YILDIZ TECHNICAL UNIVERSITY

# Homework

## Head Stabilisation Project with DC Motor

**20016928 – Aylin ÖZTÜRK**
**20016052 – Çağla ÜZÜMCÜ**

**5/13/2025**

KOM4620 Real Time Control Systems

# 1 Introduction

# Problem approach and Objectives

## 1.1. Project Overview

The primary objective of this project is to design and implement a DC motor-based control system capable of stabilizing a box-shaped head in three-dimensional (3D) space. The system aims to counteract disturbances and maintain a steady orientation of the head using real-time feedback from sensors and model-based control algorithms. Such a system has practical applications in robotics, camera stabilization platforms, and medical assistive devices where maintaining orientation is critical.

To achieve this, we integrate multiple subsystems, including motor drivers, sensors (encoders and accelerometers), and a microcontroller to manage data acquisition and control logic. The system processes incoming sensor data, computes the required response using a control algorithm, and sends appropriate commands to the motor to adjust the head's position.

## 1.2. Key Aspects of the Project

### 1.2.1. Sensor Integration

Accurate sensing of the position and movement of the head is one of the basic necessities for accurate control. For this purpose, the system incorporates several sensors:

The encoders measure the rotation position and velocity of the shaft of the DC motor. These are used to calculate the present angle and velocity of the head mechanism. An Inertial Measurement Unit (IMU) such as the MPU9250 is employed to measure 3D angular acceleration and orientation on the X, Y, and Z axes.

This provides the control system with the knowledge of how the head is moving in space and determine sudden disturbances or drifts in orientation.

Sensor data is processed in real-time and used as input for the control algorithm, causing the platform to respond quickly to rotations or motions from the outside.

### 1.2.2. Control System

For the implementation of smooth and precise stabilization, feedback control algorithm is applied. The basic intention of the controller is to identify by how much and in what direction the motor needs to rotate so as not to produce any undesired motion of the head.

Various methods of control could be used, i.e., P (Proportional), PI (Proportional-Integral), PID (Proportional-Integral-Derivative), or even better ones like PIV (Proportional-Integral with Velocity feedback) controllers.

These controllers compute the corrective motor commands through utilization of the error between current and target positions (and velocities).

One can improve the response of the system in order to reduce overshoot, settling time, and steady-state error through tuning of the control gains.

The above control logic can be coded in MATLAB/Simulink or programmed into the microcontroller which is Arduino Mega2560.

### 1.2.3. Real-Time Feedback and Response

Pulse Width Modulation (PWM) signals are generated to control the voltage (and thus speed) applied to the DC motor. In addition, the motor's physical behavior, including Coulomb friction, torque-speed characteristics, and inertia, are experimentally identified and modeled for more accurate control**.**

The effectiveness of the stabilization system depends greatly on its ability to operate in real time. Real-time response means:

-Continuously reading sensor information (angular position, speed, acceleration),

-Calculating the necessary motor response via the control algorithm in a short time,

-Updating the motor output without any noticeable delay.

This closed-loop system allows the controller to dynamically adjust the motor's position in such a way that even if any external force or vibration is applied to the head, it remains stable or quickly returns to its reference position.

# 2 Implementation

## 2.1 Subheading: explain the real-time system design implementation

### 2.1.1. Setting Circuit and Application

In this project, the primary goal is to achieve real-time control of a DC motor's position and speed using feedback from an encoder. The implementation is based on an Arduino Mega2560 microcontroller and an L298N dual H-bridge motor driver, forming the core of the motor control circuit.

### 2.1.2. Working Principle and Real-Time Feedback Loop

1. Encoder Signal Reading: The encoder connected to the motor shaft sends quadrature pulses (A and B signals), which are read via external interrupts on the Arduino. The difference in pulse counts over time is used to compute the motor speed, while the total count gives the position.

2. PWM Signal Generation: Based on the desired reference speed or position and the actual measured values, a control algorithm (usually PID) calculates the control output. This output is converted to a PWM signal that adjusts the motor's input voltage and thus controls its speed.

3. Motor Direction Control: Direction is determined using the two digital inputs of the L298N module. By setting one input HIGH and the other LOW, the motor rotates in a specific direction. Reversing these signals changes the rotation direction.

4. Real-Time Control Loop Execution:

- At fixed intervals, the Arduino executes the control loop:
    - Reads encoder values
    - Calculates speed and/or position error
    - Computes the PID output
    - Updates the PWM signal
- This loop ensures timely response to disturbances and changes in reference input.

5. Data Logging and Monitoring: For analysis, encoder readings, PWM outputs, and control errors are sent to the PC via serial communication and visualized using MATLAB or Arduino Serial Plotter.

### 2.1.3. Pin Connections

| | Description | Arduino Pin |
|---|---|---|
| IN1 | **L298N Pin** | Pin 6 |
| IN2 | Motor direction control | Pin |
| ENA | Motor speed control (PWM) | Pin 5 (PWM capable) |
| GND | Ground | GND |
| VCC | Motor power (12V input) | External 12V power supply |
| OUT1/OUT2 | Motor terminals | Connected to motor |

| **Encoder Wire** | **Function** | **Arduino Pin** |
|---|---|---|
| A (Channel A) | Encoder signal | Pin3(Interrupt ) |
| B (Channel B) | Encoder signal | Pin 2 |
| VCC | Encoder power | External 12V |
| GND | Encoder ground | GND |

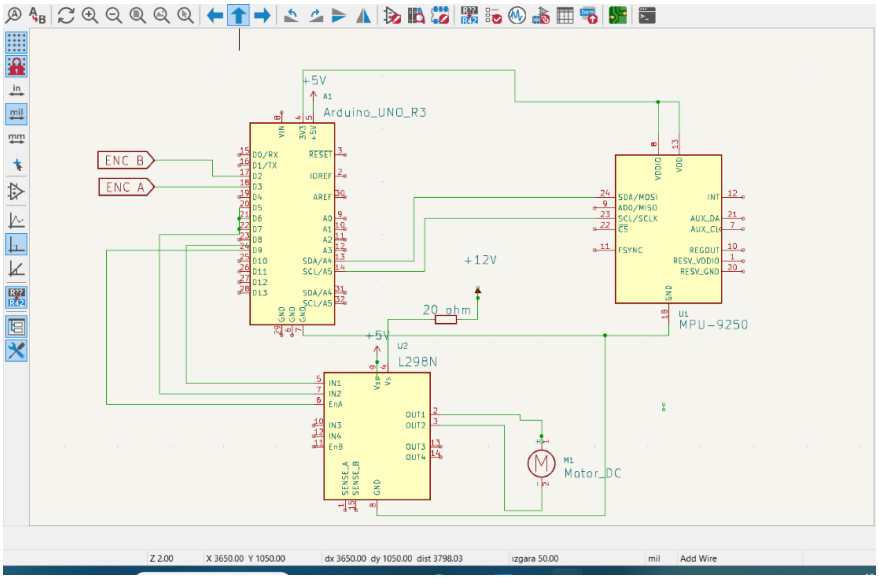



Figure : Schematics of overall system in KiCAD

Real-time systems implementation is shown as follows:

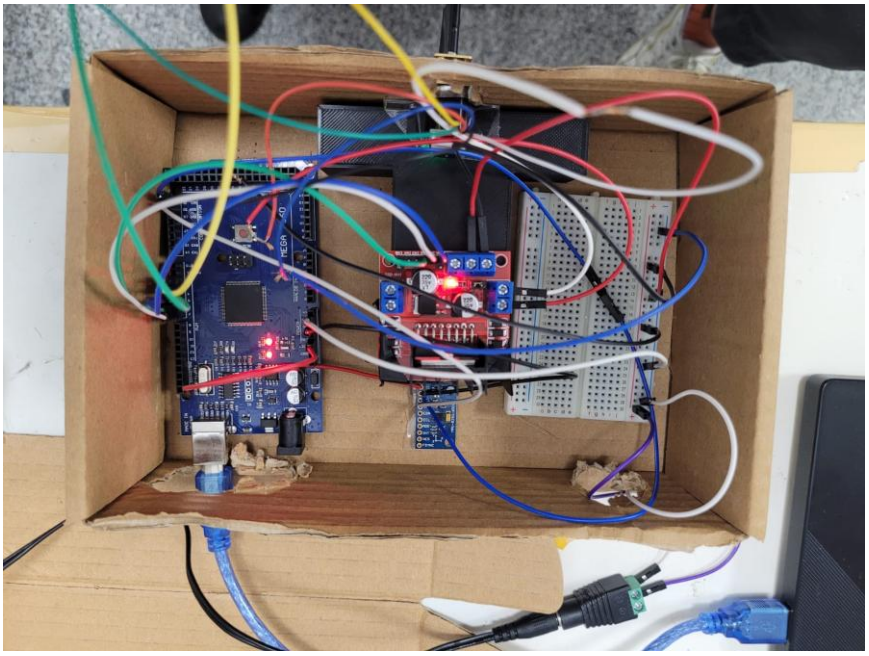


Figure : Real System implementation figure 1

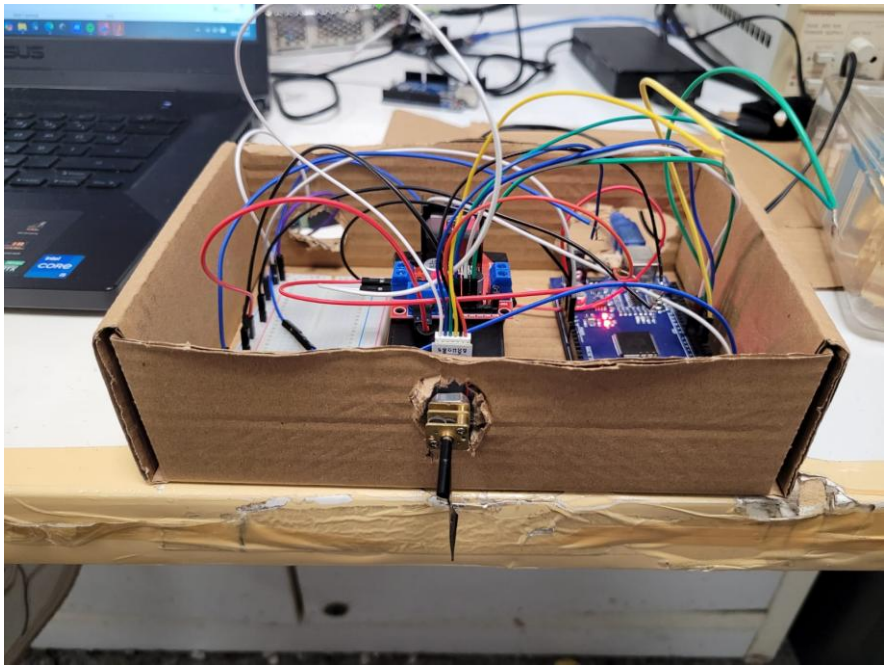


Figure: Real System implementation figure 2

## 2.2 Subheading: system dynamics and control

### Reduced DC Motor Modeling and Transfer Function Representation

A DC (Direct Current) motor is a widely used actuator in control systems, known for its simplicity and ability to provide a linear relationship between input voltage and output speed under certain assumptions. To model the behavior of a DC motor, both its electrical and mechanical components are considered

The DC motor reduced dynamics:

$$\Gamma = KV = J\ddot{\theta} + v\theta + \tau^{nlf}$$

where

Γ: Motor torque.

K: Motor constant.

v: Viscous friction.

J: Motor inertia.

$\tau^{nlf}$ : Static Coulomb friction.

V: Input (voltage).

By applying the Laplace Transform (assuming zero initial conditions), we can derive the transfer function between the input voltage V(s) and the output angular velocity ω(s):

$$KV(s) = Js^2\theta(:) + v\theta(s)$$

$$\frac{\theta(s)}{V(s)} = \frac{\frac{k}{J}}{s(s+\frac{v}{J})} = \frac{A}{s(s+B)}$$

$$A = \frac{k}{J} \quad B = \frac{v}{J}$$

## State-space Representation of a DC Motor

$$KV = J\ddot{\theta} + v\dot{\theta}$$

$$\ddot{\theta} = \frac{kV}{J} - \frac{v}{J}\dot{\theta}$$

The general form of a state-space model is:

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

where

*A: State Matrix*

*B: Input Matrix*

*C: Output Matrix*

*D: Direct Feedthrough Matrix*

By considering the state vector $x = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}$ and $\dot{x} = \begin{bmatrix} \dot{\theta} \\ \ddot{\theta} \end{bmatrix}$, and with the input being the voltage *V* and the output being the angular speed $\dot{\theta}$: the following state and output equations are obtained:

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{v}{J} \end{bmatrix} x + \begin{bmatrix} 0 \\ \frac{k}{J} \end{bmatrix} V(t)$$

$$y = [0 \quad 1]x + [0]V(t)$$

This state-space representation can be simulated in Simulink using the following diagram. But in our case, there is not any relationship between the input and the output, and therefore, *D*=0:

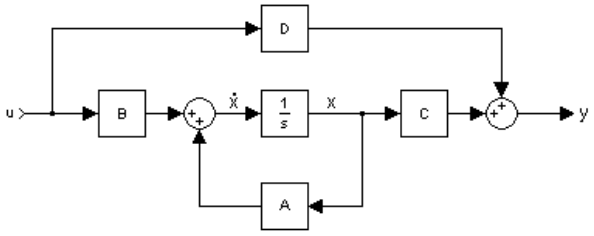



*Figure :State – space model block diagram*

Other important issue is the Motor friction's and disturbances. There is main approach with which to deal with the compensation of the overall disturbance which is the cancellation based on a model of the total disturbance, using its estimation and compensation [1].

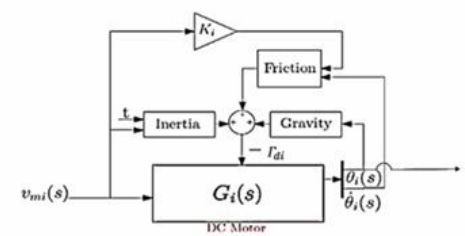



Figure: DC motor dynamics under disturbance [1]

**Real Time Implementation**

In the real-time implementation, the process starts with system identification based on experimental input-output data obtained from the physical system. Initially, rough estimates for the parameters A and B are assigned. Using the Parameter Estimation tool in Simulink, these parameters are iteratively tuned to minimize the error between the simulated and measured outputs. Once accurate values of A and B are obtained, the resulting dynamic model becomes suitable for controller design.

To characterize the motor's steady-state behavior, PWM inputs were applied from -1 to 1 in steps of 0.05 using Arduino Mega2560. For each PWM command, the motor speed was measured using a rotary encoder. The raw data was then filtered to remove non-monotonic points, allowing the creation of a clean 1D lookup table.

### 1. Create Test Data

```
maxPWM = 1.00;
incrPWM = 0.05;
PWMcmdRaw = (-maxPWM:incrPWM:maxPWM)';
```

### 2. Initialize Arduino and Motor Setup

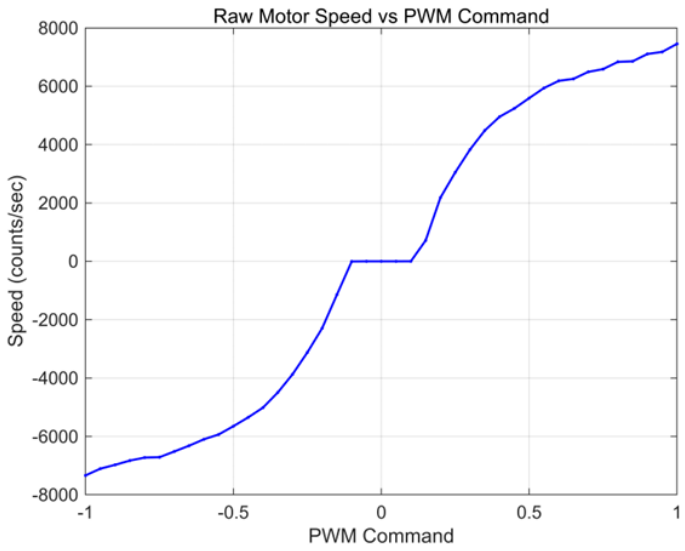
```
clear a encoderObj
a = arduino('COM5', 'Uno', 'Libraries', 'RotaryEncoder');

in1 = 'D8';
in2 = 'D7';
ena = 'D6';

configurePin(a, in1, 'DigitalOutput');
configurePin(a, in2, 'DigitalOutput');
configurePin(a, ena, 'PWM');

encoderObj = rotaryEncoder(a, 'D3', 'D2');
resetCount(encoderObj);
```

### 🔹 3. Measure Raw Motor Speed

```
speedRaw = zeros(size(PWMcmdRaw));

for ii = 1:length(PWMcmdRaw)
    resetCount(encoderObj);
    pwmVal = abs(PWMcmdRaw(ii));

    if PWMcmdRaw(ii) >= 0
        writeDigitalPin(a, in1, 1);
        writeDigitalPin(a, in2, 0);
    else
        writeDigitalPin(a, in1, 0);
        writeDigitalPin(a, in2, 1);
    end

    tic;
    writePWMVoltage(a, ena, pwmVal * 5);
    pause(1);
    dt = toc;

    count = readCount(encoderObj);
    speedRaw(ii) = count / dt;

    writePWMVoltage(a, ena, 0);
    pause(1);
end
```

### 🔹 4. Plot Raw Speed Data

```
figure;
plot(PWMcmdRaw, speedRaw, 'b.-', 'LineWidth', 1.2);
grid on;
title('Raw Motor Speed vs PWM Command');
xlabel('PWM Command');
ylabel('Speed (counts/sec)');
```



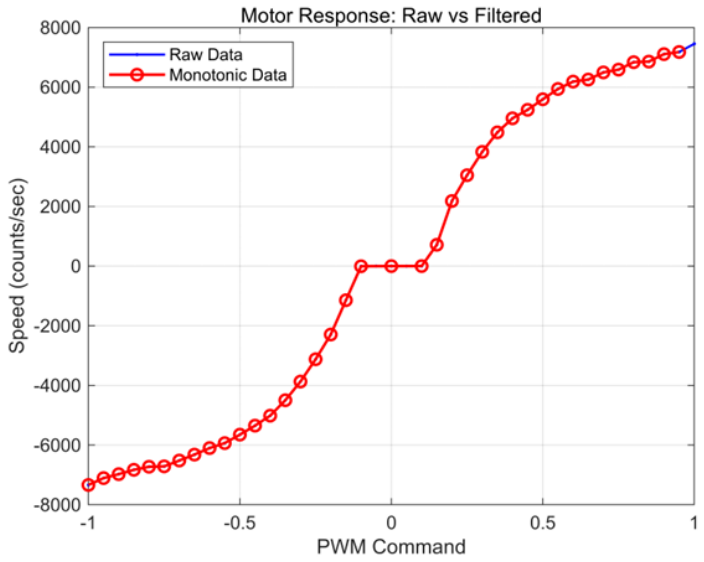
### 🔹 5. Filter Monotonic Data

```
idx = (diff(speedRaw) > 0);
speedMono = speedRaw(idx);
PWMcmdMono = PWMcmdRaw(idx);

PWMcmdMono(speedMono == 0) = 0;

save motorResponse PWMcmdMono speedMono
```

### 🔹 6. Plot Raw vs Filtered Data

```
figure;
plot(PWMcmdRaw, speedRaw, 'b.-', 'DisplayName', 'Raw Data', 'LineWidth', 1.2);
hold on;
```

```
plot(PWMcmdMono, speedMono, 'ro-', 'DisplayName', 'Monotonic Data', 'LineWidth',
1.5);
grid on;
title('Motor Response: Raw vs Filtered');
xlabel('PWM Command');
ylabel('Speed (counts/sec)');
legend('Location', 'northwest');
```




### 🔹 7. Save Final Data to File

```
save(mfilename, 'PWMcmdMono', 'speedMono')
```

### 🔹 8. Cleanup

```
clear a encoderObj
```

In the real-time Simulink model, a pulse generator is used to provide periodic reference speed commands. These commands are fed into a 1D Lookup Table block, which maps the desired speed values to appropriate PWM commands based on previously identified experimental data (PWMcmdMono and speedMono). The mapped PWM signal is then sent to the motor block, which

interfaces with the physical hardware. The encoder measures the motor's position in real time, from which the actual speed is derived using a discrete differentiation block.

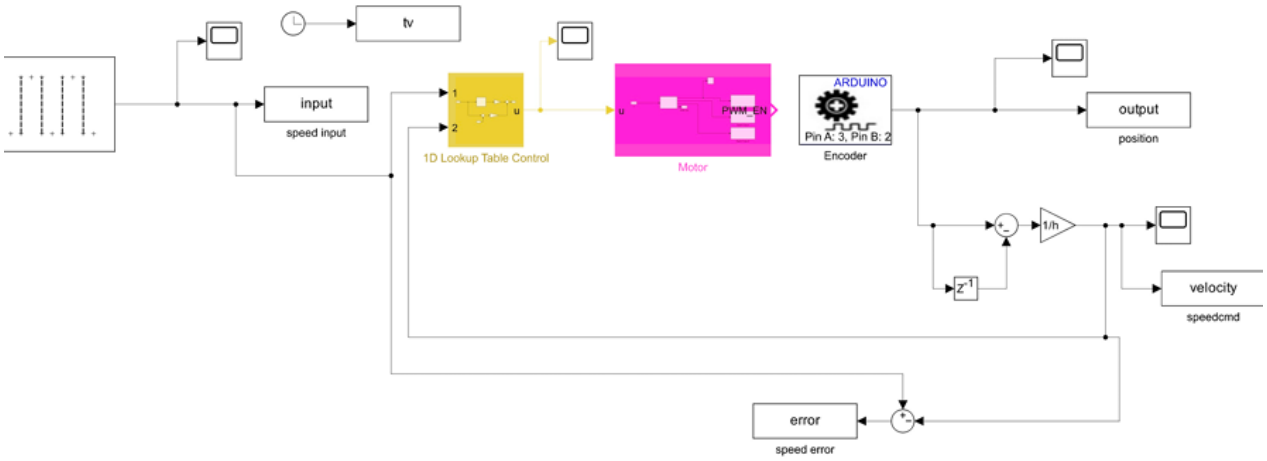



Figure : Real-Time Motor Speed Control Using 1D Lookup Table in Simulink

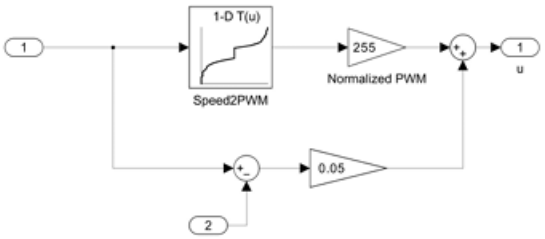



Figure : 1D Lookup Table Control

```
function [IN_PWM,INA1,INA2,PWM_EN] = fcn(u)

PWM_EN = u;
IN_PWM=abs(u);
if u<0
    INA1=0;
    INA2=1;
elseif u>0
    INA1=1;
    INA2=0;
else
    INA1=0;
    INA2=0;
end
```

Figure: fcn

**DC Motor Parameter Estimation (Pololu Motor)**

The motor's parameters A and B were identified using Simulink's Parameter Estimator tool. The measured velocity and the input duration (tv) were used for comparison. A dead zone block (±0.2) was added to account for motor nonlinearity. The estimation was done using the Nonlinear Least Squares method with:

- Max iterations: 100
- Parameter tolerance: 0.001 Accurate values of A and B were obtained for use in controller design.

Accurate values of A and B were obtained for use in controller design.

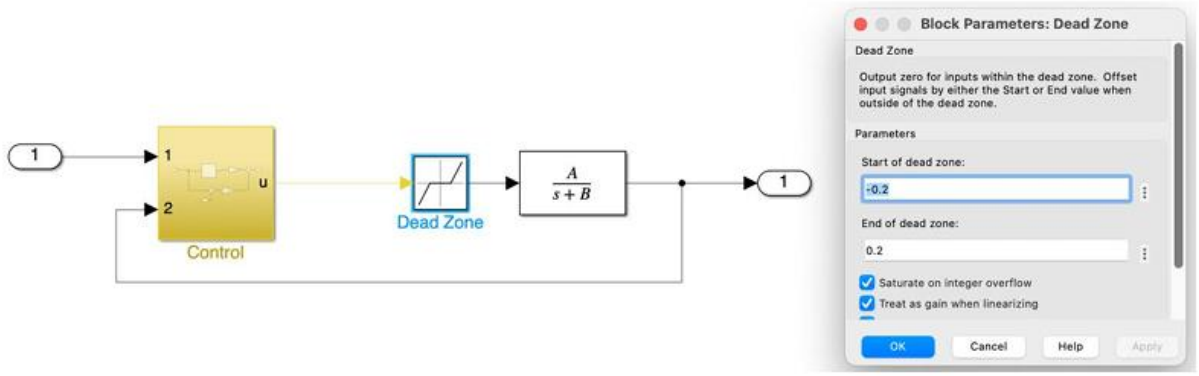



Figure: Parameter Estimation Model for A and B Using Simulink

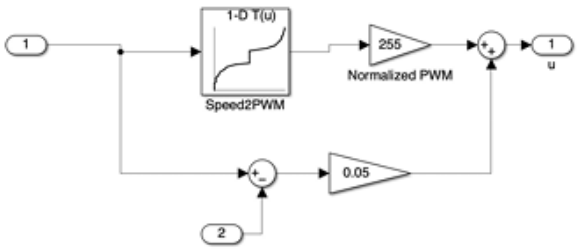



Figure: Control

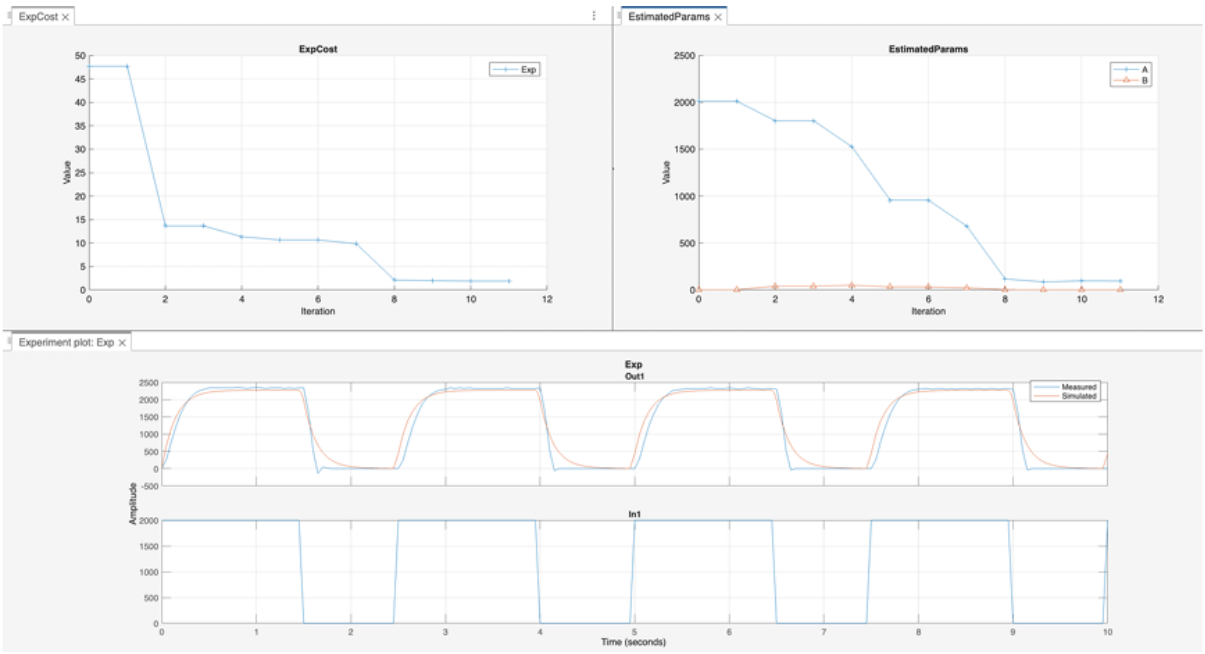



Figure: System Identification/ Parameter Estimation and Simulation Results

| Parameter | Initial Value | Estimate |
|---|---|---|
| A | 95.3222414923339 | Yes |
| B | 2.49510644698831 | Yes |

Figure: Estimated Parameter Values

## 2.3 Subheading: system results

### Open Loop Speed Controller

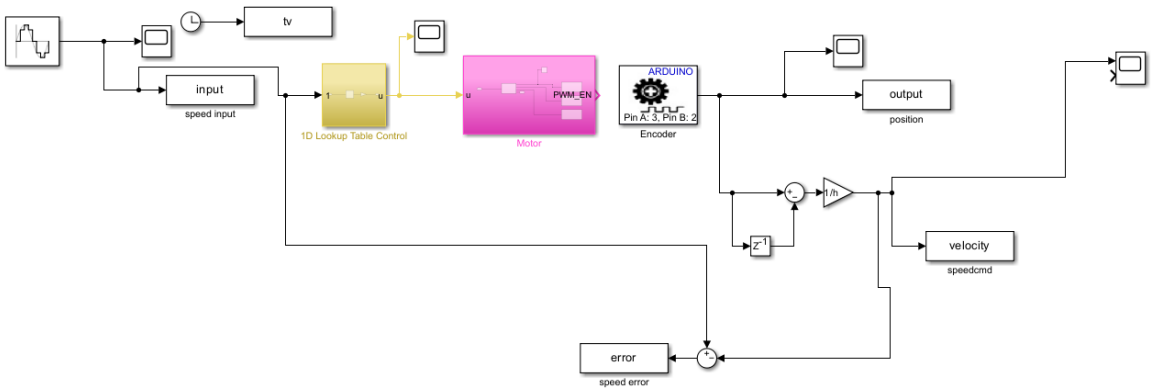
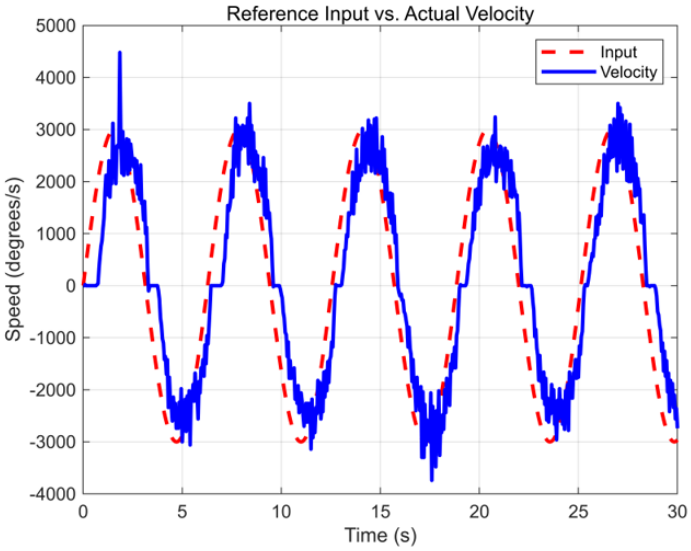
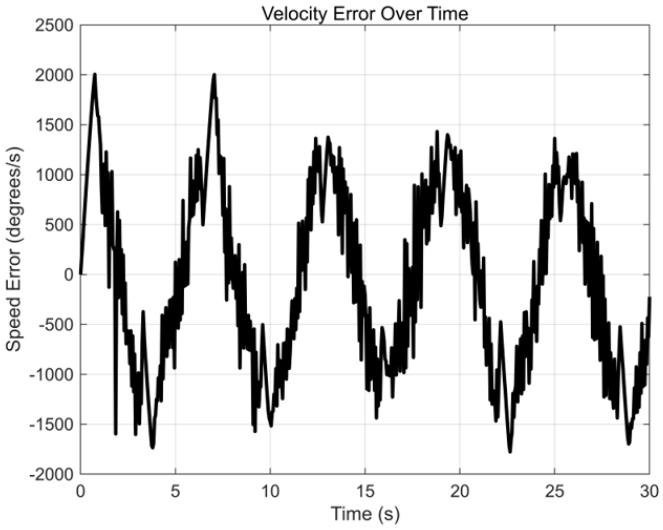
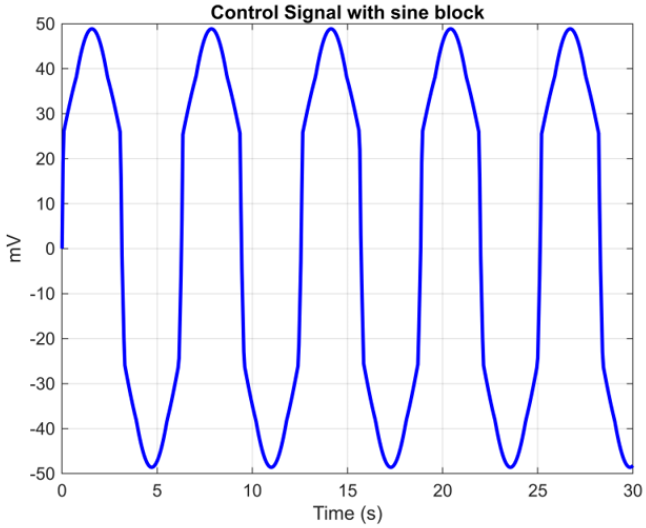



Figure: Open loop speed controller in Simulink


Reference Input vs. Actual Velocity
Input
Velocity
Speed (degrees/s)
Time (s)


Velocity Error Over Time
Speed Error (degrees/s)
Time (s)


Control Signal with sine block
mV
Time (s)

**Closed Loop Speed Controller with Sine Wave**

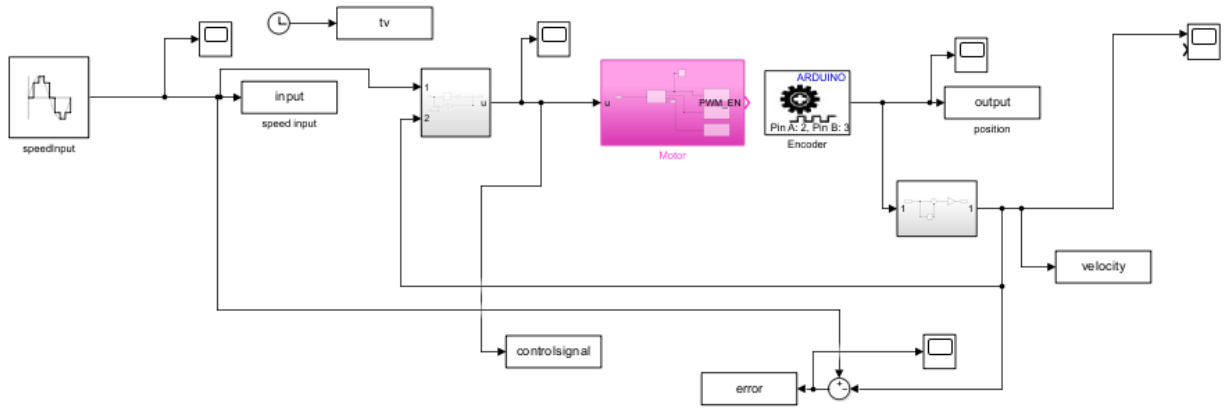
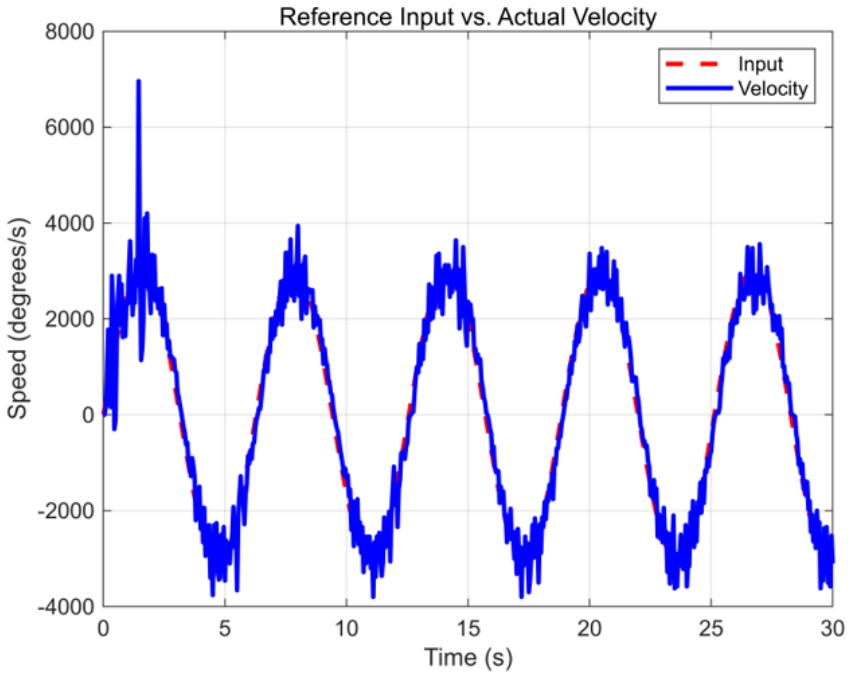
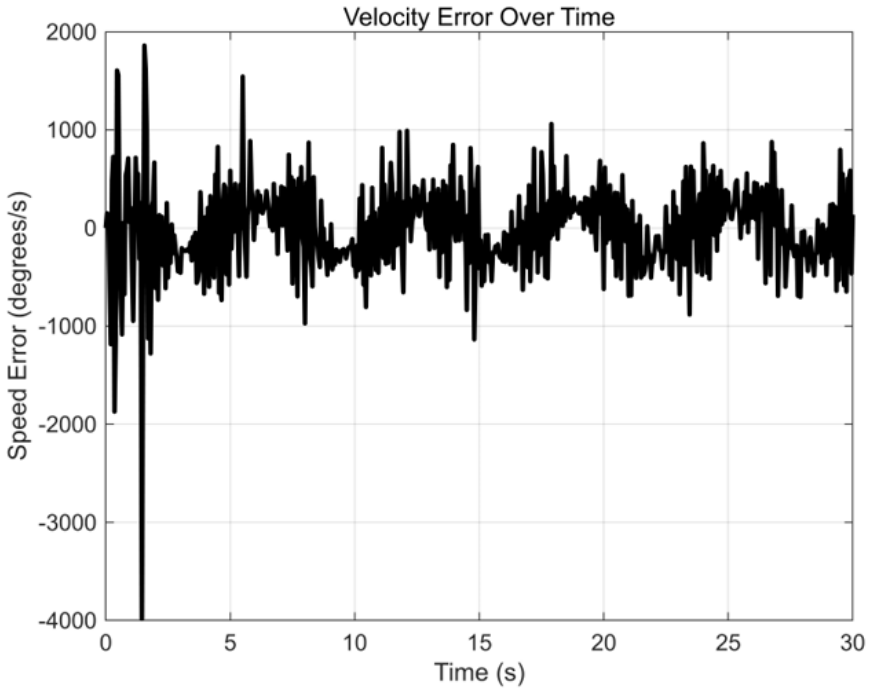
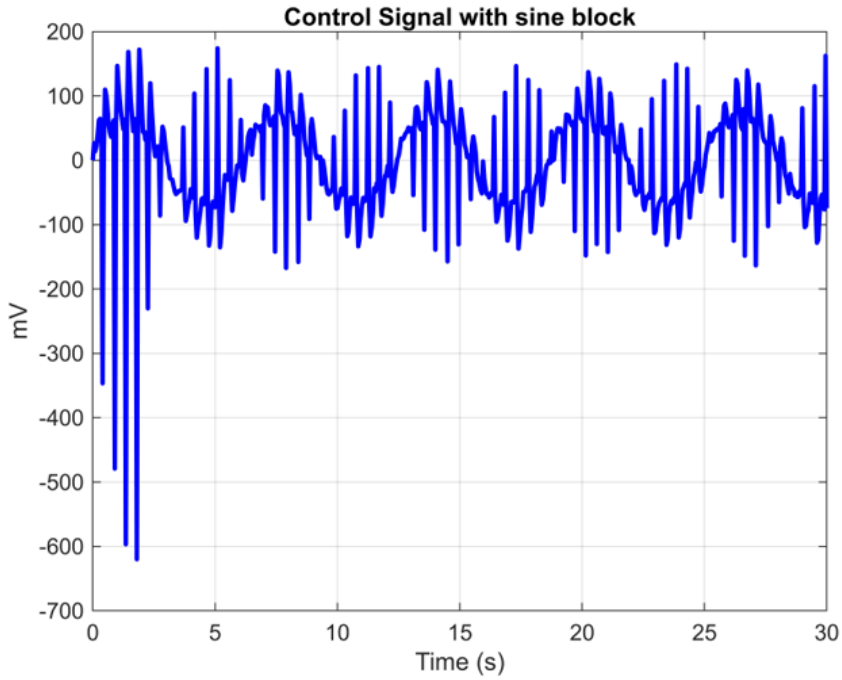



Figure: Closed loop speed controller in Simulink




Velocity Error Over Time
Speed Error (degrees/s)
2000
1000
0
-1000
-2000
-3000
-4000
0
5
10
15
20
25
30
Time (s)


Control Signal with sine block
mV
200
100
0
-100
-200
-300
-400
-500
-600
-700
0
5
10
15
20
25
30
Time (s)

**Closed Loop Speed Controller with Pulse Generation**

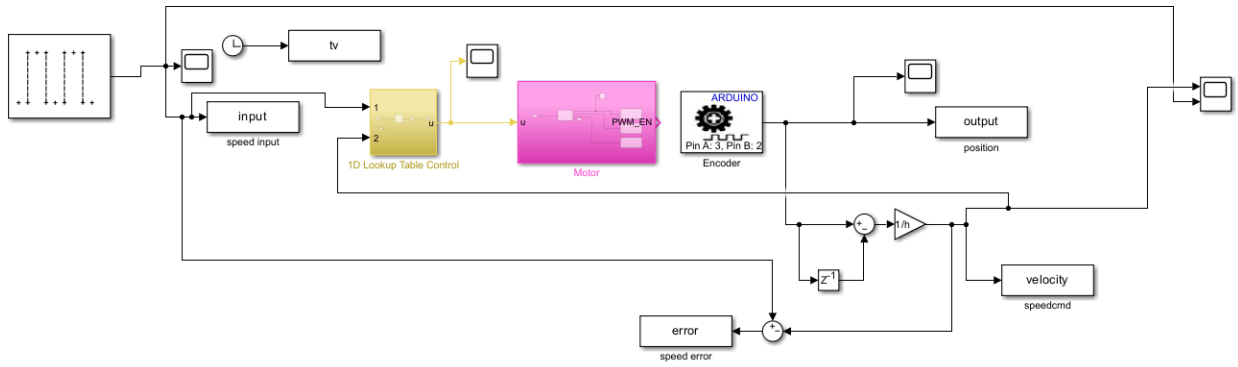



Figure: Closed loop speed controller in Simulink

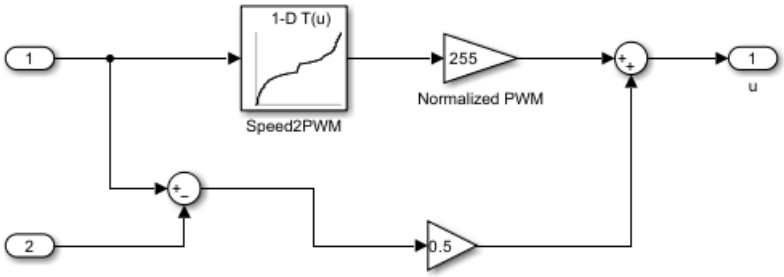



Figure: 1D Lookup Table Control

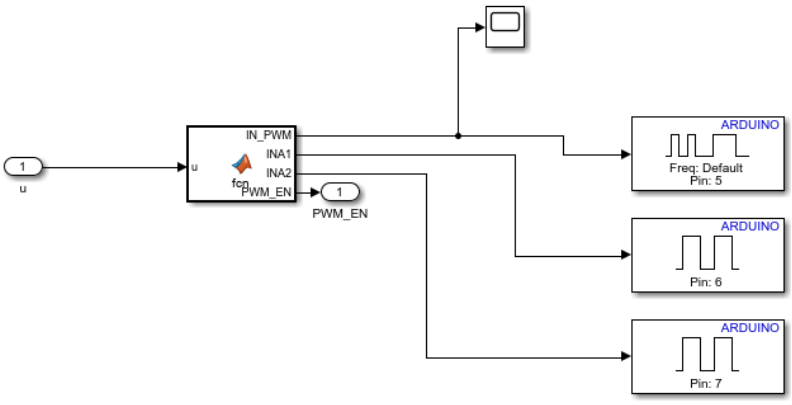
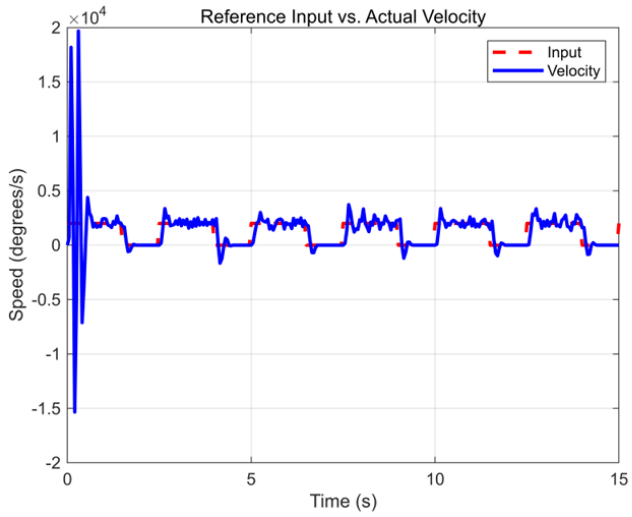
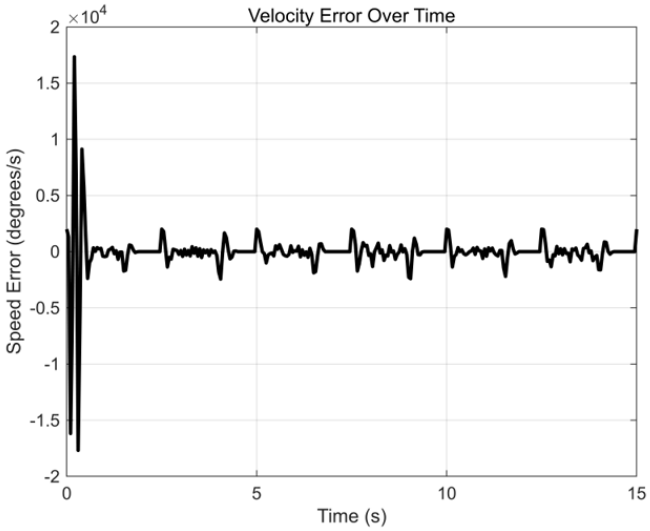
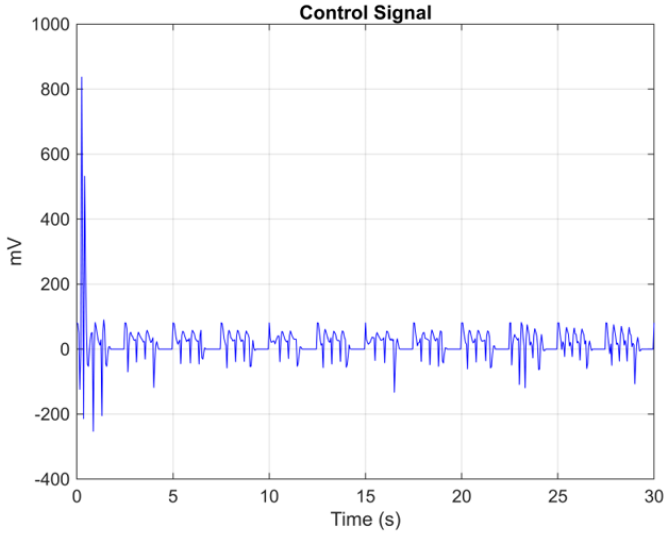



Figure: Motor Subsystem


Reference Input vs. Actual Velocity
×10^4
Input
Velocity
Speed (degrees/s)
Time (s)


Velocity Error Over Time
×10^4
Speed Error (degrees/s)
Time (s)


Control Signal
mV
Time (s)

## Closed Loop Position Control with P Controller

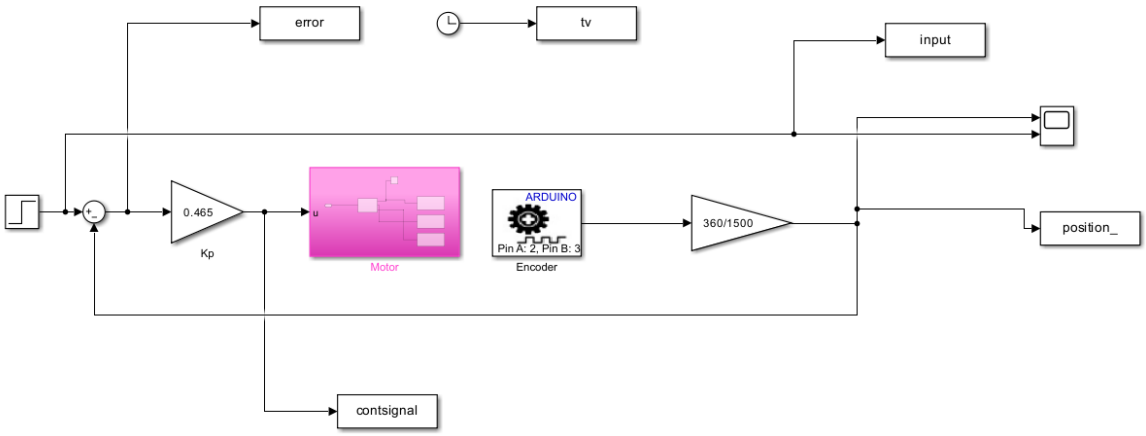
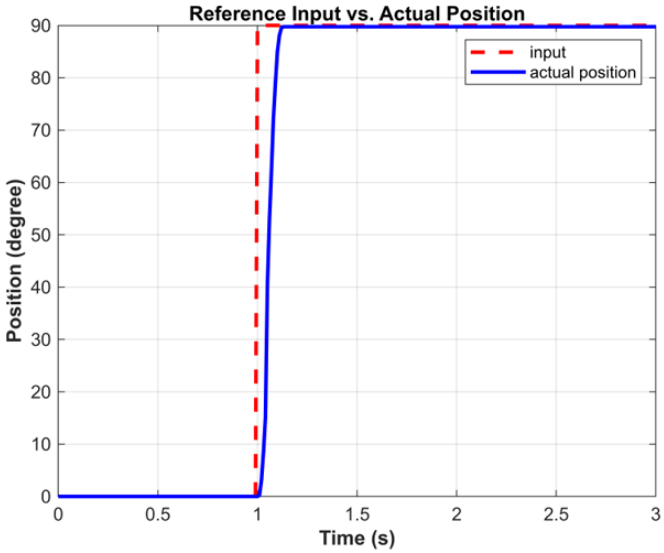
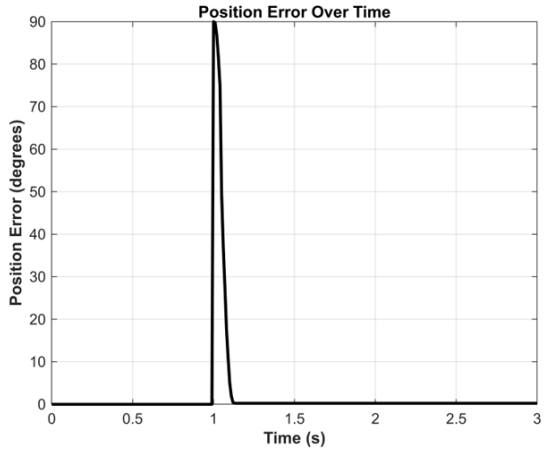



Figure: Position (P) Controller in Simulink

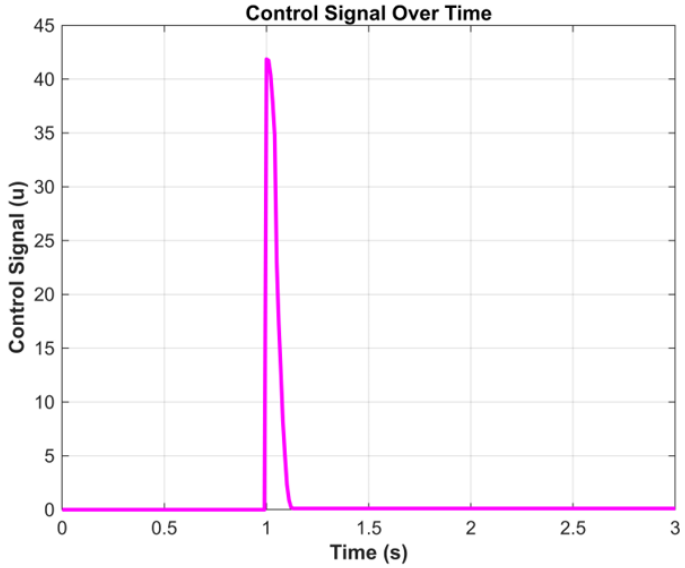
```
% --- Input ve Velocity Aynı Grafikte ---
figure;
plot(tv, input, 'r--', 'LineWidth', 2); hold on;
plot(tv, position_, 'b-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position (degree)', 'FontWeight', 'bold');
title('Reference Input vs. Actual Position', 'FontWeight', 'bold');

grid on;

legend(["input", "actual position"])
```



```
% --- Error Ayrı Grafikte ---
figure;
plot(tv, error, 'k-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position Error (degrees)', 'FontWeight', 'bold');
title('Position Error Over Time', 'FontWeight', 'bold');
grid on;
```




```
% --- Kontrol Sinyali Ayrı Grafikte ---
figure;
plot(tv, contsignal, 'm-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Control Signal (u)', 'FontWeight', 'bold');
title('Control Signal Over Time', 'FontWeight', 'bold');
grid on;
```



## Closed Loop Position Control with PI Controller

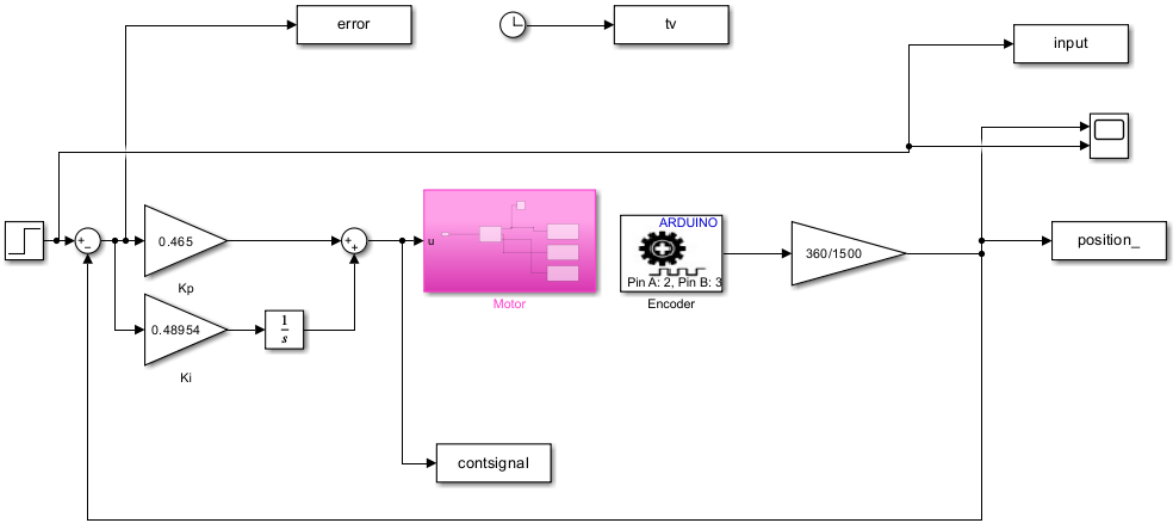
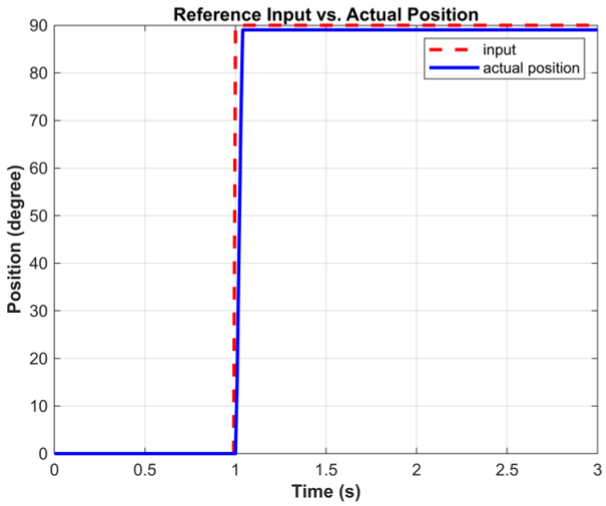
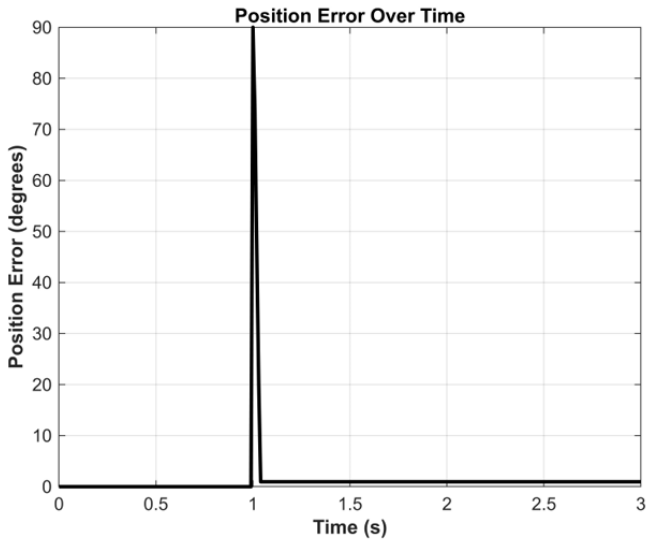



Figure: Position (PI) Controller in Simulink

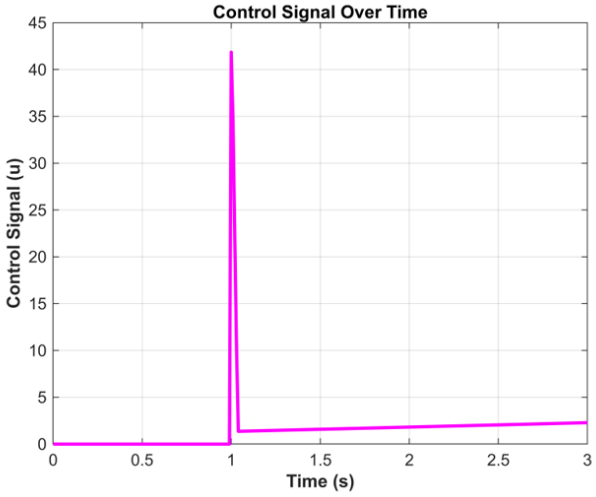
```
% --- Input ve Velocity Aynı Grafikte ---
figure;
plot(tv, input, 'r--', 'LineWidth', 2); hold on;
plot(tv, position_, 'b-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position (degree)', 'FontWeight', 'bold');
title('Reference Input vs. Actual Position', 'FontWeight', 'bold');

grid on;

legend(["input", "actual position"])
```



```
% --- Error Ayrı Grafikte ---
figure;
plot(tv, error, 'k-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position Error (degrees)', 'FontWeight', 'bold');
title('Position Error Over Time', 'FontWeight', 'bold');
grid on;
```




```
% --- Kontrol Sinyali Ayrı Grafikte ---
figure;
plot(tv, contsignal, 'm-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Control Signal (u)', 'FontWeight', 'bold');
title('Control Signal Over Time', 'FontWeight', 'bold');
grid on;
```



## Closed Loop Position Control with PID Controller

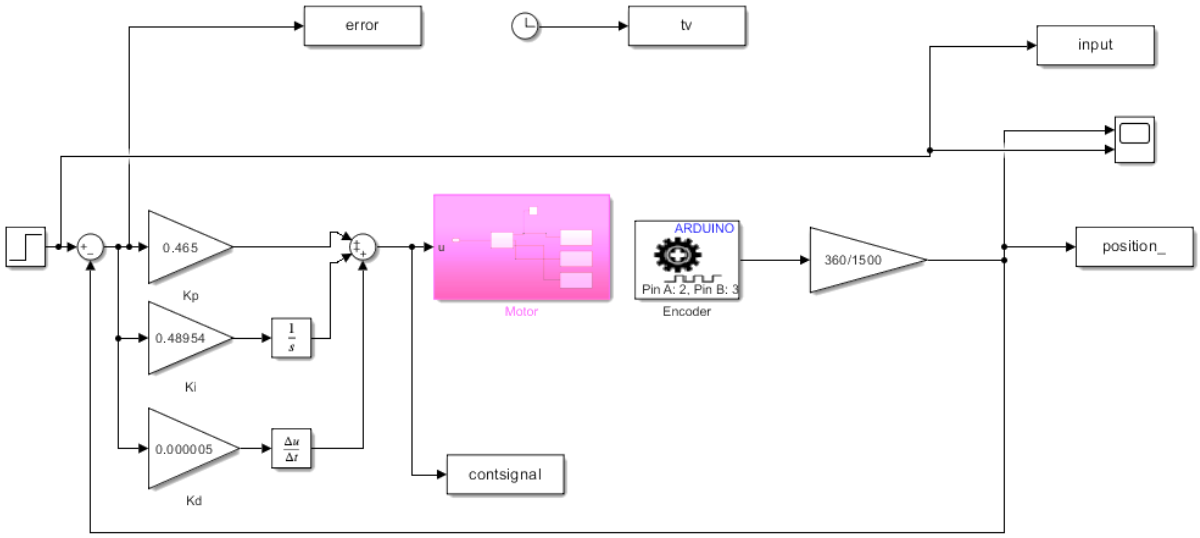
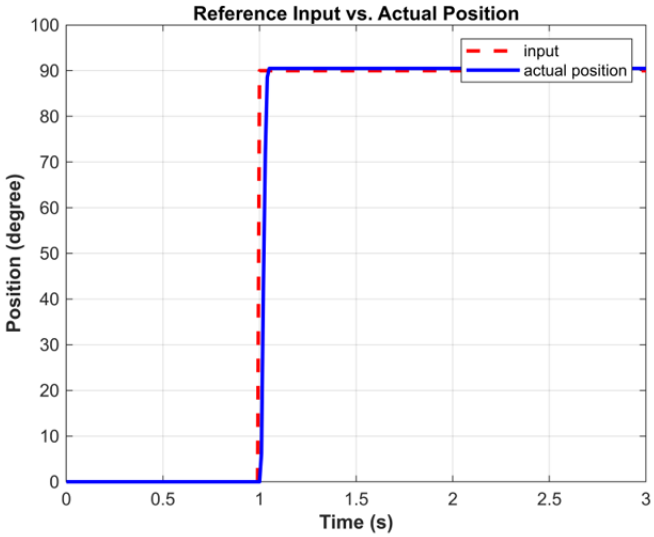
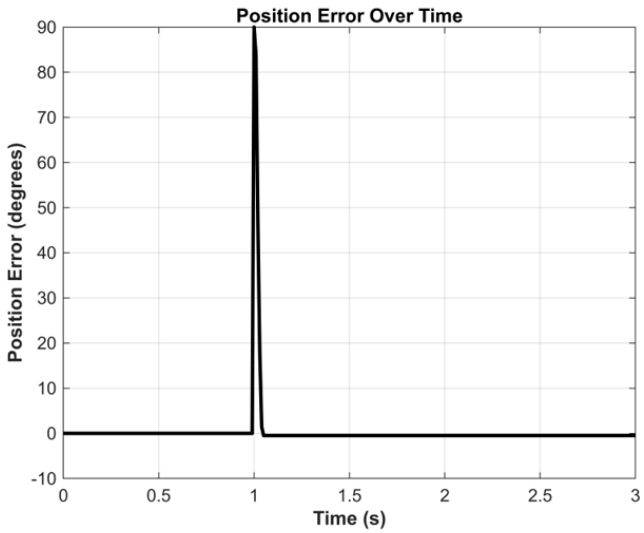



Figure: Position (PID) Controller in Simulink

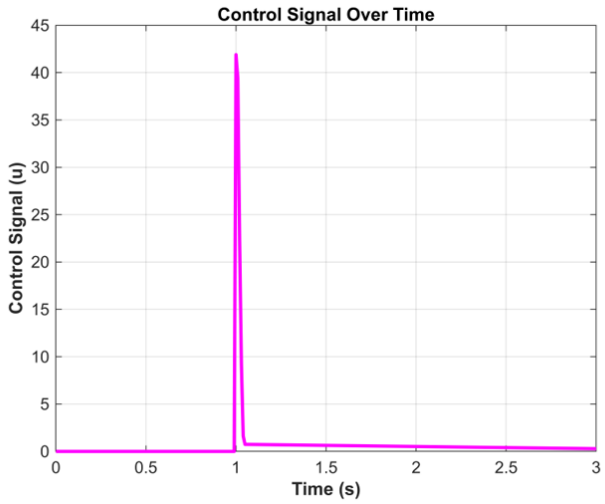
```
% --- Input ve Velocity Aynı Grafikte ---
figure;
plot(tv, input, 'r--', 'LineWidth', 2); hold on;
plot(tv, position_, 'b-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position (degree)', 'FontWeight', 'bold');
title('Reference Input vs. Actual Position', 'FontWeight', 'bold');

grid on;

legend(["input", "actual position"])
```



```
% --- Error Ayrı Grafikte ---
figure;
plot(tv, error, 'k-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position Error (degrees)', 'FontWeight', 'bold');
title('Position Error Over Time', 'FontWeight', 'bold');
grid on;
```




```
% --- Kontrol Sinyali Ayrı Grafikte ---
figure;
plot(tv, contsignal, 'm-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Control Signal (u)', 'FontWeight', 'bold');
title('Control Signal Over Time', 'FontWeight', 'bold');
grid on;
```




As a result of these graphs, it can be said that in a P controller, the system quickly approaches the target but does not fully reach it, as a steady-state error remains. In a PI controller, the system eventually reaches the target with no steady-state error; however, it may exhibit more

oscillations and a longer settling time. In a PID controller, the system reaches the target quickly and accurately, with minimal overshoot and a fast settling time

**Closed Loop Position Control with PIV Controller**

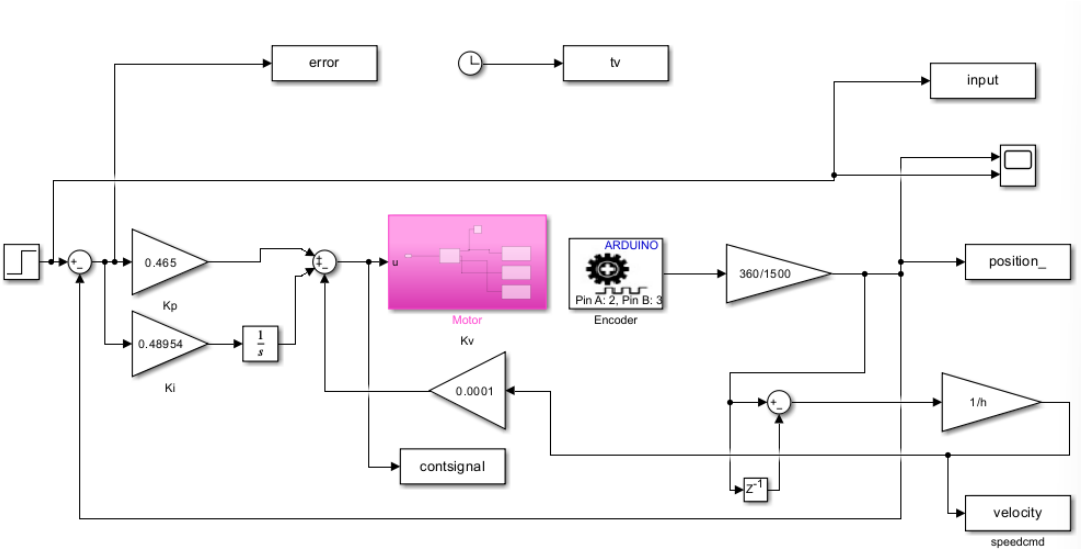
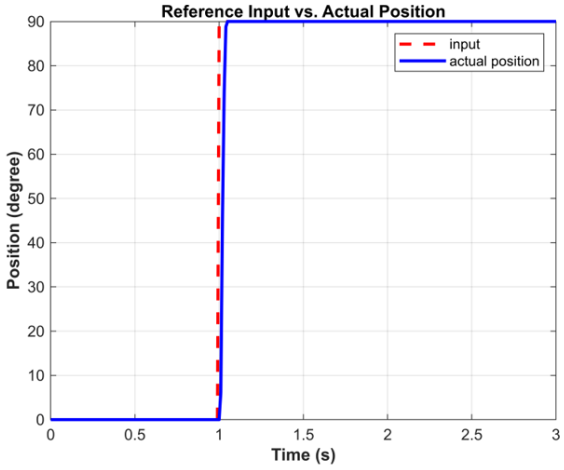
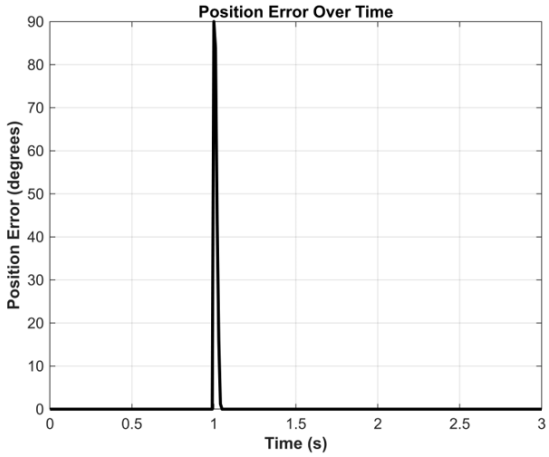



Figure: Position (PIV) Controller in Simulink

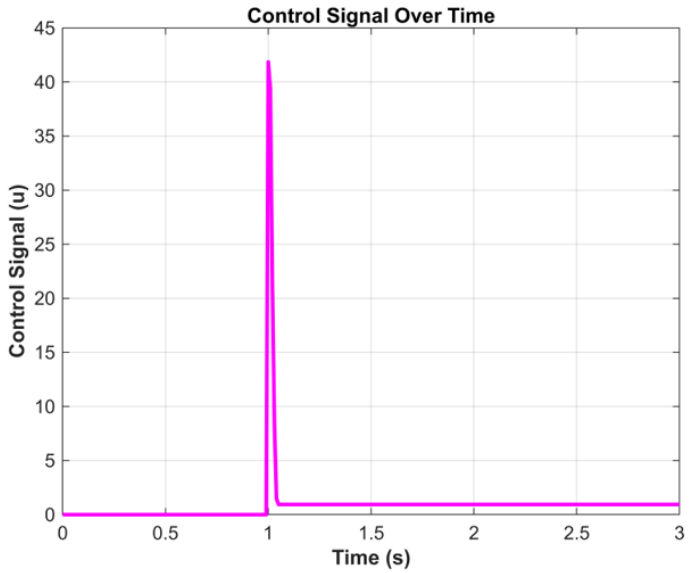
```
% --- Input ve Velocity Aynı Grafikte ---
figure;
plot(tv, input, 'r--', 'LineWidth', 2); hold on;
plot(tv, position_, 'b-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position (degree)', 'FontWeight', 'bold');
title('Reference Input vs. Actual Position', 'FontWeight', 'bold');

grid on;

legend(["input", "actual position"])
```



```
% --- Error Ayrı Grafikte ---
figure;
plot(tv, error, 'k-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position Error (degrees)', 'FontWeight', 'bold');
title('Position Error Over Time', 'FontWeight', 'bold');
grid on;
```




```
% --- Kontrol Sinyali Ayrı Grafikte ---
figure;
plot(tv, contsignal, 'm-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Control Signal (u)', 'FontWeight', 'bold');
title('Control Signal Over Time', 'FontWeight', 'bold');
grid on;
```



A PID controller generates the control signal based on the proportional, integral, and derivative of the error signal. The proportional term reacts to the current error, the integral term eliminates steady-state error by considering the accumulation of past errors, and the derivative term predicts future error behavior by considering its rate of change. This provides a fast and accurate response but can make the system sensitive to noise due to the derivative action. On the other hand, a PIV controller replaces the derivative of the error with the feedforward of the desired velocity, making it especially suitable for motion control systems. In PIV, the control signal is based on the position error, the integral of the error, and the known or commanded velocity, which allows smoother tracking performance with less sensitivity to measurement noise. The PIV controller typically results in a smoother control signal and better velocity tracking.

**PID Controller with Accelerometer (MPU9250)**

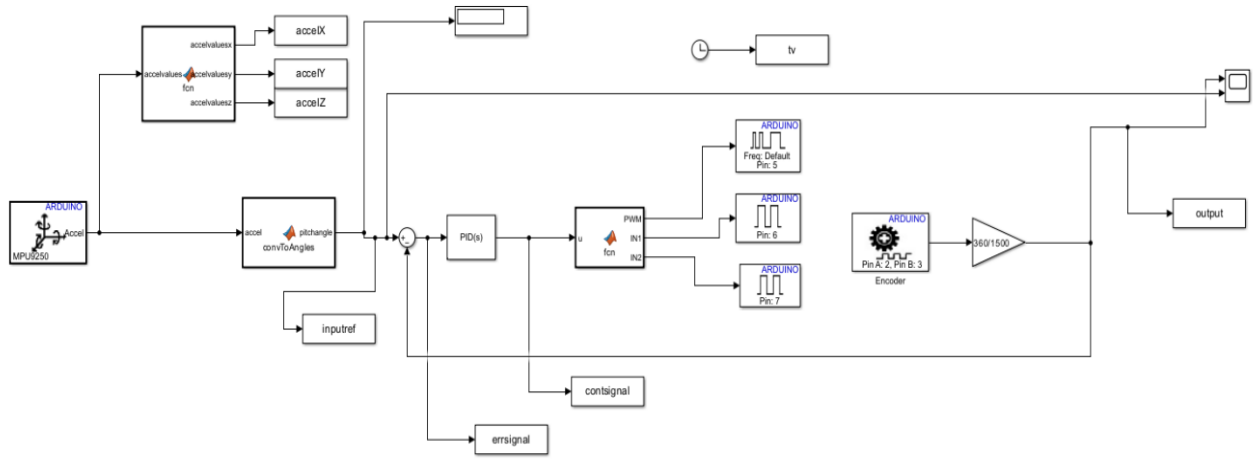



Figure: Implementation accelerometer into system schematic

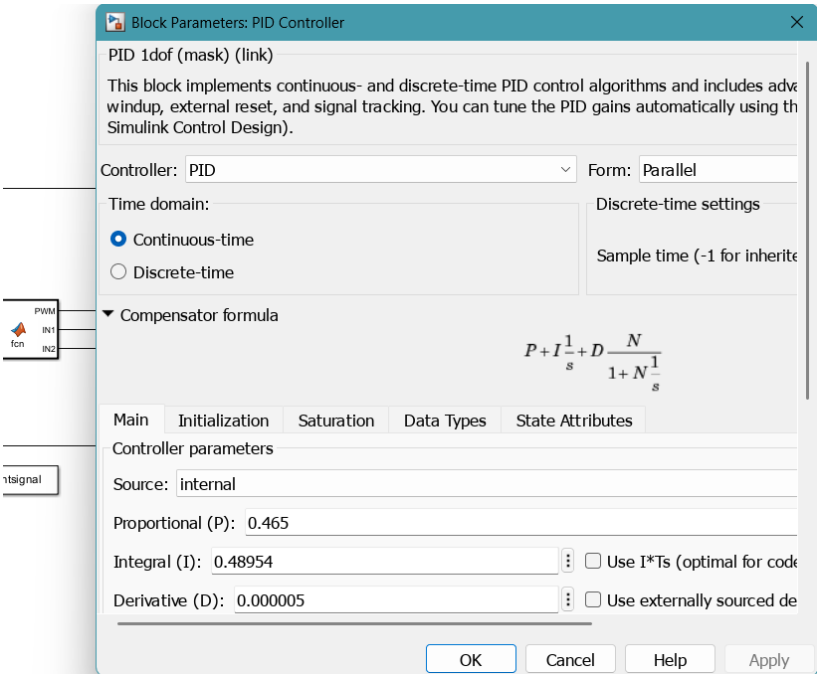



Figure: PID values

rts_project_accel ▸ MATLAB Function

```
1   function pitchangle = convToAngles(accel)
2
3
4   % Accel verilerini ayır
5   accel_x = accel(1);
6   accel_y = accel(2);
7   accel_z = accel(3);
8
9
10  % Pitch açısı hesaplama (derece cinsinden)
11  pitchangle = atan2(-accel_x, sqrt(accel_y^2 + accel_z^2)) * (180 / pi);
12  end
13
```

Figure: Matlab function – converting acceleration values into pitch angle

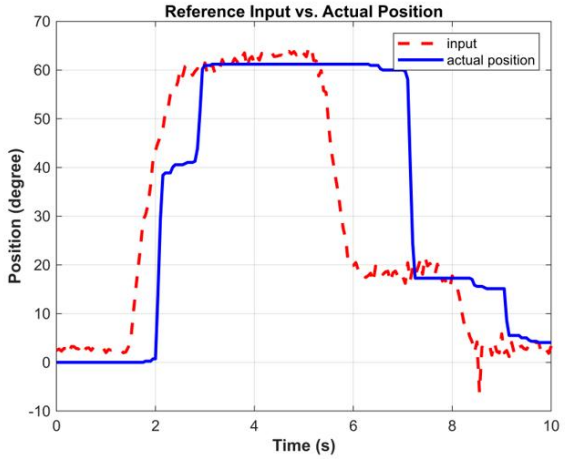
MATLAB Function2

rts_project_accel ▸ MATLAB Function2

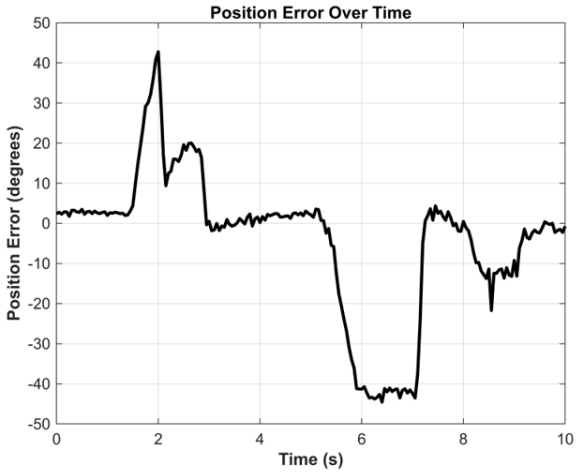
```
function [angaccelX, angaccelY, angaccelZ] = fcn(gyrovalues)
% gyrovalues: [omega_x; omega_y; omega_z] in rad/s
% Ts: sample time

Ts = 0.05; % örnekleme süresi (10 ms)

% Gyro bileşenlerini ayır
omega_x = gyrovalues(1);
omega_y = gyrovalues(2);
omega_z = gyrovalues(3);

% Önceki değerleri saklamak için persistent değişkenler
persistent prev_x prev_y prev_z

% İlk çalışmada sıfırla
if isempty(prev_x)
    prev_x = 0;
    prev_y = 0;
    prev_z = 0;
end

% Açısal ivme (türev hesabı)
angaccelX = (omega_x - prev_x) / Ts;
angaccelY = (omega_y - prev_y) / Ts;
angaccelZ = (omega_z - prev_z) / Ts;

% Önceki değerleri güncelle
prev_x = omega_x;
prev_y = omega_y;
prev_z = omega_z;

end
```

Figure: Matlab function – calculating angular acceleration

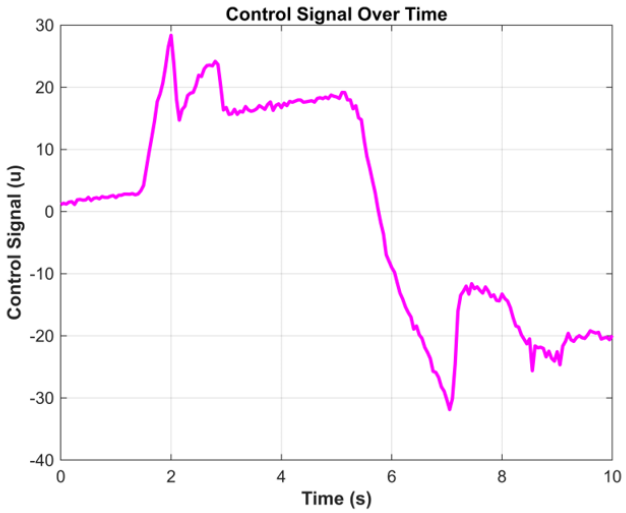
```
figure;
plot(tv, inputref, 'r--', 'LineWidth', 2); hold on;
plot(tv, output, 'b-', 'LineWidth', 2);hold on;
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position (degree)', 'FontWeight', 'bold');
title('Reference Input vs. Actual Position', 'FontWeight', 'bold');

grid on;

legend(["input", "actual position"])
```



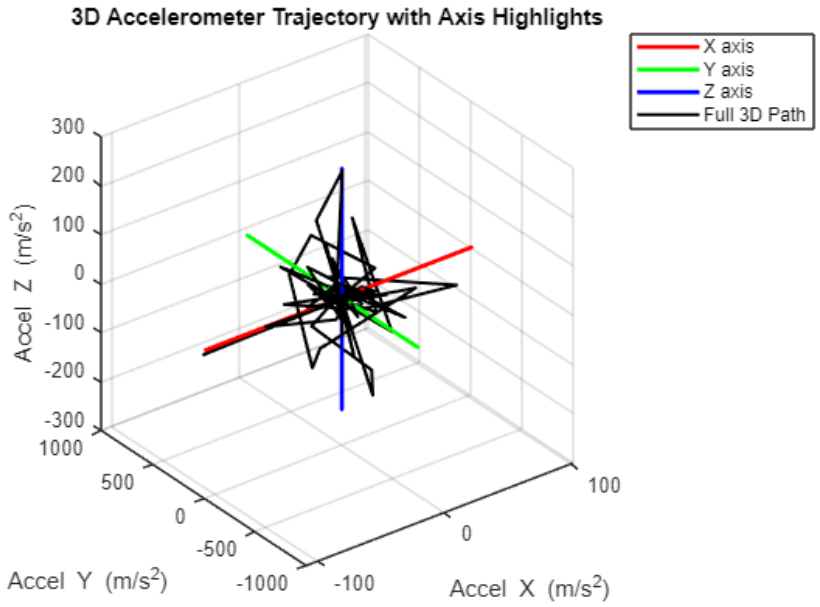
```
% --- Error Ayrı Grafikte ---
figure;
plot(tv, errsignal, 'k-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position Error (degrees)', 'FontWeight', 'bold');
title('Position Error Over Time', 'FontWeight', 'bold');
grid on;
```




```
% --- Kontrol Sinyali Ayrı Grafikte ---
figure;
plot(tv, contsignal, 'm-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Control Signal (u)', 'FontWeight', 'bold');
title('Control Signal Over Time', 'FontWeight', 'bold');
grid on;
```



```
figure;

% X yönündeki hareket (X değişiyor, Y ve Z sabit)
plot3(accelX, zeros(size(accelY)), zeros(size(accelZ)), 'r', 'LineWidth', 2); hold
on;

% Y yönündeki hareket (Y değişiyor, X ve Z sabit)
plot3(zeros(size(accelX)), accelY, zeros(size(accelZ)), 'g', 'LineWidth', 2);

% Z yönündeki hareket (Z değişiyor, X ve Y sabit)
plot3(zeros(size(accelX)), zeros(size(accelY)), accelZ, 'b', 'LineWidth', 2);

% Gerçek 3D yol: hepsini birlikte
plot3(accelX, accelY, accelZ, 'k', 'LineWidth', 1.5);

grid on;
xlabel('Accel X (m/s^2)');
ylabel('Accel Y (m/s^2)');
zlabel('Accel Z (m/s^2)');
title('3D Accelerometer Trajectory with Axis Highlights');
legend('X axis', 'Y axis', 'Z axis', 'Full 3D Path');
view(3);
```




Angular acceleration is the rate at which an object's angular velocity changes over time. It's a measure of how quickly an object is rotating faster or slower. It is calculated as follows :

$\alpha = \Delta\omega / \Delta t$

**PIV Controller with Accelerometer (MPU9250)**

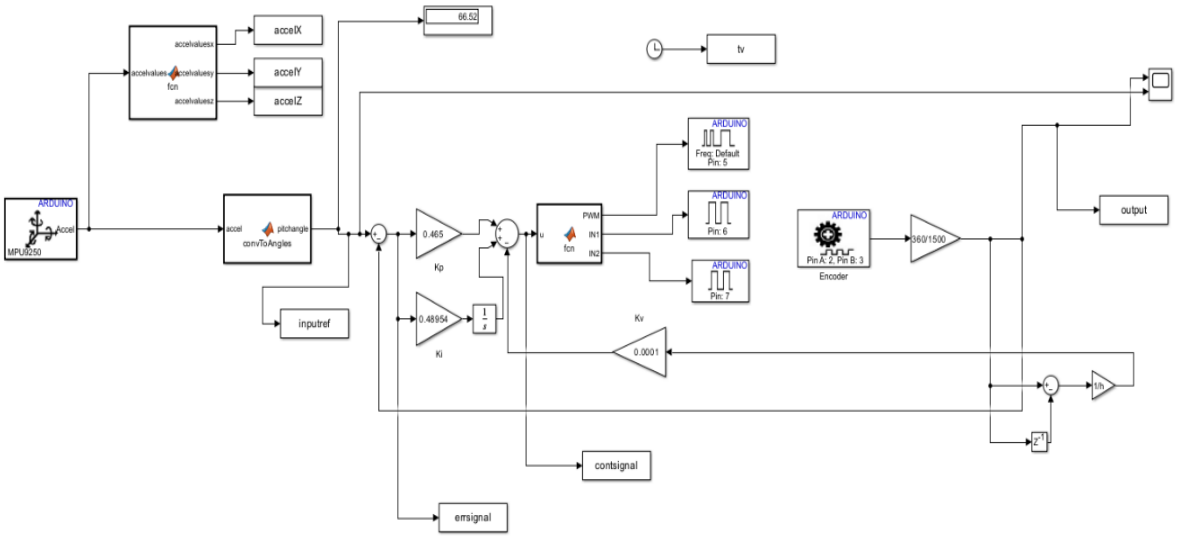
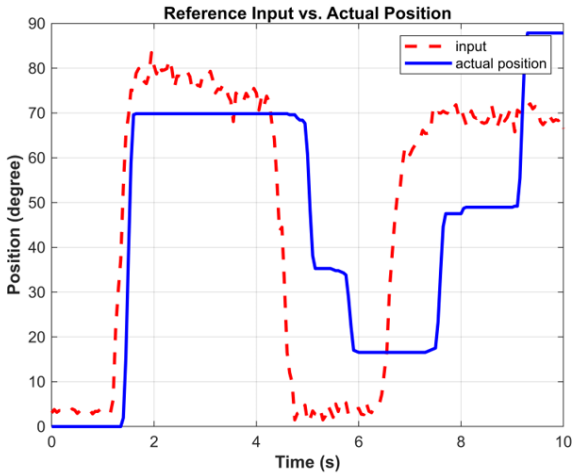
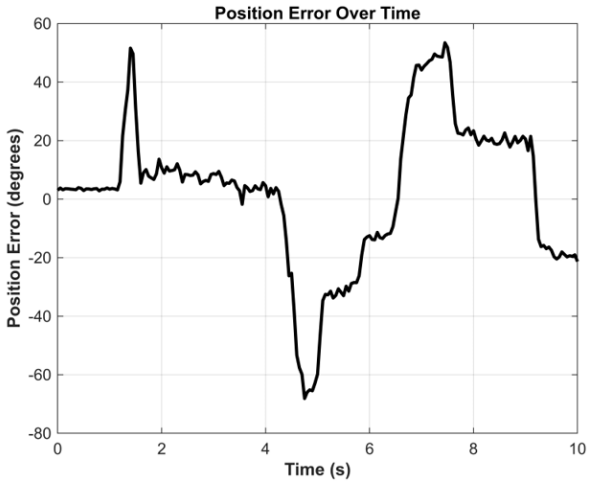



Figure: PIV implementation into system in Simulink

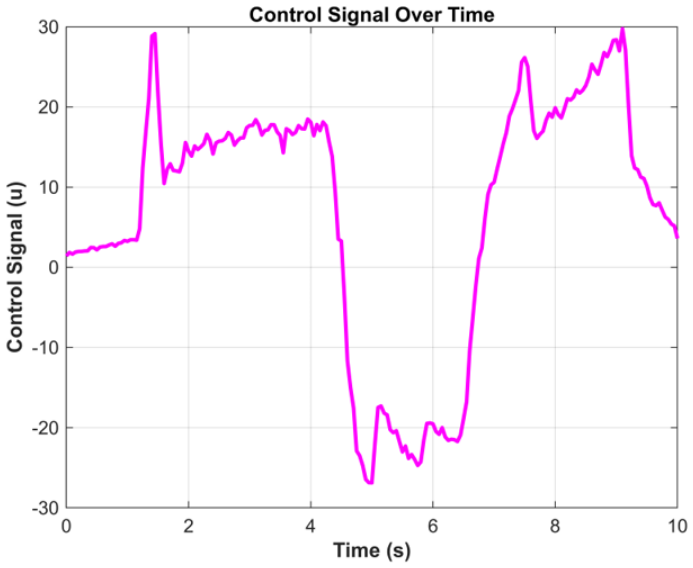
```
figure;
plot(tv, inputref, 'r--', 'LineWidth', 2); hold on;
plot(tv, output, 'b-', 'LineWidth', 2);hold on;
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position (degree)', 'FontWeight', 'bold');
title('Reference Input vs. Actual Position', 'FontWeight', 'bold');

grid on;

legend(["input", "actual position"])
```



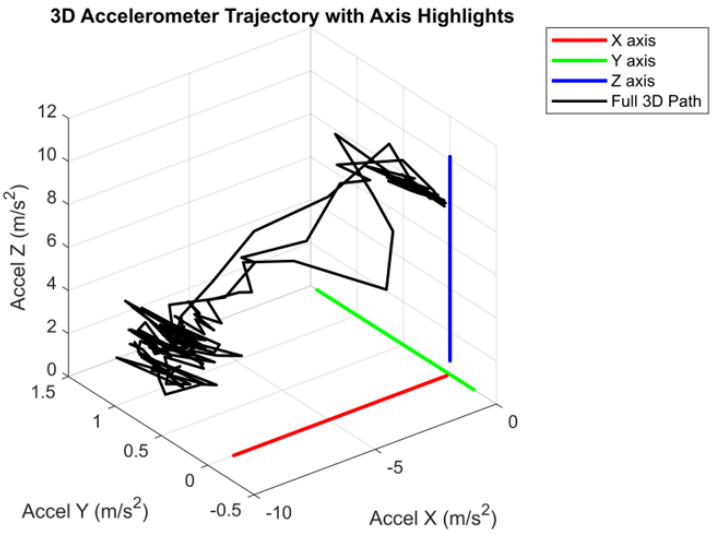

```
% --- Error Ayrı Grafikte ---
figure;
plot(tv, errsignal, 'k-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position Error (degrees)', 'FontWeight', 'bold');
title('Position Error Over Time', 'FontWeight', 'bold');
grid on;
```




```
% --- Kontrol Sinyali Ayrı Grafikte ---
figure;
plot(tv, contsignal, 'm-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Control Signal (u)', 'FontWeight', 'bold');
title('Control Signal Over Time', 'FontWeight', 'bold');
grid on;
```



```
figure;

% X yönündeki hareket (X değişiyor, Y ve Z sabit)
plot3(accelX, zeros(size(accelY)), zeros(size(accelZ)), 'r', 'LineWidth', 2); hold
on;

% Y yönündeki hareket (Y değişiyor, X ve Z sabit)
plot3(zeros(size(accelX)), accelY, zeros(size(accelZ)), 'g', 'LineWidth', 2);

% Z yönündeki hareket (Z değişiyor, X ve Y sabit)
plot3(zeros(size(accelX)), zeros(size(accelY)), accelZ, 'b', 'LineWidth', 2);

% Gerçek 3D yol: hepsini birlikte
plot3(accelX, accelY, accelZ, 'k', 'LineWidth', 1.5);

grid on;
xlabel('Accel X (m/s^2)');
ylabel('Accel Y (m/s^2)');
zlabel('Accel Z (m/s^2)');
title('3D Accelerometer Trajectory with Axis Highlights');
legend('X axis', 'Y axis', 'Z axis', 'Full 3D Path');
view(3);
```




**PID Controller with using Low-Pass Filter**

In this section, the low-pass filter is used to make the input (angle data from accelerometer) smoother. A low-pass filter is an electronic filter that allows low-frequency signals to pass through while attenuating (reducing) the amplitude of signals with frequencies higher than a certain cutoff frequency.

Gain is taken as 1 and Tau is taken as 0.1 . A gain of 1 implies a "unity gain" meaning the signal's magnitude stays the same throughout the system. Tau with 0.1 suggests the system is relatively fast in responding to changes and in the case of a filter, it will allow higher frequencies to pass more easily than a system with a larger tau. These parameters is selected experimentally with real-time system.

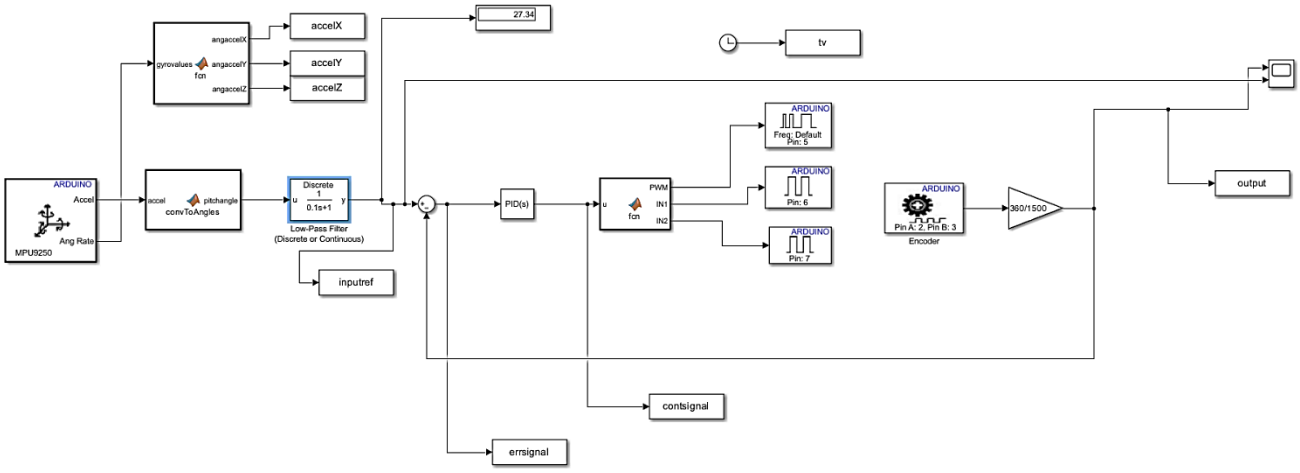
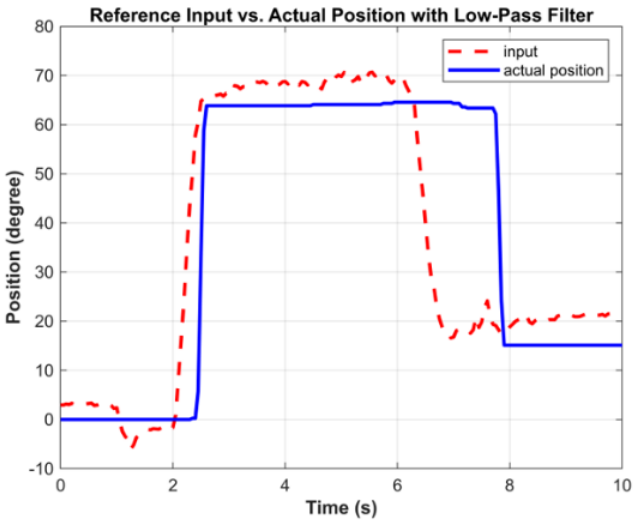
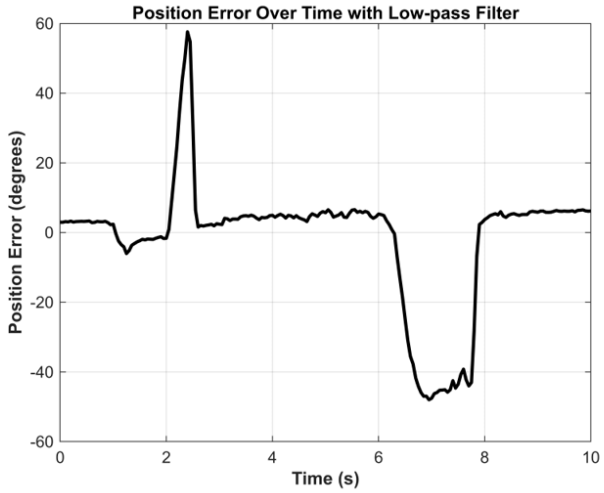



Figure: Low-pass filter implementation into input

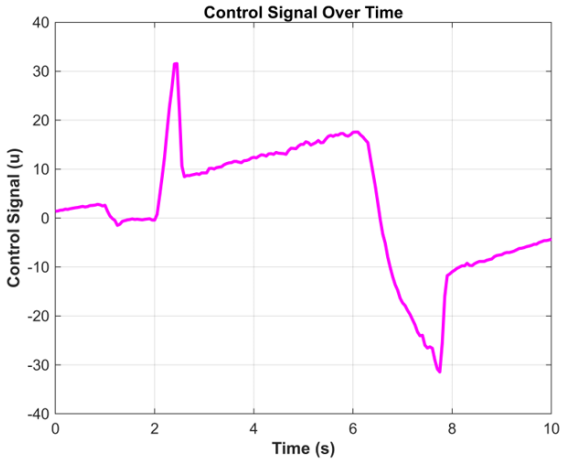
```
figure;
plot(tv, inputref, 'r--', 'LineWidth', 2); hold on;
plot(tv, output, 'b-', 'LineWidth', 2);hold on;
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position (degree)', 'FontWeight', 'bold');
title('Reference Input vs. Actual Position with Low-Pass Filter', 'FontWeight',
'bold');

grid on;

legend(["input", "actual position"])
```



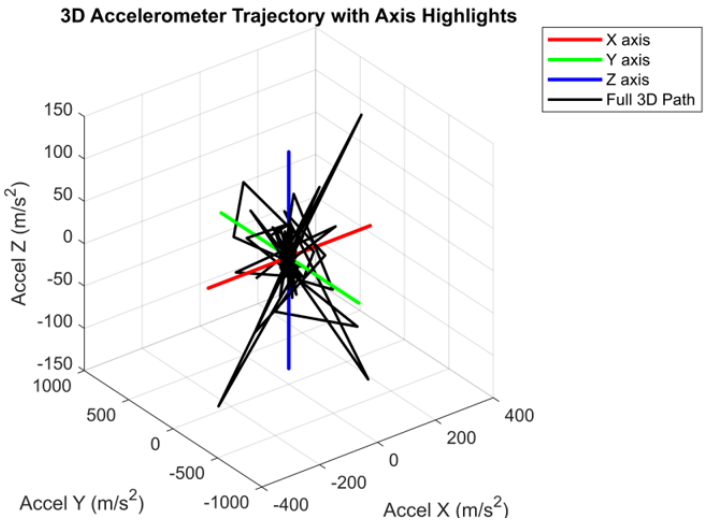

```
% --- Error Ayrı Grafikte ---
figure;
plot(tv, errsignal, 'k-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Position Error (degrees)', 'FontWeight', 'bold');
title('Position Error Over Time with Low-pass Filter', 'FontWeight', 'bold');
grid on;
```




```
% --- Kontrol Sinyali Ayrı Grafikte ---
figure;
plot(tv, contsignal, 'm-', 'LineWidth', 2);
xlabel('Time (s)', 'FontWeight', 'bold');
ylabel('Control Signal (u)', 'FontWeight', 'bold');
title('Control Signal Over Time', 'FontWeight', 'bold');
grid on;
```




```
figure;

% X yönündeki hareket (X değişiyor, Y ve Z sabit)
plot3(accelX, zeros(size(accelY)), zeros(size(accelZ)), 'r', 'LineWidth', 2); hold
on;

% Y yönündeki hareket (Y değişiyor, X ve Z sabit)
plot3(zeros(size(accelX)), accelY, zeros(size(accelZ)), 'g', 'LineWidth', 2);

% Z yönündeki hareket (Z değişiyor, X ve Y sabit)
plot3(zeros(size(accelX)), zeros(size(accelY)), accelZ, 'b', 'LineWidth', 2);

% Gerçek 3D yol: hepsini birlikte
plot3(accelX, accelY, accelZ, 'k', 'LineWidth', 1.5);

grid on;
xlabel('Accel X (m/s^2)');
ylabel('Accel Y (m/s^2)');
zlabel('Accel Z (m/s^2)');
title('3D Accelerometer Trajectory with Axis Highlights');
legend('X axis', 'Y axis', 'Z axis', 'Full 3D Path');
view(3);
```




With the implementation of low-pass filter to input, the error signal and the control signal is smoother.

**MRAC + PI Controller Implementation (with Pololu DC Motor)**

The main objective of MRAC is to make the output of the controlled system follow the behavior of a reference model, which represents the desired closed-loop response. The system parameters are updated in real-time to minimize the error between the plant output and the reference model output.

The PID controller provides a baseline level of control with its well-known ability to stabilize the system and respond to setpoint changes, while the MRAC (Model Reference Adaptive Control) component dynamically adjusts the control parameters to reduce the tracking error between the plant output and a predefined reference model.

the adaptation gain (γ) and learning rate are selected carefuult to ensure both stability and effective learning performance. The value of γ directly influences how quickly the adaptive parameters update in response to the tracking error. A higher γ value results in faster adaptation, but it can also lead to instability or oscillations if chosen too large. Additionally, the reference model was designed to be critically damped, as requested, in order to achieve a smooth and fast response without overshoot. Natural frequency of the system is found experimentally.

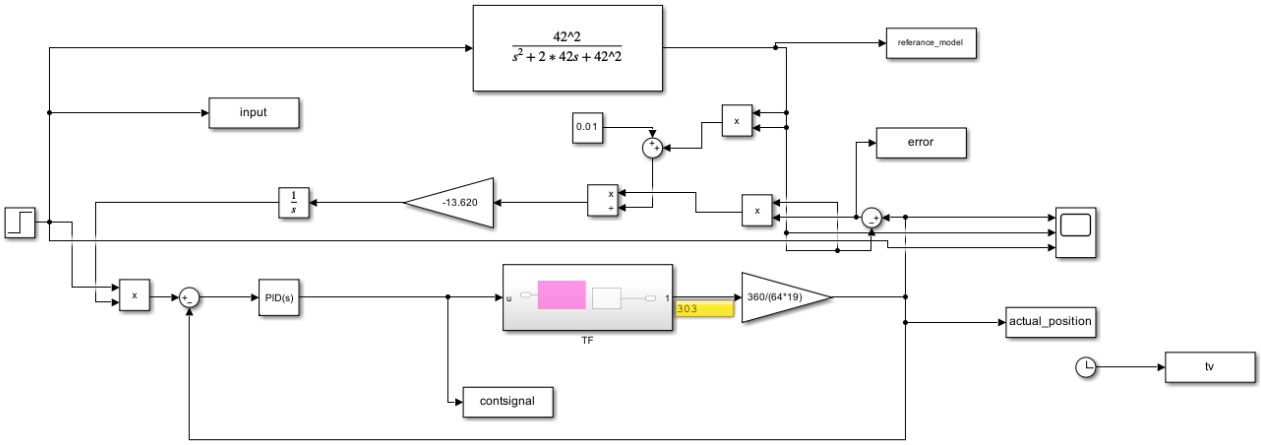
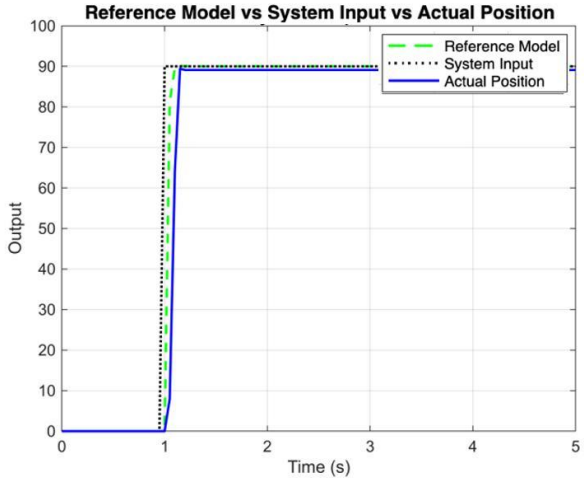
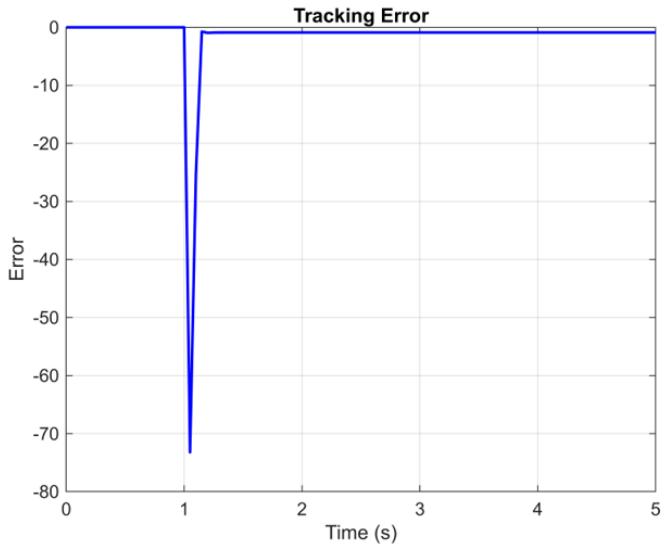



Figure: MRAC-PI controller design in Simulink

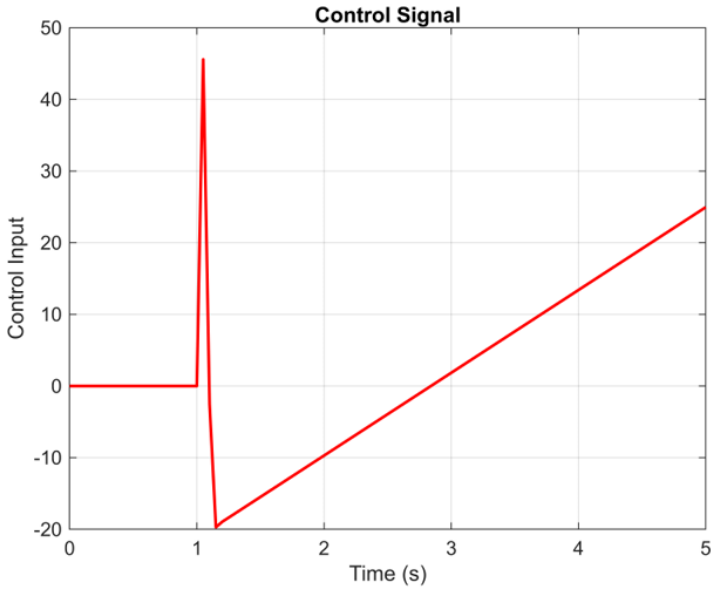
```
figure;
plot(tv, referance_model, 'g--', 'LineWidth', 1.5); hold on;
plot(tv, input, 'k:', 'LineWidth', 1.5);
plot(tv, actual_position, 'b-', 'LineWidth', 1.5); % Yeni eklendi
xlabel('Time (s)');
ylabel('Output');
title('Reference Model vs System Input vs Actual Position');
legend('Reference Model', 'System Input', 'Actual Position');
grid on;
```




```
% Error plot
figure;
plot(tv, error, 'b', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Error');
title('Tracking Error');
grid on;
```




```
% Control signal plot
figure;
plot(tv, contsignal, 'r', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Control Input');
title('Control Signal');
grid on;
```



## MRAC + PID Controller Implementation

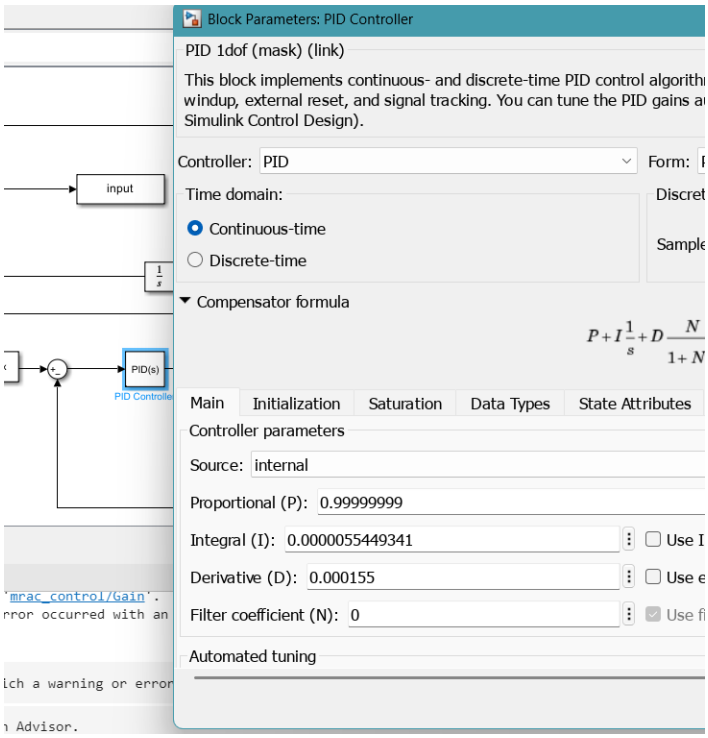
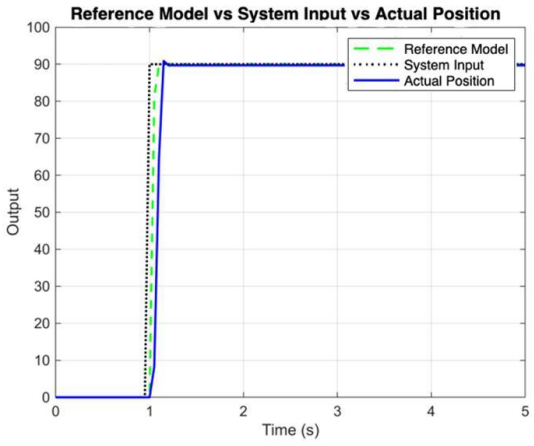
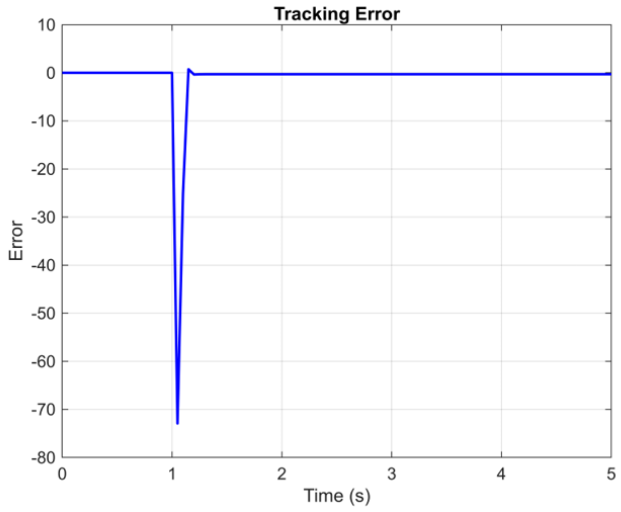



Figure: PID values for MRAC-PID Controller design

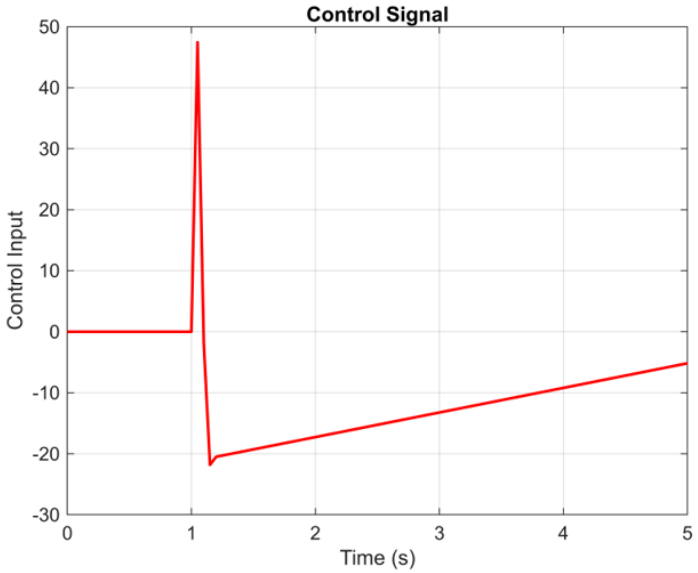
```
figure;
plot(tv, referance_model, 'g--', 'LineWidth', 1.5); hold on;
plot(tv, input, 'k:', 'LineWidth', 1.5);
plot(tv, actual_position, 'b-', 'LineWidth', 1.5); % Yeni eklendi
xlabel('Time (s)');
ylabel('Output');
title('Reference Model vs System Input vs Actual Position');
legend('Reference Model', 'System Input', 'Actual Position');
grid on;
```



```
% Error plot
figure;
plot(tv, error, 'b', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Error');
title('Tracking Error');
grid on;
```




```
% Control signal plot
figure;
plot(tv, contsignal, 'r', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Control Input');
title('Control Signal');
grid on;
```



## 2.4 Subheading: Videos' link

Videos of platform is included in the following drivers:

Matlab Drive:

https://drive.mathworks.com/sharing/4b9812c8-2e17-406e-92fb-753037b6b199

Google Drive:

https://drive.google.com/drive/folders/1WsemH1UKuEjCPNn17U7dBj9YBq5PWllO?usp=sharing

# 3 Conclusion

This project involved the step-by-step development of a DC motor platform for precise speed and position control, including head stabilization using sensor feedback and control design. PWM signals were used to drive the motor, and encoder data provided speed and position feedback. Coulomb friction was identified, and motor parameters were estimated using MATLAB Parameter Estimation Toolbox. A PWM vs. speed table was also created. Open-loop and closed-loop speed and position controllers were designed and tested. An accelerometer measured 3D angular acceleration (X, Y, Z), and an end-point controller was developed for head stabilization.

Experimental results showed accurate motor control and effective stabilization, with control signals staying within limits and low error values achieved.

# 4 Personal Comment

Aylin Öztürk:

Working on this project has been a valuable and educational experience for me. Through the development of the DC motor experimental platform and the head stabilization system, I gained hands-on knowledge in areas such as real-time control systems, feedback loops, sensor integration, and motor driver circuitry. I learned how to work with encoders, PWM signals, and accelerometers, and I became more confident in implementing control algorithms using Arduino and MATLAB.

One of the most rewarding aspects was seeing how theory from control systems and electronics courses could be applied in a real-world application. This project has increased my interest in embedded systems and robotics. In the future, I hope to continue developing similar projects and contribute to more complex applications, such as robotic arms or autonomous vehicles.

Çağla Üzümcü:

This project has been an incredibly educational and inspiring experience for me in terms of applying theoretical knowledge to practice. During the development of the DC motor experimental setup and the head stabilization system, we not only implemented fundamental control techniques but also

applied parameter estimation methods to better understand the system dynamics. This allowed us to create more accurate models of the motor and improve overall control performance.

We designed a PID controller to enhance the system's stability and response time. For motor control, we used the L298N motor driver, which enabled us to manage both the direction and speed of the motor effectively. By developing real-time control applications on the Arduino platform, I gained valuable experience in microcontroller programming, sensor data processing, and motor control. Additionally, I used MATLAB for system analysis and simulations, which helped me better understand the integration of software and hardware.

# 5 Reference List

GitHub - ClaudiaYasar/DC-Motor-Dynamic-Modeling-and-Position-Control: This Live Script validates theoretical approaches for monophasic DC motor control through experiments and simulations, serving as a practical guide for engineers and researchers. It includes reduced-order modeling, state-space representation, transfer function analysis, parameter estimation, and PI control.

[1] C. F. Yaşar, "Climbing with Robots: A Second Order Controller Design for Accurate Wheel Motion Positioning," *Çukurova Univ. J. Fac. Eng.*, vol. 39, no. 1, pp. 175–187, Mar. 2024.