Part 1

Q1)

Theoritical

Step 1: Find the Transfer Function

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

$$H(s) = C(sI - A)^{-1}B + D$$

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

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:

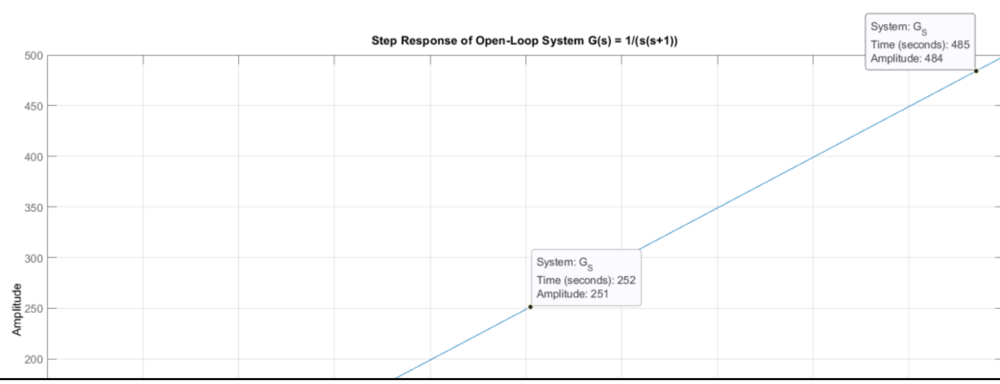
```
G_S =

 
      1
  -------
  s^2 + s
 
Continuous-time transfer function.


H_S =
 
  1
 
Static gain.
```

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:




```
Poles of Open-Loop System G(s):
     0
    -1

System is marginally stable (poles on imaginary axis)
```

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:

```
Closed-Loop TF using feedback():

T_feedback =

        1
  -----------
  s^2 + s + 1

Continuous-time transfer function.

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

T_manual =

        1
  -----------
  s^2 + s + 1

Continuous-time transfer function.
```

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:

	Before $G_c(s)$	with $G_c(s)$
$e_{s.s}\Big\|_{due\ to\ dist.}$	½	zero
ζ	0.4	0.69
ω_n	14.8	58
M_p	25%	5%
t_s	0.67	0.1

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

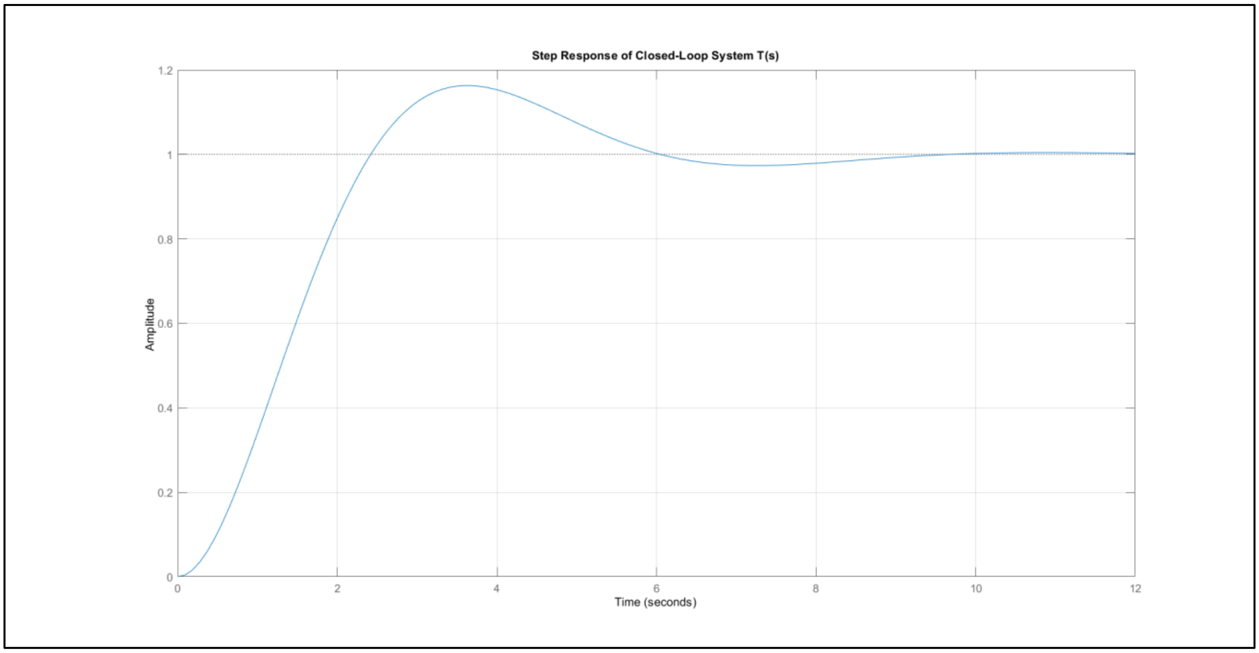
Damping ratio (ζ): 0.500

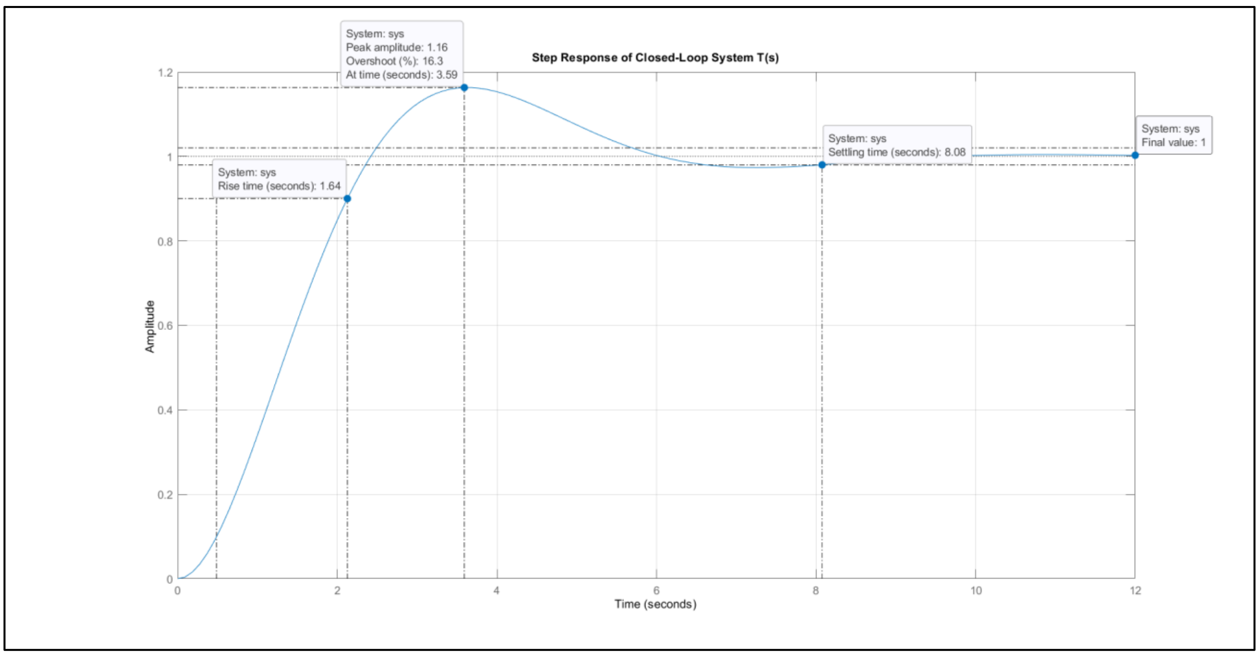
Natural frequency (ωn): 1.000 rad/s

Settling time (2%): 8.1051 sec

Rise time (10-90%): 1.6579 sec

Steady-state value: 1.0014


Step Response of Closed-Loop System T(s)
Amplitude
Time (seconds)
0
0.2
0.4
0.6
0.8
1
1.2
2
4
6
8
10
12

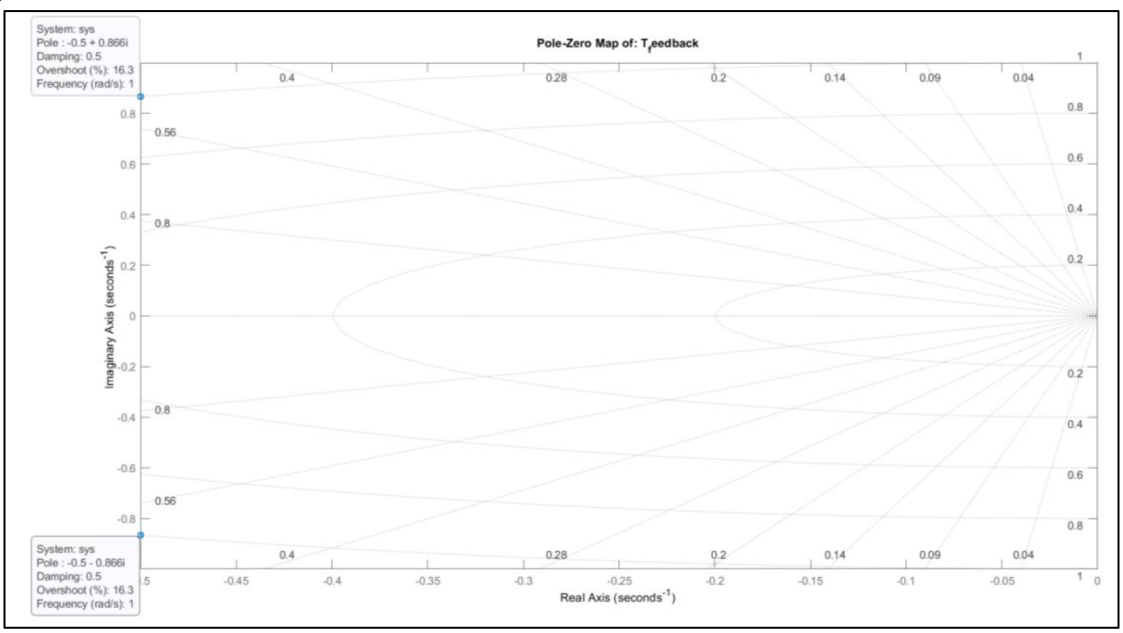

Step Response of Closed-Loop System T(s)
System: sys
Peak amplitude: 1.16
Overshoot (%): 16.3
At time (seconds): 3.59
System: sys
Rise time (seconds): 1.64
System: sys
Settling time (seconds): 8.08
System: sys
Final value: 1
Amplitude
Time (seconds)
0
0.2
0.4
0.6
0.8
1
1.2
2
4
6
8
10
12

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
```

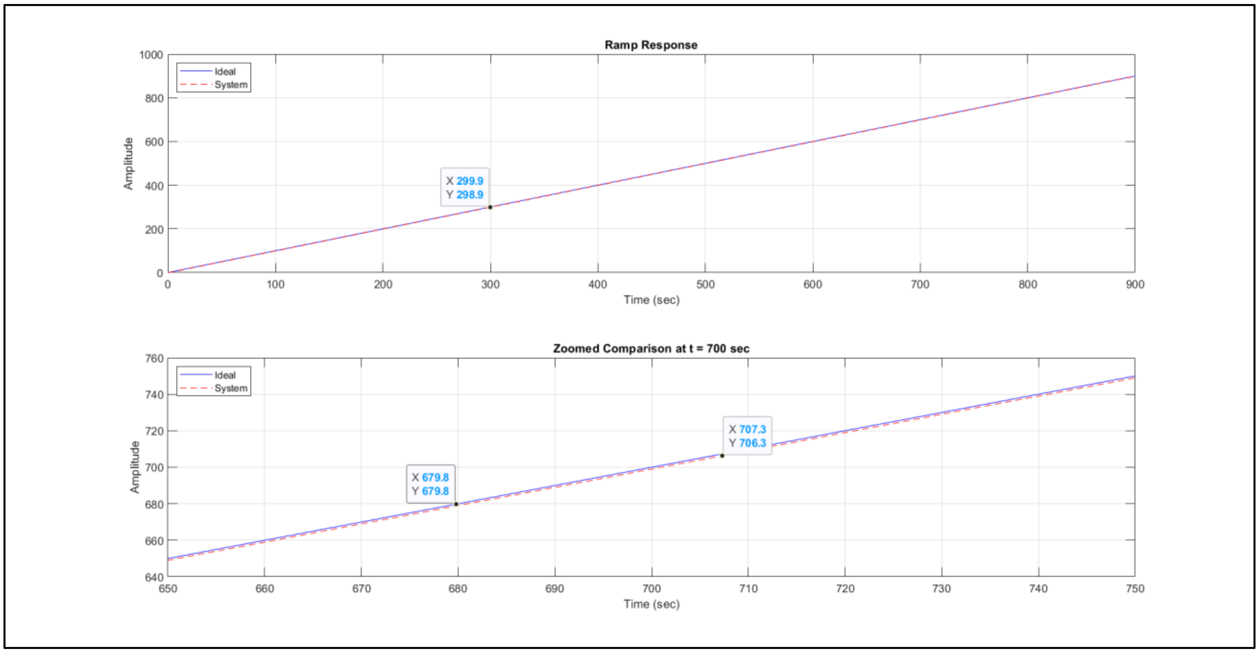
Q6,Q7 is done

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:



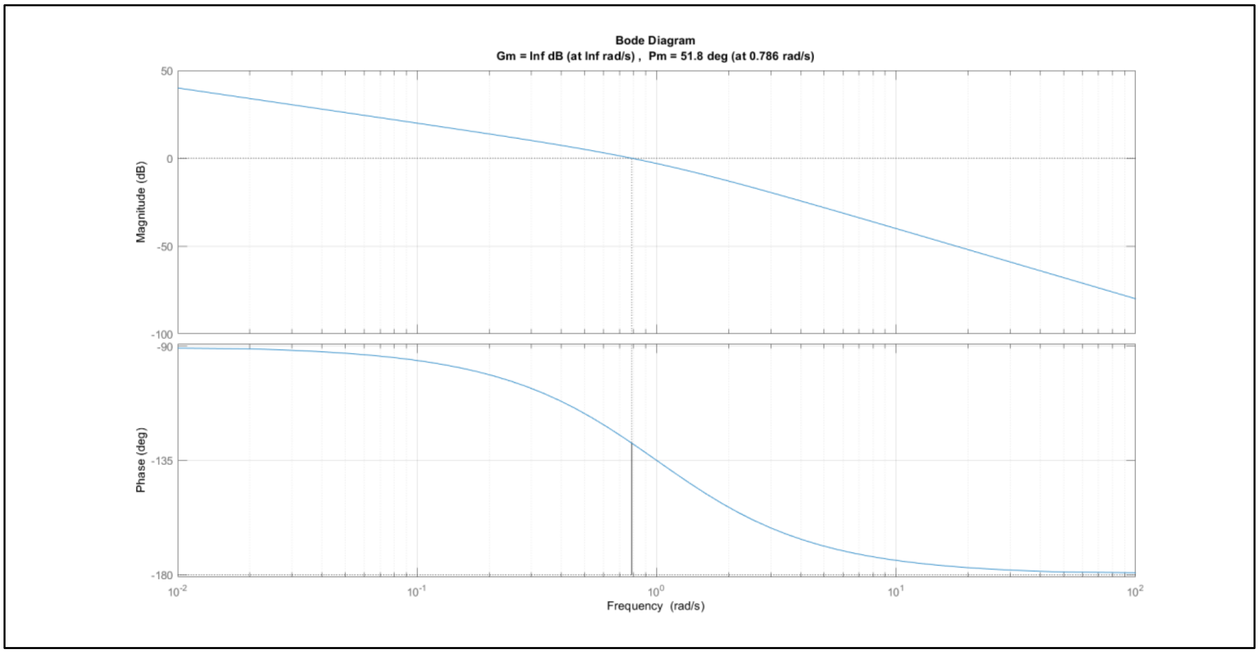
Steady-state error (ess): 1

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:

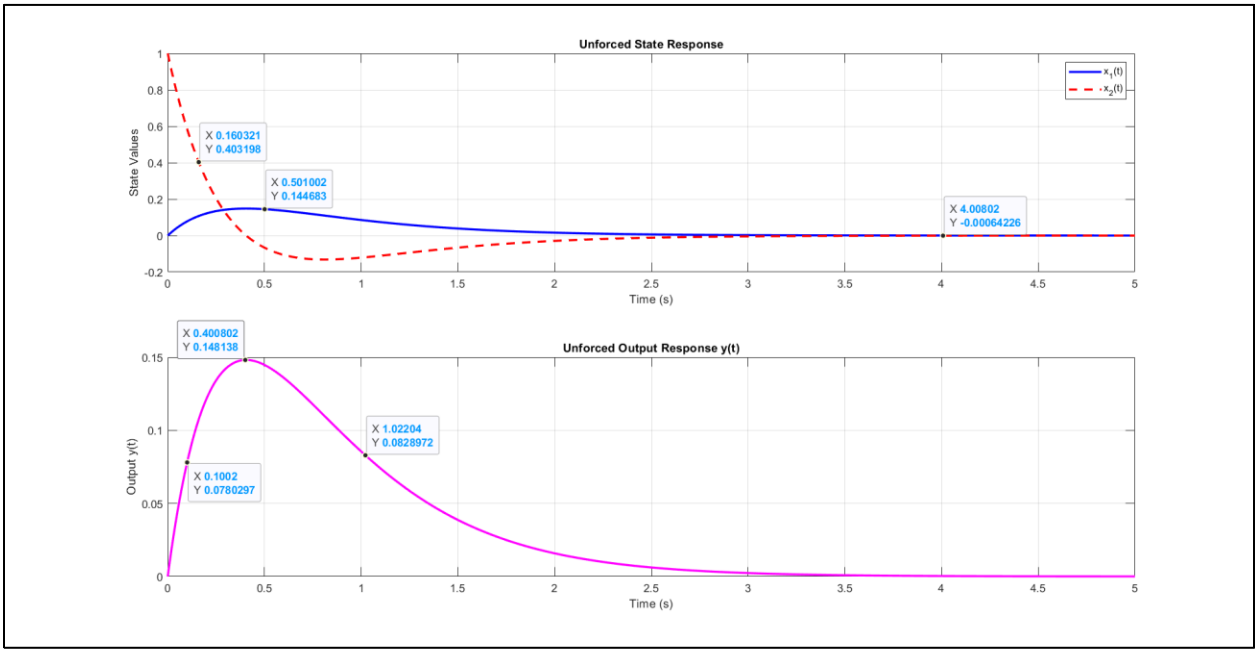
```
Controllability Matrix:
    0    1
    1   -5

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 State Response
State Values
$x_1(t)$
$x_2(t)$
X 0.160321
Y 0.403198
X 0.501002
Y 0.144683
X 4.00802
Y -0.00064226
Time (s)
Unforced Output Response y(t)
Output y(t)
X 0.400802
Y 0.148138
X 1.02204
Y 0.0828972
X 0.1002
Y 0.0780297
Time (s)

=== 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) =

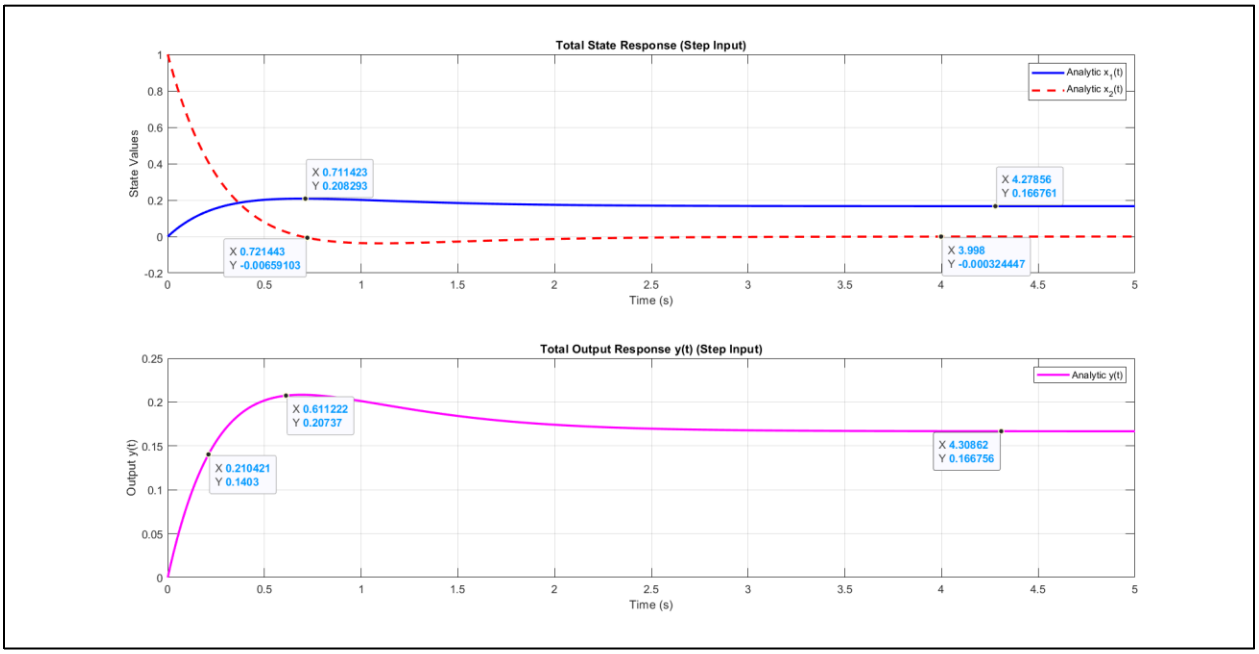
exp(-2 t) - exp(-3 t)

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:



Total State Response (Step Input)
State Values
Time (s)
Analytic $x_1(t)$
Analytic $x_2(t)$
X 0.711423
Y 0.208293
X 0.721443
Y -0.00659103
X 4.27856
Y 0.166761
X 3.998
Y -0.000324447
Total Output Response y(t) (Step Input)
Output y(t)
Time (s)
Analytic y(t)
X 0.611222
Y 0.20737
X 0.210421
Y 0.1403
X 4.30862
Y 0.166756

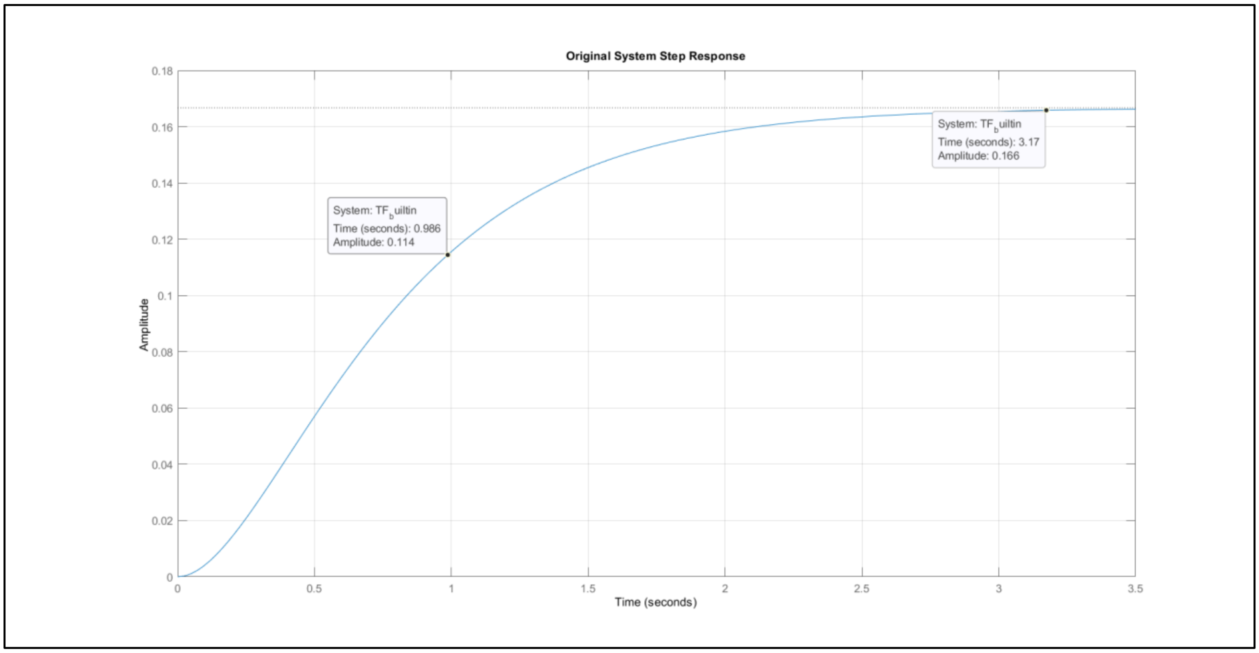
```
=== Forced Response Analysis ===
Forced state solution (from step input):
/ exp(-3 t)   exp(-2 t)   1 \
| --------- - --------- + - |
|     3           2       6 |
|                           |
\   exp(-2 t) - exp(-3 t)   /


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
```

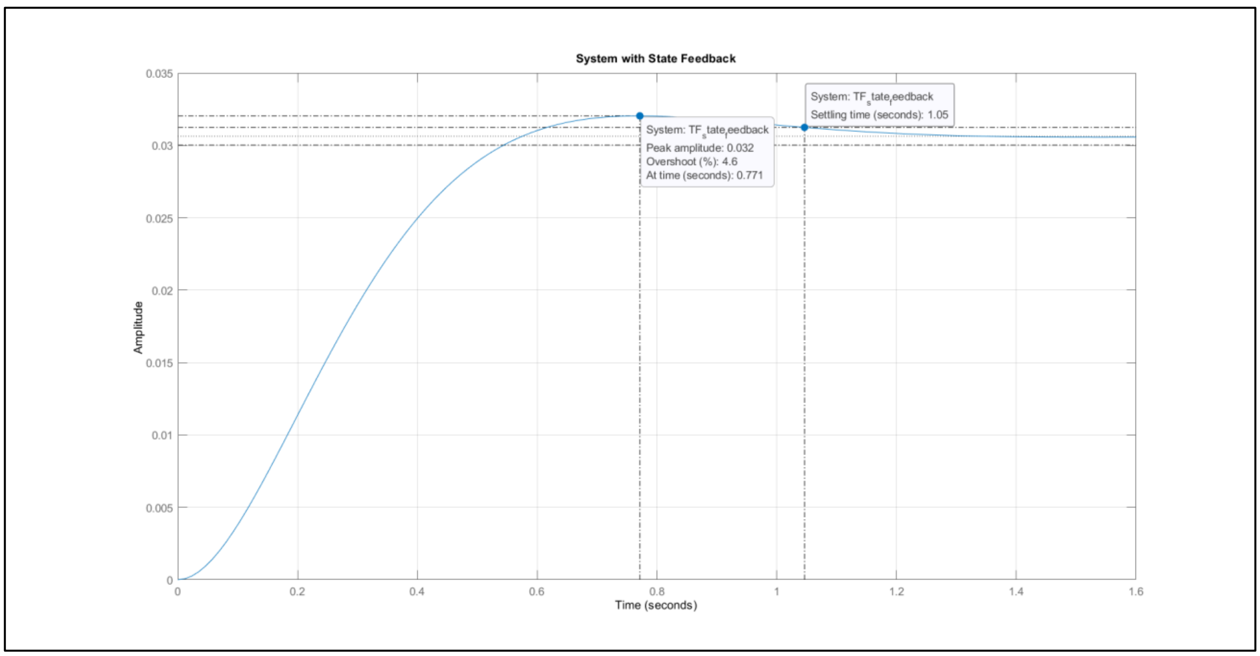
Q8)

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:


Original System Step Response
System: TF_builtin
Time (seconds): 0.986
Amplitude: 0.114
System: TF_builtin
Time (seconds): 3.17
Amplitude: 0.166
Amplitude
Time (seconds)
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0.5
1
1.5
2
2.5
3
3.5


System with State Feedback
System: TF_state_feedback
Settling time (seconds): 1.05
System: TF_state_feedback
Peak amplitude: 0.032
Overshoot (%): 4.6
At time (seconds): 0.771
Amplitude
Time (seconds)
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6

```
=== 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), '%']);
```