K.N. Toosi University

Aidin Sahneh(40120243)

Professor: Dr. Taghirad

Course: Linear Control Systems

1 Introduction

In this report, we aim to systematically progress through the project step by step and overcome its challenges.

2 Q1

Using the given data and the following code, we plot the Bode diagram of the system.

MATLAB Code

```matlab
omega = Data.omega;
magnitude = Data.magnitude;
phase = Data.phase;

figure;

subplot(2,1,1);
semilogx(omega, 20*log10(magnitude), 'b', 'LineWidth', 1.5);
grid on;
xlabel('Frequency (rad/s)');
ylabel('Magnitude (dB)');
title('Bode Plot - Magnitude Response');

subplot(2,1,2);
semilogx(omega, phase, 'r', 'LineWidth', 1.5);
grid on;
xlabel('Frequency (rad/s)');
ylabel('Phase (degrees)');
title('Bode Plot - Phase Response');
```

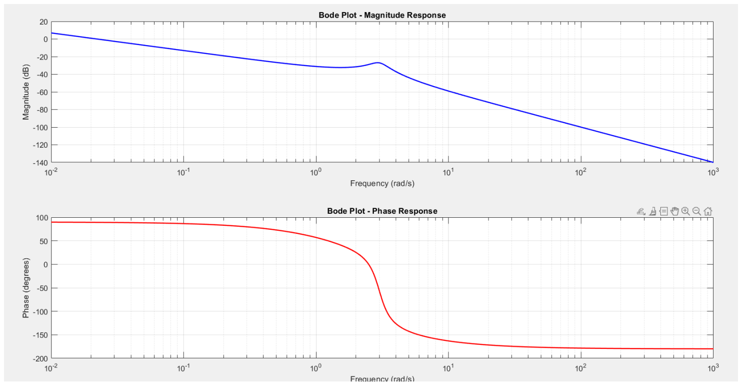


Figure 1: Bode Plot Output

3 Q2

As observed, the magnitude of the Bode plot starts with a slope of -20, indicating that the system is of type one. According to the formula:

$$20 \log \left(\frac{k}{\omega^p} \right)$$

where the initial magnitude of the plot represents the system gain, we can determine k as:

$$k = 0.022$$

Furthermore, there is no noticeable delay in the system. Systems with delay exhibit a continuously decreasing phase, whereas here, the phase converges to a fixed value, confirming the absence of delay.

Next, we aim to determine the system's transfer function. Observing the magnitude plot at a frequency of 1, we notice a 20 dB increase in slope, indicating the presence of a zero.

To determine whether this zero is minimum-phase or non-minimum-phase, we analyze the phase plot. At a frequency of 0.1, the slope decreases by 45 degrees, whereas in a minimum-phase system, it should increase by 45 degrees. Similarly, at a frequency of 10, the slope increases by 45 degrees, whereas in a minimum-phase system, it should decrease by 45 degrees. This confirms that the zero is non-minimum-phase, and thus the system is non-minimum-phase.

Thus, the numerator contains $s - 1$.

Examining the magnitude plot further, we observe an overshoot at approximately 2.93 rad/s, indicating the presence of a second-order factor in the system. The standard second-order factor is given by:

$$\frac{\omega_n^2}{s^2 + 2\eta\omega_n s + \omega_n^2}$$

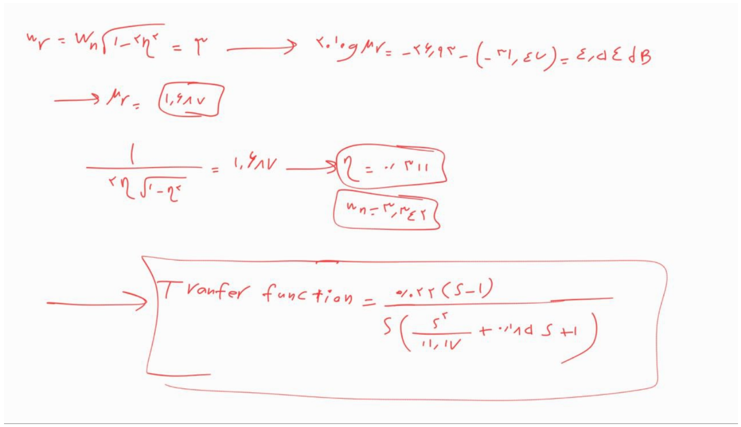
Using the resonance peak, the resonance frequency, and the following formulas:

$$\mu_r = \frac{1}{2\eta\sqrt{1 - \eta^2}}$$

$$\omega_r = \omega_n \sqrt{1 - 2\eta^2}$$

we can compute ω_n and η.

The manual calculations are shown below, where we successfully derive the system's transfer function:



$$\omega_r = \omega_n\sqrt{1-۲\eta^۲} = ۳ \longrightarrow ۲۰\log M_r = -۲۶٫۹۳ - (-۳۱٫۴۷) = ۴٫۵۴\,dB$$

$$\longrightarrow M_r = \boxed{۱٫۶۸۷}$$

$$\frac{۱}{۲\eta\sqrt{۱-\eta^۲}} = ۱٫۶۸۷ \longrightarrow \boxed{\eta = ۰٫۳۱۱}$$

$$\boxed{\omega_n = ۳٫۳۴۲}$$

$$\longrightarrow \boxed{\text{Tranfer function} = \frac{۰٫۰۲۲(S-1)}{S\left(\frac{S^۲}{۱۱٫۱۷} + ۰٫۱۸۵\,S + ۱\right)}}$$

Figure 2: Manual Calculations

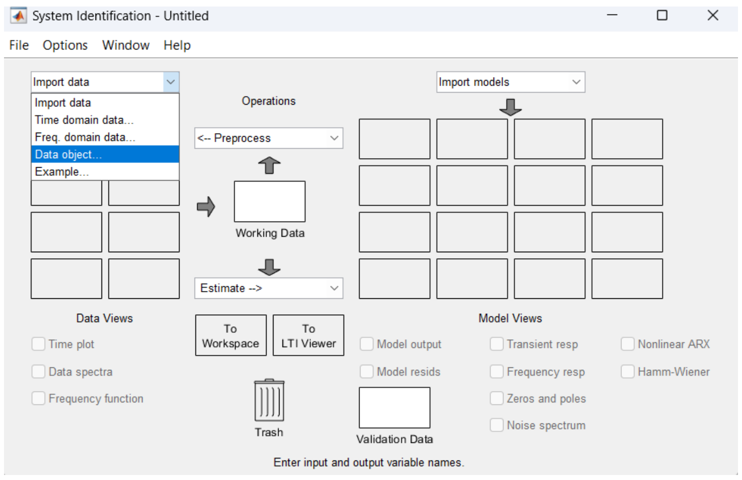
As a result, the system has an order of 3, is of type 1, has no delay, and is non-minimum-phase.

4 Q3

To fit an appropriate transfer function using the frequency response, we use the following code and then follow the steps shown in the image:

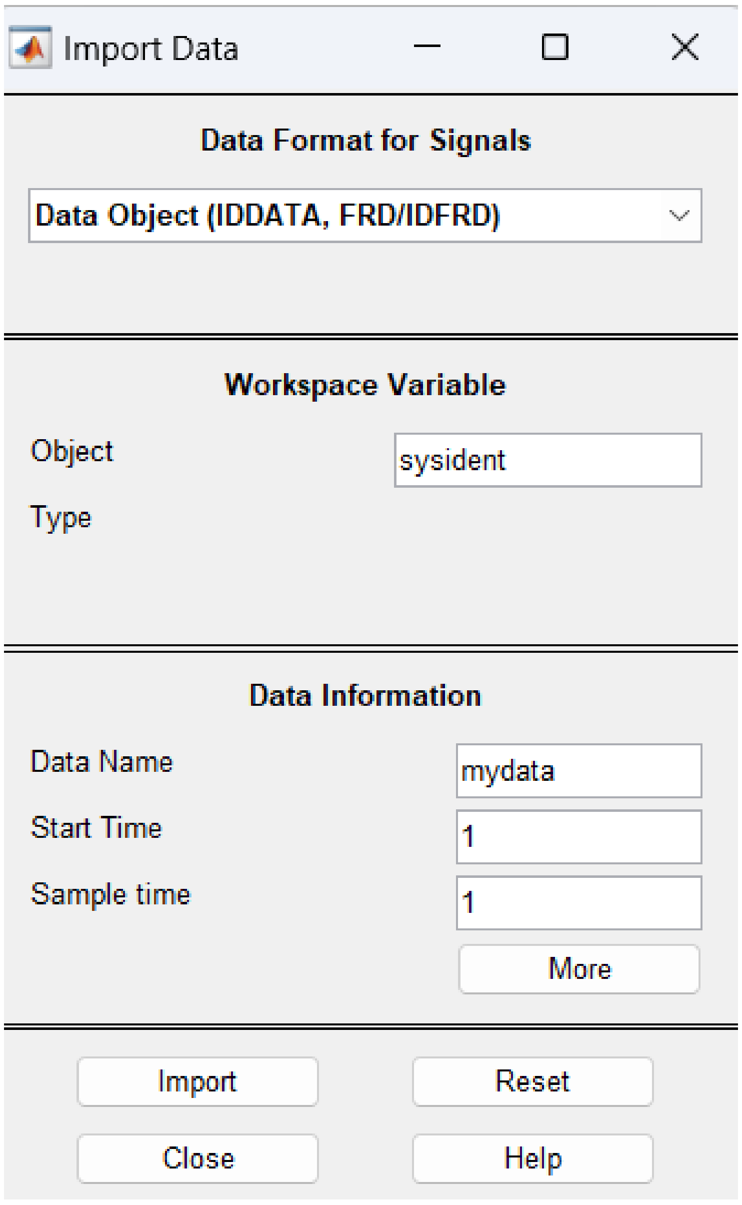
MATLAB Code

```matlab
        load('Data.mat');

        omega = Data.omega;
        magnitude = Data.magnitude;
        phase = Data.phase;

        phase_rad = deg2rad(phase);

        frequency_response_data = magnitude .* exp(1i * phase_rad
            );

        sysident = frd(frequency_response_data, omega, '
            FrequencyUnit', 'rad/s');

        systemIdentification
```



Enter input and output variable names.

Figure 3: Step 1 - System Identification


Import Data

Data Format for Signals

Data Object (IDDATA, FRD/IDFRD)

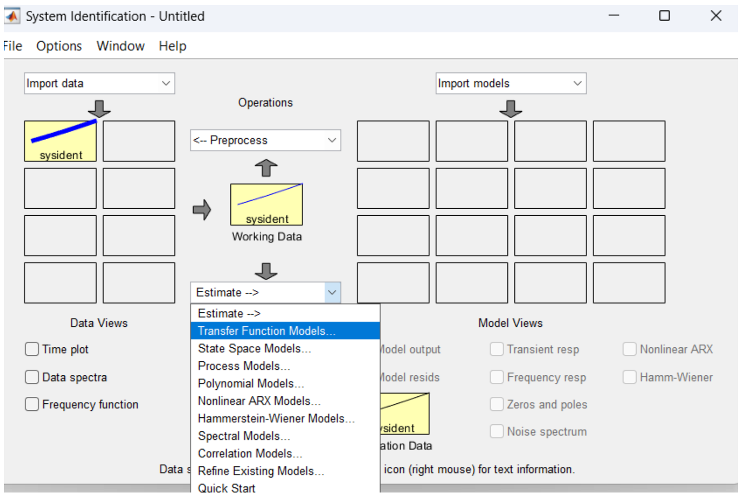
Workspace Variable

Object	sysident
Type	

Data Information

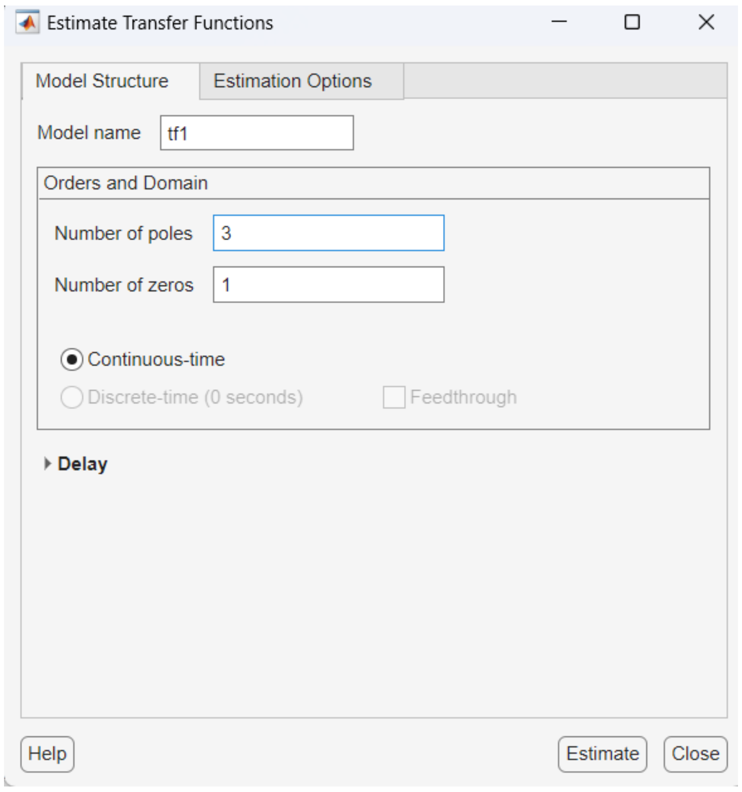
Data Name	mydata
Start Time	1
Sample time	1

Figure 4: Step 2 - Parameter Estimation



Estimate -->
[✓ Transfer Function Models...]
State Space Models...
Process Models...
Polynomial Models...
Nonlinear ARX Models...
Hammerstein-Wiener Models...
Spectral Models...
Correlation Models...
Refine Existing Models...
Quick Start

Figure 5: Step 3 - Model Fitting



Model name: tf1

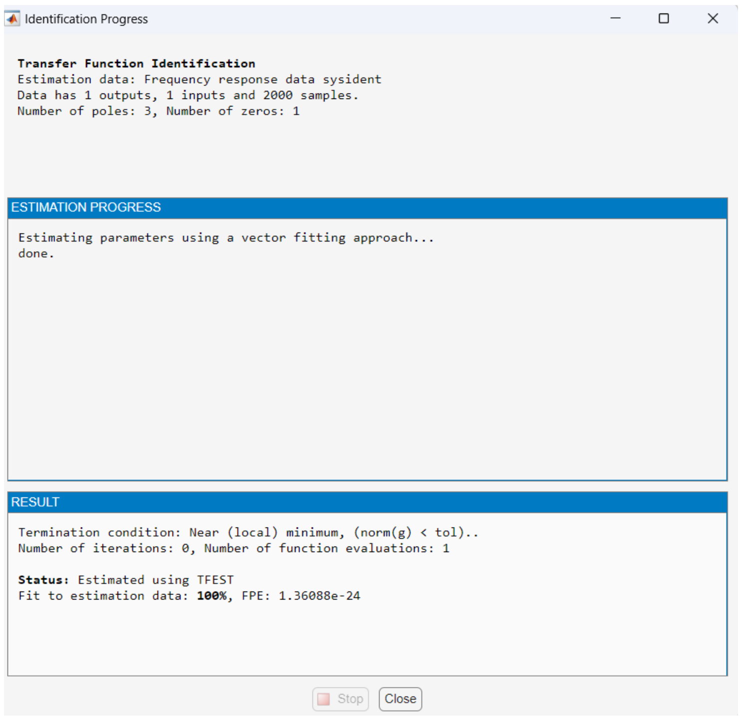
Orders and Domain

Number of poles: 3

Number of zeros: 1

[✓ Continuous-time] Discrete-time (0 seconds) Feedthrough

Figure 6: Step 4 - Validation

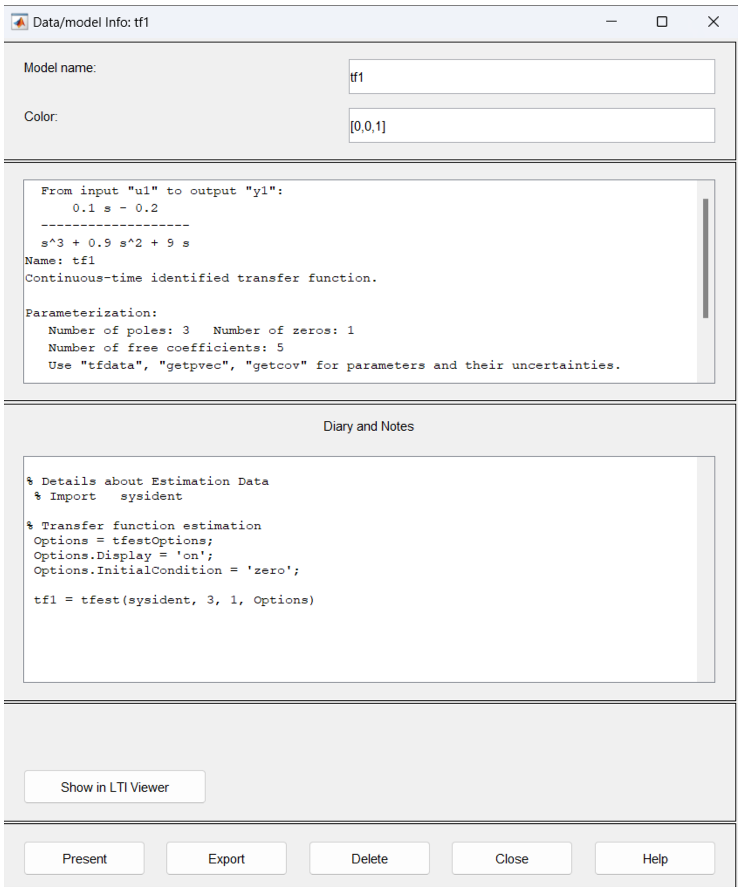


```
Transfer Function Identification
Estimation data: Frequency response data sysident
Data has 1 outputs, 1 inputs and 2000 samples.
Number of poles: 3, Number of zeros: 1
```

```
Estimating parameters using a vector fitting approach...
done.
```

```
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 0, Number of function evaluations: 1

Status: Estimated using TFEST
Fit to estimation data: 100%, FPE: 1.36088e-24
```

Figure 7: Step 5 - Comparison with Experimental Data



Model name: tf1

Color: [0,0,1]

```
  From input "u1" to output "y1":
      0.1 s - 0.2
  -------------------
  s^3 + 0.9 s^2 + 9 s
Name: tf1
Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 1
   Number of free coefficients: 5
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.
```

```
% Details about Estimation Data
 % Import   sysident

% Transfer function estimation
 Options = tfestOptions;
 Options.Display = 'on';
 Options.InitialCondition = 'zero';

 tf1 = tfest(sysident, 3, 1, Options)
```

Figure 8: Step 6 - Final Transfer Function

As we can see, the obtained transfer function is given by:

$$G(s) = \frac{0.1s - 0.2}{s^3 + 0.9s^2 + 9s}$$

5 Q4

For stability analysis using the Routh-Hurwitz method, we first need to determine the characteristic equation:

$$\Delta(s) = 1 + KG(s)$$

Substituting:

$$G(s) = \frac{0.1s - 0.2}{s^3 + 0.9s^2 + 9s}$$

into the equation and simplifying, we obtain:

$$\Delta(s) = s^3 + 0.9s^2 + (0.1K + 9)s - 0.2K$$

$$\begin{array}{c|cc} s^3 & 1 & (9+0.1k) \\ s^2 & 0.9 & -0.2k \\ s^1 & B & 0 \\ s^0 & -0.2k & 0 \end{array}$$

In this table:

$$B = \frac{0.9(9+0.1k) - (-0.2k)}{0.9}$$

By simplifying:

$$B = \frac{8.1 + 0.09k + 0.2k}{0.9} = \frac{8.1 + 0.29k}{0.9}$$

Stability Analysis

To ensure stability, all elements in the first column of the Routh array must be positive:

1. $-0.2k > 0 \Rightarrow k < 0$
2. $B > 0 \Rightarrow k > -27.93$

Thus, the stability condition for the system is:

$$-27.93 < k < 0$$

6 Q5

In this section, we first plot the root locus in MATLAB using the following code:

MATLAB Code

```
num = [0.1 -0.2];
den = [1 0.9 9 0];
figure;
rlocus(tf(num, den));
title('Root Locust');
```

And we obtain the following root locus:

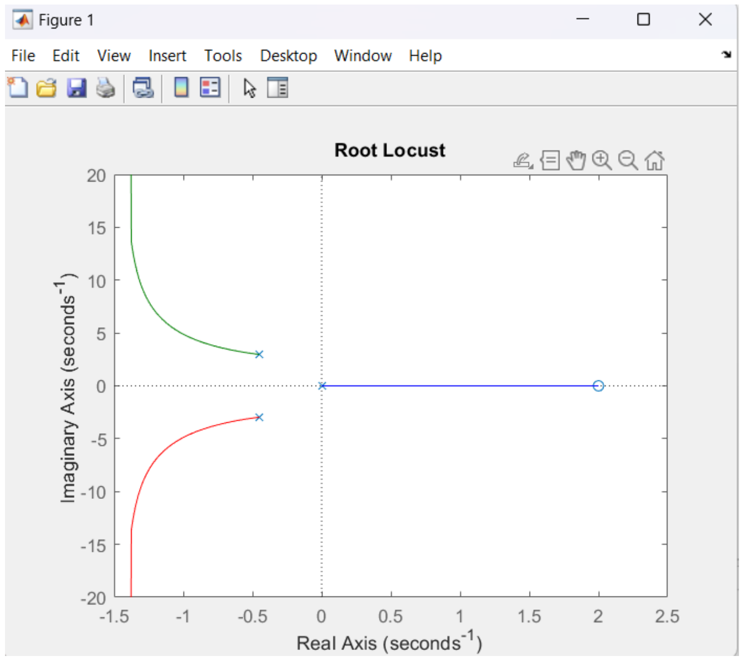


Figure 9: Step 1 - Root Locust

6.1 Stability with Proportional Controller

As observed in the root locus plot, two branches are located on the left side of the axis, while one branch is on the right side. The right-side branch is responsible for instability. Therefore, we need to either eliminate its effect or shift it towards the left side.

Using a proportional controller alone cannot stabilize the system because this type of controller only affects the gain and does not alter the poles, zeros, or the root locus plot.

As a result, the system remains unstable, and this controller is not suitable.

Stability Analysis for PI and PD Controllers

6.2 PI Controller Analysis

For $K > 0$: The PI controller is given by:

$$C(s) = K_p + \frac{K_i}{s}$$

By adding this controller, a pole at the origin and a zero are introduced to the system. While this controller may help in reducing the steady-state error, it does not positively impact the root locus and system stability. This is because, for $K > 0$, the pole continues to move towards the right, leading to an unstable pole in the right-half plane.

For $K < 0$: The controller takes the form:

$$C(s) = -(K_p + \frac{K_i}{s})$$

This inversion causes the controller's effect to be reversed, meaning that instead of moving towards the right, the poles shift towards the left-half plane of the s-axis. Consequently, the system becomes stable within certain ranges.

We can verify this behavior in MATLAB by adjusting the value of K. We set $K_p = 5$ and $K_i = 1$ and observe the impact:

MATLAB Code

```
clc; clear; close all;

num = [0.1 -0.2];
den = [1 0.9 9 0];
G = tf(num, den);

K_values = [5, -5];

for i = 1:length(K_values)
K = K_values(i);

Kp = 5;
Ki = 1;
C = K * tf([Kp Ki], [1 0]);

G_new = C * G;

figure;
rlocus(G_new);
title(['Root Locus with PI Controller (K = ', num2str(K), ')']);
grid on;
end
```

And the ouput is as follows:

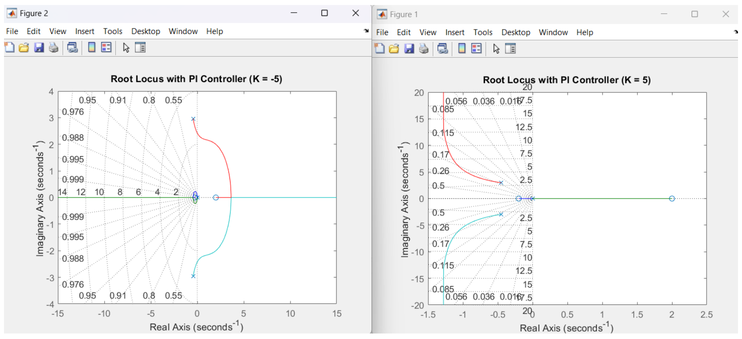


Figure 10: Root Locust

As observed, for $K = 5$, the system remains unstable at all times. However, for $K = -5$, the system can achieve stability within certain ranges.

6.3 PD Controller Analysis

The PD controller is defined as follows:

For $K > 0$:

$$C(s) = K_p + K_d s$$

By adding this controller, a zero is introduced into the system. However, for $K > 0$, this does not positively impact stability or the root locus, as a pole still moves towards the right-half plane, making the system unstable.

For $K < 0$:

$$C(s) = -(K_p + K_d s)$$

This inversion reverses the controller's effect, causing the poles to shift towards the left-half plane of the s-axis instead of moving right. Consequently, the system can become stable within certain ranges.

We can validate this behavior in MATLAB by varying K and setting $K_p = 5$ and $K_d = 1$:

MATLAB Code

```
clc; clear; close all;

num = [0.1 -0.2];
den = [1 0.9 9 0];
G = tf(num, den);

K_values = [5, -5];

for i = 1:length(K_values)
K = K_values(i);

Kp = 5;
Kd = 2;
C = K * tf([Kd Kp], [1]);

G_new = C * G;

figure;
rlocus(G_new);
title(['Root Locus with PD Controller (K = ', num2str(K), ')']);
grid on;
end
```

And the ouput is as follows:

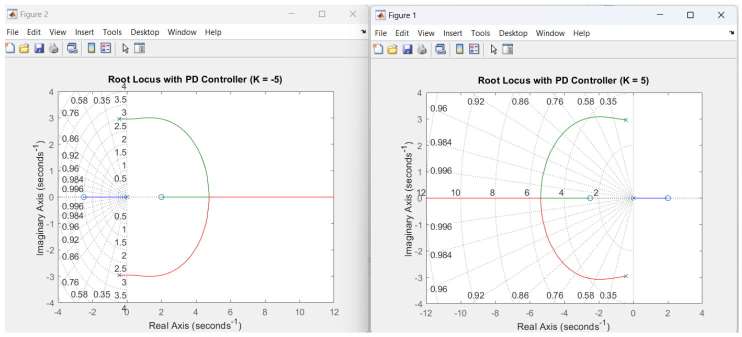


Figure 11: Root Locust

As observed, for $K = 5$, the system remains unstable at all times. However, for $K = -5$, the system can achieve stability within certain ranges.

7 Q6

By eliminating the required values, we obtain the following equation:

$$G(s) = \frac{0.1}{s^2 + 0.9s + 9}$$

The Bode plot is as follows:

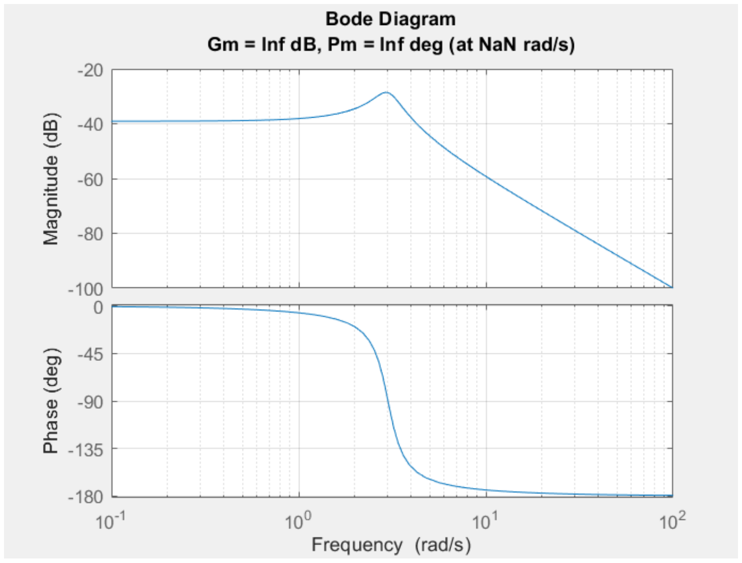


Figure 12: Bode Plot

As we can see, the magnitude plot never crosses 0 dB. Therefore, we multiply it by a gain to shift the magnitude plot slightly upward. We set the gain to 50:

$$G(s) = \frac{50 \times 0.1}{s^2 + 0.9s + 9} = \frac{5}{s^2 + 0.9s + 9}$$

The Bode plot of this transfer function is as follows:

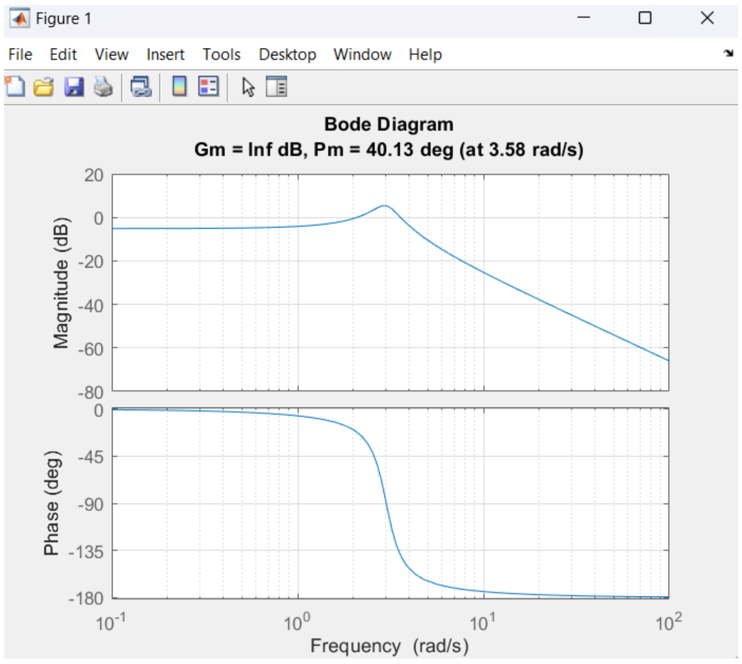
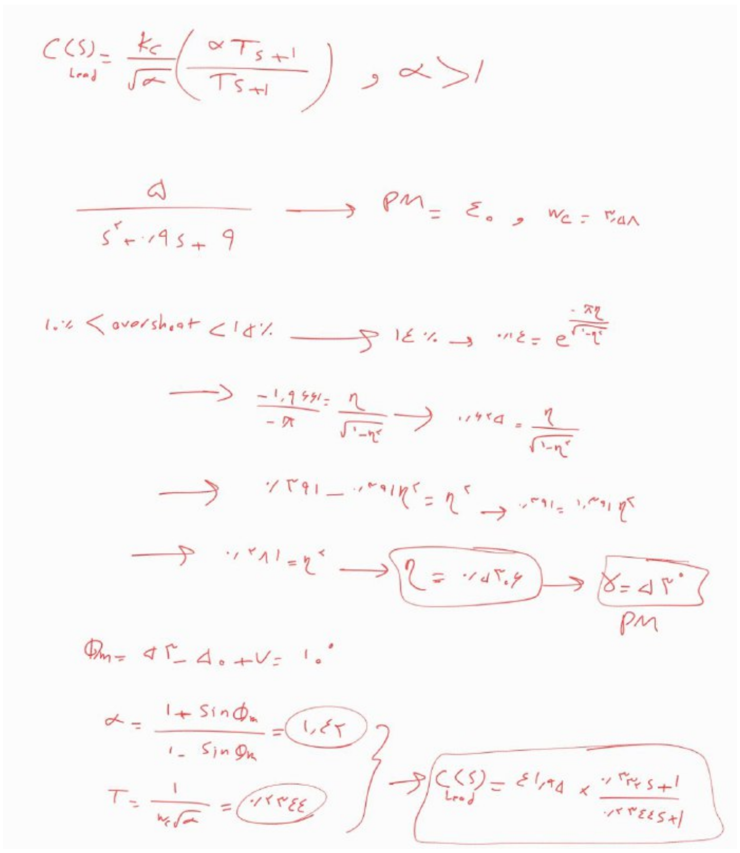


Figure 13: Bode Plot

To achieve an appropriate overshoot, we use a lead compensator:



$$C_{\text{Lead}}(s) = \frac{k_c}{\sqrt{\alpha}}\left(\frac{\alpha T s + ۱}{T s + ۱}\right) \text{ ، } \alpha > ۱$$

$$\frac{۵}{s^{۲} + ۰٫۹ s + ۹} \longrightarrow PM = ۴۰ \text{ ، } w_c = ۳٫۵۸$$

$$۱۰٪ < \text{overshoot} < ۱۵٪ \longrightarrow ۱۴٪ \rightarrow ۰٫۱۴ = e^{\frac{-\pi\eta}{\sqrt{۱-\eta^{۲}}}}$$

$$\longrightarrow \frac{-۱٫۹۶۶۱}{-\pi} = \frac{\eta}{\sqrt{۱-\eta^{۲}}} \longrightarrow ۰٫۶۲۵ = \frac{\eta}{\sqrt{۱-\eta^{۲}}}$$

$$\longrightarrow ۰٫۳۹۱ - ۰٫۳۹۱\eta^{۲} = \eta^{۲} \rightarrow ۰٫۳۹۱ = ۱٫۳۹۱\eta^{۲}$$

$$\longrightarrow ۰٫۲۸۱ = \eta^{۲} \longrightarrow \boxed{\eta = ۰٫۵۳۰۶} \longrightarrow \underbrace{\boxed{\gamma = ۵۳^{\circ}}}_{PM}$$

$$\Phi_m = ۵۳ - ۵۰ + ۷ = ۱۰^{\circ}$$

$$\alpha = \frac{۱ + \sin\Phi_m}{۱ - \sin\Phi_m} = \boxed{۱٫۴۲}$$

$$T = \frac{۱}{w_c\sqrt{\alpha}} = \boxed{۰٫۲۳۴۴}$$

$$\longrightarrow \boxed{C_{\text{Lead}}(s) = ۴۱٫۹۵ \times \frac{۰٫۳۳۲ s + ۱}{۰٫۲۳۴۴ s + ۱}}$$

Figure 14: Manual Calculations

Applying the Controller:

$$G(s)C_{\text{lead}}(s) = 41.95 \times \frac{0.332s + 1}{0.2344s + 1} \times \frac{0.1}{s^2 + 0.9s + 9}$$

Output:

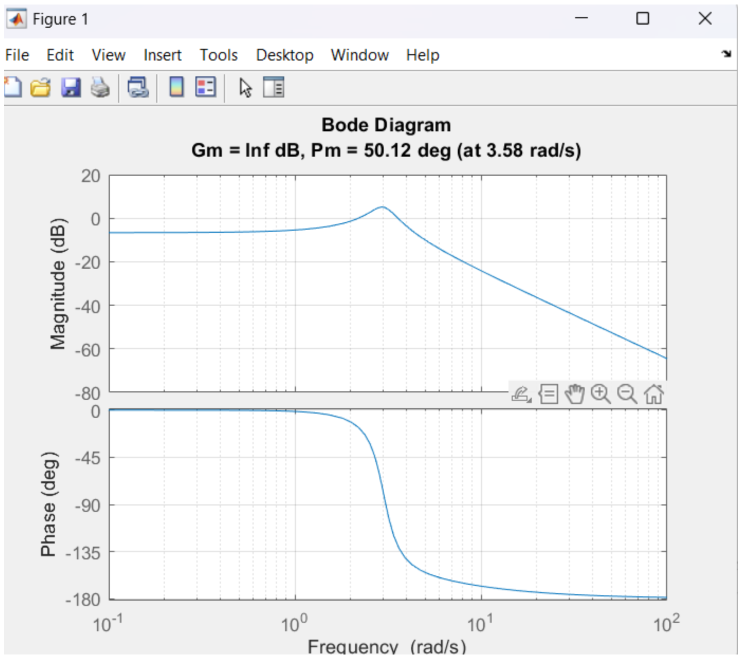


Figure 15: Utilizng Lead Controller

Step Response:

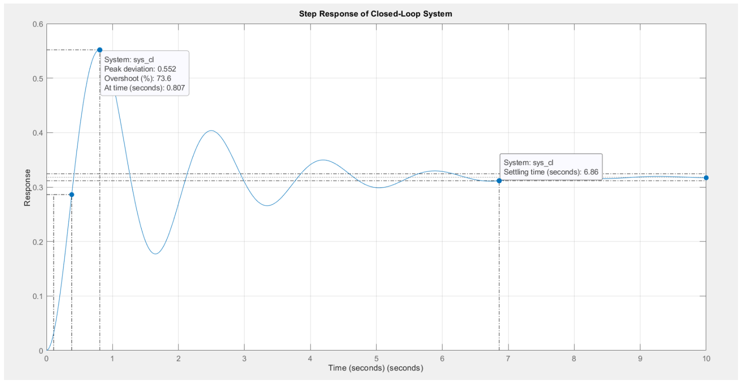
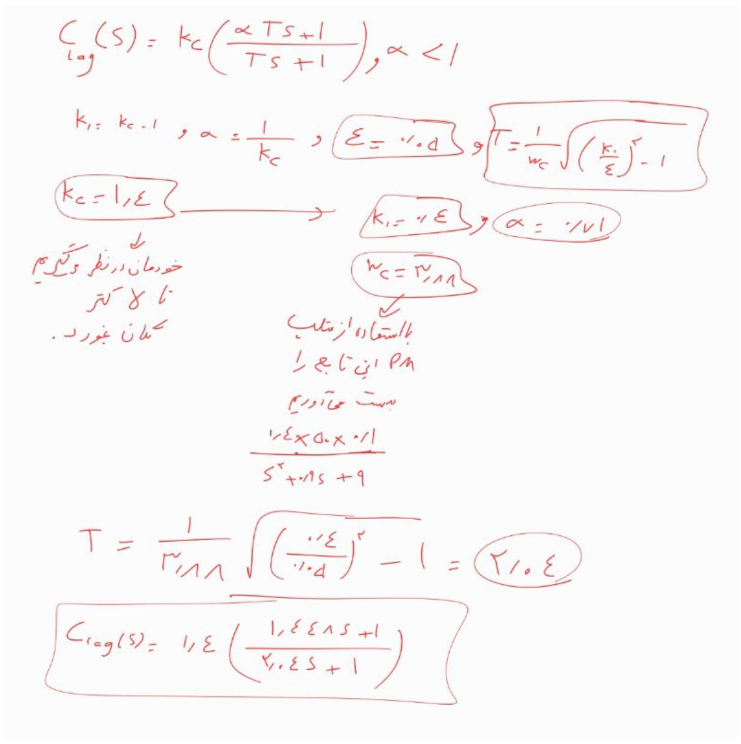


Figure 16: Step Response

As we can see, the overshoot is 73%, which is undesirable, but the settling time is 6.86 seconds, which is acceptable. Therefore, we use a **Lag** controller:



$$C_{lag}(s) = k_c\left(\frac{\alpha T s+1}{T s+1}\right), \alpha < 1$$

$k_1 = k_c - 1$, $\alpha = \frac{1}{k_c}$ و $\boxed{\varepsilon = ۰٫۰۵}$ و $\boxed{T = \frac{1}{w_c}\sqrt{\left(\frac{k_1}{\varepsilon}\right)^۲ - 1}}$

$\boxed{k_c = ۱٫۴} \longrightarrow \boxed{k_1 = ۰٫۴}$ و $\boxed{\alpha = ۰٫۷۱}$

خودمان در نظر میگیریم
تا لاگ کمتر
کمان بخورد.

$\boxed{w_c = ۳٫۸۸}$

با استفاده از متلب
PM این تابع را
بدست می‌آوریم

$$\frac{۱٫۴ \times ۵۰ \times ۰٫۱}{s^۲ + ۰٫۹s + ۹}$$

$$T = \frac{1}{۳٫۸۸}\sqrt{\left(\frac{۰٫۴}{۰٫۰۵}\right)^۲ - 1} = \boxed{۲٫۰۴}$$

$$\boxed{C_{lag}(s) = ۱٫۴\left(\frac{۱٫۴۴۸s+1}{۲٫۰۴s+1}\right)}$$

Figure 17: Manual Calculations

Applying Both Controllers:

$$C_{\text{lag}}(s)C_{\text{lead}}(s)G(s) = 1.4 \times \frac{1.448s+1}{2.04s+1} \times 41.95 \times \frac{0.332s+1}{0.2344s+1} \times \frac{0.1}{s^2+0.9s+9}$$

Bode Plot After Applying Both Controllers:

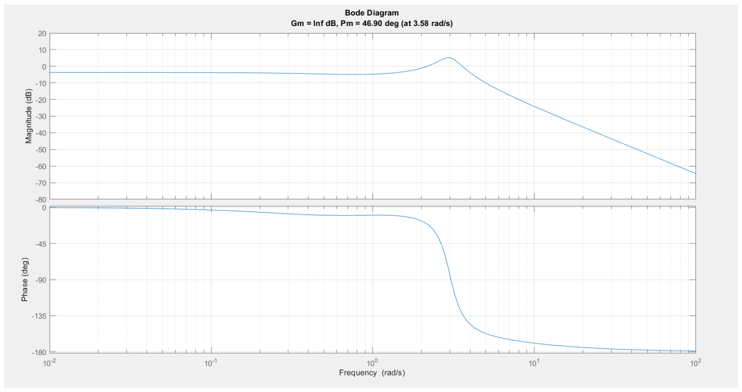


Figure 18: Bode plot

Step Response:

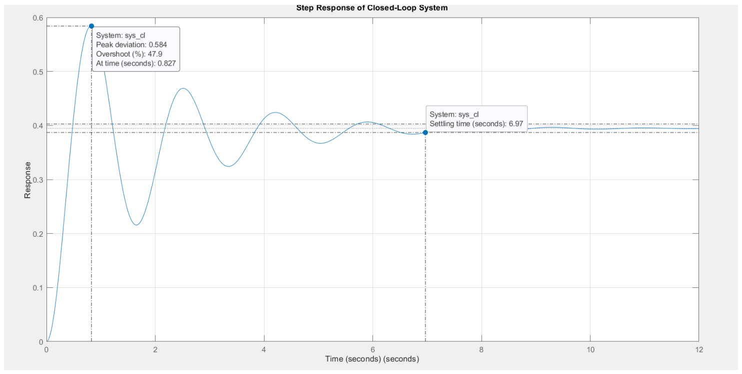


Figure 19: Step Response

As we can see, the overshoot has further improved, reducing from 73% to 47.9%. Now, we make slight adjustments (in the range of tenths) to the numerator and denominator of the **Lead** and **Lag** controllers to achieve an overshoot of 15%.

Our final design is as follows:

$$C_{\text{lag}}(s)C_{\text{lead}}(s)G(s) = 1.4 \times \frac{1.2s+1}{2.5s+1} \times 41.95 \times \frac{0.28s+1}{0.3s+1} \times \frac{0.1}{s^2+0.9s+9}$$

Bode Plot:

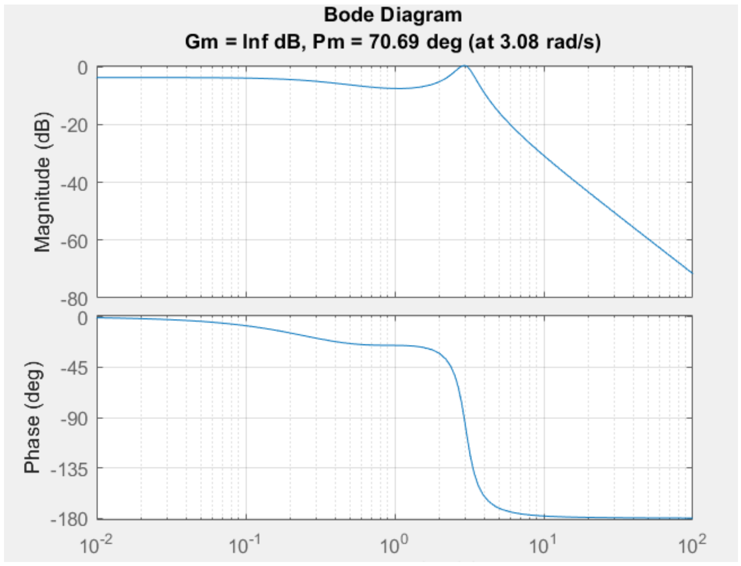


Figure 20: Bode Plot

Step Response:

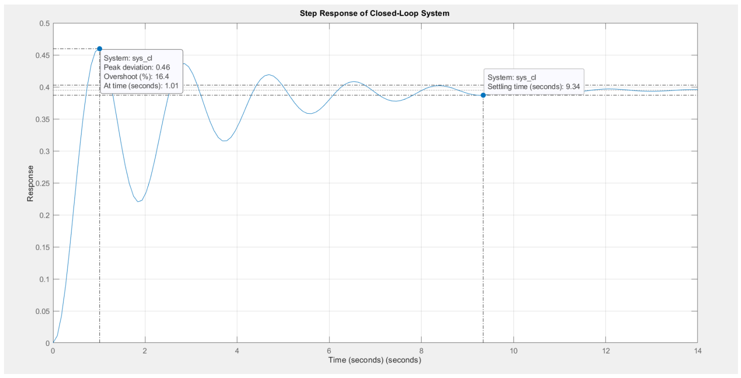


Figure 21: Step Response

MATLAB Code for the Final System with Lead and Lag Controllers (Step Response):

MATLAB Code

```
        clc; clear; close all;

        num_G = [0.1];
        den_G = [1 0.9 9];
        G = tf(num_G, den_G);

        num_Clag = 1.4 * [1.2 1];
        den_Clag = [2.5 1];
        C_lag = tf(num_Clag, den_Clag);

        num_Clead = 41.95 * [0.28 1];
        den_Clead = [0.3 1];
        C_lead = tf(num_Clead, den_Clead);

        G_total = C_lag * C_lead * G;

        sys_cl = feedback(G_total, 1);

        figure;
        step(sys_cl);
        grid on;
        title('Step Response of Closed-Loop System');
        xlabel('Time (seconds)');
        ylabel('Response');
```

Comparison of Our Design with the Uncontrolled System

Step Response of the Uncontrolled System:

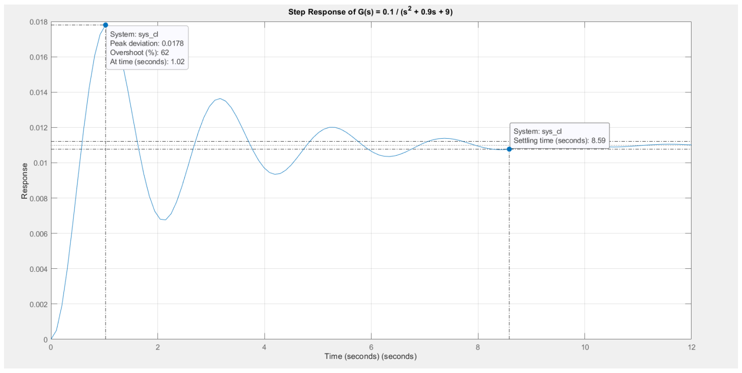


Figure 22: Step Response

As observed:

1. The overshoot has been reduced from 62% to 16.4%.
2. The settling time has improved from 8.59 seconds to 9.34 seconds.
3. The phase margin has increased from infinity to 70.69 degrees.

This confirms that our design has been successfully implemented.

8 Q7

To solve this problem, we first address **Part 2** by selecting a suitable controller using the sensitivity function. Then, for **Part 1**, we improve the compensated system by implementing a **PI controller**, whose purpose is to reduce the steady-state error.

8.1 Part2

We know that our transfer function is given by:

$$\frac{0.1s - 0.2}{s^3 + 0.9s^2 + 9s}$$

The relative degree is **two**, and we have an interpolation condition (a non-minimum phase zero):

$$T(2) = 0$$

As a result, the degree of the complementary sensitivity function is **three**:

$$T_d(s) = \frac{w_c^3(s/\tau + 1)}{(s + w_c)^3}$$

We set $w_c = 1$:

$$T_d(s) = \frac{(s/\tau + 1)}{(s + 1)^3}$$

By solving $T_d(2) = 0$, we obtain $\tau = -2$.

Substituting τ and simplifying, we get:

$$T_d(s) = \frac{-s + 2}{2(s^3 + 3s^2 + 3s + 1)}$$

Now, we determine $S_d(s)$:

$$S_d(s) = 1 - T_d(s) = \frac{2s^3 + 6s^2 + 7s}{2(s^3 + 3s^2 + 3s + 1)}$$

Next, we find the controller:

$$C(s) = \frac{T_d(s)}{S_d(s)P(s)} = \frac{-10(s^3 + 0.9s^2 + 9s)}{2s^3 + 6s^2 + 7s}$$

Now, we compute the **open-loop transfer function**:

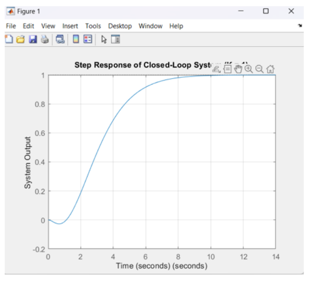
$$L(s) = C(s)P(s) = \frac{-(s - 2)}{2s^3 + 6s^2 + 7s}$$

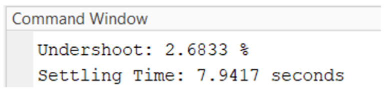
Finally, we obtain the **closed-loop step response** using the following MATLAB code:

MATLAB Code

```matlab
clc; clear; close all;

numG = [-1 2];
denG = [2 6 7 0];

K = 1;
G = K * tf(numG, denG);

T = feedback(G, 1);

figure;
step(T);
grid on;
title(['Step Response of Closed-Loop System (K = ', num2str(K), ')']);
xlabel('Time (seconds)');
ylabel('System Output');

stepInfo = stepinfo(T);
undershoot = stepInfo.Undershoot;
settlingTime = stepInfo.SettlingTime;

disp(['Undershoot: ', num2str(undershoot), ' %']);
disp(['Settling Time: ', num2str(settlingTime), ' seconds']);
```

The output is as follows:



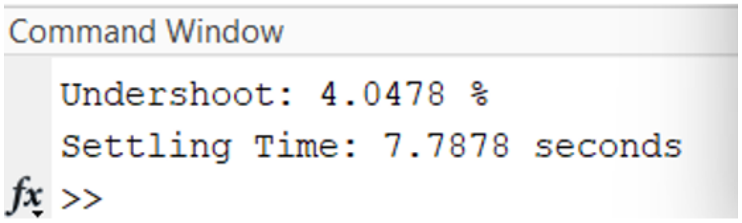

Command Window

```
Undershoot: 2.6833 %
Settling Time: 7.9417 seconds
```

Figure 23: Output

As we can see, the **undershoot** is below 6%, but the **settling time** needs to be improved. In the previous system, we set the **gain** K **to 1**, but now we adjust K to achieve a **settling time of 6 seconds**. Increasing K speeds up the response.

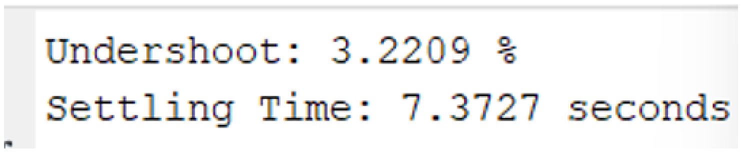
First, we set $K = 1.5$:



```
Command Window
  Undershoot: 4.0478 %
  Settling Time: 7.7878 seconds
fx >>
```

Figure 24: Output

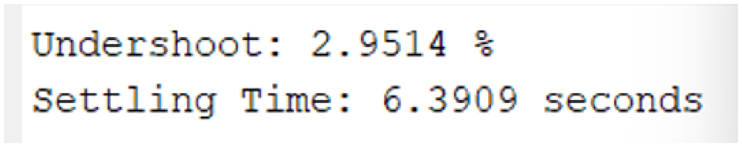
We observe that this value is not suitable. Now, we set $K = 1.2$:



```
Undershoot: 3.2209 %
Settling Time: 7.3727 seconds
```

Figure 25: Output

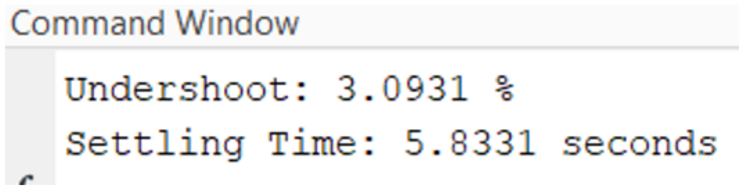
It is still not suitable. Now, we set $K = 1.1$:



```
Undershoot: 2.9514 %
Settling Time: 6.3909 seconds
```

Figure 26: Output

It improved, but the settling time did not drop below 6 seconds. Now, we set $K = 1.15$:



```
Undershoot: 3.0931 %
Settling Time: 5.8331 seconds
```

Figure 27: Output

We have achieved the desired result. Now, for $K = 1.15$, we plot the **step response** and the **slope**:

MATLAB Code

```
clc; clear; close all;

numG = [-1 2];
denG = [2 6 7 0];

K = 1;
G = K * tf(numG, denG);

T = feedback(G, 1);


t = 0:0.1:50;


[y_step, t_step] = step(T, t);


ramp_input = t;
[y_ramp, t_ramp] = lsim(T, ramp_input, t);


figure;
subplot(2,1,1);
plot(t_step, y_step, 'b', 'LineWidth', 1.5);
grid on;
title(['Step Response of Closed-Loop System (K = ', num2str(K), '
    )']);
xlabel('Time (seconds)');
ylabel('System Output');
legend('Step Response');

subplot(2,1,2);
plot(t_ramp, y_ramp, 'r', 'LineWidth', 1.5);
grid on;
title(['Ramp Response of Closed-Loop System (K = ', num2str(K), '
    )']);
xlabel('Time (seconds)');
ylabel('System Output');
legend('Ramp Response');
```

Output:

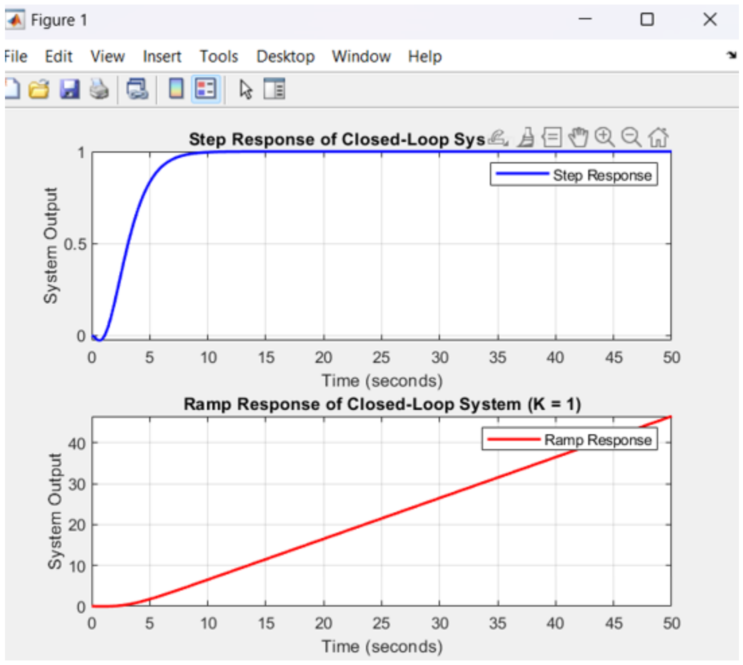


Figure 28: Output

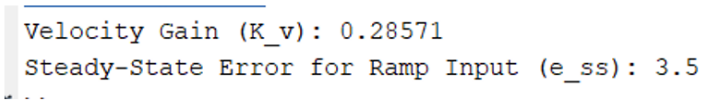
8.2 Part1

In this section, we aim to reduce the **steady-state error** for a **ramp input** to below **2%** and improve the steady-state error. To achieve this, we use a **PI controller**.

First, we determine the steady-state error of the compensated system obtained from the sensitivity function:

$$L(s) = C(s)P(s) = \frac{-(s-2)}{2s^3 + 6s^2 + 7s}$$

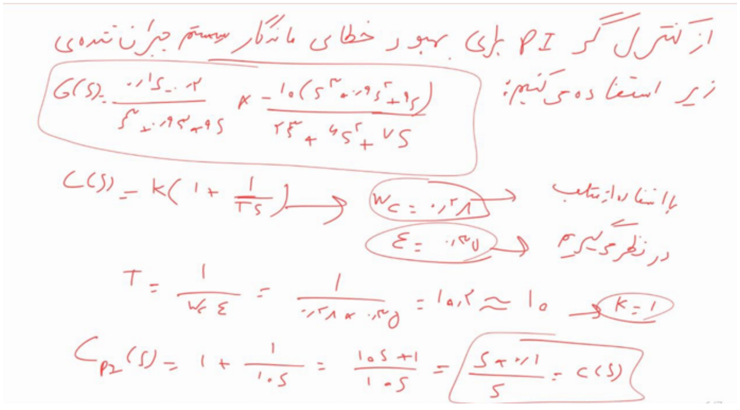
Steady-State Error:



```
Velocity Gain (K_v): 0.28571
Steady-State Error for Ramp Input (e_ss): 3.5
```

Figure 29: Steady-State Error

As we can see, the steady-state error is **350%**, which is excessively high. Now, we introduce the **PI controller**:



از کنترل گر PI برای بهبود خطای ماندگار سیستم جبران شده ی زیر استفاده می کنیم:

$$\boxed{G(s) = \frac{0.1s - 0.2}{s^3 + 0.9s^2 + 9s} \times \frac{-10(s^3 + 0.9s^2 + 9s)}{2s^3 + 6s^2 + 7s}}$$

$$C(s) = k\left(1 + \frac{1}{Ts}\right) \longrightarrow \boxed{\omega_c = 0.238} \longrightarrow$$ با استفاده از متلب

$$\boxed{\varepsilon = 0.412} \longrightarrow$$ در نظر می گیریم

$$T = \frac{1}{\omega_c \varepsilon} = \frac{1}{0.238 \times 0.412} = 10.2 \simeq 10 \longrightarrow \boxed{k = 1}$$

$$C_{PI}(s) = 1 + \frac{1}{10s} = \frac{10s + 1}{10s} = \boxed{\frac{s + 0.1}{s} = C(s)}$$

Figure 30: Manual Calculations

Thus, our **PI controller** is given by:

$$C_{\mathrm{PI}}(s) = \frac{s + 0.1}{s}$$

The compensated system is:

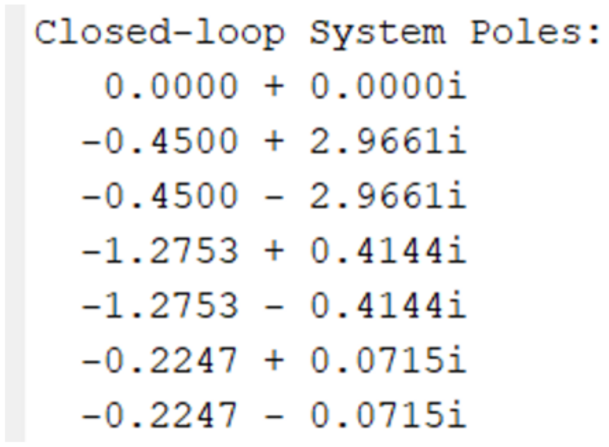
$$C_{\mathrm{PI}}(s)C_d(s)P(s) = \frac{(s + 0.1)}{s} \times \frac{-10(s^3 + 0.9s^2 + 9s)}{2s^3 + 6s^2 + 7s} \times \frac{0.1s - 0.2}{s^3 + 0.9s^2 + 9s}$$

Now, we analyze the **steady-state error** of this system for a **ramp input**. Additionally, we compute the **poles of the closed-loop transfer function** to ensure system stability:

MATLAB Code

```matlab
clc; clear; close all;

K = 1;

s = tf('s');

numG1 = K * (0.1 * s - 0.2);
denG1 = (s^3 + 0.9 * s^2 + 9 * s);

numG2 = (s + 0.1);
denG2 = s;

numG3 = -10 * (s^3 + 0.9 * s^2 + 9 * s);
denG3 = (2 * s^3 + 6 * s^2 + 7 * s);

G = (numG1 / denG1) * (numG2 / denG2) * (numG3 / denG3);

T = feedback(G, 1);

poles = pole(T);

Kv = dcgain(s * G);

if Kv ~= 0
ess = 1 / Kv;
else
ess = inf;
end

disp(['For K = ', num2str(K)]);
disp('Closed-loop Transfer Function T(s):');
T

disp('Closed-loop System Poles:');
disp(poles);

disp(['Velocity Gain (K_v): ', num2str(Kv)]);
disp(['Steady-State Error for Ramp Input (e_ss): ',
    num2str(ess)]);
```

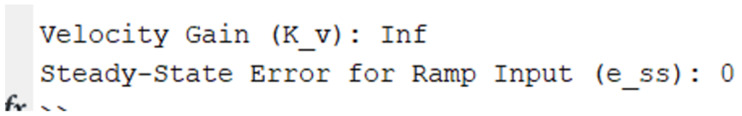
The closed-loop system poles:



```
Closed-loop System Poles:
   0.0000 + 0.0000i
  -0.4500 + 2.9661i
  -0.4500 - 2.9661i
  -1.2753 + 0.4144i
  -1.2753 - 0.4144i
  -0.2247 + 0.0715i
  -0.2247 - 0.0715i
```

Figure 31: The closed-loop system poles

Steady-State Error:



```
Velocity Gain (K_v): Inf
Steady-State Error for Ramp Input (e_ss): 0
```

Figure 32: Steady-State Error

As we can see, the system is **stable**, and the steady-state error for a **ramp input** is **zero**, which confirms that our design has been successfully implemented.

Step Response of the System:

MATLAB Code

```matlab
clc; clear; close all;

s = tf('s');

numG1 = (s + 0.1);
denG1 = s;

numG2 = -10 * (s^3 + 0.9 * s^2 + 9 * s);
denG2 = (2 * s^3 + 6 * s^2 + 7 * s);

numG3 = (0.1 * s - 0.2);
denG3 = (s^3 + 0.9 * s^2 + 9 * s);

L = (numG1 / denG1) * (numG2 / denG2) * (numG3 / denG3);

T = feedback(L, 1);

figure;
step(T);
grid on;
title('Step Response of the Closed-loop System');
xlabel('Time (seconds)');
ylabel('System Output');
```

Step Response:

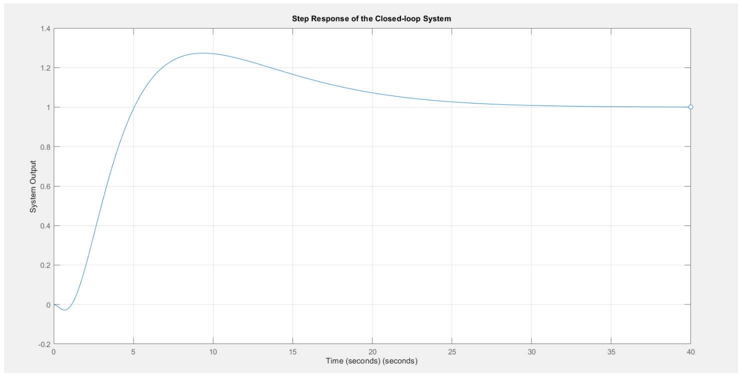


Figure 33: Step Response

As observed, the **step response** of the system is also excellent. Thus, our design is correct.

Good Luck