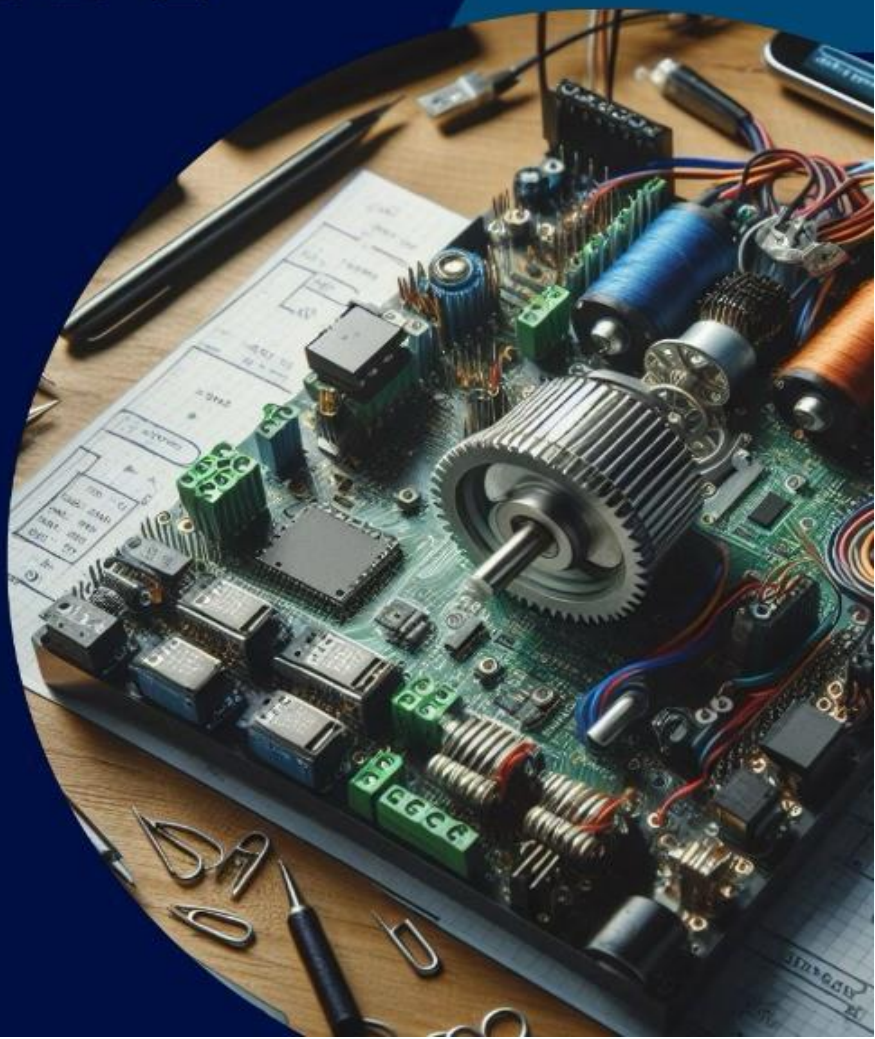
NUST COLLEGE OF EME

LINEAR CONTROL SYSTEMS

PROJECT REPORT



PREPARED BY :

- Huzaifa Zafar

PRESENTED TO :

- Sir Azmat Saeed

Title

POSITION CONTROL OF DC MOTOR

Project Statement

Design and Implementation of closed loop control for shaft angular position of DC Motor

- DC Motor with Shaft encoder
- Drive carefully with H-Bridge
- Microcontroller for PWM Generation
- Microcontroller interface with Simulink

Objectives

- ✓ Model the DC encoder motor.
- ✓ Derive its transfer function.
- ✓ Designed the controller using root locus or “sisotool” of MATLAB.
- ✓ Implemented the controller on Arduino Uno.
- ✓ Displayed the output on Simulink using Arduino Uno.

Introduction

A control law is a set of mathematical tools or algorithms that are designed to manipulate the inputs or actions of a control system based on the system’s current state or error signals. They are typically based on feedback from sensors that measure the system’s output and compare it to the desired reference. Then the control law computes the appropriate control actions to minimize the error and drive the system toward the desired state. One of the simplest control laws is Proportional-Integral-Derivative (PID) controller. In this project, we will control the position of a DC encoder motor using the P, I, PI, PD, and PID controllers and compare the outputs of all the controllers.

Theoretical Background

Proportional Controller (P):

P or proportional controller is the simplest form of feedback control and provides control actions proportional to the error between the desired setpoint and the actual system output. It multiplies the error signal by a constant gain factor, known as the proportional gain (K_p), to compute the control action. The control action is then applied to the system to reduce the error and bring the output closer to the set point. The transfer function of a P controller is:

$$C(s) = k_p$$

- It is a simple controller that eliminates errors, but it cannot eliminate the steady-state error.

Integral Controller (I):

An integral controller is designed to reduce steady-state errors by integrating the error signal over time and applying a control action based on the accumulated error. It operates by continuously summing the error between the setpoint and the system output over time. This accumulated error, known as the integral of the error, is multiplied by a constant gain factor, called the integral gain (Ki), to compute the control action. The control action is then applied to the system to correct for any steady-state errors and drive the output closer to the setpoint. The transfer function of a PID controller is:

$$C(s) = \frac{Ki}{s}$$

- Although it eliminates the steady-state error, if not properly tuned it can cause instability and cause the system to approach infinity.

- **Proportional – Integral (PI) Controller:**

The proportional-integral (PI) controller combines the proportional (P) control and integral (I) control strategies to provide improved control performance compared to using either strategy individually. A PI controller operates by considering both the current error between the desired setpoint and the system output, as well as the accumulated error over time. The proportional term provides an immediate response to the current error, while the integral term addresses any steady-state error that may exist by integrating the error over time. The transfer function of a PI controller is:

$$C(s) = k_p + \frac{k_i}{s}$$

- PI controllers strike a balance between simplicity and performance, providing effective control in many scenarios. However, in some cases, additional control strategies, such as derivative (D) control, may be required to further enhance the control system's performance and stability.

- **Proportional – Derivative (PD) Controller:**

The PD controller combines the proportional (P) control and derivative (D) control strategies to improve control performance compared to using only proportional control. The derivative controller is based on the derivative of the error between the desired setpoint and the system output and provides a control action proportional to the rate of change of the error. It helps identify the sudden changes in the error and helps achieve stability. A PD controller operates by considering both the current error between the desired setpoint and the system output, as well as the rate of change of the error. The proportional term provides an immediate response to the current error, while the derivative term provides a dampening effect by considering the rate of change of the error. The transfer function of a PD controller is:

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

- PD controllers are effective in reducing overshoot, improving stability, and increasing the system's ability to track setpoint changes. However, it's important to note that PD controllers may not be able to eliminate steady-state errors, as they lack an integral term.

➢ **Proportional – Integral - Derivative (PID) Controller:**

The PID controller combines three control strategies—proportional (P), integral (I), and derivative (D)—to provide accurate and robust control of a system. The PID controller calculates the control action based on the current error, the integral of the error over time, and the rate of change of the error. The proportional term provides an immediate response to the current error, the integral term eliminates steady-state errors by integrating the error over time, and the derivative term provides dampening and anticipates future changes in the error. The transfer function of a PID controller is:

$$C(s) = k_p + \frac{k_i}{s} + k_d s$$

- PID controllers offer a versatile and effective control strategy by combining the strengths of proportional, integral, and derivative control.

Motor Specifications

Model No.	KG-47C-SG-100-E221
Operating Voltage	24 V DC
Rated Current	1.5 A
Power	36 W
Armature Resistance	16 ohms
Speed	2000 rpm
Torque	1.5 Nm
Gear Ratio	100:1
Encoder Type	Incremental, 221 pulse per revolution
Time Constant	0.008 sec

Micro-Controller

- In this project, Arduino Uno is utilized as the microcontroller to implement the closed-loop control system for the position control of the DC motor. Arduino is chosen due to its ease of use, extensive library support, and strong community backing. Here is the detailed explanation of it:

1. PWM Signal Generation

The Arduino Uno generates Pulse Width Modulation (PWM) signals through its digital output pins. The duty cycle of the PWM signal can be varied, thus controlling the average voltage supplied to the motor and subsequently its speed. This is crucial for precise control of the motor's position.

2. Interfacing with Shaft Encoder

The Arduino reads the output from the shaft encoder, which provides feedback in the form of pulses corresponding to the motor's angular position and speed.

3. Control Algorithm Implementation

The control algorithms (P, I, PI, PD, PID) are implemented in the Arduino's code. These algorithms calculate the necessary control actions based on the error between the desired setpoint and the actual position of the motor shaft. The Arduino adjusts the PWM signal based on these calculations to minimize the error.

4. Real-time Data Processing

The Arduino processes the feedback data from the encoder in real-time, updating the control actions rapidly to ensure precise control. This real-time capability is essential for maintaining the desired position of the motor shaft with high accuracy.

5. Simulink Interface

Arduino can be interfaced with Simulink via the Arduino Support Package for Simulink. This allows for easy design, simulation, and implementation of control algorithms. The Arduino receives control signals from Simulink and applies them to the motor. Additionally, it sends real-time data back to Simulink for monitoring and analysis. The communication is typically handled via the serial port.

6. Communication with H-Bridge

The H-Bridge circuit, controlled by the Arduino, allows the motor to run in both forward and reverse directions. The Arduino sends appropriate signals to the H-Bridge to switch the motor's direction as needed. This is typically done using digital I/O pins that control the H-Bridge's input pins.

Motor Encoder

The motor encoder is essential for providing real-time feedback on the motor's position and speed, enabling precise control. Here's a brief overview of its role:

1. Position Measurement

The encoder generates pulses corresponding to the motor shaft's rotation. By counting these pulses, the system determines the exact angular position of the motor, which is crucial for comparing to the desired setpoint.

2. Speed Calculation

The rate of pulse generation indicates the motor's speed. This information is used in control algorithms, particularly for the derivative component in PD and PID controllers.

3. Feedback for Control Algorithms

The encoder feedback is used by the control algorithms (P, I, PI, PD, PID) to adjust the control actions. Accurate feedback ensures effective error correction and precise positioning.

4. Closed-Loop Control

The encoder enables a closed-loop control system where the actual motor position is continuously compared to the desired position. The control algorithms use this feedback to minimize the error and achieve the desired position.

5. Improving Stability and Accuracy

Accurate position feedback from the encoder enhances system stability and accuracy, reducing overshoot, settling time, and steady-state error.

Mathematical Calculation

For the controller design, we will first need a mathematical model of our DC encoder motor. The DC motor we are using for our project is KG-47C-SG-100-E221 made by Tsukasa. It is a geared motor, and we will need to model it to design an appropriate controller.

The general transfer function of a DC motor for position control is written as:

$$G(s) = \frac{K}{s(\tau s + 1)}$$

Where $\tau\tau$ is the time constant of the DC motor and K is the gain or motor's constant. It is equal to the inverse of the motor's torque constant (km) which can be calculated as:

$$k_m = \frac{V_{max} - I_{max} * \mathrm{R}}{\omega_{max}}$$

Where, Vmax, Imax, R, and ωmax are the maximum voltage, maximum current, motor resistance, and maximum motor speed. Using the datasheet,

$$V_{max} = 24V \;\&\; \quad I_{max} = 1A$$

$$R = 23ohm\ (Using\ DMM)$$

$$\omega_{max} = 87.2\frac{rev}{s} = 9.1315\frac{rad}{s}$$

$$k_m = \frac{24-23}{9.1315} = 0.6881$$

And by taking the inverse of *km*, we get **K = 1.4533**. Similarly, for the time constant we will first need to calculate the equivalent moment of inertia of the DC motor, which can be calculated using the following equation:

$$J_{eq} = J_{motor} + J_{gear}$$

Using the general equations to calculate the moment of inertia of certain shapes,

$$J_{eq} = \frac{m * r^2}{2} + \frac{\left(m_{gear} * \frac{(upper\ diameter - lower\ diameter)}{4}\right)}{2}$$

Using the dimensions and weight mentioned in the datasheet and motor's description on Amazon,

$$J_{eq} = \frac{0.11 * 3mm^2}{2} + \frac{\left(0.076kg * \frac{(37mm - 7mm)}{4}\right)}{2} = 4.95 * 10^{-7} + 2.1375 * 10^{-6}$$

$$J_{eq} = 2.6325 * 10^{-6} kgm^2$$

So, now we can calculate the time constant of the DC motor using the following equation:

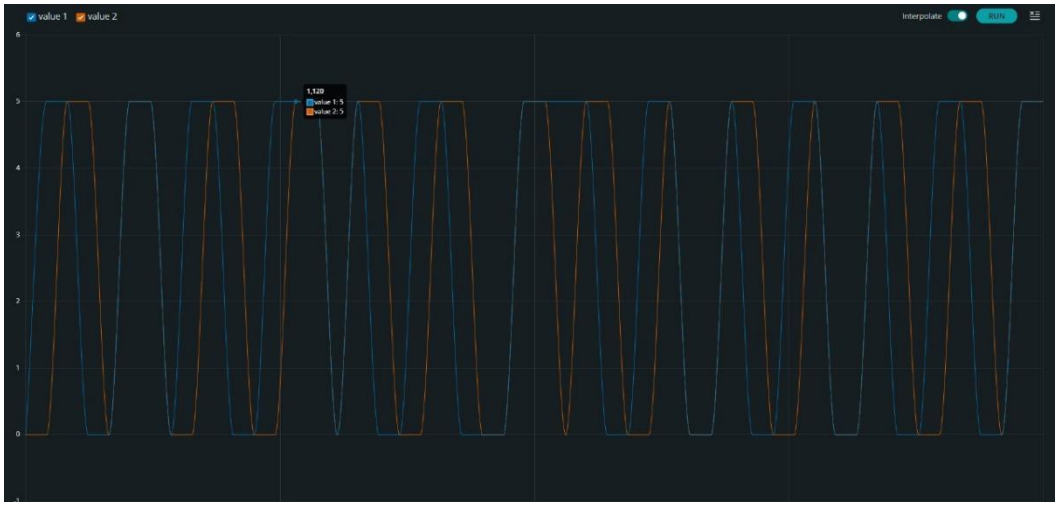
$$\tau = \frac{R * J_{eq}}{k_m} = \frac{23 * 2.6325 * 10^{-6}}{0.6881} = 8.799 * 10^{-5}s$$

Transfer Function

The transfer function of a DC motor describes the relationship between the input voltage and the resulting angular position or speed of the motor. To design an effective control system, it's essential to determine this transfer function. In this project, we derive the transfer function experimentally by using encoder values to measure the motor's response to a step input.

- Setup the Arduino & Motor Encoder to take readings.

Motor Encoder Readings:



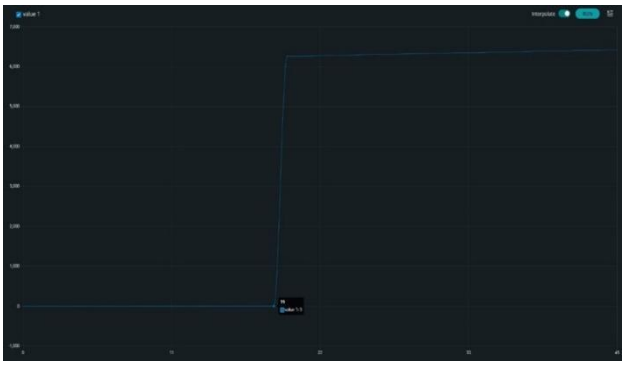
- Write a code that sets the PWM output to a specific value and maintains it.

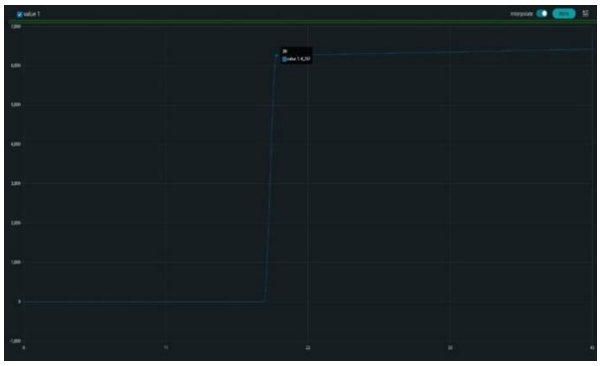
Code:

```
const int pwmPin = 9;
const int encoderPinA = 2;
const int encoderPinB = 3;
volatile long encoderCount = 0;

void setup() {
  pinMode(pwmPin, OUTPUT);
  pinMode(encoderPinA, INPUT);
  pinMode(encoderPinB, INPUT);
  attachInterrupt(digitalPinToInterrupt(encoderPinA), updateEncoder, CHANGE);
```

```
  attachInterrupt(digitalPinToInterrupt(encoderPinB), updateEncoder, CHANGE);
  Serial.begin(9600);
}

void loop() {
  analogWrite(pwmPin, 128); // Apply a step input (50% duty cycle)
  delay(5000); // Wait for 5 seconds
  analogWrite(pwmPin, 0); // Stop the motor
  Serial.println(encoderCount); // Print encoder count
  delay(1000);
}

void updateEncoder() {
  int stateA = digitalRead(encoderPinA);
  int stateB = digitalRead(encoderPinB);
  if (stateA == stateB) {
    encoderCount++;
  } else {
    encoderCount--;
  }
}
```

- As the motor responds to the step input, record the encoder value.





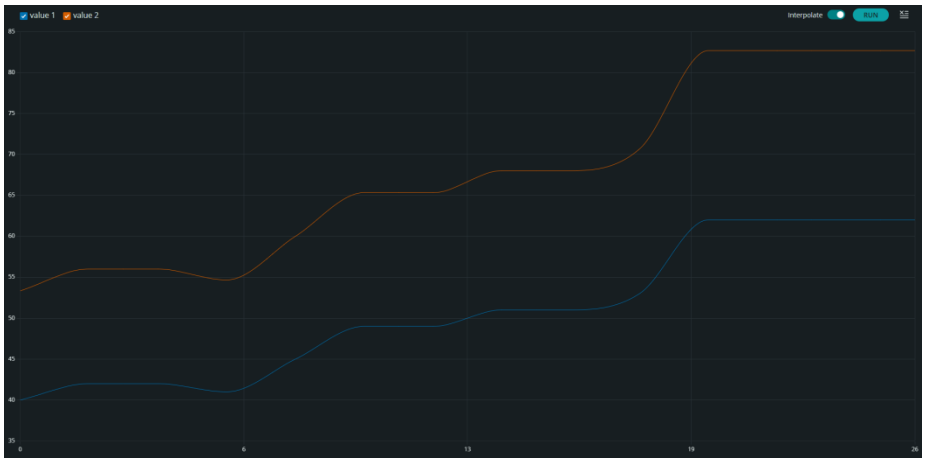
- Conversion:

$$\text{Angular Position} = \left(\frac{\text{Encoder Counts}}{\text{Counts Per Revolution}} \right) \times 360^\circ$$

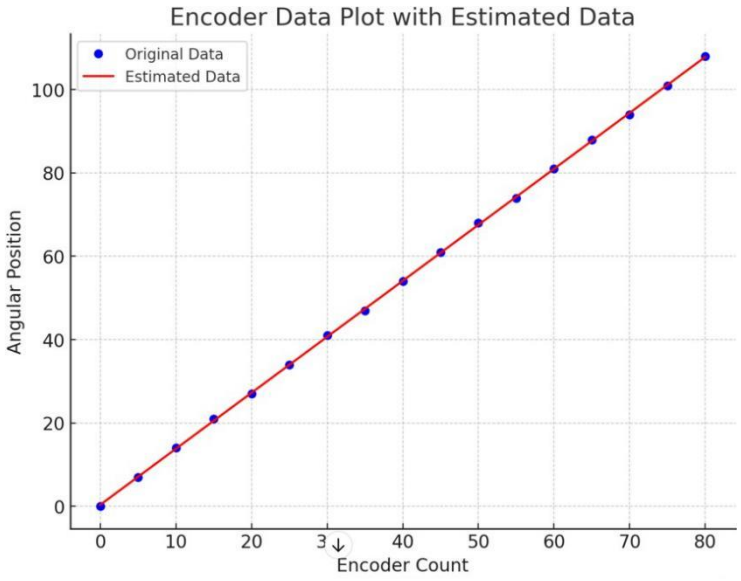
Code:

```
const int countsPerRevolution = 360; // Example value
  float angularPosition = (encoderCount / (float)countsPerRevolution) * 360.0;
```

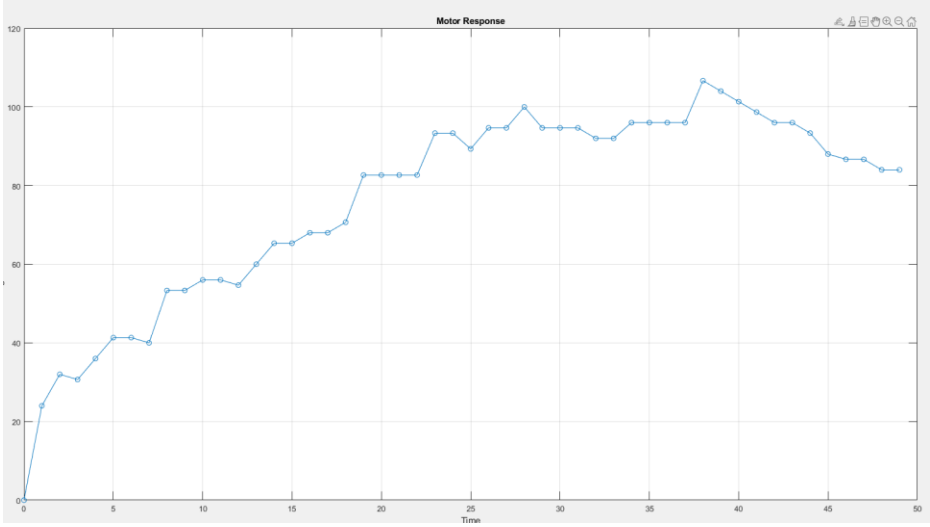
At 180 Degrees:



Step Input Graph at 180 Degrees:



Motor Response:



Derivation:

For a Second-order system:

$$G(s) = \frac{K}{s(\tau s + 1)}$$

- K is the system gain.
- τ is the time constant.

Encoder Values:

```
#include <util/atomic.h> // For the ATOMIC_BLOCK macro

#define ENCA 2 // YELLOW
#define ENCB 3 // WHITE
#define PWM 5
#define IN2 6
#define IN1 7

volatile int posi = 0; // specify posi as volatile

void setup() {
```

```
  Serial.begin(9600);
  pinMode(ENCA, INPUT);
  pinMode(ENCB, INPUT);
  attachInterrupt(digitalPinToInterrupt(ENCA), readEncoder, RISING);

  pinMode(PWM, OUTPUT);
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
}

void loop() {
  // Read the position in an atomic block to avoid a potential
  // misread if the interrupt coincides with this code running
  int pos = 0;
  ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
    pos = posi;
  }

  setMotor(1, 25, PWM, IN1, IN2); // Set motor direction forward with PWM value 25
  delay(200);
  Serial.println(pos); // Print the current position
  setMotor(-1, 25, PWM, IN1, IN2); // Set motor direction backward with PWM value 25
  delay(200);
  Serial.println(pos); // Print the current position
  setMotor(0, 25, PWM, IN1, IN2); // Stop the motor
  delay(20);
  Serial.println(pos); // Print the current position
}

void setMotor(int dir, int pwmVal, int pwm, int in1, int in2) {
  analogWrite(pwm, pwmVal);
  if (dir == 1) {
    digitalWrite(in1, HIGH);
    digitalWrite(in2, LOW);
  } else if (dir == -1) {
    digitalWrite(in1, LOW);
    digitalWrite(in2, HIGH);
  } else {
    digitalWrite(in1, LOW);
    digitalWrite(in2, LOW);
  }
}

void readEncoder() {
  int b = digitalRead(ENCB);
  if (b > 0) {
    posi++;
  } else {
```

```
    posi--;
  }
}
```

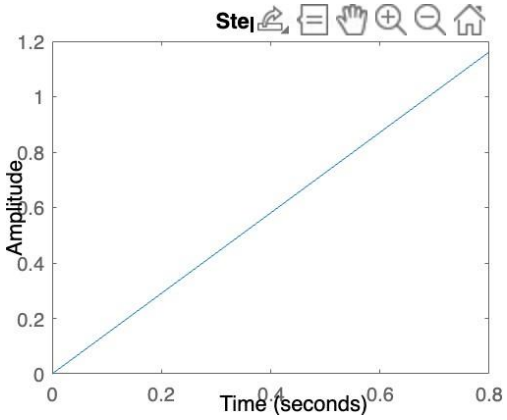
Serial Plotter:

MATLAB Simulation

Step Response:

```
k = 1.4522;
t = 8.79E-5 ;
s = tf('s');
P_motor = k/(s*((t*s+1)));
pole(P_motor);
step(P_motor)
stepinfo(P motor);
```

Graph:

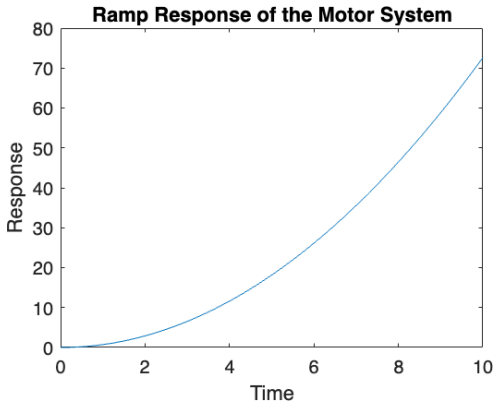


Ramp Response:

```
k = 1.4522;
t = 8.79E-5;
s = tf('s');
P_motor = k/(s*((t*s+1)));

t = 0:0.01:10;


ramp_input = t;


ramp_response = lsim(P_motor, ramp_input, t);


plot(t, ramp_response);
xlabel('Time');
ylabel('Response');
```

```
title('Ramp Response of the Motor System');
```

Graph:

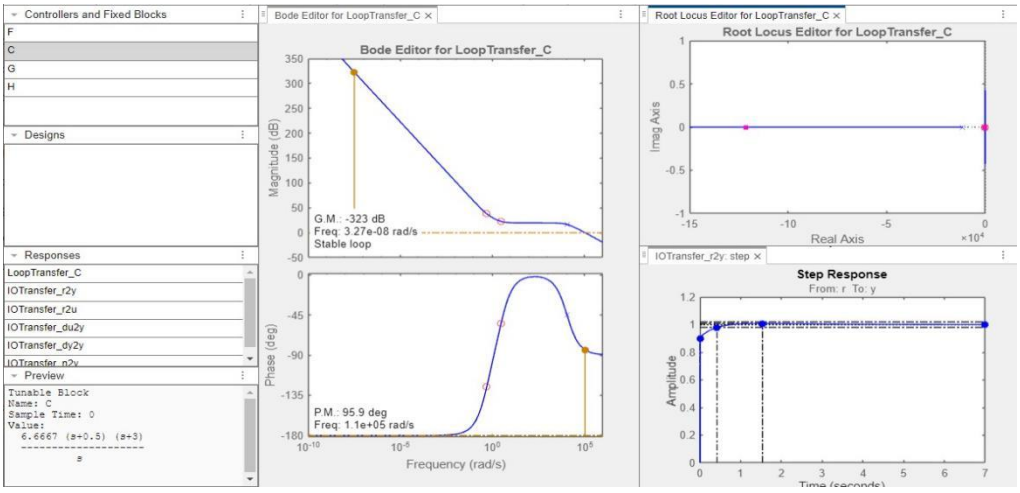


PID Controller

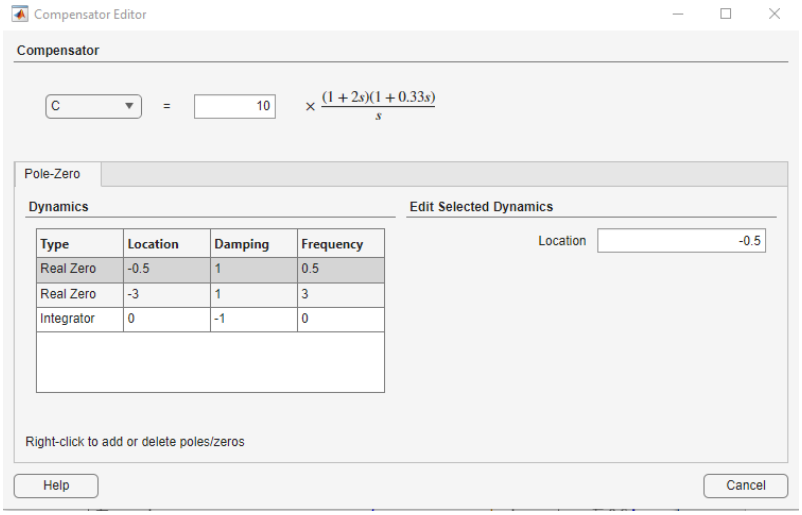
Requirements:

- %OS = 5%
- Rising Time = 0.56 sec
- Steady State Error = 0.001

PID Using SISOTOOL:



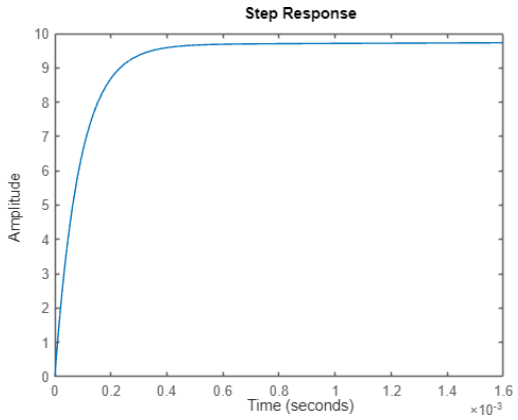
Parameters:



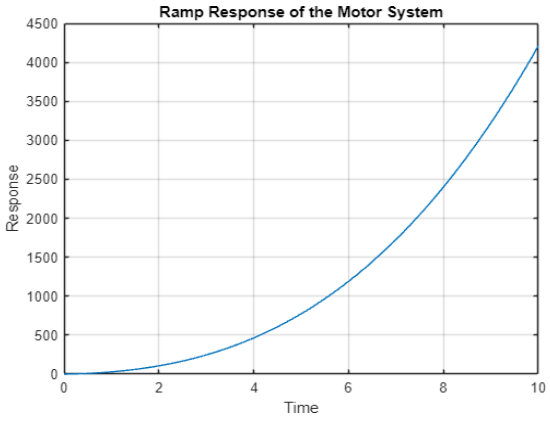
Verification Using MATLAB:

```
t = 0:0.01:10;


ramp_input = t;


ramp_response = lsim(G*C, ramp_input, t);


plot(t, ramp_response);
xlabel('Time');
ylabel('Response');
title('Ramp Response of the Motor System');
grid on;


pole(G*C)
step(G*C)
stepinfo(G*C)
```

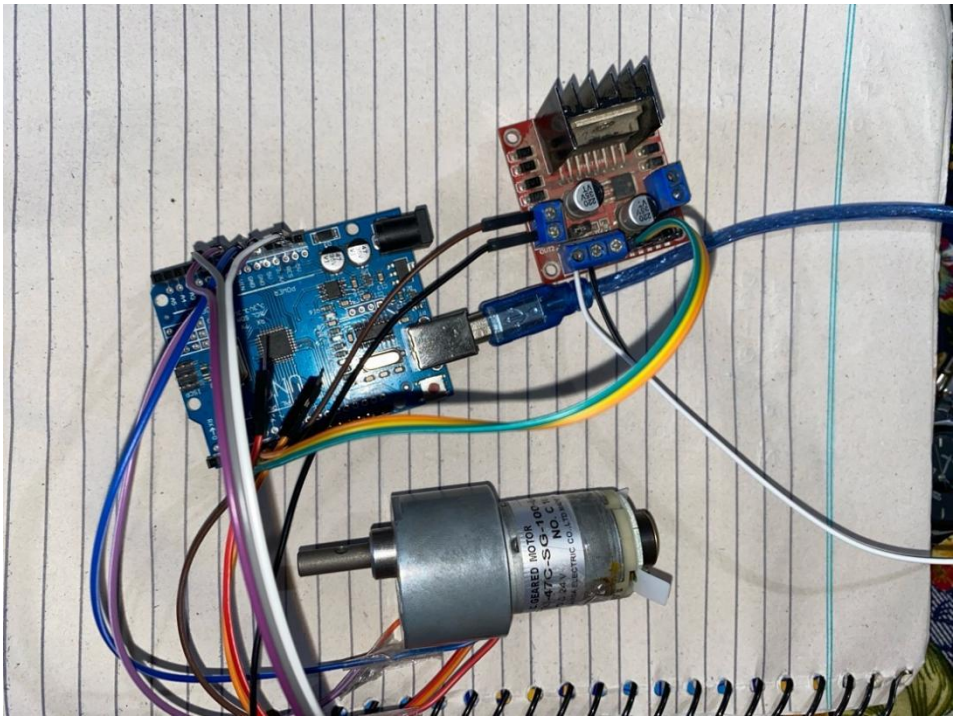
Step Response:



Ramp Response:



Implementation



- **Components Needed:**
  - Arduino (e.g., Arduino Uno)
  - DC motor
  - Motor driver (e.g., L298N or L293D)
  - Power supply (e.g., 12V battery for the motor driver)
  - Connecting wires

- **Connections:**

1. Connecting the DC Motor and Motor Driver:

Motor Driver to Arduino:

- PWM Pin: Connect the motor driver's PWM input to one of the Arduino's PWM-capable pins (e.g., pin 9).
- Direction Pins: Connect the motor driver's direction control pins to two digital pins on the Arduino (e.g., pins 7 and 6).

Motor Driver to Motor:

- Motor Terminals: Connect the motor driver's output terminals to the DC motor's terminals.

Power Supply:

- Motor Driver Power Input: Connect the motor driver's power input to the positive terminal of the 12V battery.
- Common Ground: Connect the ground of the motor driver, the Arduino, and the battery together to ensure they share a common ground.

2. Connecting the Encoder:

Encoder to Arduino:

- Output Pins: Connect the encoder's output pins A and B to two digital pins on the Arduino that support interrupts (e.g., pins 2 and 3).
- Power Supply Pins: Connect the encoder's power supply pins to the Arduino's 5V and GND pins.

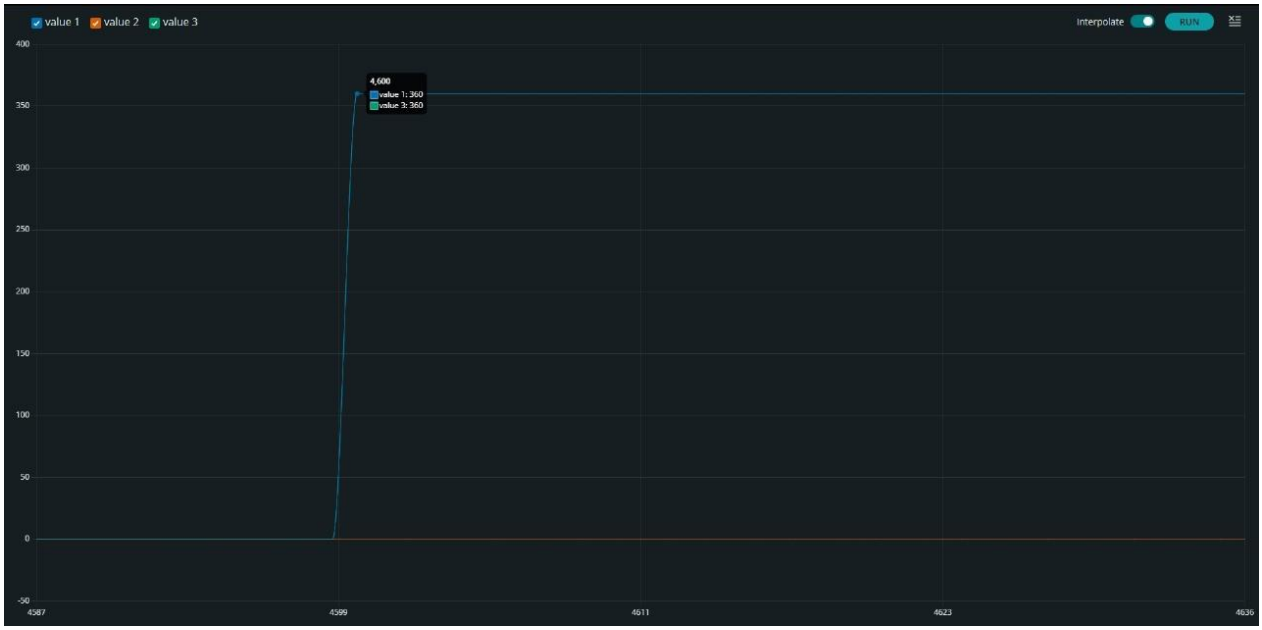
PID Controller Design:

```
// Define pins for PWM control, encoder output A and B, and the number of counts per revolution
const int pwmPin = 9;
const int encoderPinA = 2;
const int encoderPinB = 3;
const int countsPerRevolution = 360; // Update with your encoder's CPR

// Declare variables for PID coefficients, desired position, and encoder count
volatile long encoderCount = 0;
float kp = 1.0, ki = 0.1, kd = 0.01; // PID coefficients
long setpoint = 1000; // Desired position in encoder counts
long previousError = 0;
long  integral  =  0;
unsigned long lastTime = 0;
```

```
void setup() {
  // Set pin modes and attach interrupt routines
  pinMode(pwmPin, OUTPUT); // PWM pin for motor speed control
  pinMode(encoderPinA, INPUT); // Encoder output A
  pinMode(encoderPinB, INPUT); // Encoder output B
  attachInterrupt(digitalPinToInterrupt(encoderPinA), updateEncoder, CHANGE); //
Interrupt on change for encoder A
  attachInterrupt(digitalPinToInterrupt(encoderPinB), updateEncoder, CHANGE); //
Interrupt on change for encoder B
  Serial.begin(9600); // Initialize serial communication
}

void loop() {
  // Record current time and elapsed time since last loop iteration
  unsigned long currentTime = millis();
  unsigned long elapsedTime = currentTime - lastTime;

  // Calculate PID terms
  long error = setpoint - encoderCount;
  integral += error * elapsedTime;
  long derivative = (error - previousError) / elapsedTime;
  long controlSignal = kp * error + ki * integral + kd * derivative;

  // Constrain control signal to the allowable range (0-255 for analogWrite)
  controlSignal = constrain(controlSignal, 0, 255);

  // Write control signal to the PWM pin to control motor speed
  analogWrite(pwmPin, controlSignal);

  // Update variables for next loop iteration
  previousError = error;
  lastTime = currentTime;

  // Convert encoder counts to angular position in degrees
  float angularPosition = (encoderCount / (float)countsPerRevolution) * 360.0;

  // Print encoder count and angular position
  Serial.print("Encoder Count: ");
  Serial.print(encoderCount);
  Serial.print(" Angular Position: ");
  Serial.println(angularPosition);

  // Delay for stability
  delay(100);
}
```

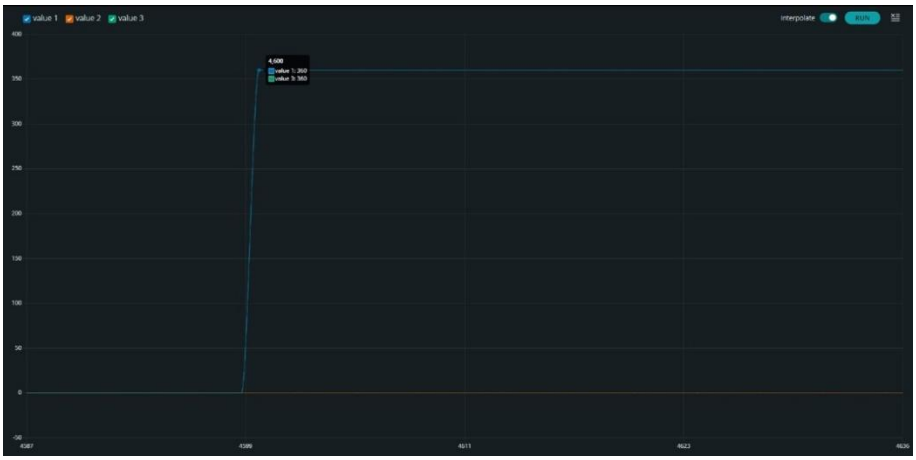
```
// Interrupt service routine to update encoder count based on changes in encoder pins
A and B
void updateEncoder() {
  int stateA = digitalRead(encoderPinA);
  int stateB = digitalRead(encoderPinB);
  if (stateA == stateB) {
    encoderCount++; // Increment count if A and B are in the same state
  } else {
    encoderCount--; // Decrement count if A and B are in different states
  }
}
```

PID Controller Output:

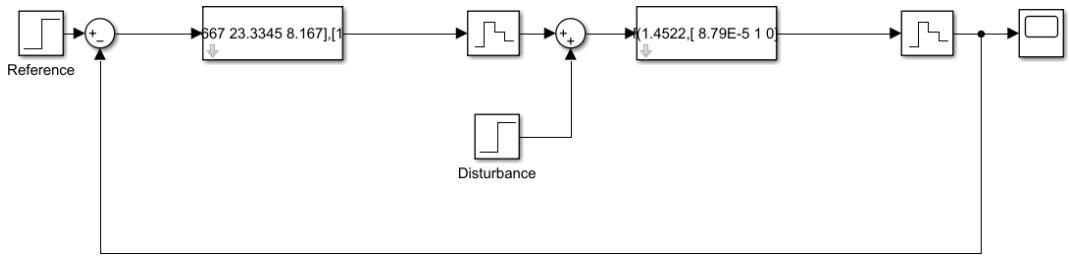


PID Control Application:

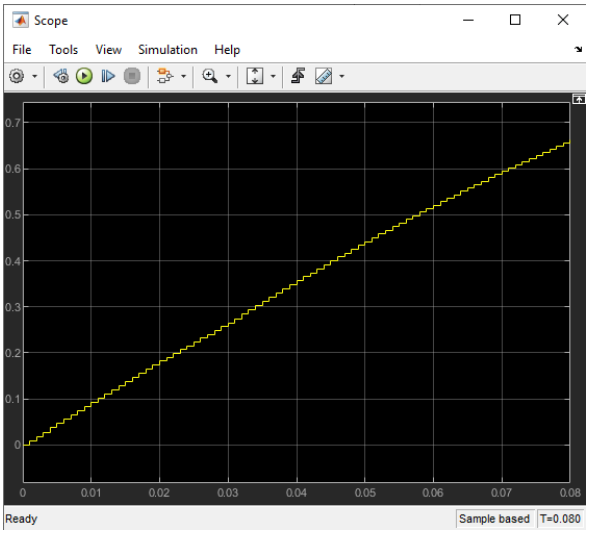
- The PID controller was implemented and tuned using the PID coefficients: $K_p = 23.33$, $K_i = 10$, $K_d = 6.6667$.
- Tests were repeated to measure the motor's response with the PID controller active.



Simulink Model



Output:



Discussion

We focused on implementing closed-loop control for the position of a DC motor using various control algorithms, including P, I, PI, PD, and PID controllers. The process involved modeling the DC motor, deriving its transfer function, designing the controller using MATLAB's root locus or "sisotool," and implementing it on an Arduino Uno. The system utilized a shaft encoder for feedback, enabling precise positioning of the motor.

Key components of the project included:

- **Arduino Uno:** Utilized as the microcontroller to implement the closed-loop control system.
- **DC Motor with Encoder:** Used for position control, with feedback provided by the shaft encoder.
- **Motor Driver:** Controlled by the Arduino to drive the DC motor, enabling bidirectional movement.
- **Power Supply:** Provided power to the motor driver for motor operation.
- **Simulink Interface:** Enabled design, simulation, and implementation of control algorithms, with the Arduino receiving control signals and sending real-time data to Simulink.

The main focus was on designing and implementing various control algorithms to achieve precise position control of the DC motor. This involved deriving the transfer function of the motor, tuning the PID controller using MATLAB, and verifying its performance through simulations and real-world tests.

Conclusion

In summary, this project demonstrated the successful implementation of closed-loop control for precise positioning of a DC motor. By modeling the motor, designing PID controllers, and interfacing with an Arduino Uno, we showcased the effectiveness of control algorithms in achieving accurate motor control. Through simulations and real-world tests, we validated the system's performance, highlighting the practicality of Arduino-based control systems in engineering applications. Overall, this project bridges theoretical concepts with practical implementation, offering valuable insights into control engineering and its real-world applications.