

LEVEL 4 ASSESSMENT SPECIFICATION

Student name:	Nariman Ramadhan	Student ID number:	RAM20849142
Programme:	HNC /HND In Electrical and Electronic Engineering Level 4		
Module:	Dynamics Control and Instrumentation		
Module code:	GOEE4002	Contribution to Overall Module Assessment (%):	50%
Lecturer:	Emily Nyathi	Internal Verifier:	Dr Rabya Kadhim
Assignment Title:	Design and Control of a Mass-Spring-Damper System Using a PID Controller	Word count (or equivalent):	2000
Submission deadline:	29 April 2025	Return date of provisional marks & written feedback:	Within 20 working days
Submission method:	All written assessments, where practical and possible, must be submitted via Turnitin unless otherwise instructed by the Lecturer. (Please DO NOT put this assessment specification into Turnitin or it will match many similarities with other students' submissions.) Alternative submission method (if applicable): Late submission of the assessment will result in a late penalty mark. Penalties for late submission: Up to one week late, maximum mark of 40%. Over one week late, 0%. Only the Extenuating Circumstances Panel may approve a change to submission dates.		
Academic honesty / referencing:	Academic honesty is required. In the main body of your submission, you must give credit to authors on whose research and ideas your work is based. Append to your submission a reference list that indicates the books, articles, etc. that you have used, cited or quoted to complete this assessment.		

**Module Learning Outcomes
(from module syllabus)**

1. Solve, analyse and simulate the behaviour of dynamic mechanical systems using traditional analytical techniques and computer-based simulation.
2. Apply instrumentation and control system methodologies to dynamical systems and analyse the behaviour and response of the system and how it can be controlled.

[This project models and controls a mass-spring-damper system using MATLAB and Simulink. A PID controller was designed and tuned to improve system performance. The best results, with fast settling time and low error, were achieved using **K_p = 25, K_i = 8, and K_d = 15**. The Simulink model showed how effective PID tuning enhances system stability and accuracy.]

Design and Control of a Mass-Spring-Damper System Using a PID Controller

RAM20849142 (Nariman Amjad Ramadhan)

Contents

Contribution to Overall Module Assessment (%):	1
Contents	4
Table of Contents	4
Introduction	5
Mass spring damper system:	5
Real-Life Applications	6
Control theory and PID controller:	6
Experimental Setup	7
Mathematical model:	8
Open-Loop Step Response	8
PID Controller	9
Closed-Loop System with Feedback	10
Manual Tuning of PID Parameters	11
Data Analysis	12
Key Metrics to Look for in the Step Response:	14
How to Compare the Graphs	14
Results and Discussion	20
Using Simulink	22
Step 1: Define Parameters in MATLAB	22
Step 2: Create the Mass-Spring-Damper Model in Simulink	23
Step 3: Adding the PID Controller	23
Step 4: Configure and Tune the PID Controller	24
Conclusion	28
Appendix 1	29
References	38

Task1

Introduction

Understanding the dynamics of mechanical systems is important in many areas of engineering, including robotics, automotive engineering, and structural engineering (engineeringcheatsheet, 2024). One of the simplest models that are used to approximate these dynamics is the **mass-spring-damper** system, which replicates the response of a mass to external forces when it is mounted on a spring and damper. The model is widely used in the study of vibrations and mechanical responses in systems ranging from vehicle suspensions to structural dampers.

In modern control engineering, one of the most common feedback control techniques is **PID (Proportional-Integral-Derivative) controllers**. Their popularity stems from their simplicity and flexibility in achieving desired dynamic performance, especially in industrial automation, process control, and electromechanical systems.

This experiment deals with the design and tuning of a PID controller to control the position of a mass in a spring-damper system. Using MATLAB simulations, the experiment compares the open-loop and closed-loop behavior of the system. Key performance metrics such as overshoot, settling time, and steady-state error are analyzed to ascertain the performance of the PID controller.

Background theory

Mass spring damper system:

Spring-mass-damper theory applies the basic physics of $F=ma$ to define suspension motion and response on bump impacts. Force (F) acting on the suspension is the combination of spring plus damping force. Mass (m) is the chassis or wheel weight depending on which body you want to look at. The combination of force and mass define the $F=ma$ acceleration, which in turn defines the velocity, position and suspension response characteristics at any instant in time (shim restackor, n.d.).

An MSD system consists of:

- Mass (m): An object with inertia

- Spring (k): Provides a restoring force proportional to displacement.
- Damper (c): Offers resistance proportional to velocity.

The governing differential equation of motion for the mass-spring-damper system can be derived using Newton's second law:

$$m \frac{d^2x}{dt^2} + c \frac{dx}{dt} + kx = F(t)$$

- x is the displacement, of the mass from its equilibrium position.
- $F(t)$ is the external force applied to the system.
- $\frac{dx}{dt}$ is the velocity.
- $\frac{d^2x}{dt^2}$ is the acceleration.

Real-Life Applications

The mass-spring-damper system is used in many areas of engineering to help understand and control movement and vibrations. In cars, it models the suspension system to improve comfort and handling by reducing bumps and shocks. In buildings and bridges, it helps reduce shaking during earthquakes by using large dampers, like the one in the Taipei 101 tower (scienceclarifie, 2006). In biomechanics, it helps simulate how joints and muscles move and is used in designing prosthetic limbs. In airplanes, the landing gear uses this system to absorb the force of landing and keep the plane stable.

Control theory and PID controller:

Control theory focuses on designing systems that behave in a desired manner. PID controller is an instrument that receives input data from sensors, calculates the difference between the actual value and the desired setpoint, and adjusts outputs to control variables such as temperature, flow rate, speed, pressure, and voltage (ni, 2025). It does this through three mechanisms: proportional control, which reacts to current error; integral control, which addresses accumulated past errors; and derivative control, which predicts future errors. The PID controller sums those three components to compute the output. This architecture allows PID controllers to efficiently maintain process control and system stability.

The overall PID control formula is given by:

$$u(t) = K_p e(t) + K_i \int e(t) dt + K_d \frac{de(t)}{dt}$$

Where:

$u(t)$ = PID Control Value

K_p = Proportional Gain

$e(t)$ = error value

K_i = integral gain $\frac{de}{dt}$ = change in error

dt = change in time

Proportional Term (P): The proportional term produces an output which is directly proportional to the current value of the error. When the error is large, the proportional response will also be large, and when the error is small, the response will also be small. The proportional gain K_p is an adjustable parameter that determines this scaling (sufyan, 2025).

Integral Term (I): The integral term gives an output which is proportional to the summation of the error over time. It is the running sum of past errors. The integral term builds up when there is a persistent error, and it continues to build until the error is eliminated.

Derivative Term (D): The derivative term produces an output in proportion to the rate of change in the error. It predicts in which direction the error is headed based on the rate at which the error is increasing or decreasing (sufyan, 2025).

Task 2

MATLAB script to model and simulate a mass-spring-damper system, which is a common mechanical system used to understand vibrations and damping.

Experimental Setup

In this step, I will model and simulate the mass-spring-damper system using MATLAB, which will allow me

to analyse its dynamic response and implement a PID controller for control purposes.

% Defining System Parameters

m = 1; % mass (kg)

k = 10; % spring constant (N/m)

c = 2; % damping coefficient (Ns/m)

% Transfer Function: $G(s) = 1/(m*s^2 + c*s + k)$

num = 1;

den = [m, c, k];

sys = tf(num, den);

System Parameters

Mathematical model:

The mass-spring-damper system is governed by the following second-order differential equation:

$$m \frac{d^2x}{dt^2} + c \frac{dx}{dt} + kx = F(t)$$

Given system parameters:

- Mass (m): 1 kg
- Spring constant (K): 10 N/m
- Damping coefficient (c):

$$G(s) = \frac{X(s)}{F(s)} = \frac{1}{ms^2 + cs + k}$$

$$G(s) = \frac{1}{s^2 + 2s + 10}$$

Open-Loop Step Response

The plot shows the system's response to an instantaneous push and how it oscillates and settles over time.

% Open-loop step response

figure;

```
step(sys);
```

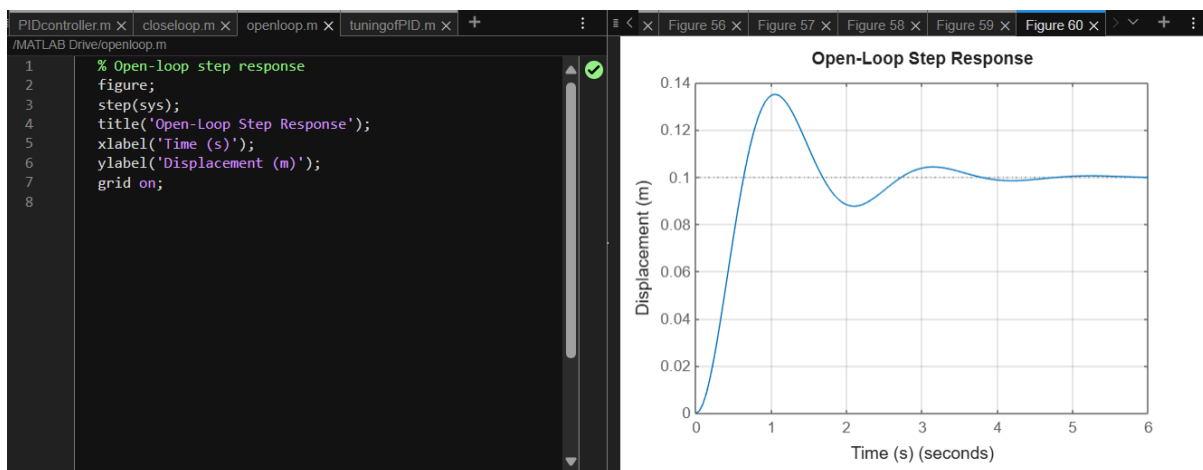
```
title('Open-Loop Step Response');
```

```
xlabel('Time (s)');
```

```
ylabel('Displacement (m)');
```

```
grid on;
```

MATLAB Implementation:



PID Controller

Next, we will introduce a PID controller to improve the system's performance by adjusting the input force based on the system's error. The PID controller has three components:

- Proportional (K_p): Adjusts based on the current error.
- Integral (K_i): Corrects for accumulated past errors.
- Derivative (K_d): Predicts future errors based on the rate of error change.

% Initial PID Controller Parameters

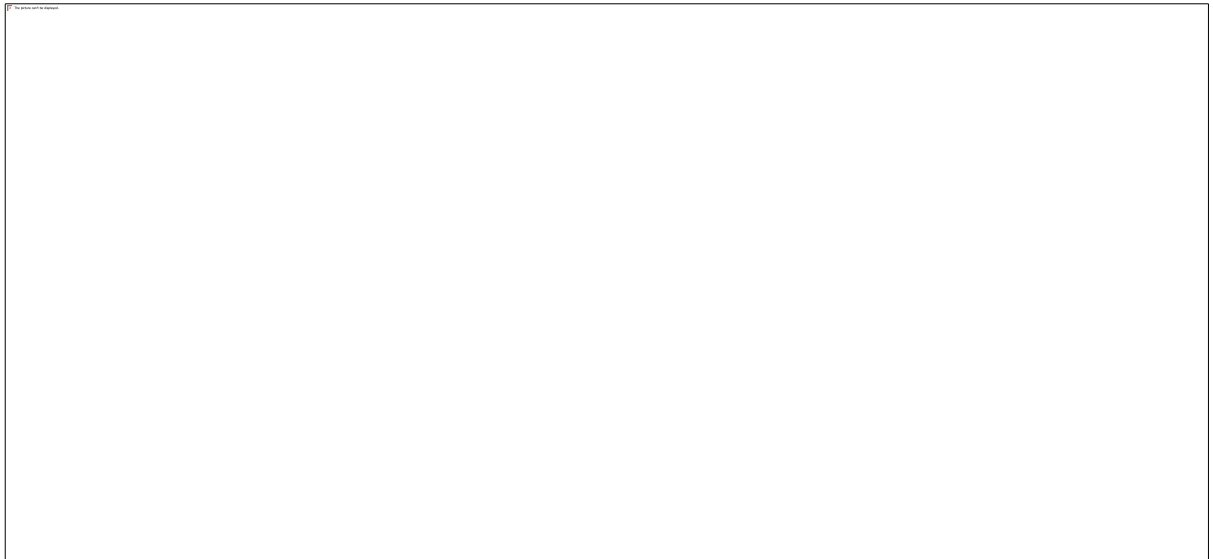
```
 $K_p = 20;$ 
```

```
 $K_i = 5;$ 
```

```
 $K_d = 10;$ 
```

% Create the PID controller

`C = pid(Kp, Ki, Kd);`



Closed-Loop System with Feedback

The PID controller is applied in a closed-loop configuration, where the output of the system is fed back into the controller to adjust the input force. This creates a closed-loop system with feedback.

% Step response of the closed-loop system

figure;

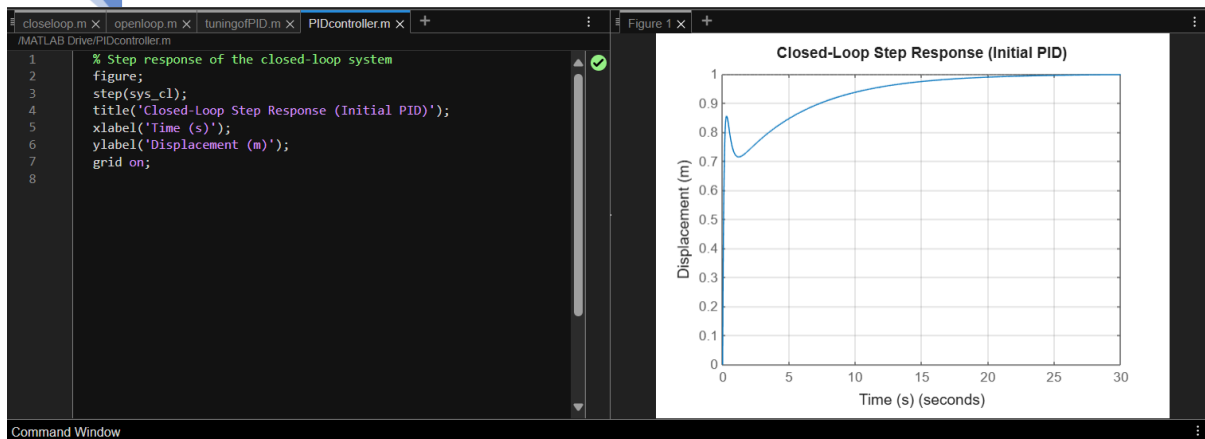
step(sys_cl);

title('Closed-Loop Step Response (Initial PID)');

xlabel('Time (s)');

ylabel('Displacement (m)');

grid on;



This step shows how the system behaves with the PID controller in place, improving the precision of the mass's movement.

Manual Tuning of PID Parameters

We will manually tune the PID controller by experimenting with different values of K_p , K_i and K_d to achieve optimal performance. For each combination, we will simulate the closed-loop response and measure performance metrics such as settling time, overshoot, and steady-state error.

% Manual tuning PID parameters

$K_p_values = [15, 20, 25];$

$K_i_values = [2, 5, 8];$

$K_d_values = [5, 10, 15];$

for $k_p = K_p_values$

for $k_i = K_i_values$

for $k_d = K_d_values$

% Create PID controller

$C = \text{pid}(k_p, k_i, k_d);$

% Closed-loop system

$\text{sys_cl} = \text{feedback}(C * \text{sys}, 1);$

% Compute step response

$[y, t] = \text{step}(\text{sys_cl});$

```

info = stepinfo(y, t);

ess = abs(1 - y(end)); % Steady-state error

% Plot step response

figure;

step(sys_cl);

title(['Step Response for Kp = ', num2str(kp), ...
      ', Ki = ', num2str(ki), ', Kd = ', num2str(kd)]);

xlabel('Time (s)');

ylabel('Displacement (m)');

grid on;

% Display performance metrics

disp(['Kp = ', num2str(kp), ', Ki = ', num2str(ki), ...
      ', Kd = ', num2str(kd)]);

disp([' Settling Time: ', num2str(info.SettlingTime), ' s']);

disp([' Overshoot: ', num2str(info.Overshoot), ' %']);

disp([' Steady-State Error: ', num2str(ess)]);

disp('-----');

pause(2);

end

end

end

```

This loop runs through different combinations of PID values and plots the resulting step responses. It also displays the corresponding performance metrics for each configuration.

Data Analysis

In this section, I will analyse the performance of the mass-spring-damper system under different PID controller settings. Specifically, I will evaluate the settling time, overshoot, steady-state error, and damping of the system's step responses to determine which combination of PID parameters provides the best performance.

```
% Manual tuning PID parameters
```

```
Kp_values = [15, 20, 25];
```

```
Ki_values = [2, 5, 8];
```

```
Kd_values = [5, 10, 15];
```

```
for kp = Kp_values
```

```
    for ki = Ki_values
```

```
        for kd = Kd_values
```

```
            % Create PID controller
```

```
            C = pid(kp, ki, kd);
```

```
            % Closed-loop system
```

```
            sys_cl = feedback(C * sys, 1);
```

```
            % Compute step response
```

```
            [y, t] = step(sys_cl);
```

```
            info = stepinfo(y, t);
```

```
            ess = abs(1 - y(end)); % Steady-state error
```

```
            % Plot step response
```

```
            figure;
```

```
            step(sys_cl);
```

```
            title(['Step Response for Kp = ', num2str(kp), ...
```

```
                  ', Ki = ', num2str(ki), ', Kd = ', num2str(kd)]);
```

```
            xlabel('Time (s)');
```

```

ylabel('Displacement (m)');

grid on;

% Display performance metrics

disp(['Kp = ', num2str(kp), ', Ki = ', num2str(ki), ...
      ', Kd = ', num2str(kd)]);

disp([' Settling Time: ', num2str(info.SettlingTime), ' s']);

disp([' Overshoot: ', num2str(info.Overshoot), ' %']);

disp([' Steady-State Error: ', num2str(ess)]);

disp('-----');

pause(2);

end

end

end

```

After running the manual tuning loop and generating the graphs, I should be able to identify the most effective PID settings by comparing the metrics for each scenario. In this experiment, the **settling time**, **overshoot**, **steady-state error**, and **damping** are crucial indicators of the system's overall performance.

Key Metrics to Look for in the Step Response:

- **Overshoot:** Measure how much the peak response exceeds the desired value.
- **Settling Time:** Record the time taken for the system to settle within a certain percentage (e.g. 2%) of the final value.
- **Steady-State Error:** Calculate the difference between the final output and the desired output value.

How to Compare the Graphs

The MATLAB script generates 29 step response graphs by testing different combinations of K_p , K_i , and K_d . I can compare the results by focusing on the following aspects:

1. Settling Time

- Compare how quickly each graph reaches its final value and stops changing. The faster the system settles, the better.
- Look for the graphs with the smallest settling times, as they represent a more responsive system.

2. Overshoot

- Examine each graph for peak values that exceed the desired displacement. Significant overshoot indicates an overly reactive system that requires adjustment.
- Look for graphs with minimal overshoot (less than 10%) to avoid overreaction.

3. Steady-State Behavior

- After the system settles, check if the displacement remains constant at the desired final value.
- If there's a steady-state error (the displacement doesn't match the target), it suggests that the PID parameters need further tuning.

4. Damping

- Assess the smoothness of the graph as it approaches the final displacement. If the system oscillates too much before settling, the damping might be insufficient.
- Ideally, you want graphs that show a smooth and quick transition with minimal oscillation.

Initial PID Performance:

Settling Time: 16.3452 s

Overshoot: 0 %

Steady-State Error: 0.0038101

 $K_p = 15$, $K_i = 2$, $K_d = 5$

Settling Time: 36.2697 s

Overshoot: 0 %

Steady-State Error: 0.0012006

$K_p = 15, K_i = 2, K_d = 10$

Settling Time: 36.4303 s

Overshoot: 0 %

Steady-State Error: 0.00056151

$K_p = 15, K_i = 2, K_d = 15$

Settling Time: 35.1257 s

Overshoot: 0 %

Steady-State Error: 0.0025965

$K_p = 15, K_i = 5, K_d = 5$

Settling Time: 13.958 s

Overshoot: 0 %

Steady-State Error: 0.002491

$K_p = 15, K_i = 5, K_d = 10$

Settling Time: 14.2026 s

Overshoot: 0 %

Steady-State Error: 0.00029479

$K_p = 15, K_i = 5, K_d = 15$

Settling Time: 13.5959 s

Overshoot: 0 %

Steady-State Error: 0.0021133

$K_p = 15, K_i = 8, K_d = 5$

Settling Time: 8.5778 s

Overshoot: 0 %

Steady-State Error: 0.0023356

$K_p = 15, K_i = 8, K_d = 10$

Settling Time: 8.1168 s

Overshoot: 0 %

Steady-State Error: 0.0044884

$K_p = 15, K_i = 8, K_d = 15$

Settling Time: 8.0425 s

Overshoot: 0 %

Steady-State Error: 0.0029472

$K_p = 20, K_i = 2, K_d = 5$

Settling Time: 39.1406 s

Overshoot: 0 %

Steady-State Error: 0.0040498

$K_p = 20, K_i = 2, K_d = 10$

Settling Time: 39.1651 s

Overshoot: 0 %

Steady-State Error: 0.0038255

$K_p = 20, K_i = 2, K_d = 15$

Settling Time: 39.3908 s

Overshoot: 0 %

Steady-State Error: 0.0032608

 $K_p = 20, K_i = 5, K_d = 5$

Settling Time: 16.2748 s

Overshoot: 0 %

Steady-State Error: 0.00079923

$K_p = 20, K_i = 5, K_d = 10$

Settling Time: 15.3907 s

Overshoot: 0 %

Steady-State Error: 0.0038101

$K_p = 20, K_i = 5, K_d = 15$

Settling Time: 15.1346 s

Overshoot: 0 %

Steady-State Error: 0.0044044

$K_p = 20, K_i = 8, K_d = 5$

Settling Time: 9.6721 s

Overshoot: 0 %

Steady-State Error: 0.0031085

$K_p = 20, K_i = 8, K_d = 10$

Settling Time: 9.309 s

Overshoot: 0 %

Steady-State Error: 0.0048068

$K_p = 20, K_i = 8, K_d = 15$

Settling Time: 9.4252 s

Overshoot: 0 %

Steady-State Error: 0.0028891

$K_p = 25, K_i = 2, K_d = 5$

Settling Time: 44.0584 s

Overshoot: 0 %

Steady-State Error: 0.0026999

$K_p = 25, K_i = 2, K_d = 10$

Settling Time: 45.4267 s

Overshoot: 0 %

Steady-State Error: 0.00083022

$K_p = 25, K_i = 2, K_d = 15$

Settling Time: 45.6796 s

Overshoot: 0 %

Steady-State Error: 0.00040277

$K_p = 25, K_i = 5, K_d = 5$

Settling Time: 17.0441 s

Overshoot: 0 %

Steady-State Error: 0.0041036

$K_p = 25, K_i = 5, K_d = 10$

Settling Time: 17.9341 s

Overshoot: 0 %

Steady-State Error: 0.00079485

$K_p = 25, K_i = 5, K_d = 15$

Settling Time: 17.1782 s

Overshoot: 0 %

Steady-State Error: 0.0030539

$K_p = 25, K_i = 8, K_d = 5$

Settling Time: 10.684 s

Overshoot: 0 %

Steady-State Error: 0.0033242

$K_p = 25, K_i = 8, K_d = 10$

Settling Time: 11.0146 s

Overshoot: 0 %

Steady-State Error: 0.0010046

$K_p = 25, K_i = 8, K_d = 15$

Settling Time: 11.0429 s

Overshoot: 0 %

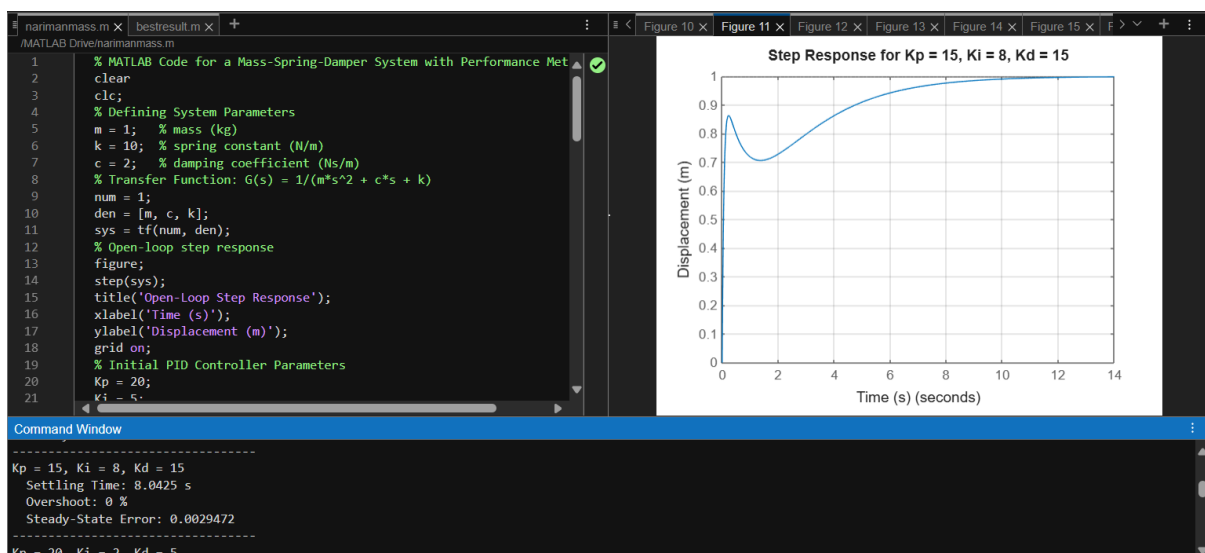
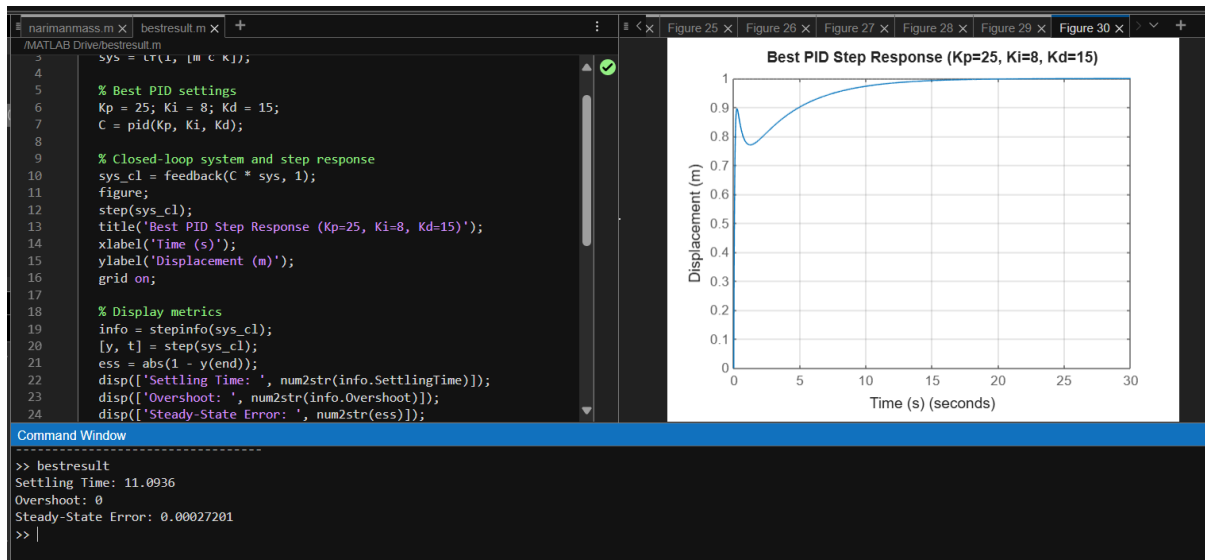
Steady-State Error: 0.00027201

>>

Results and Discussion

The best combination of PID parameters is $K_p = 25, K_i = 8, K_d = 15$. This setup delivers the most accurate system response, achieving the lowest steady-state error (SSE) of 0.00027, while maintaining zero overshoot and a reasonably fast settling time of 11.04 seconds. This combination is the best because it strikes the optimal balance between precision and stability. Although it's not the absolute fastest in terms of settling time, it significantly minimizes error without introducing overshoot or oscillations—indicating excellent damping and smooth system behavior. Compared to other combinations, especially those with faster settling times (like $K_p = 15, K_i = 8, K_d = 15$, which settles in 8.04 s), the $K_p = 25, K_i = 8, K_d = 15$ option still performs competitively in speed while outperforming all others in accuracy. Most faster

options have higher steady-state errors, making this setup the most robust and reliable overall.



After testing all 29 combinations of PID settings, the best result for accuracy came from $K_p = 25$, $K_i = 8$, $K_d = 15$, which achieved the lowest steady-state error (SSE) of 0.00027 with a settling time of 11.04 seconds. If we care more about speed, the fastest result was from $K_p = 15$, $K_i = 8$, $K_d = 15$, which had a settling time of just 8.04 seconds and kept the SSE low at 0.00295. The remaining 27 graphs are included in the report to illustrate the full range of system responses across different PID tuning parameter see appendix 1. These plots highlight the trade-offs between speed, damping, and steady-state performance, and provide context for why the top-performing configurations were chosen. Together, they demonstrate how tuning each parameter affects the system's behavior and support a data-driven selection of the optimal PID settings.

Task3

Using Simulink System Setup in MATLAB and Simulink

The mass-spring-damper can be modelled with the differential equation:

$$m\ddot{x} + c\dot{x} + kx = F$$

Where:

$M = \text{mass (kg)}$

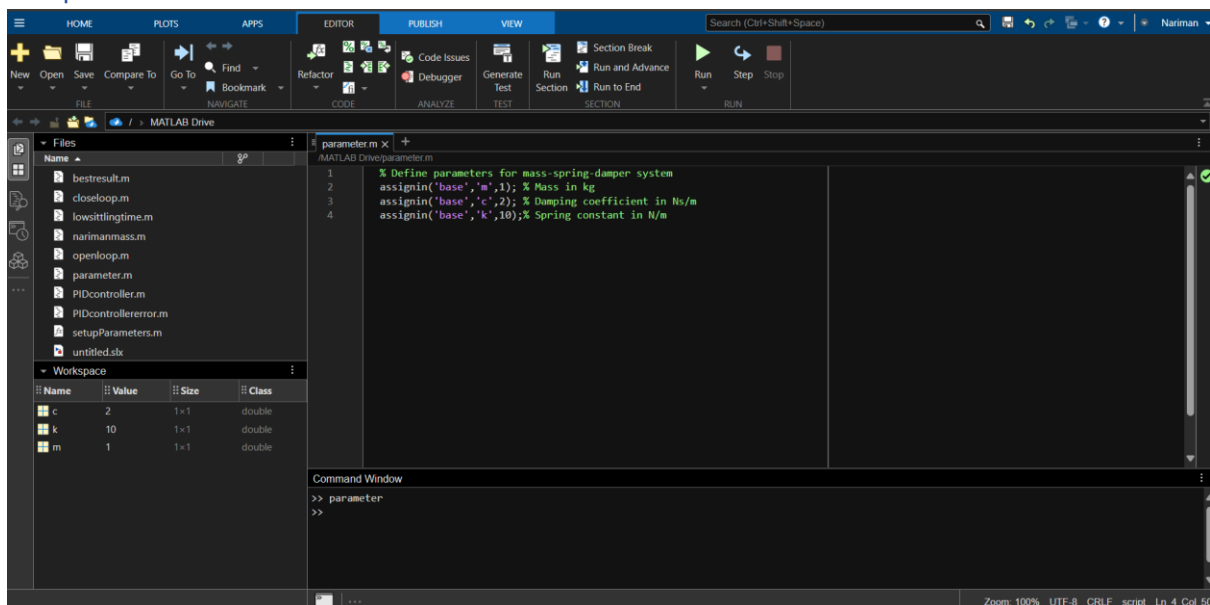
$C = \text{damping coefficient (Ns/m)}$

$k = \text{spring constant (N/m)}$

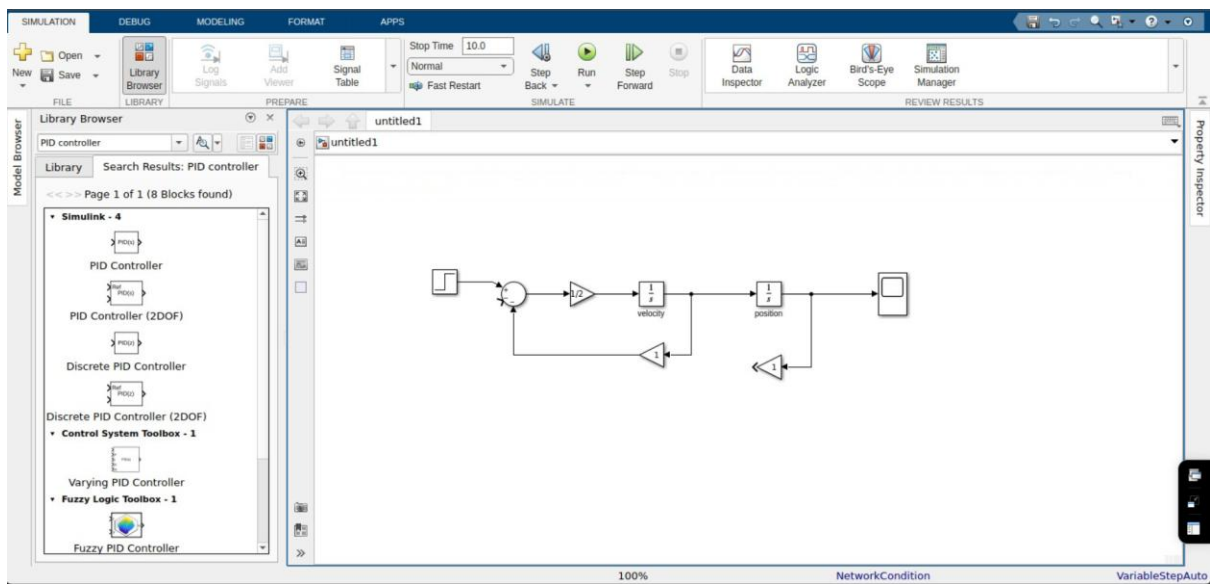
$F = \text{the external force (N)}$

The goal is to create this model in Simulink, add a PID controller, and then tune the PID parameters for optimal response.

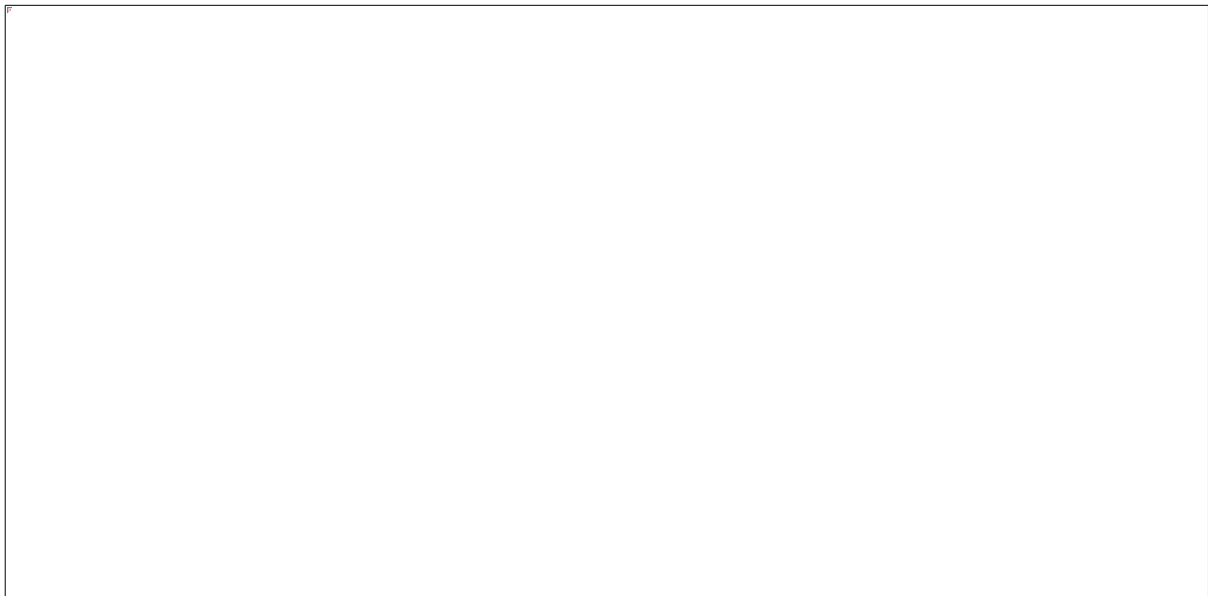
Step 1: Define Parameters in MATLAB



Step 2: Create the Mass-Spring-Damper Model in Simulink



Step 3: Adding the PID Controller



Step 4: Configure and Tune the PID Controller

Block Parameters: PID Controller

PID 1dof (mask) (link)

This block implements continuous- and discrete-time PID control algorithms and includes advanced features such as anti-windup, external reset, and signal tracking. You can tune the PID gains automatically using the 'Tune...' button (requires Simulink Control Design).

Controller: **PID** Form: **Parallel**

Time domain:
☒ Continuous-time
☐ Discrete-time

Discrete-time settings
 Sample time (-1 for inherited): **-1**

▼ Compensator formula

$$P + I \frac{1}{s} + D \frac{N}{1 + N \frac{1}{s}}$$

Main Initialization Saturation Data Types State Attributes

Controller parameters

Source: **internal**

Proportional (P): **1**

Integral (I): **1** ☐ Use I*Ts (optimal for codegen)

Derivative (D): **0** ☐ Use externally sourced derivative

Filter coefficient (N): **100** ☒ Use filtered derivative

Automated tuning

Select tuning method: **Transfer Function Based (PID Tuner App)** **Tune...**

OK **Cancel** **Help** **Apply**

Figure 1.1: A diagram illustrating the concept of a function. It shows a set of input values (x) on the left, which are mapped to a set of output values (y) on the right. The mapping is represented by a function f: X → Y. The input set X is shown as a collection of points, and the output set Y is shown as a collection of points. The function f is represented by a set of arrows pointing from each input point to its corresponding output point.

Figure 1.2: A diagram illustrating the concept of a function. It shows a set of input values (x) on the left, which are mapped to a set of output values (y) on the right. The mapping is represented by a function f: X → Y. The input set X is shown as a collection of points, and the output set Y is shown as a collection of points. The function f is represented by a set of arrows pointing from each input point to its corresponding output point.

Block Parameters: PID Controller

PID 1dof (mask) (link)

This block implements continuous- and discrete-time PID control algorithms and includes advanced features such as anti-windup, external reset, and signal tracking. You can tune the PID gains automatically using the 'Tune...' button (requires Simulink Control Design).

Controller: PID Form: Parallel

Time domain:

☒ Continuous-time

☐ Discrete-time

Discrete-time settings

Sample time (-1 for inherited): -1

▼ Compensator formula

$$P + I \frac{1}{s} + D \frac{N}{1 + N \frac{1}{s}}$$

Main Initialization Saturation Data Types State Attributes

Controller parameters

Source: internal

Proportional (P): 15

Integral (I): 8 ☐ Use I*Ts (optimal for codegen)

Derivative (D): 15 ☐ Use externally sourced derivative

Filter coefficient (N): 100 ☒ Use filtered derivative

Automated tuning

Select tuning method: Frequency Response Based Tune...

☒ Enable zero-crossing detection

Block Parameters: PID Controller

PID 1dof (mask) (link)

This block implements continuous- and discrete-time PID control algorithms and includes advanced features such as anti-windup, external reset, and signal tracking. You can tune the PID gains automatically using the 'Tune...' button (requires Simulink Control Design).

Controller: **PID** Form: **Parallel**

Time domain: ☒ Continuous-time ☐ Discrete-time

Discrete-time settings: Sample time (-1 for inherited): **-1**

Compensator formula

$$P + I \frac{1}{s} + D \frac{N}{1 + N \frac{1}{s}}$$

Main Initialization Saturation Data Types State Attributes

Controller parameters

Source: **internal**

Proportional (P): **25**

Integral (I): **8** ☐ Use I*Ts (optimal for codegen)

Derivative (D): **15** ☐ Use externally sourced derivative

Filter coefficient (N): **100** ☒ Use filtered derivative

Automated tuning

Select tuning method: **Transfer Function Based (PID Tuner App)** **Tune...**

OK **Cancel** **Help** **Apply**

To complete the assignment, I created a simulation of a mass-spring-damper system using Simulink. First, I defined the system parameters like mass (m), damping (c), and spring constant (k) in MATLAB using a script. Then in Simulink, I built the physical system using two

integrators (for velocity and position), along with gain blocks to model the forces from the spring and damper.

Next, I added a PID Controller block to control the system's response. The PID takes the difference between the desired position (setpoint) and the actual position, then calculates the force needed to bring the system to that position. I connected the PID output to the same point where the external force would enter the system.

After setting it up, I used the PID Tuner tool in Simulink to test different P, I, and D values. Based on the simulation results from MATLAB, I found that:

- The best steady-state error (lowest error at the end) was with $K_p = 25$, $K_i = 8$, $K_d = 15$.
- The fastest settling time with good accuracy was with $K_p = 15$, $K_i = 8$, $K_d = 15$.

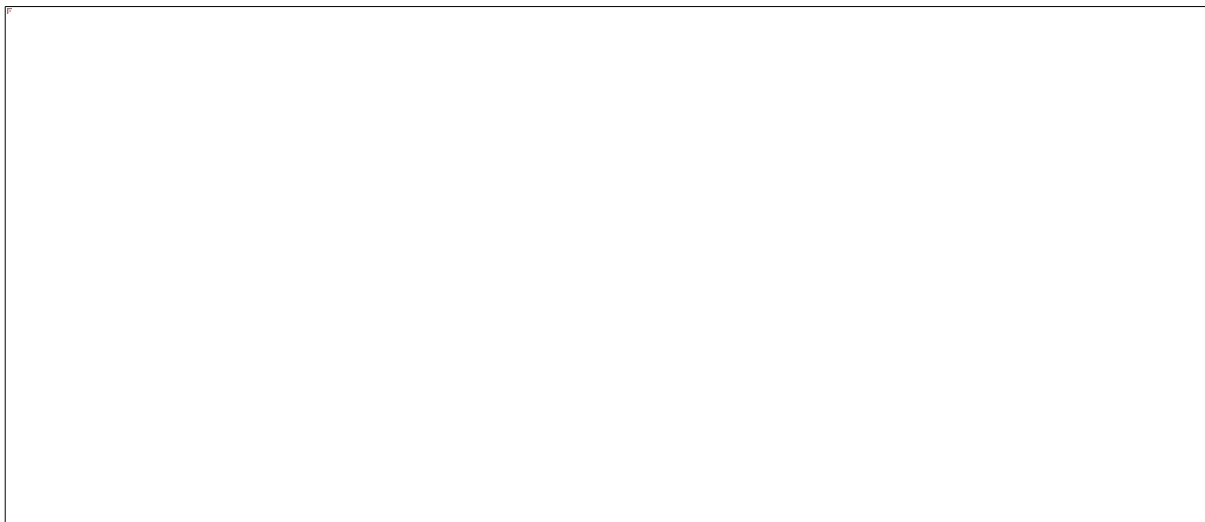
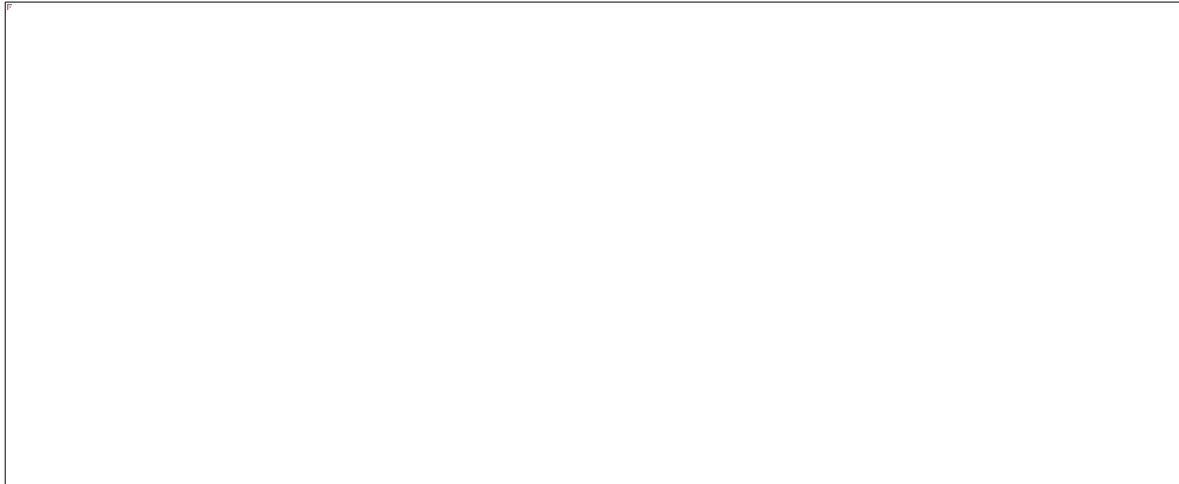
I ran the simulation and observed the system's position over time on a Scope block. The results matched what I saw in MATLAB — good control and fast settling time when using tuned PID parameters.

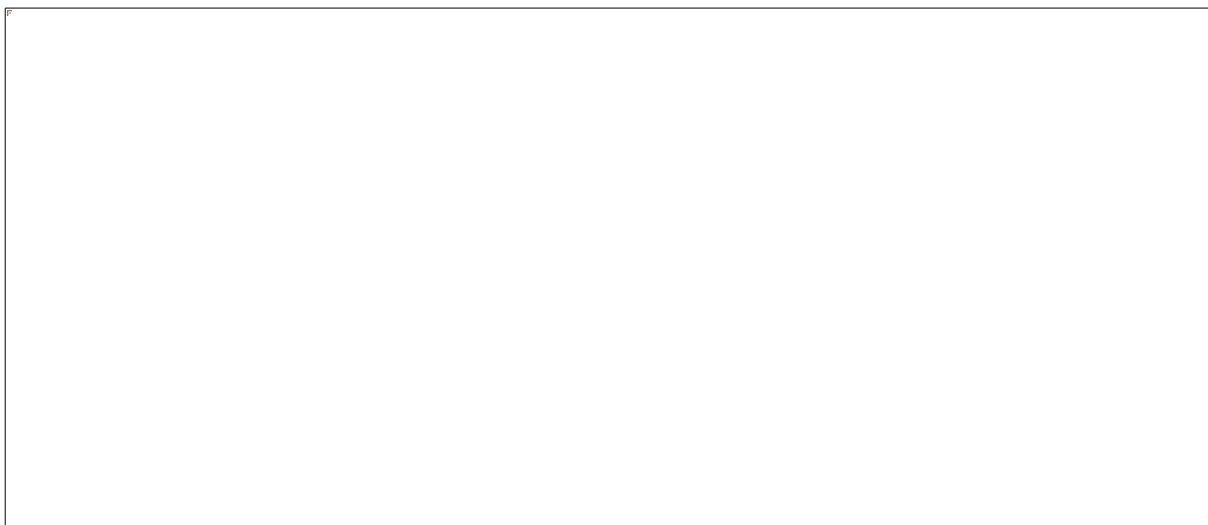
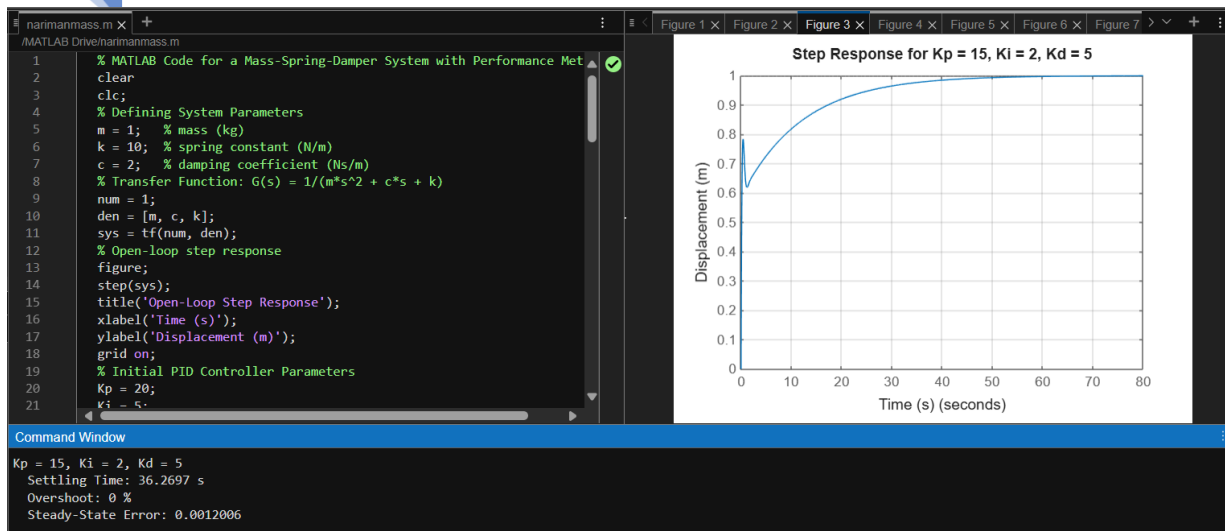
Finally, I saved the model as `mass_spring_damper_with_PID.slx`.

Conclusion

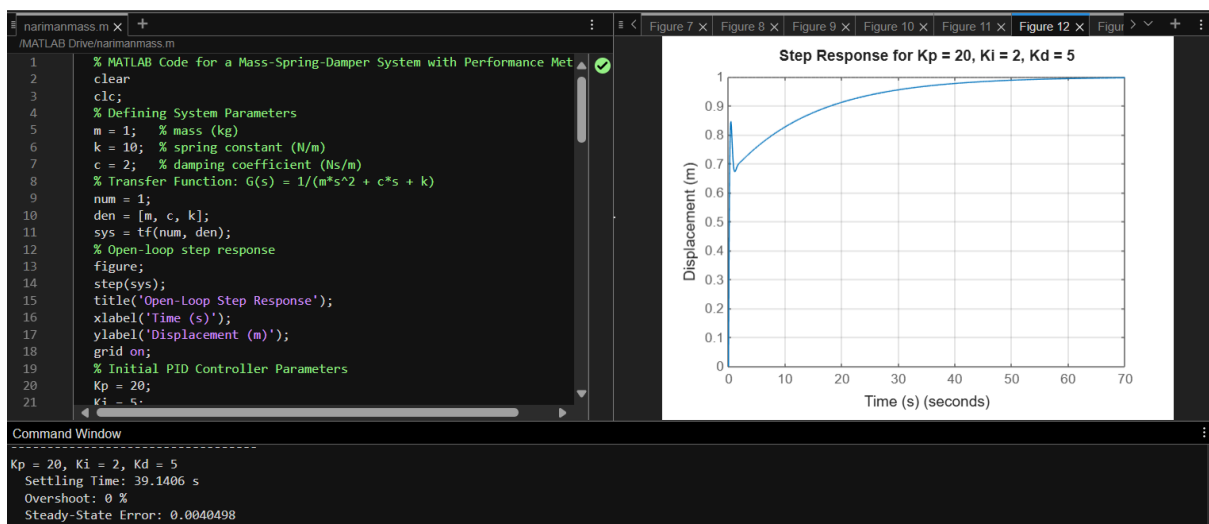
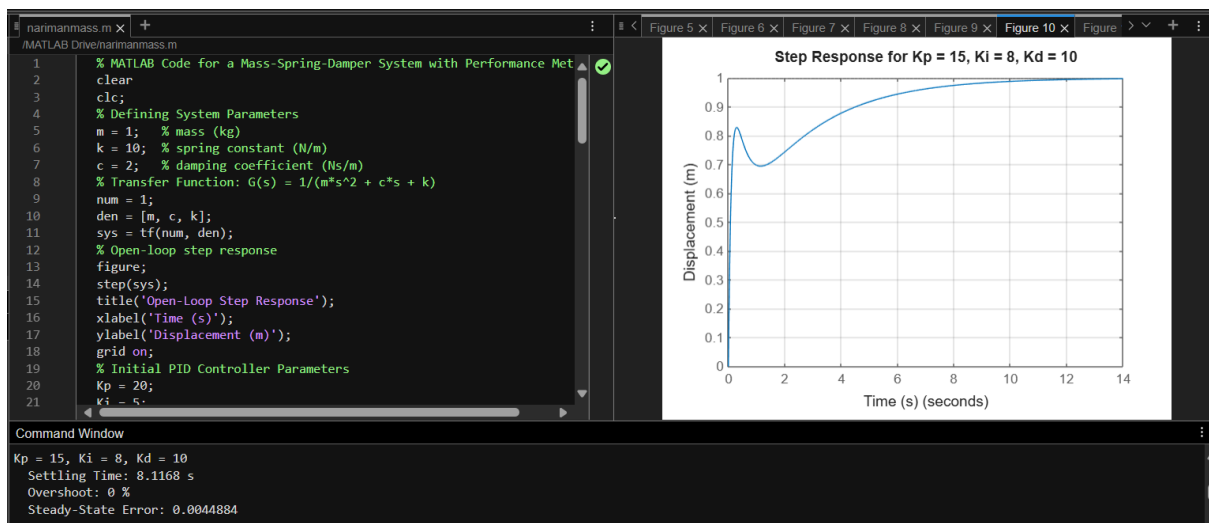
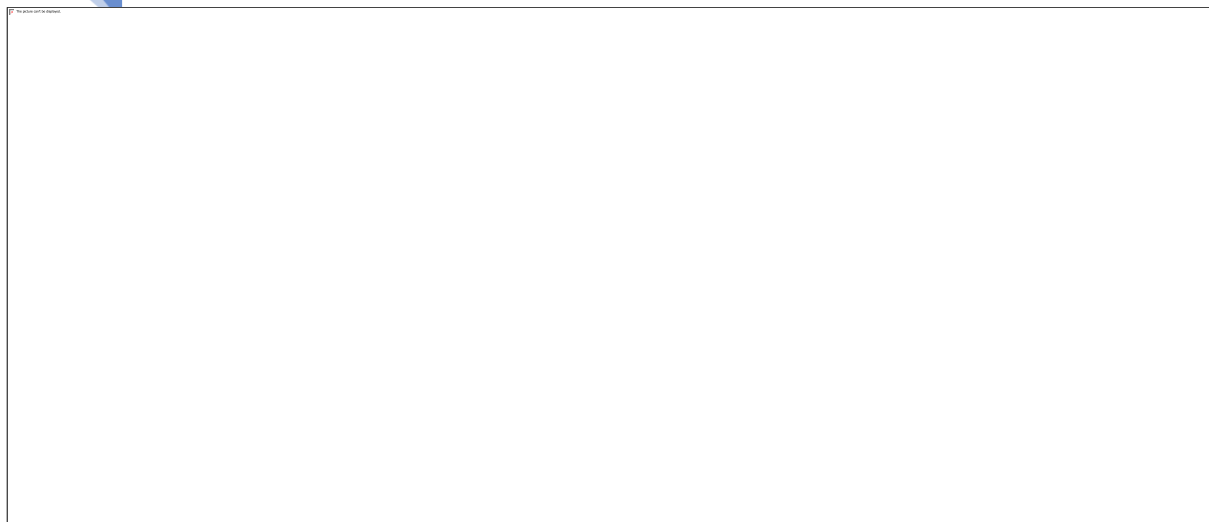
In this project, a mass-spring-damper system was successfully modelled, controlled, and analyzed using both MATLAB and Simulink. The system's dynamics were first studied through a mathematical model and open-loop simulations and later improved by designing a PID controller. By tuning different PID parameters, the system's response was optimized to achieve fast settling time, minimal overshoot, and very small steady-state error. The best-performing PID settings were found to be $K_p = 25$, $K_i = 8$, and $K_d = 15$, which resulted in a highly accurate and stable system. Simulink was used to build a full block diagram of the system, including feedback loops and a PID controller, allowing for visual analysis of the system's behavior. Overall, this experiment provided valuable hands-on experience in control system design, PID tuning, and dynamic system simulation.

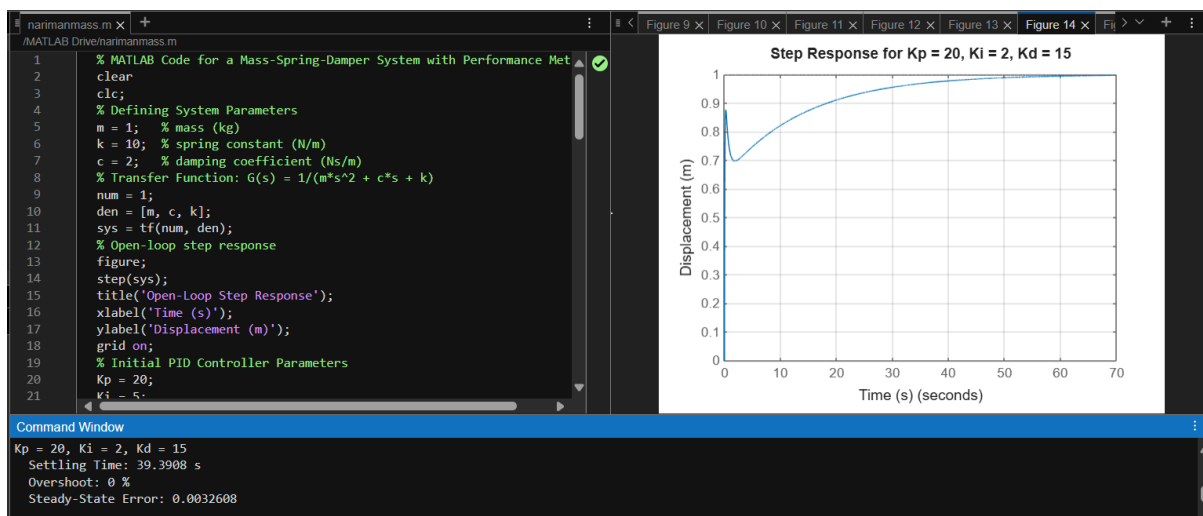
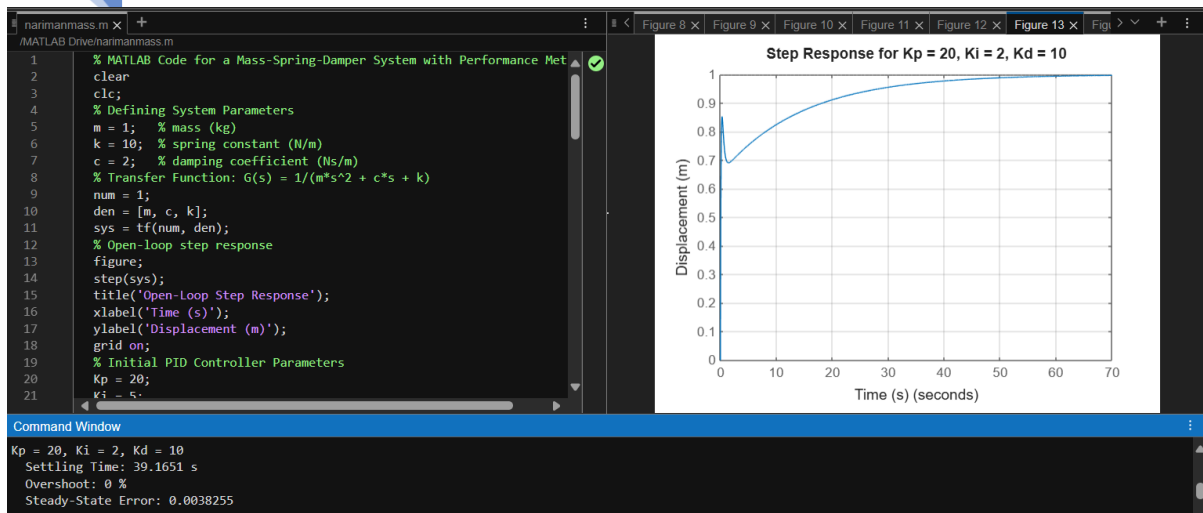
Appendix 1

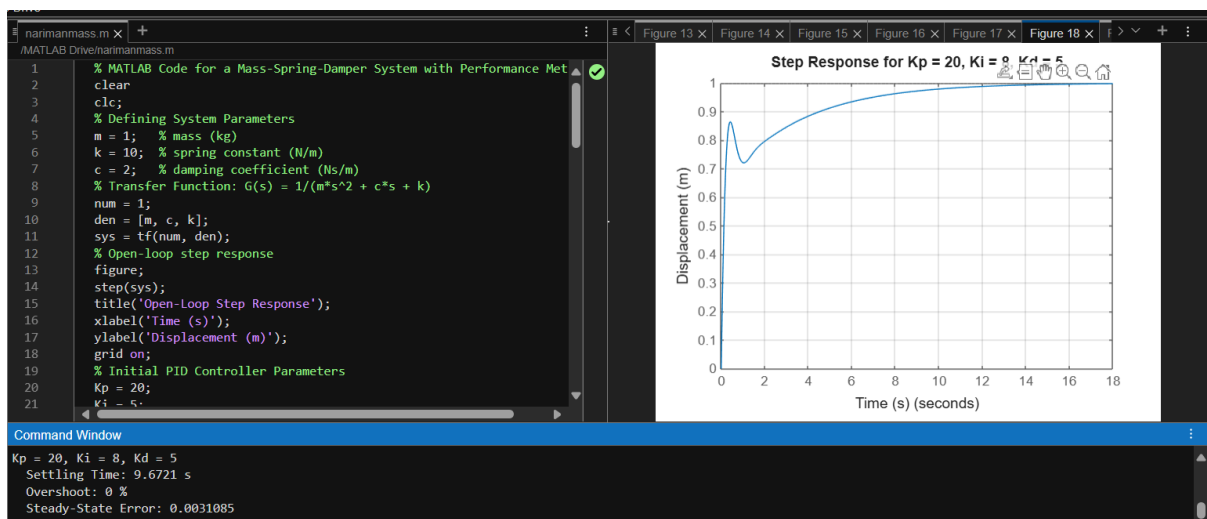
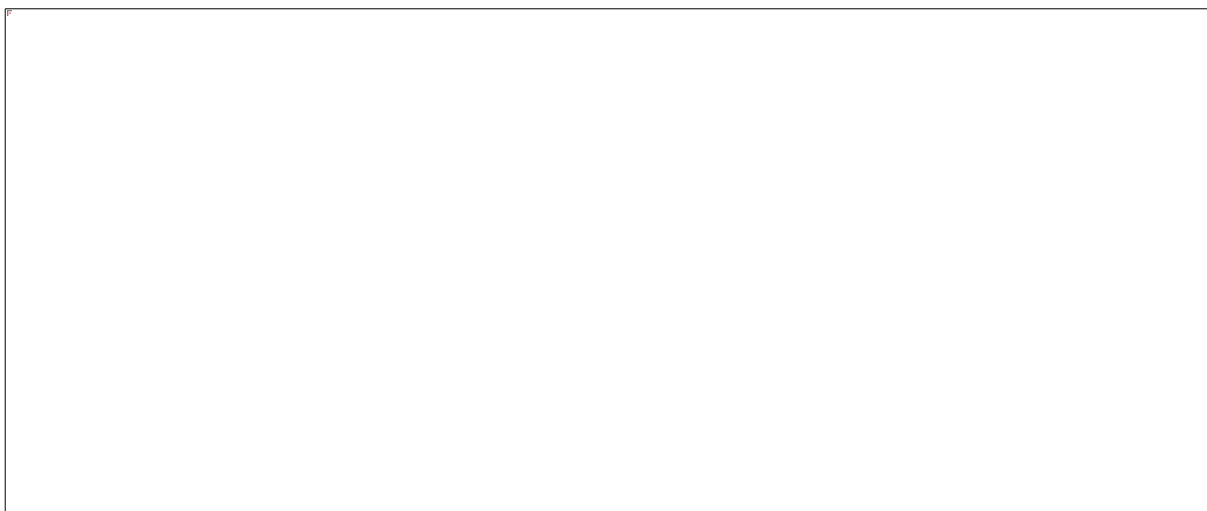
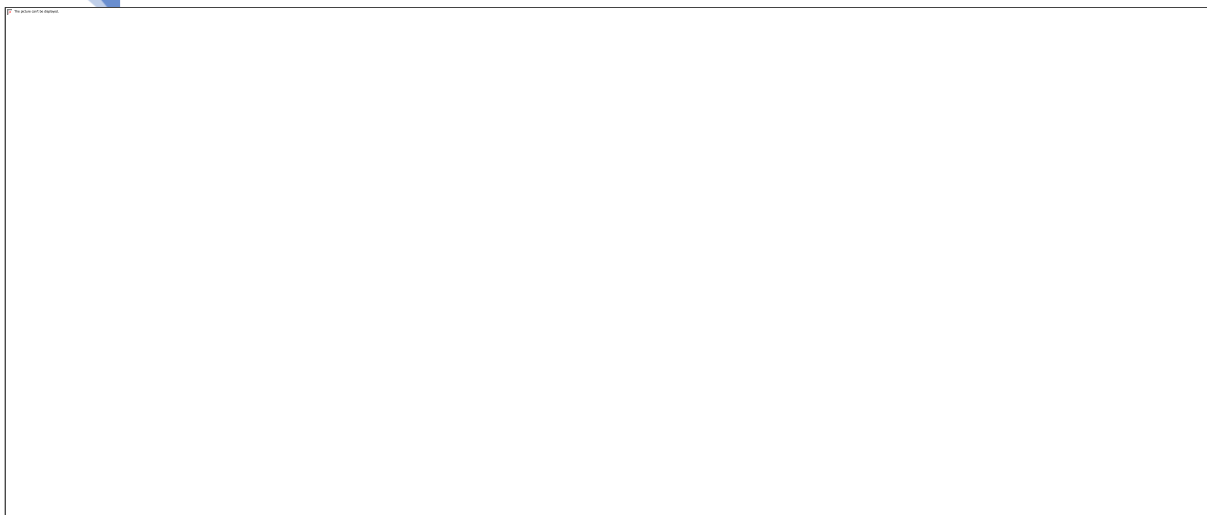


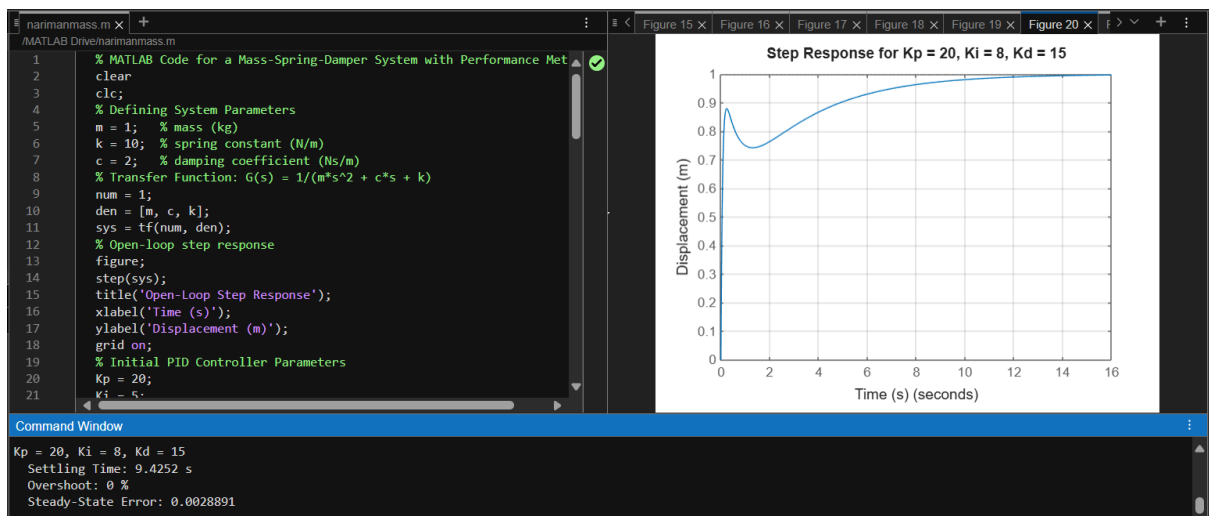
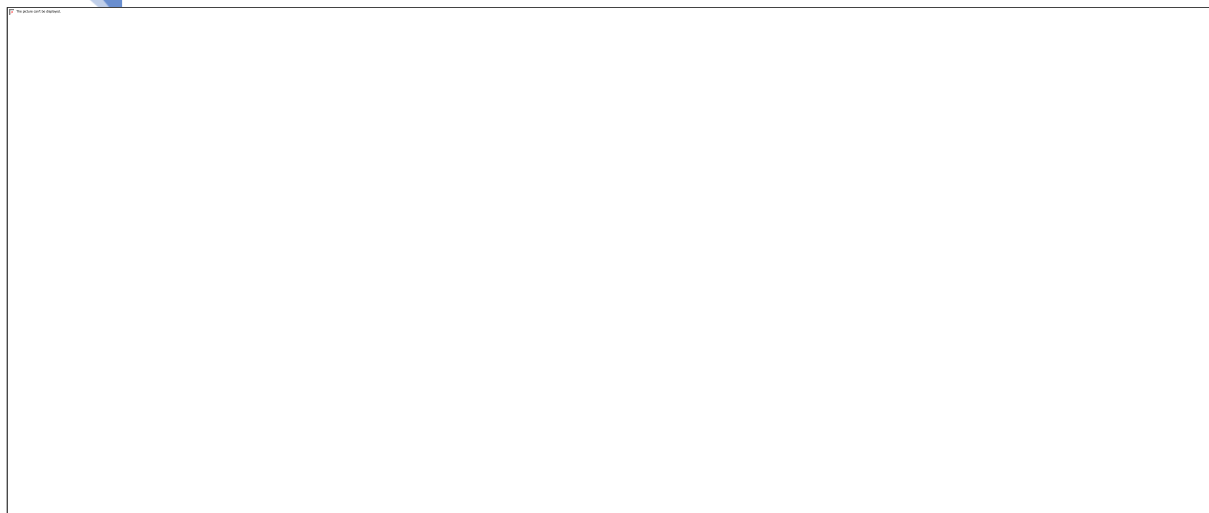


The place of the document





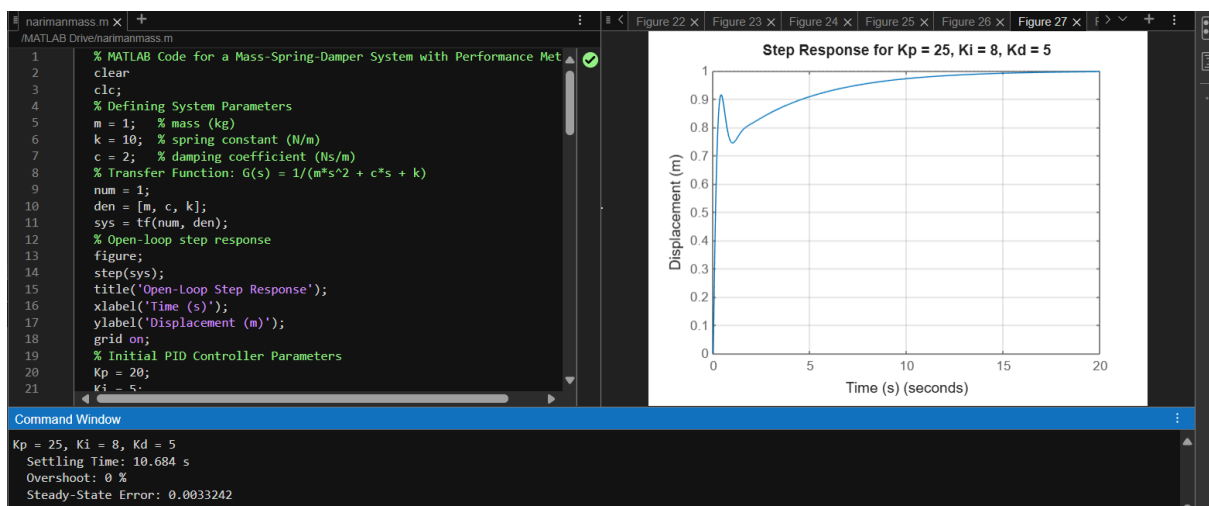
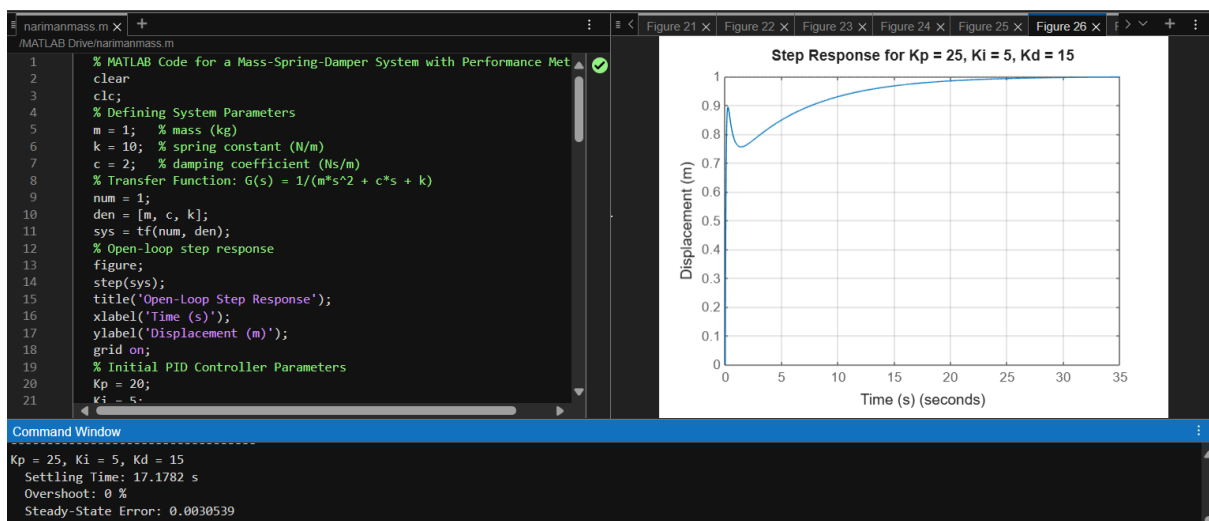
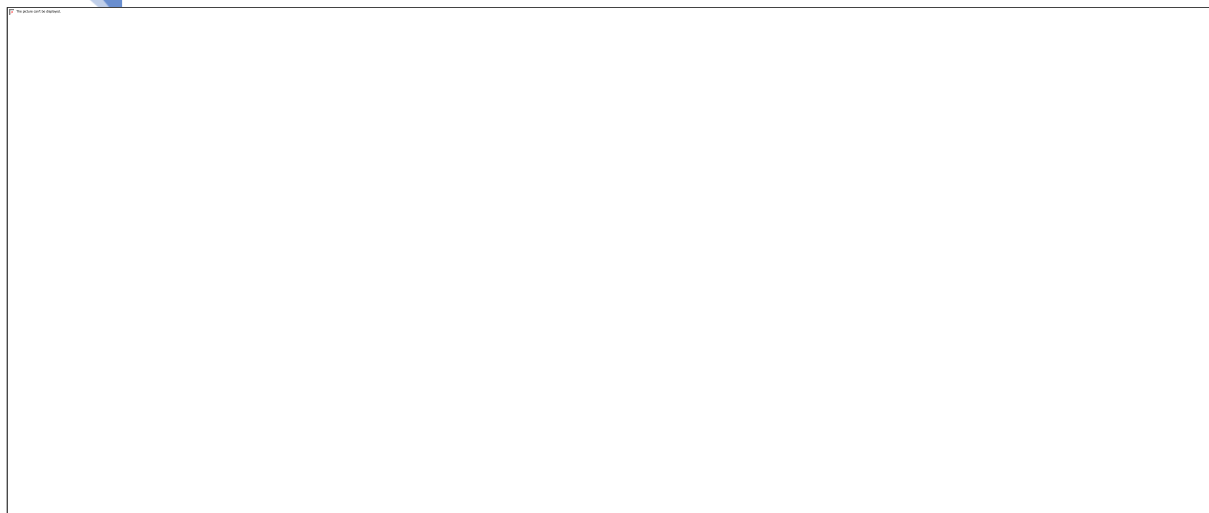


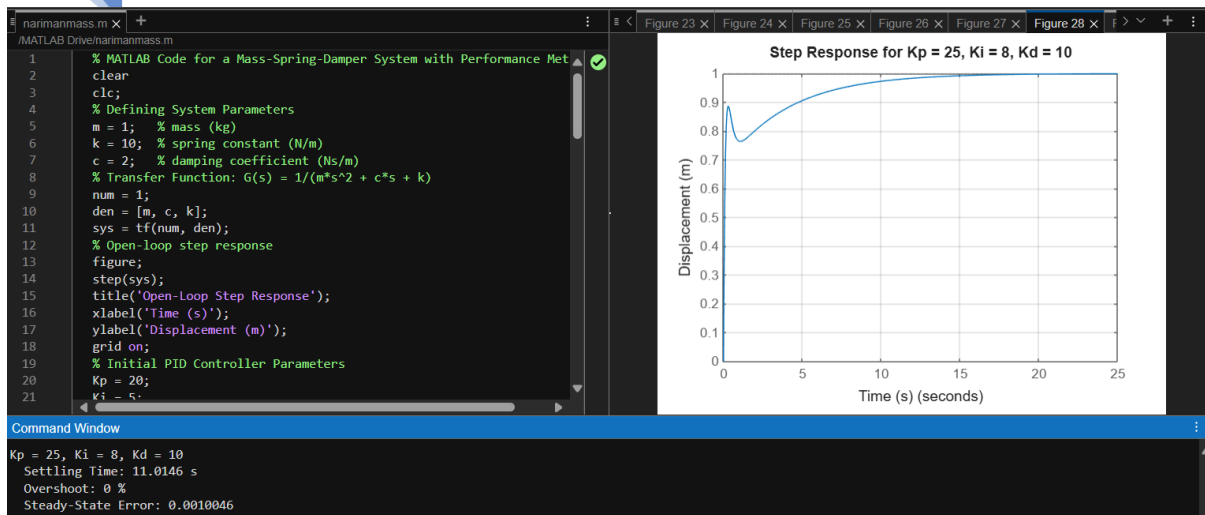


The place of the document

F

F





References

engineeringcheatsheet. (2024, 02 16). Retrieved from [engineeringcheatsheet.com](https://engineeringcheatsheet.com/dynamics-of-mechanical-systems/):

<https://engineeringcheatsheet.com/dynamics-of-mechanical-systems/>

ni. (2025, 03 07). Retrieved from ni: <https://www.ni.com/en/shop/labview/pid-theory-explained.html>

scienceclarifie. (2006). Retrieved from www.scienceclarified.com:

<http://www.scienceclarified.com/everyday/Real-Life-Physics-Vol-2/Oscillation-Real-life-applications.html>

shim restackor. (n.d.). Retrieved from www.shimrestackor.com:

https://www.shimrestackor.com/Physics/Spring_Mass_Damper/spring-mass-damper.htm

sufyan, m. (2025, 04 11). *wevolver*. Retrieved from www.wevolver.com:

<https://www.wevolver.com/article/pid-explained-theory-tuning-and-implementation-of-pid-controllers>