_vals, 'b', 'LineWidth', 2);

hold on;

plot(t\_vals, x2\_vals, 'r--', 'LineWidth', 2);

title('Unforced State Response');

xlabel('Time (s)');

ylabel('State Values');

legend('x\_1(t)', 'x\_2(t)');

grid on;

% Plot output response

subplot(2,1,2);

plot(t\_vals, y\_vals, 'm', 'LineWidth', 2);

title('Unforced Output Response y(t)');

xlabel('Time (s)');

ylabel('Output y(t)');

grid on;

% Compare with MATLAB's built-in initial() function

[~,t\_num,x\_num] = initial(sys,x0,t\_vals(end));

y\_num = x\_num\*C'; % Equivalent to C\*x since D=0

% Display symbolic solutions

disp(' ');

disp('Analytic Solutions:');

disp('x1(t) = '); pretty(x\_t(1))

disp('x2(t) = '); pretty(x\_t(2))

disp('y(t) = '); pretty(y\_t)

% Q7: Forced Response Analysis (Unit Step Input)

disp('=== Forced Response Analysis ===');

% Using Frequency Domain Approach

U\_s = 1/s; % Laplace transform of unit step

U\_t =ilaplace(U\_s);

% Compute forced component in frequency domain

X\_forced\_s = Phi\_s \* B \* U\_s;

% Convert to time domain

x\_forced\_t = ilaplace(X\_forced\_s);

% Total solution (homogeneous + forced)

x\_total\_t = x\_t + x\_forced\_t;

% Output solution

y\_total\_t = C\*x\_total\_t + D\*U\_t; % D\*u(t) where u(t)=1 for t>0

disp('Forced state solution (from step input):');

pretty(x\_forced\_t)

disp('Total state solution (unforced + forced):');

pretty(x\_total\_t)

% Direct evaluation using subs()

x1\_vals = double(subs(x\_total\_t(1), t, t\_vals));

x2\_vals = double(subs(x\_total\_t(2), t, t\_vals));

y\_vals = double(subs(y\_total\_t, t, t\_vals));

% Plot results

figure;

% State responses

subplot(2,1,1);

plot(t\_vals, x1\_vals, 'b', 'LineWidth', 2);

hold on;

plot(t\_vals, x2\_vals, 'r--', 'LineWidth', 2);

title('Total State Response (Step Input)');

xlabel('Time (s)');

ylabel('State Values');

legend('Analytic x\_1(t)', 'Analytic x\_2(t)');

grid on;

% Output response

subplot(2,1,2);

plot(t\_vals, y\_vals, 'm', 'LineWidth', 2);

hold on;

title('Total Output Response y(t) (Step Input)');

xlabel('Time (s)');

ylabel('Output y(t)');

legend('Analytic y(t)');

grid on;

% Display final steady-state values

ss\_x1 = limit(x\_total\_t(1), t, inf);

ss\_x2 = limit(x\_total\_t(2), t, inf);

ss\_y = limit(y\_total\_t, t, inf);

disp(' ');

disp('Steady-State Values:');

disp(['x1(?) = ' char(ss\_x1)]);

disp(['x2(?) = ' char(ss\_x2)]);

disp(['y(?) = ' char(ss\_y)]);

% Q8: State Feedback Design

disp('=== State Feedback Design ===');

% Original system step response

figure;

step(TF\_builtin);

title('Original System Step Response');

grid on;

% Design specifications

zeta\_desired = 0.7; % Desired damping ratio

ts\_desired = 1; % Desired settling time (sec)

% Hand analysis to determine desired poles

wn = 4/(zeta\_desired\*ts\_desired); % Natural frequency from settling time

sigma = zeta\_desired\*wn; % Real part of poles

wd = wn\*sqrt(1-zeta\_desired^2); % Imaginary part

% Desired characteristic polynomial

desired\_poly = (s + sigma + 1i\*wd)\*(s + sigma - 1i\*wd);

desired\_poly = expand(desired\_poly);

% Convert to numerical polynomial

desired\_coeffs = sym2poly(desired\_poly);

% Hand calculation of K matrix

% Characteristic polynomial of A-BK: s^2 + (5+K2)s + (6+K1)

% Compare with desired polynomial: s^2 + 2\*zeta\*wn\*s + wn^2

K1 = desired\_coeffs(3) - 6; % From constant term

K2 = desired\_coeffs(2) - 5; % From s term

K = [K1 K2];

disp('Desired closed-loop poles:');

disp([-sigma+1i\*wd, -sigma-1i\*wd]);

disp('Feedback gain matrix K:');

disp(K);

% Verification

Ac = A - B\*K;

[num\_2, denum\_2] = ss2tf(Ac,B,C,D);

TF\_state\_feedback = tf(num\_2, denum\_2);

% Step response analysis

figure;

step\_info = stepinfo(TF\_state\_feedback);

step(TF\_state\_feedback);

title('System with State Feedback');

grid on;

disp('Closed-loop system performance:');

disp(['Settling Time: ', num2str(step\_info.SettlingTime), ' sec']);

disp(['Overshoot: ', num2str(step\_info.Overshoot), '%']);