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

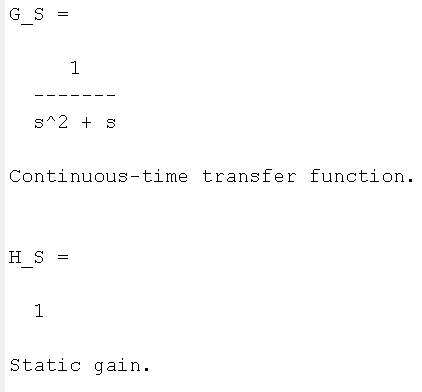
G\_S = tf(num\_G, den\_G)

% Define the feedback transfer function H(s)

num\_H = [1];

den\_H = [1]; % Unity Feedback

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

Output:  


Q2) the 𝑠𝑡𝑒𝑝() command to plot the output of 𝐺(𝑆)

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)

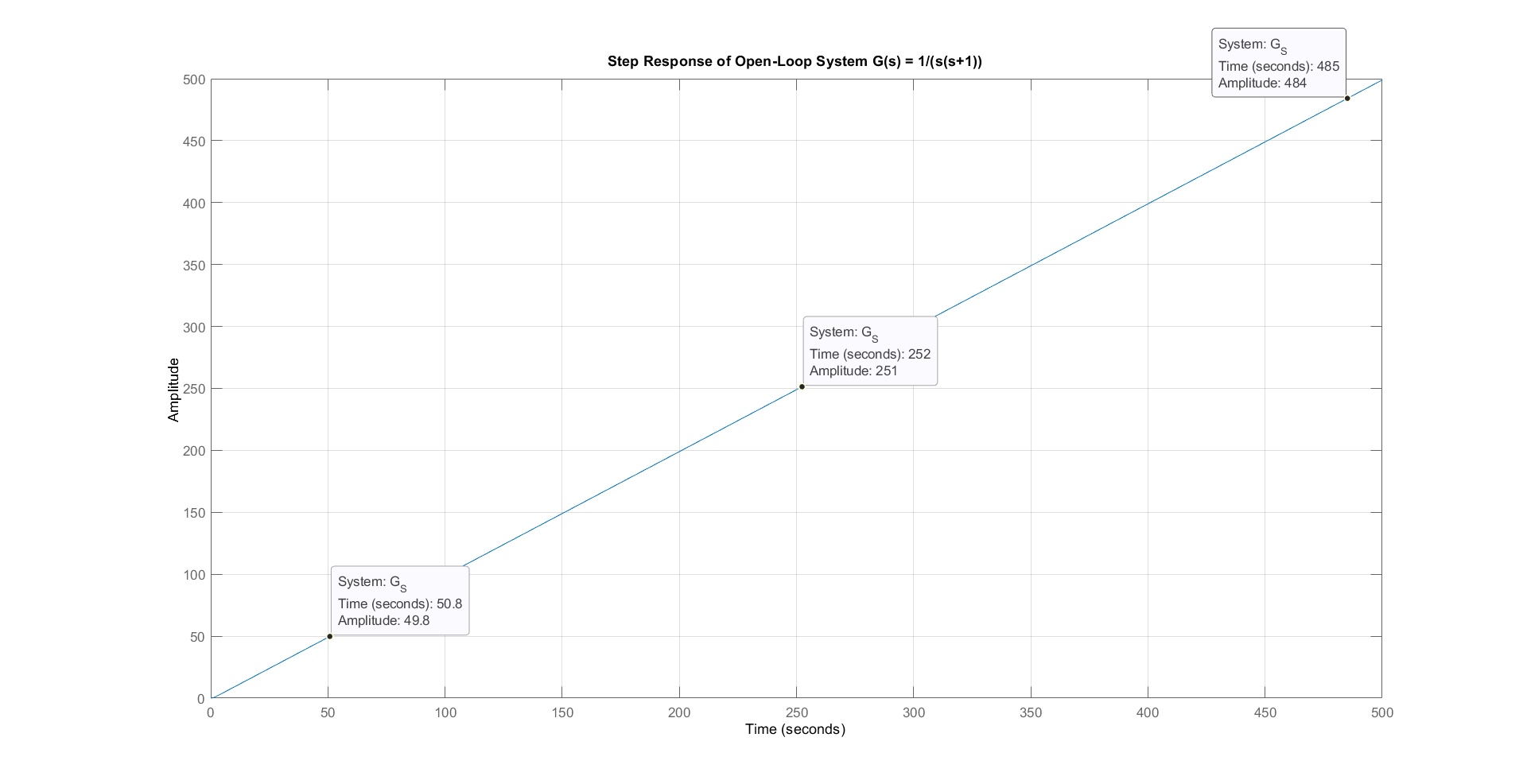
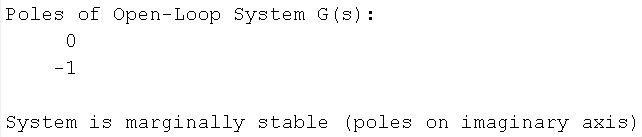
disp('System is unstable (at least one pole in RHP)');

else

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

end

end

Output:

Q3)

Code:

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

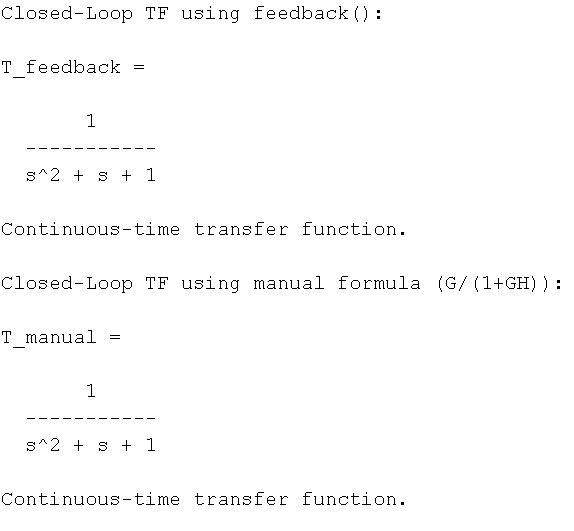
disp('Closed-Loop TF using feedback():');

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:

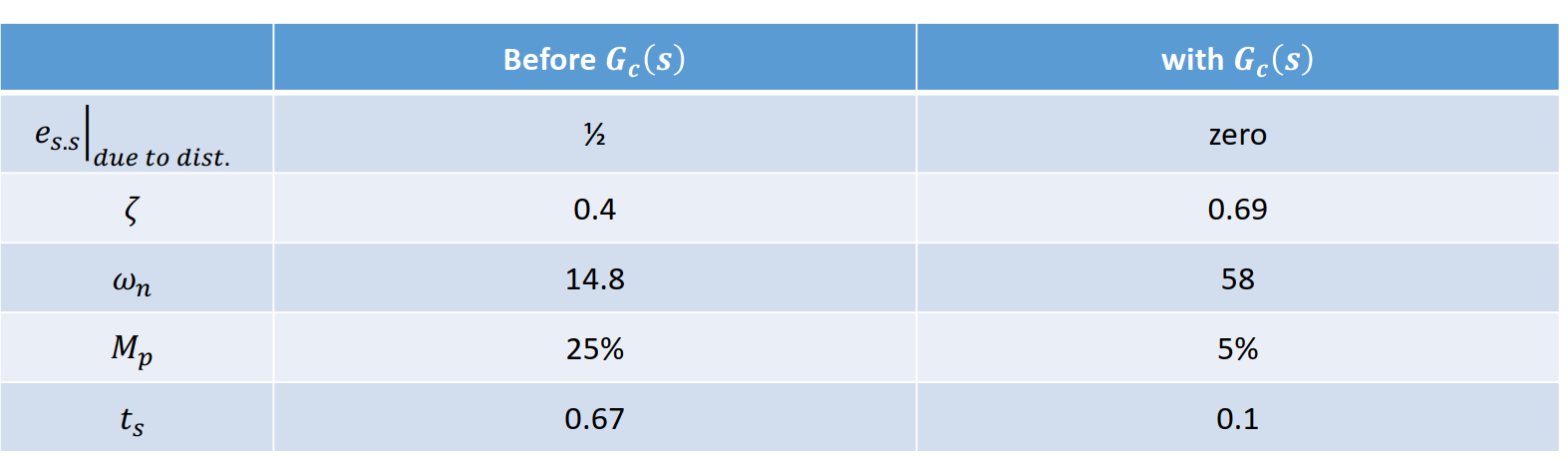
Q4)

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

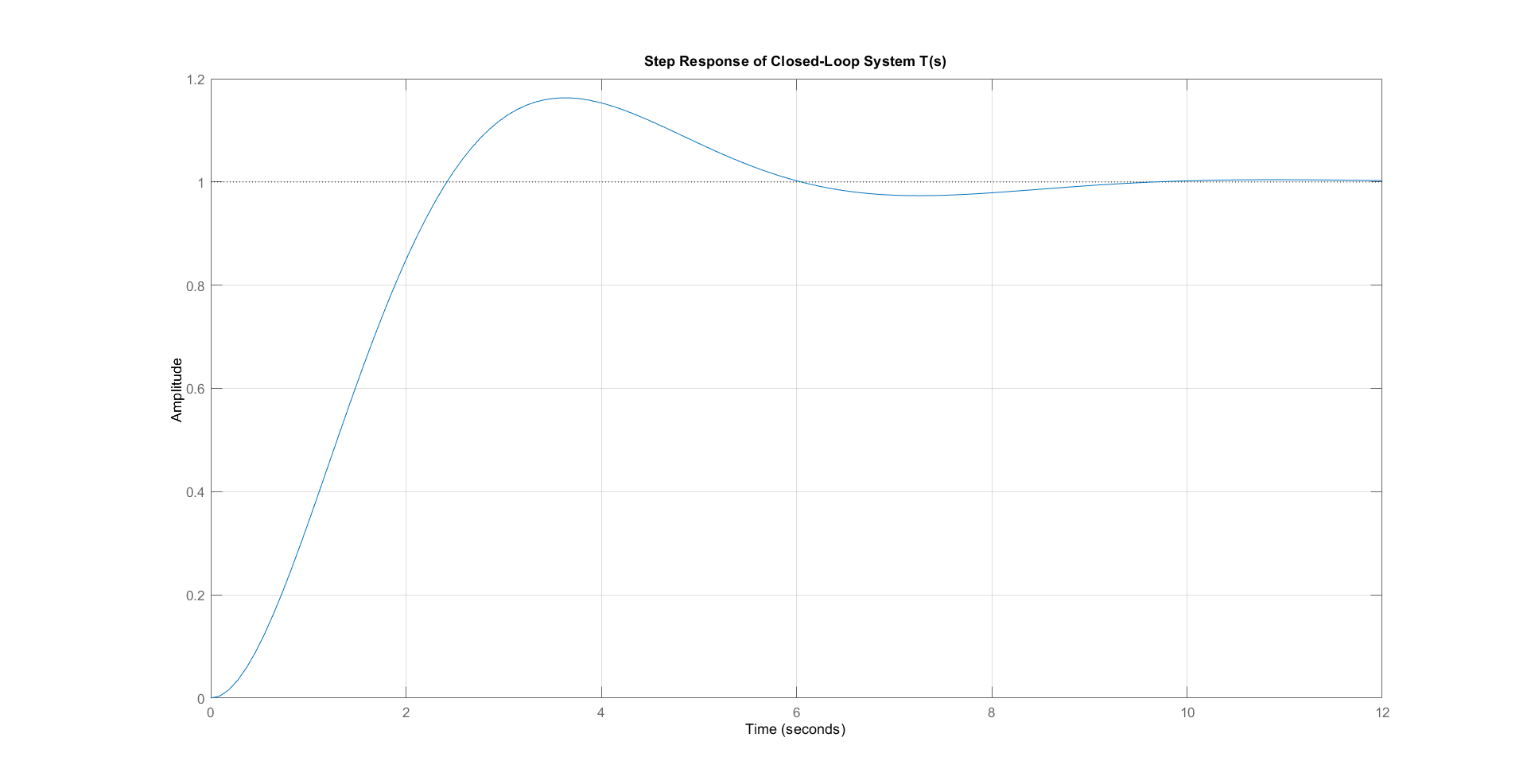
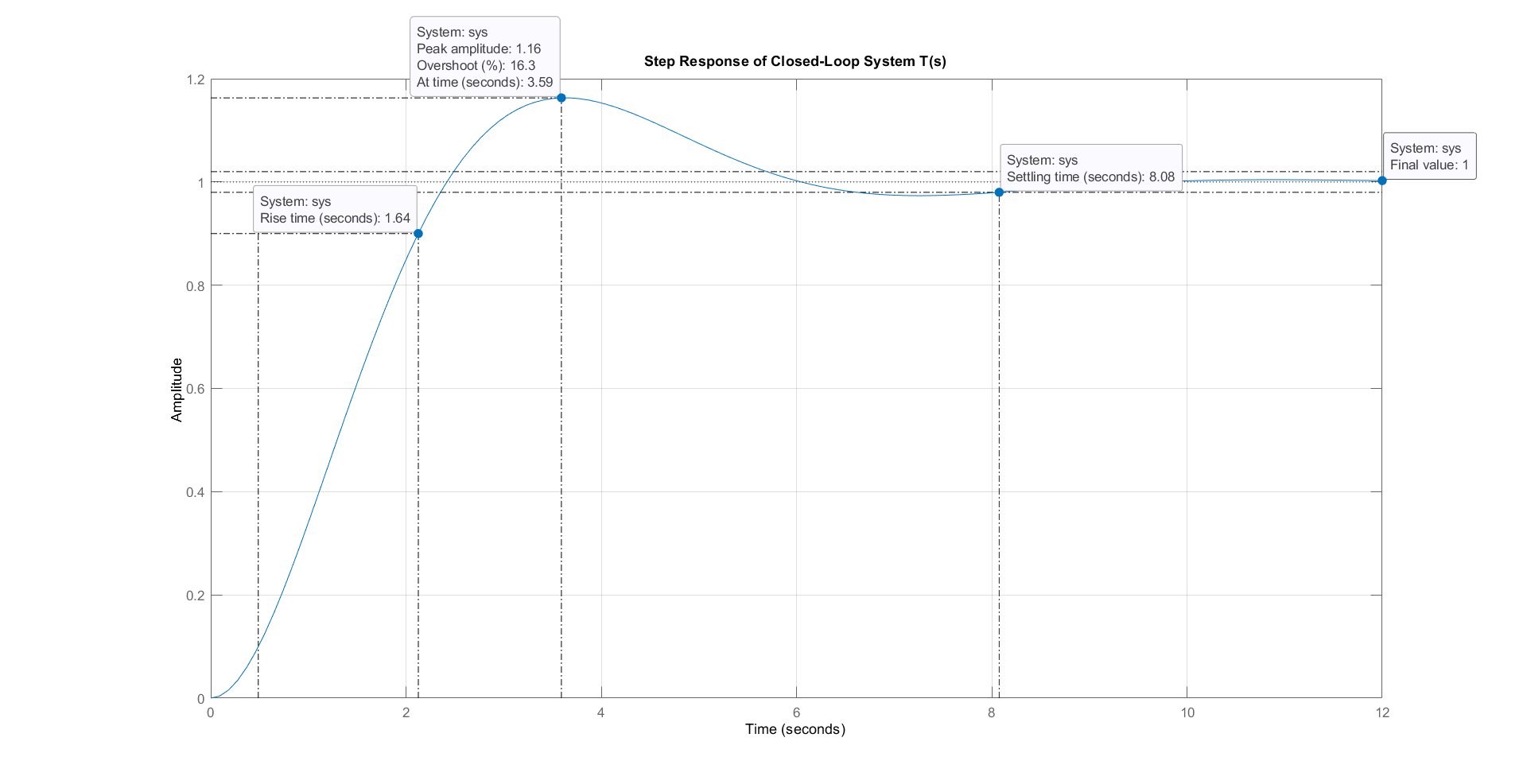
Damping ratio (ζ): 0.500

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)

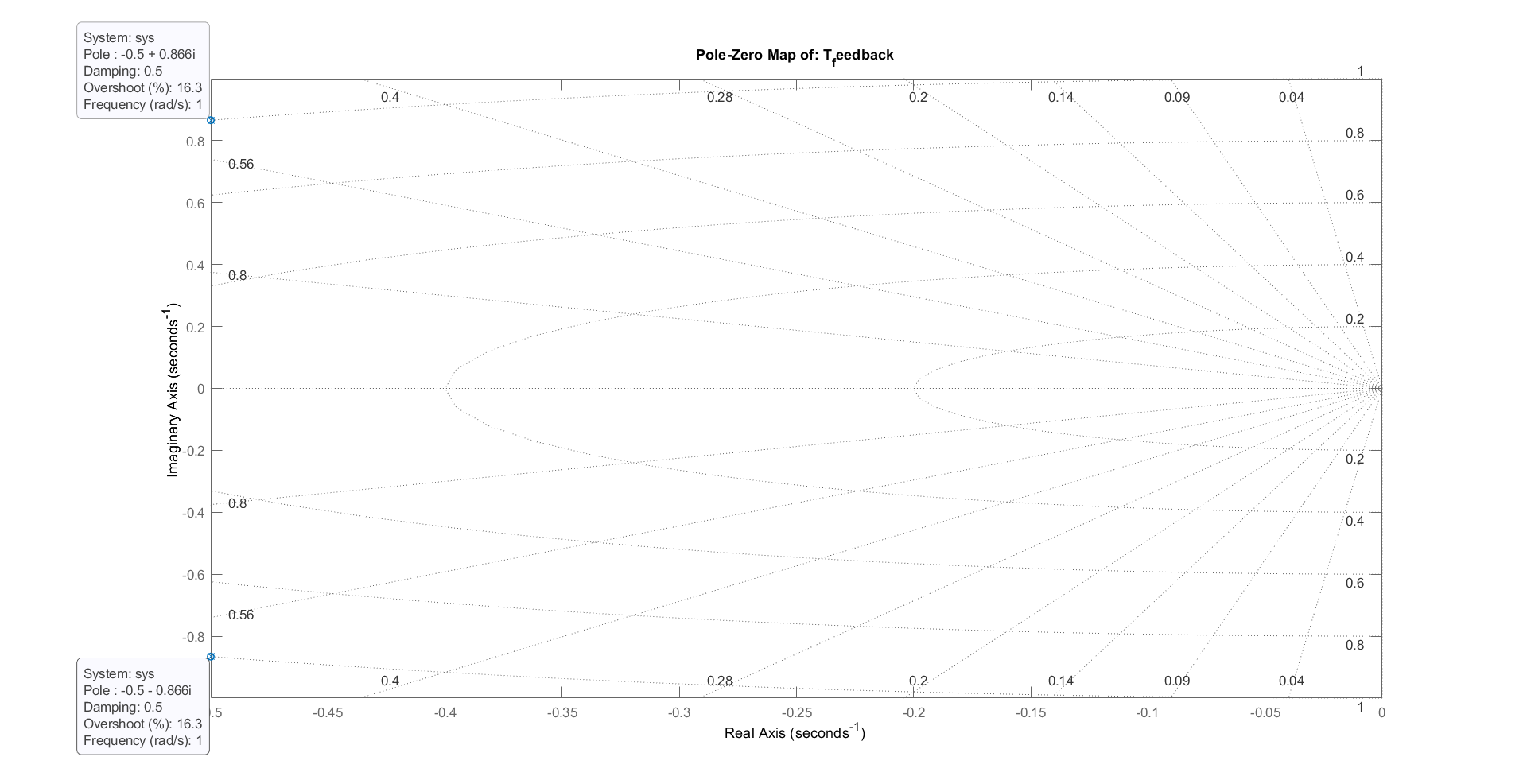
[wn, zeta] = damp(sys);

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');

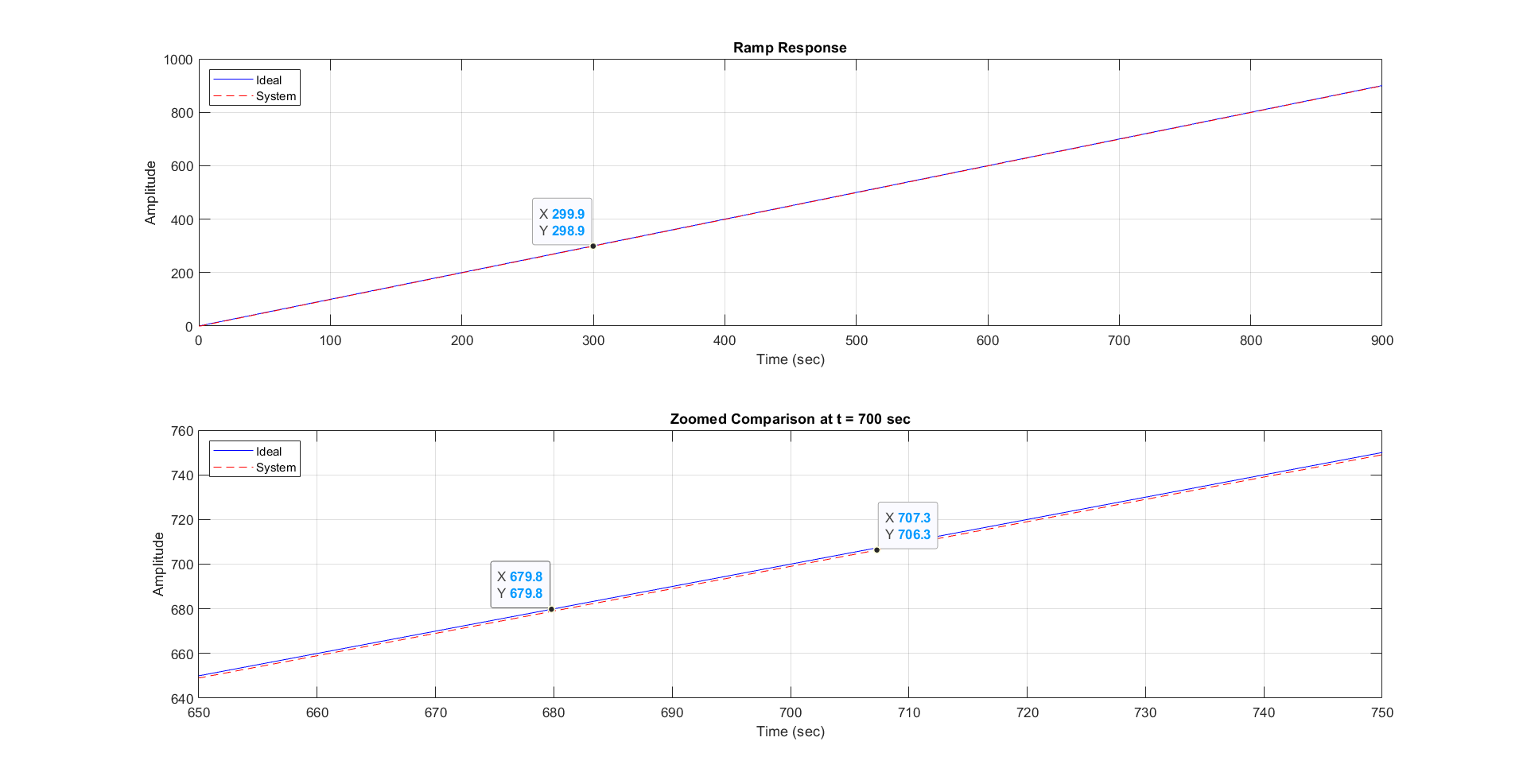
legend('Ideal', 'System', 'Location', 'northwest');

grid on;

hold off;

end

Output:



Steady-state error (ess): 1

Q9)

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

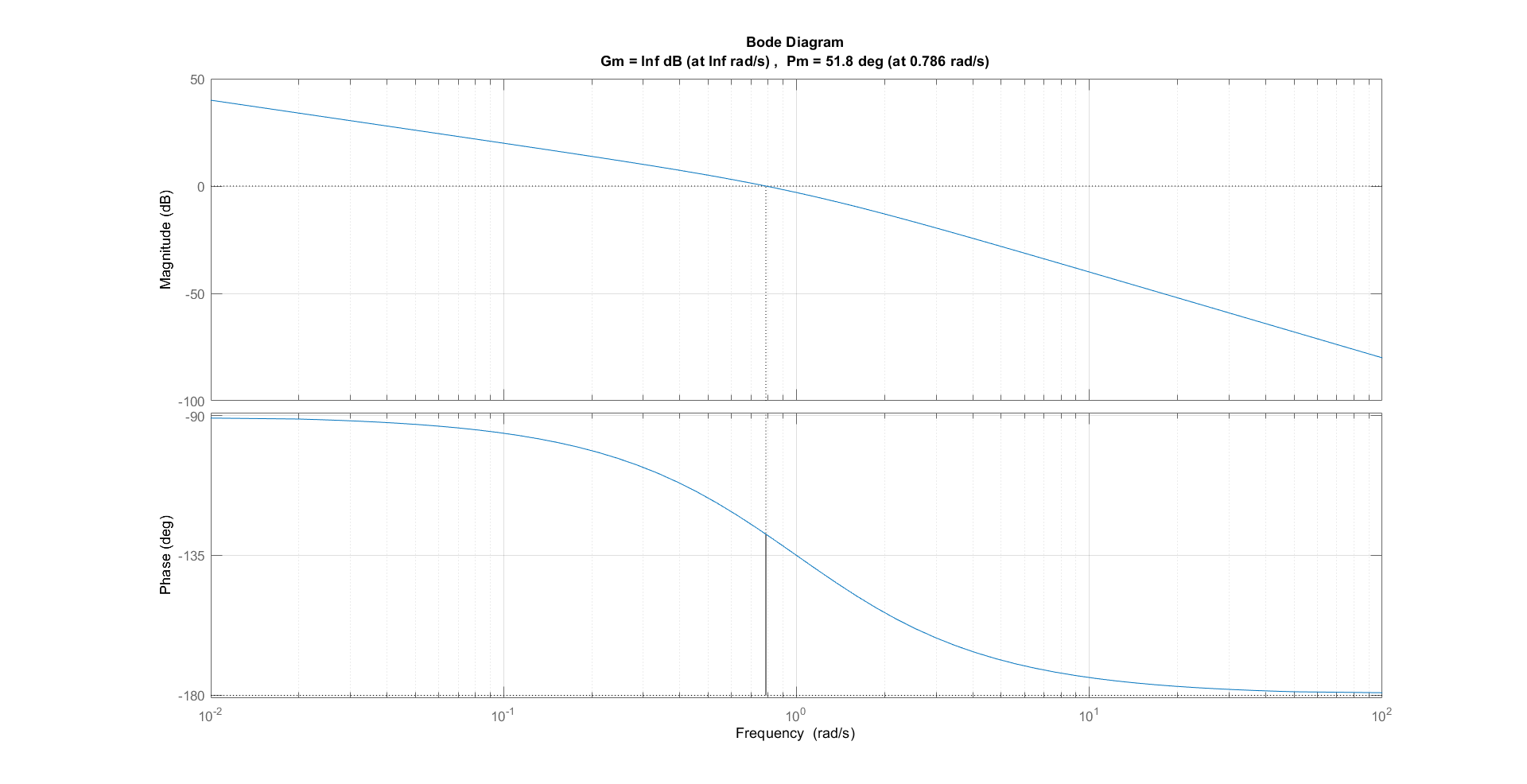
disp(['=== Stability Margins for ' inputname(1) ' ===']);

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

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:

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

State transition matrix in s-domain (Φ(s)):

/ s + 5 1 \

| ------------, ------------ |

| 2 2 |

| s + 5 s + 6 s + 5 s + 6 |

| |

| 6 s |

| - ------------, ------------ |

| 2 2 |

\ s + 5 s + 6 s + 5 s + 6 /

State transition matrix in time domain (Φ(t)):

/ exp(-2 t) 3 - exp(-3 t) 2, exp(-2 t) - exp(-3 t) \

| |

\ exp(-3 t) 6 - exp(-2 t) 6, exp(-3 t) 3 - exp(-2 t) 2 /

Verification of Φ(0) = I:

[1, 0]

[0, 1]

Q4)

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:

Time derivative of state transition matrix (Φ̇(t)):

/ exp(-3 t) 6 - exp(-2 t) 6, exp(-3 t) 3 - exp(-2 t) 2 \

| |

\ exp(-2 t) 12 - exp(-3 t) 18, exp(-2 t) 4 - exp(-3 t) 9 /

A\*Φ(t):

/ exp(-3 t) 6 - exp(-2 t) 6, exp(-3 t) 3 - exp(-2 t) 2 \

| |

\ exp(-2 t) 12 - exp(-3 t) 18, exp(-2 t) 4 - exp(-3 t) 9 /

Verification successful: Φ̇(t) = AΦ(t)

Q5)

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)

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(' ');

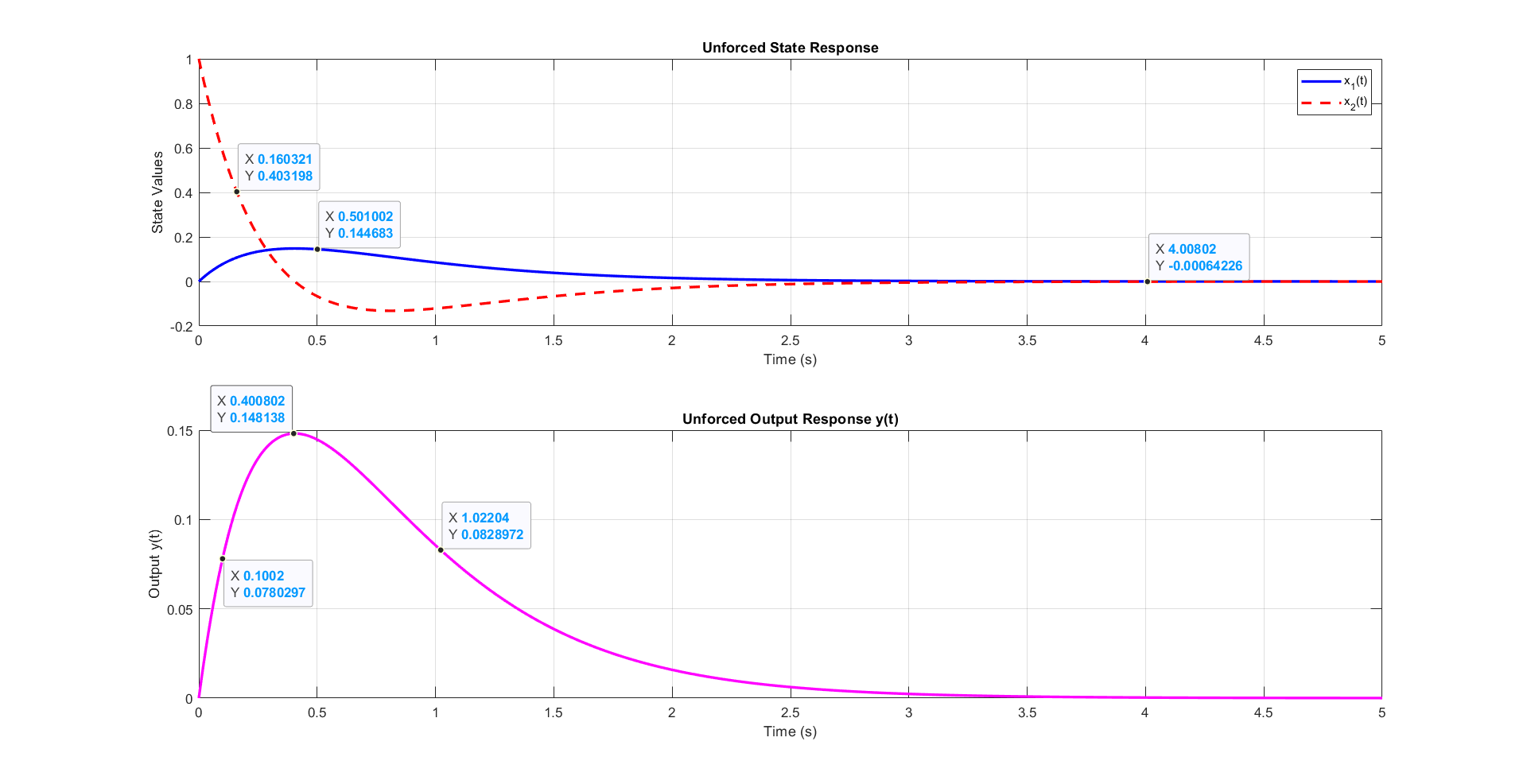
disp('Analytic Solutions:');

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

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

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

Output:



=== Unforced Response Analysis ===

Unforced state solution x(t):

/ exp(-2 t) - exp(-3 t) \

| |

\ exp(-3 t) 3 - exp(-2 t) 2 /

Unforced output response y(t):

exp(-2 t) - exp(-3 t)

Analytic Solutions:

x1(t) =

exp(-2 t) - exp(-3 t)

x2(t) =

exp(-3 t) 3 - exp(-2 t) 2

y(t) =

exp(-2 t) - exp(-3 t)

Q7)

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(' ');

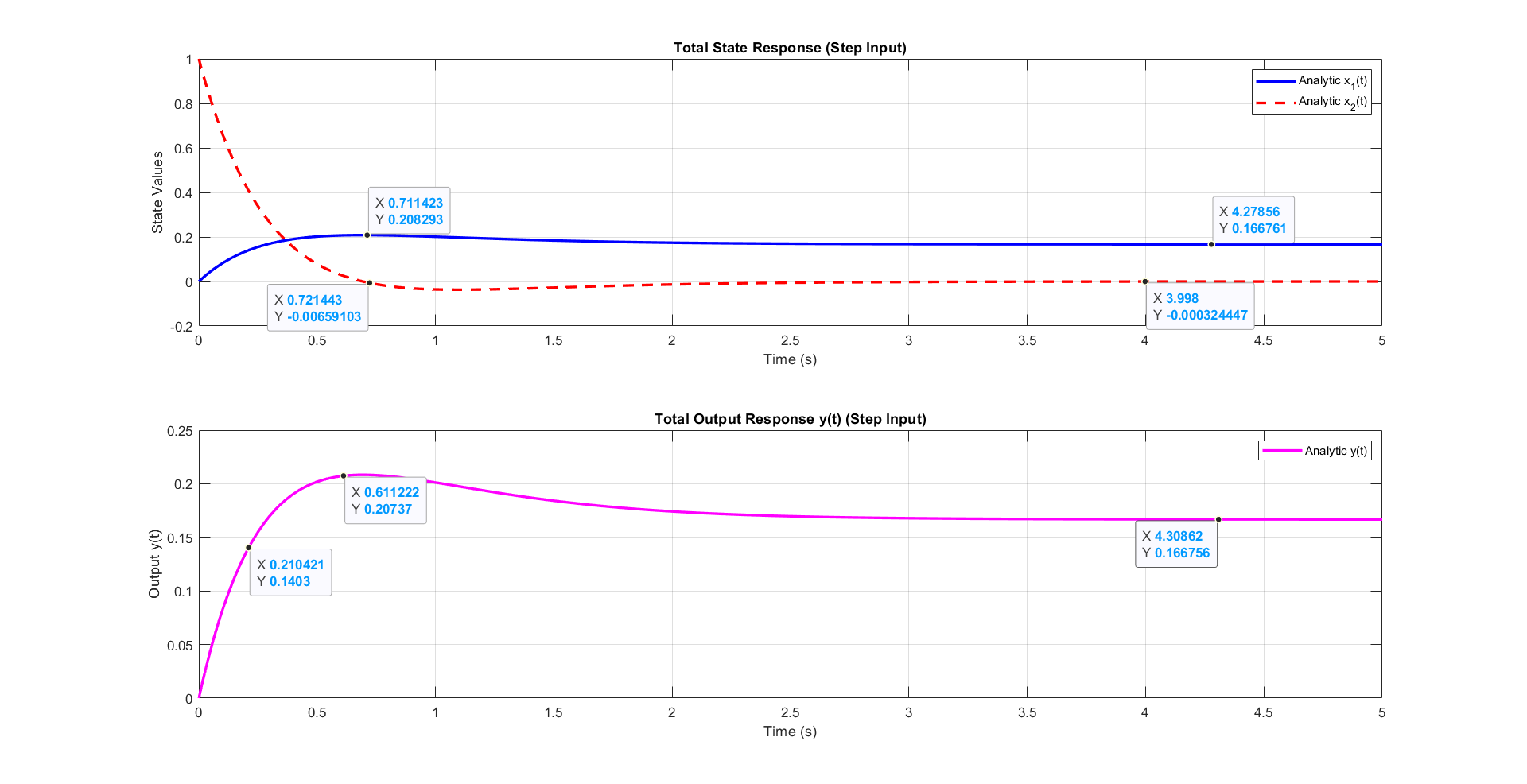
disp('Steady-State Values:');

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

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

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

Output:



=== Forced Response Analysis ===

Forced state solution (from step input):

/ exp(-3 t) exp(-2 t) 1 \

| --------- - --------- + - |

| 3 2 6 |

| |

\ exp(-2 t) - exp(-3 t) /

Total state solution (unforced + forced):

/ exp(-2 t) exp(-3 t) 2 1 \

| --------- - ----------- + - |

| 2 3 6 |

| |

\ exp(-3 t) 2 - exp(-2 t) /

Steady-State Values:

x1(∞) = 1/6

x2(∞) = 0

y(∞) = 1/6

Q8)

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');

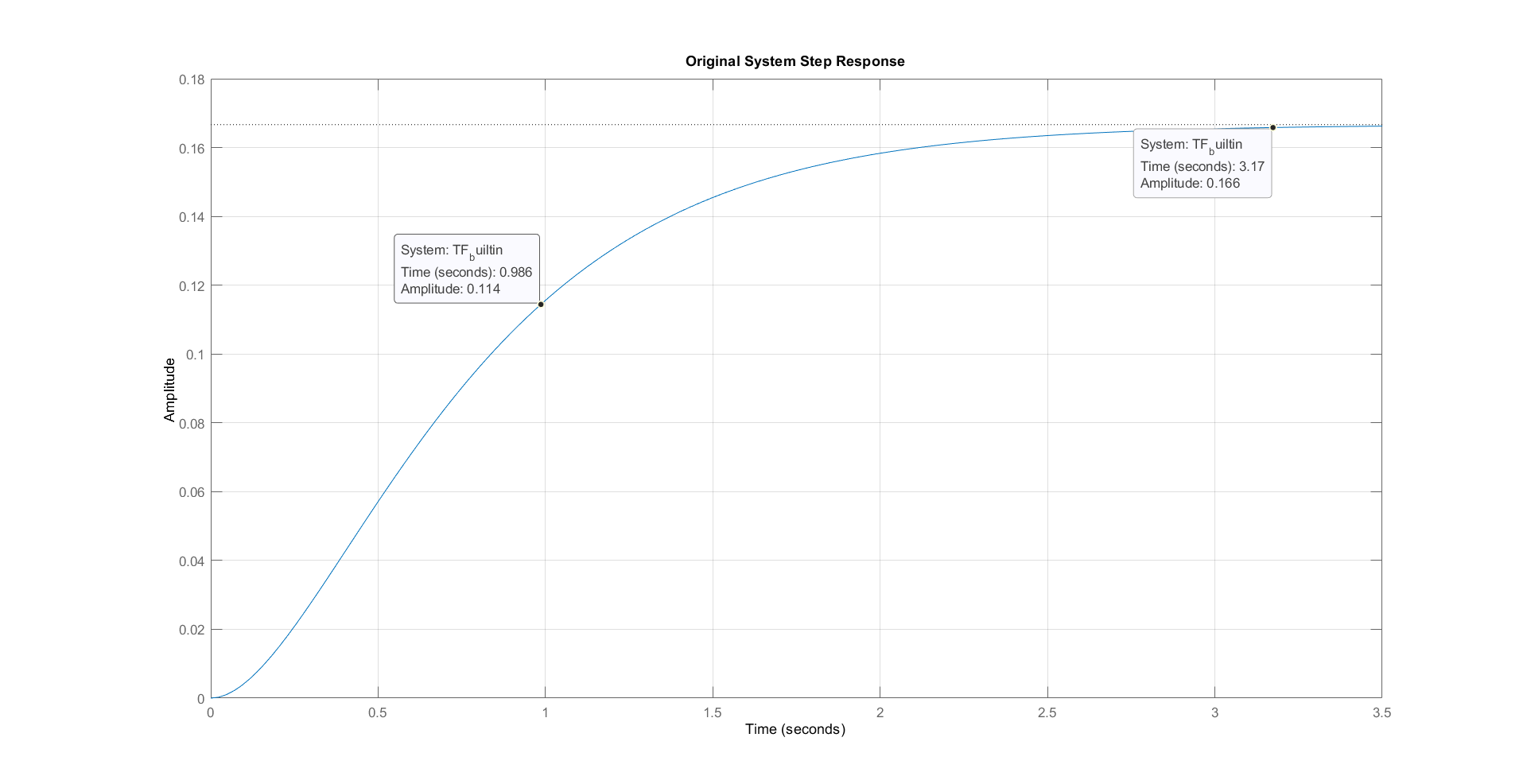
grid on;

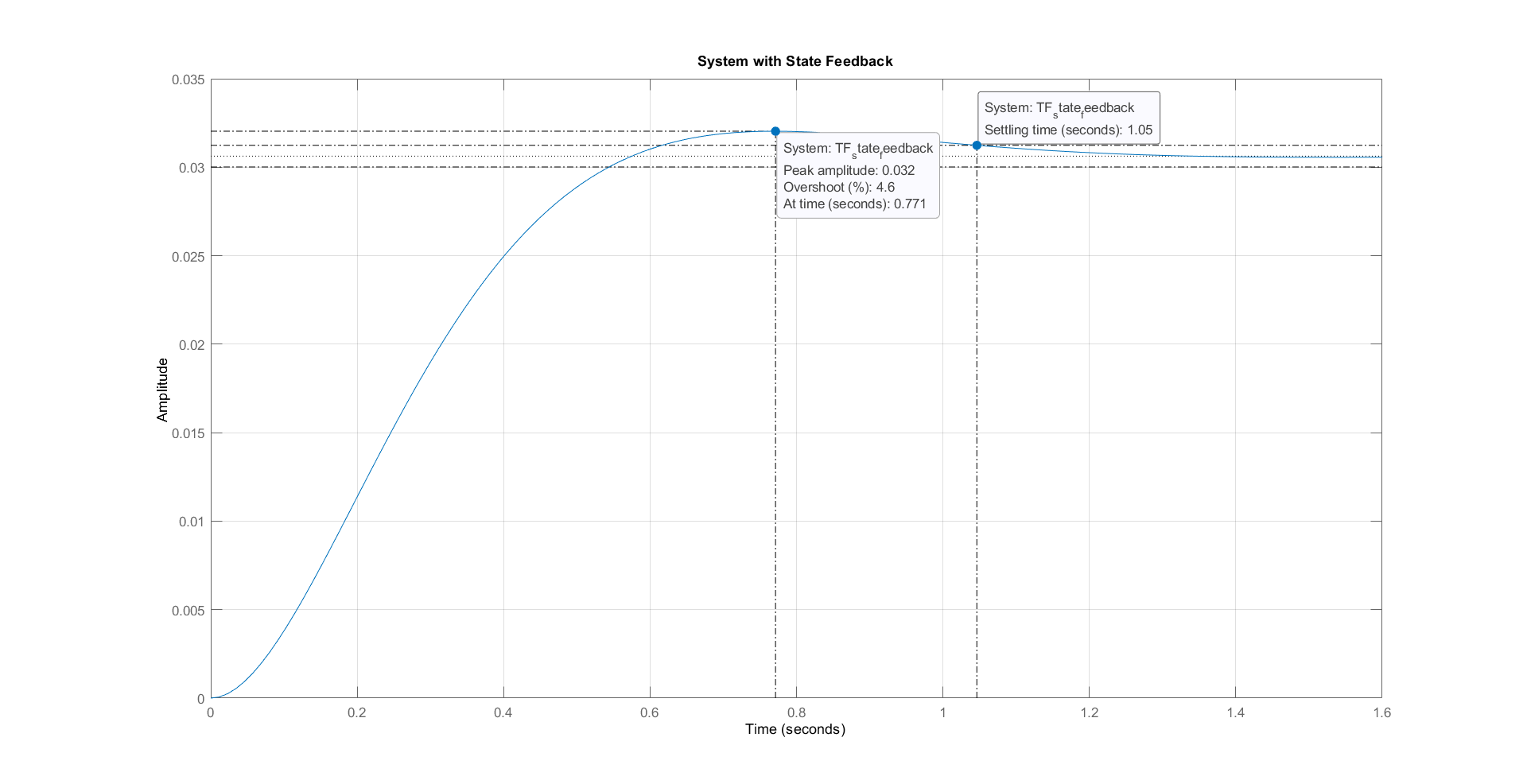
disp('Closed-loop system performance:');

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), '%']);