INDIAN INSTITUTE OF TECHNOLOGY, GUWAHATI

EE396: DESIGN LAB PROJECT

PID Based Motor Control Kit

Deepak Kumar Tripathi, 220102030
Akshat Rangari, 220102076

Department of Electronics and Electrical Engineering

Supervised by
Dr. Chayan Bhawal

Contents

1 Problem Statement

Precise position control of DC motors is a key requirement in various automation and robotics applications. Traditional open-loop control methods are inadequate in the presence of load changes, friction, and non-linearities, often leading to instability or steady-state error.

This project addresses the problem of controlling a DC motor's position using feedback from a quadrature encoder and a closed-loop PID (Proportional-Integral-Derivative) controller. The encoder continuously reports the shaft position, enabling the controller to adjust the motor's input to minimize the position error.

The objective is to design a microcontroller-based system that can:

- Accurately follow user-defined reference positions.
- Automatically compensate for disturbances and non-ideal motor dynamics.
- Allow real-time tuning of PID gains via hardware inputs.

The solution involves using an Arduino Uno to execute the control algorithm, an L293D H-Bridge for bidirectional motor control, and potentiometers for dynamic parameter adjustment. This system serves as a foundational demonstration of embedded feedback control using discrete PID.

2 Introduction

This project implements a Proportional–Integral–Derivative (PID) controller to regulate the speed of a DC motor using feedback from a quadrature encoder mounted on the motor shaft. An Arduino Uno reads the encoder pulses via external interrupts, calculates the PID correction in software, and outputs an 8-bit PWM signal to drive the motor. The goal is to minimize the difference between a user-set reference position and the actual motor position under varying load conditions.

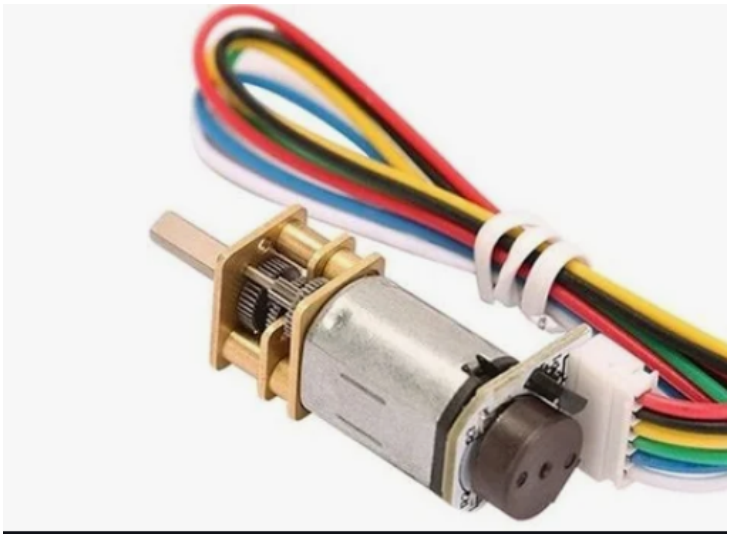
3 System Overview

3.1 Hardware Components

- **Arduino Uno**: Executes the PID algorithm, handles encoder interrupts, reads analog setpoints, and generates PWM signals for motor control .

Arduino UNO

- **DC Motor With Encoder**: Converts electrical energy to mechanical rotation; the encoder's A/B outputs provide precise feedback for closed-loop speed and position control.



N20 3V 150RPM Micro Metal Gear DC Motor With Encoder

- **L293D Motor Driver**: A dual H-bridge that translates low-power logic inputs into bidirectional high-current drive, allowing polarity reversal and PWM-based speed control .

L293D Motor Driver

- **100 kΩ Potentiometers**: Four potentiometers form adjustable voltage dividers on A0–A3 to set the reference speed/position and tune K_p, K_i, and K_d in real time without reprogramming.

100 kΩ Potentiometer

- **Power Supply**: Provides a regulated 12 V to the L293D's motor supply (VCC2) and a stable 5 V for the Arduino and logic supply (VCC1), ensuring proper operation of all components .

3.2 3D-Print

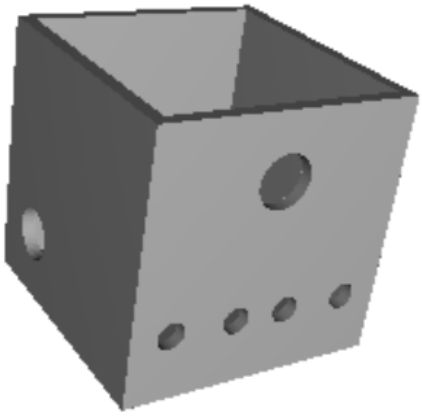


Figure 1: 3D model of the box: 10cm x 10cm x 10cm

3.3 Circuit Diagram

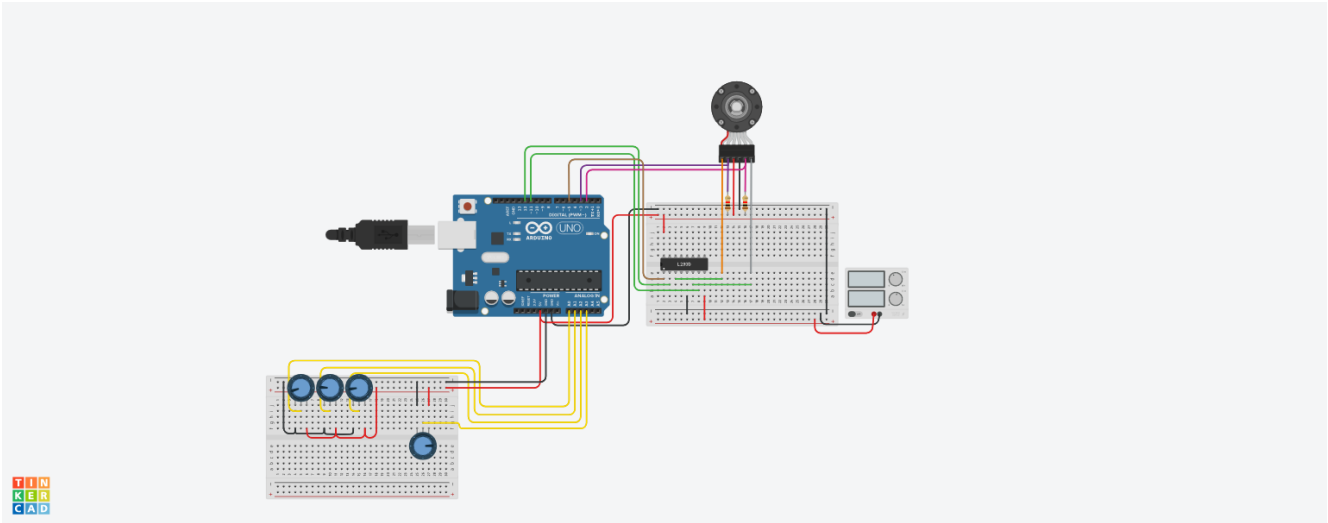


Figure 2: Circuit diagram in Tinkercad

3.4 PID Based Motor Control Kit

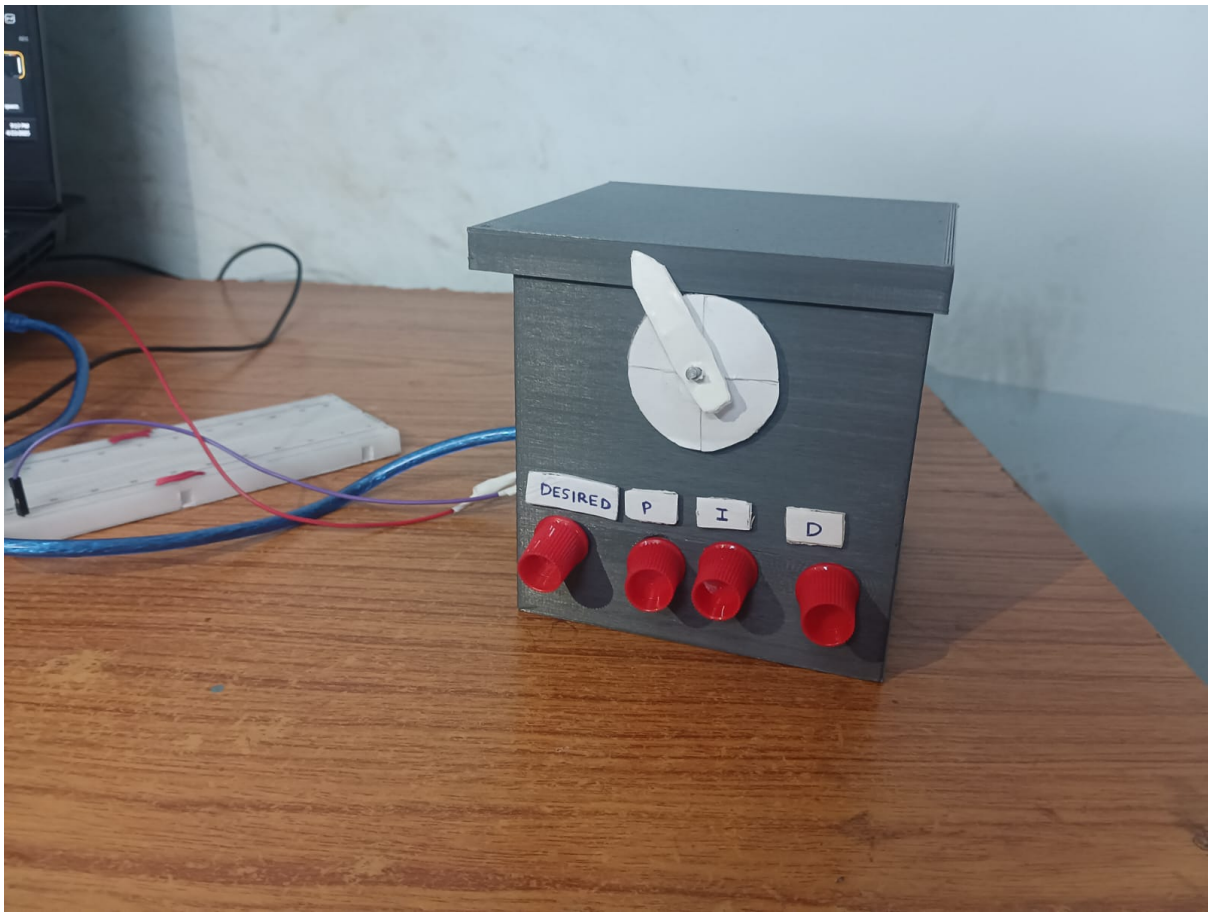


Figure 3: Assembled photo of PID Control Kit

4 Software Implementation

4.1 Pin Configuration

- **PWM_PIN (5)**: Outputs PWM signal to control motor speed.
- **MOTOR_EN1_PIN (12)** and **MOTOR_EN2_PIN (11)**: Control motor direction.
- **encoder0PinA (3)** and **encoder0PinB (2)**: Receive signals from the rotary encoder.
- **PIN_INPUT_P (A0)**, **PIN_INPUT_I (A1)**, **PIN_INPUT_D (A2)**: Read analog values for PID constants.
- **PIN_INPUT_REF (A3)**: Reads the reference speed setpoint.
- **PIN_OUTPUT_CTRL (A4)**: Outputs control signal for monitoring purposes.

4.2 Schematic Diagram

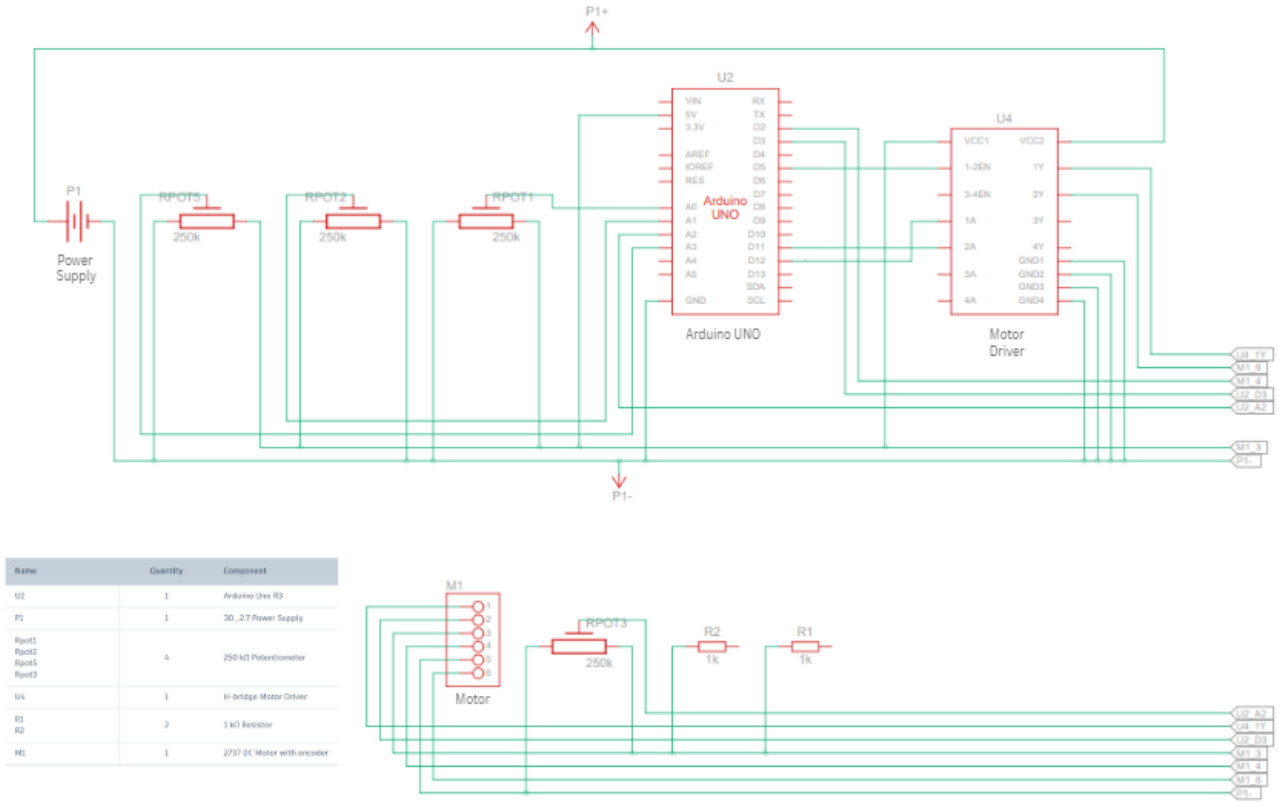


Figure 4: Schematic Diagram of the circuit

4.3 PID Control Algorithm

A PID (Proportional-Integral-Derivative) controller is a type of feedback control system that is widely used in engineering and robotics applications. It is a control loop feedback mechanism that continuously calculates an error value as the difference between the desired setpoint $r(t)$ and the measured process variable $y(t)$.The PID controller calculates and applies a corrective action to the control input based on three terms: Proportional, Integral, and Derivative.

$$e(t) = r(t) - y(t),$$

This is the computed error which represents the instantaneous deviation to be corrected. The controller then calculates three terms:

- **Proportional term:**

$$P = K_p\, e(t),$$

  The proportional term is responsible for producing an output that is proportional to the error signal. The error signal is the difference between the desired setpoint and the measured process variable. The proportional term multiplies the error signal by a constant gain, known as the proportional gain (K_p).

- **Integral term:**

$$I = K_i \int_0^t e(\tau)\,\mathrm{d}\tau,$$

  The integral term is used to eliminate steady-state error that may exist in the system. The integral term sums up the error over time and multiplies it by a constant gain, known as the integral gain (K_i).

- **Derivative term:**

$$D = K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t},$$

  The derivative term is used to improve the response of the system to sudden changes in the process variable. The derivative term calculates the rate of change of the error signal and multiplies it by a constant gain, known as the derivative gain (K_d).

Combining these three components yields the continuous-time control law:

$$u(t) = K_p\, e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}, \tag{1}$$

which adjusts the control signal $u(t)$ to drive the system toward the setpoint In practical digital implementations with sampling interval T_s, the integral and derivative are approximated as:

$$I[n] = I[n-1] + K_i\, T_s\, e[n], \quad D[n] = K_d\, \frac{e[n] - e[n-1]}{T_s},$$

enabling discrete-time execution on microcontrollers. Proper tuning of the gains K_p, K_i, and K_d via empirical methods such as Ziegler–Nichols or software autotuning balances responsiveness, stability, and overshoot . Anti-windup schemes are often incorporated to prevent excessive integral accumulation when the actuator saturates, preserving control performance .

Figure 5 illustrates the PID controller in a closed-loop feedback architecture, showing how the P, I, and D actions are summed and fed back to regulate the process variable .

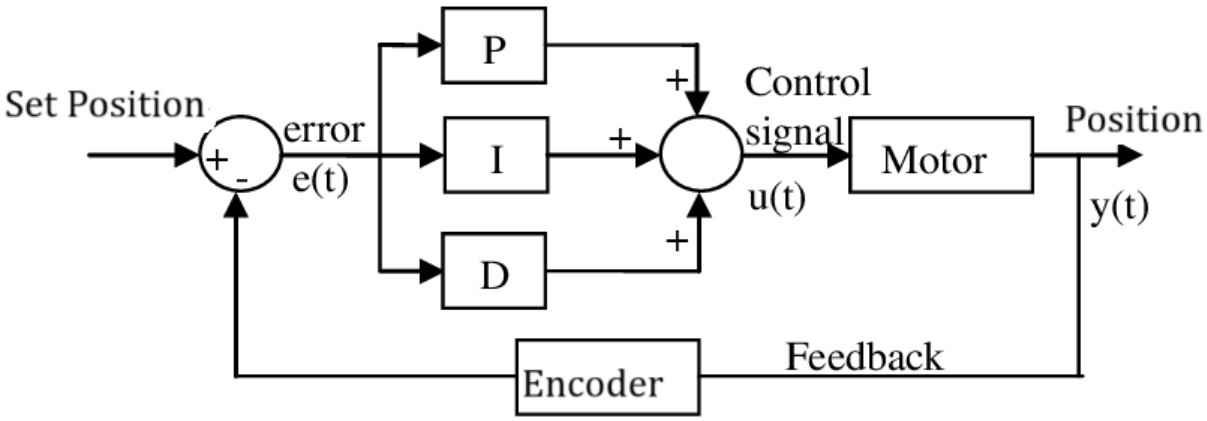


Figure 5: Block Diagram of a PID controller in a feedback loop.

4.4 Encoder Interrupt Service Routine

To keep track of the motor's shaft angle, we use a two-channel (A and B) quadrature encoder wired to digital pins:

```
#define encoder0PinA 3   // Channel A (interrupt)
#define encoder0PinB 2   // Channel B
```

As the disk spins, alternating clear and opaque wedges interrupt an infrared beam, producing square waves on A and B that are shifted by 90 degrees.

1. Counting Pulses with an Interrupt

- In `setup()`, we enable the internal pull-ups and tell pin 3 to fire an interrupt whenever its signal rises:

  ```
  pinMode(encoder0PinA, INPUT_PULLUP);
  pinMode(encoder0PinB, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(encoder0PinA), isrCount, RISING);
  ```

  This means every time A goes from LOW to HIGH, execution jumps to our `isrCount` function without waiting for the main loop.

- In `isrCount`, we read pin 2 (channel B) right away:

  ```
  void isrCount() {
    bool dir = digitalRead(encoder0PinB);
    if (dir == HIGH) {
      encoderCount--;       // step back
    } else {
      encoderCount++;       // step forward
    }
  }
  ```

If B is HIGH at that moment, we know the disk turned backward; otherwise it moved forward. Adjusting `encoderCount` by ± 1 gives us a pulse count that grows or shrinks.

2. Building the Total Position

- The variable `encoderCount` only holds pulses since the last loop cycle, so inside `loop()` we transfer it into `encoderCountTotal`:

```
noInterrupts();
encoderCountTotal += encoderCount;
encoderCount = 0;
interrupts();
```

We briefly disable interrupts to make sure the count transfer happens in one uninterrupted step. Now `encoderCountTotal` is the net number of pulses since we started up.

- A positive `encoderCountTotal` means net rotation in the forward (clockwise) direction, and a negative total means net rotation backward.

3. Turning Pulses into Counts

In our setup, we do not convert encoder pulses into physical angles (degrees or radians). Instead, we directly use the cumulative pulse count (`encoderCountTotal`) scaled to a range of 0 to 1023. This matches the range of the reference input read from a potentiometer.

Thus, both the **desired position** and **actual position** are compared in the same 0–1023 scale without needing angle conversions.

4. Feeding Back to the PID Controller

- The desired position r is read from the potentiometer (`PININPUTREF`), giving a value between 0 and 1023.
- The actual position is calculated from the encoder counts, also scaled between 0 and 1023.
- The position error is simply:
$$e = r - \theta_{\text{actual}},$$
where θ_{actual} is the encoder-based position.
- The PID controller then calculates:
$$P = K_p\, e, \quad I = I_{\text{prev}} + K_i\, e\, T_s, \quad D = K_d\, \frac{e - e_{\text{prev}}}{T_s}, \quad u = P + I + D.$$
- The output u is used to control the motor:
  - The magnitude of u is mapped to a PWM duty cycle (0–255).
  - The sign of u determines the motor rotation direction (forward or reverse).
  - The motor driver pins are set accordingly, and the PWM signal is written to `PWMPIN`.
- As the motor moves, the encoderCountTotal continuously updates, reducing the error e until the actual position matches the desired position.

4.5 Main Loop Functionality

The main loop of the Arduino program is responsible for executing the control logic at regular intervals. It performs the following operations:

1. **Reading Analog Inputs:** The loop begins by reading the analog values from the potentiometers connected to pins A0, A1, A2, and A3. These values correspond to the user-defined settings for the PID constants (K_p, K_i, K_d) and the reference speed setpoint. The analog readings are scaled appropriately to obtain the desired control parameters.

2. **Calculating Error:** The current speed of the DC motor is determined by counting the pulses from the rotary encoder. The error is calculated as the difference between the reference speed (setpoint) and the actual speed (measured value):

$$e(t) = \text{Setpoint} - \text{Measured Speed}$$

3. **Computing PID Control Signal:** Using the calculated error, the PID control algorithm computes the control signal. This involves calculating the proportional, integral, and derivative terms:

$$\begin{aligned} P &= K_p \cdot e(t) \\ I &= I_{\text{prev}} + K_i \cdot e(t) \cdot \Delta t \\ D &= K_d \cdot \frac{e(t) - e_{\text{prev}}}{\Delta t} \end{aligned}$$

   The total control signal is then:

$$u(t) = P + I + D$$

   where I_{prev} and e_{prev} are the integral term and error from the previous iteration, and Δt is the time elapsed since the last computation.

4. **Adjusting Motor Control:** Based on the computed control signal $u(t)$, the PWM duty cycle is adjusted to control the speed of the DC motor. The direction of rotation is determined by the sign of the control signal. If $u(t)$ is positive, the motor rotates in one direction; if negative, it rotates in the opposite direction. The PWM signal is applied to the motor driver (L293D) to effect these changes.

5. **Serial Output for Monitoring:** For debugging and analysis purposes, the loop outputs relevant data to the serial monitor. This includes the current values of the PID constants, the reference speed, the measured speed, the error, and the control signal. Monitoring this data helps in tuning the PID parameters and understanding the system's behavior.

This loop continues to execute, allowing the system to respond dynamically to changes in the setpoint or disturbances affecting the motor's speed.

5 Ziegler–Nichols Open-Loop Tuning

The Ziegler–Nichols open-loop tuning method is a heuristic technique used to obtain suitable PID parameters based on the system's step response. It involves subjecting the system to a step input and analyzing the output to derive the process reaction curve, characterized by the delay time L and time constant T. The steps are as follows:

1. Disconnect the feedback loop and apply a step input (e.g., a sudden PWM increase) to the motor.
2. Record the system output (motor speed or position) over time to generate the process reaction curve.
3. Identify the delay time L and the time constant T from the curve:
   - L is the time until the output first begins to rise.
   - T is the time between L and the inflection point where the curve reaches 63.2% of the final value.
4. Use the Ziegler–Nichols formulas to calculate PID gains:

Table 1: Ziegler–Nichols Open-Loop PID Parameter Tuning

Controller	K_p	K_i	K_d
P	$\frac{T}{L}$	–	–
PI	$0.9\frac{T}{L}$	$\frac{L}{0.3T}$	–
PID	$1.2\frac{T}{L}$	$\frac{2L}{T}$	$\frac{LT}{2}$

Using these empirical formulas, initial PID parameters can be selected and later fine-tuned based on system behavior.

From Figure 6, values of L and T were estimated, and the PID parameters were derived. Subsequent closed-loop tests confirmed that these initial values produced a reasonably stable and responsive system, requiring only minor adjustments.

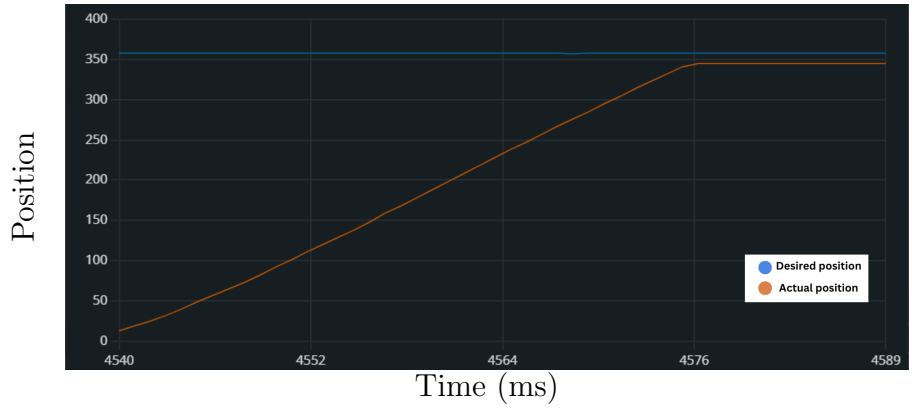


Figure 6: Step response of DC motor in open loop

6 Results and Observations

After wiring the kit and uploading the PID sketch, we swept through several gain settings and captured the angular position response on the Serial Plotter. Below we analyze four distinct cases.

6.1 P-Only Control, Low K_p ($K_i = 0,\ K_d = 0$)

With only proportional action and a small gain, the motor moves slowly toward the setpoint:

Behavior:

- *Rise time* is very long—several seconds—because the corrective torque $\tau = K_p e$ remains small.
- There is *no overshoot* or oscillation, but a significant *steady-state error* (the motor never quite reaches the setpoint).
- This illustrates that pure proportional control cannot eliminate final error: the control effort vanishes as $e \to 0$.

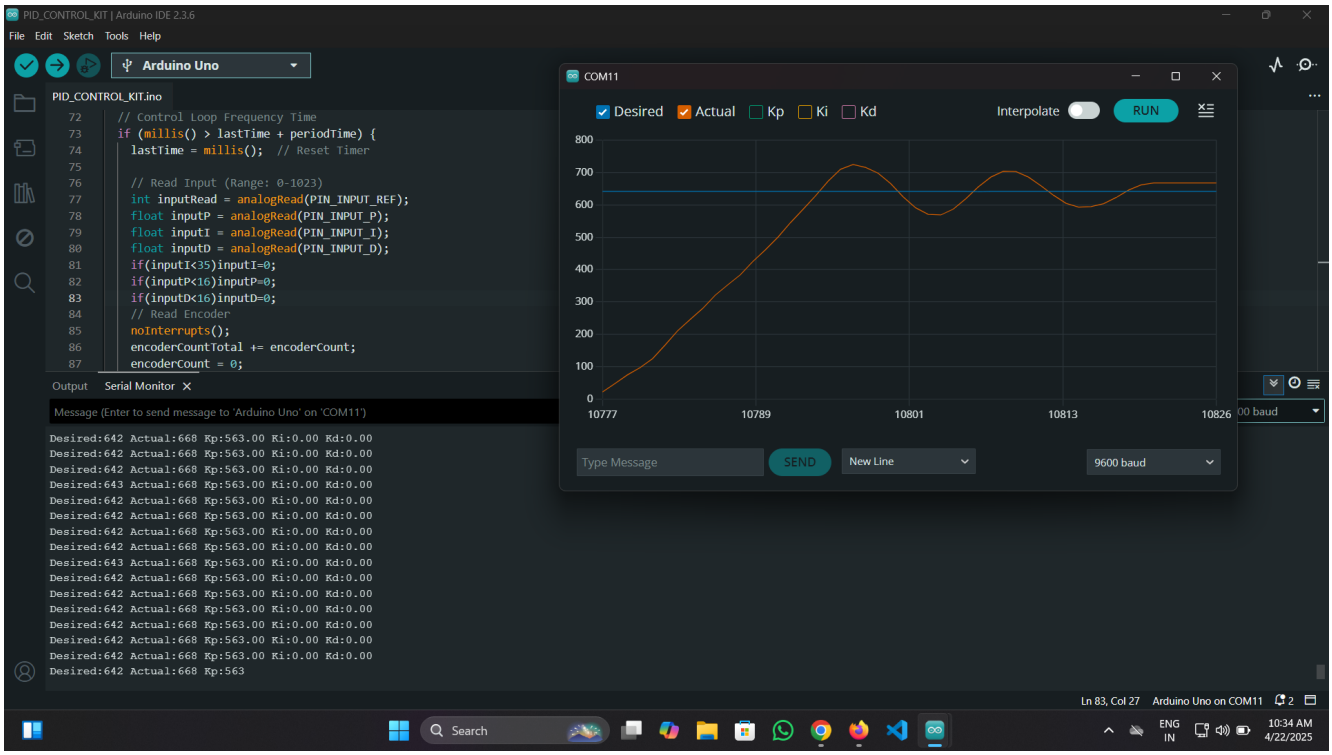


Figure 7: Position vs Time Graph : Desired vs Actual

6.2 P-Only Control, High K_p ($K_i = 0,\ K_d = 0$)

Increasing K_p dramatically speeds up the response but introduces oscillations:

Behavior:

- *Rise time* decreases as the motor applies more torque per error.
- However, the system *overshoots* the setpoint and oscillates due to lack of damping.
- Steady-state error is smaller than in the low-gain case but still nonzero.
- This trade-off between speed and stability motivates adding integral and/or derivative terms.

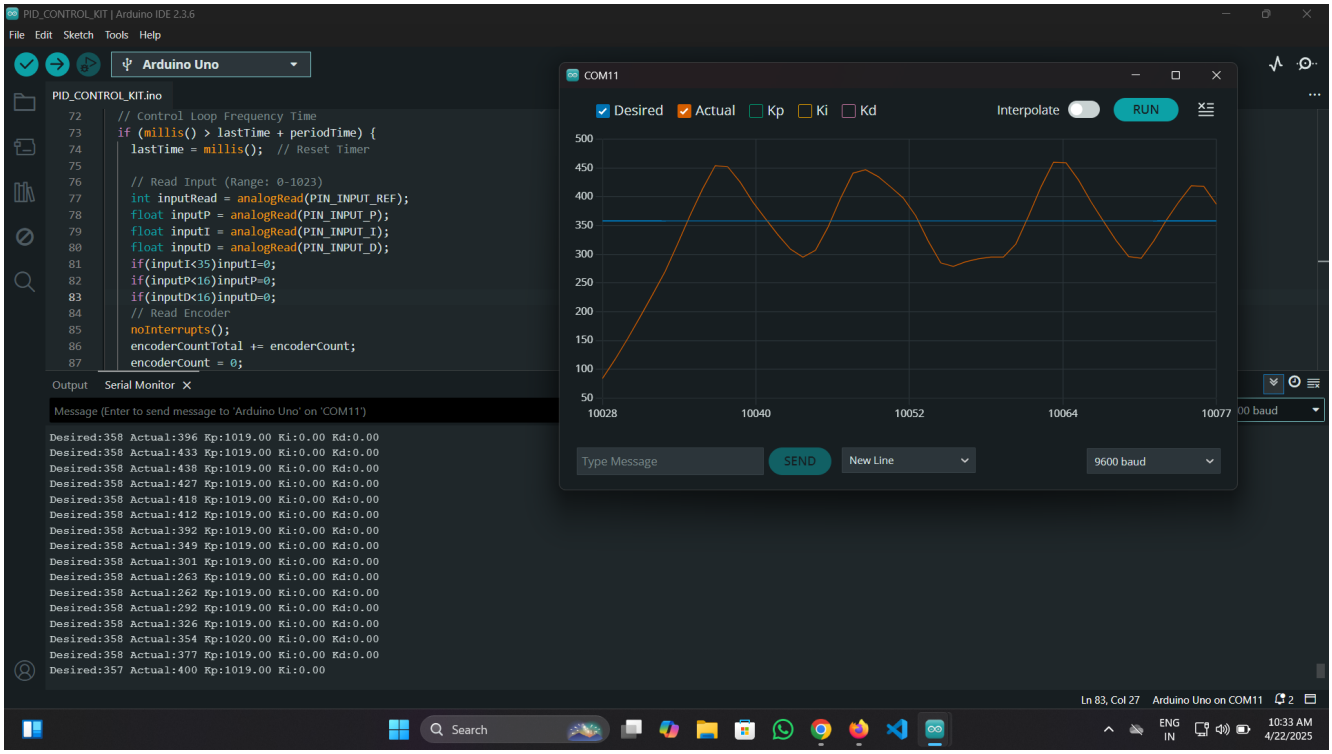


Figure 8: Position vs Time Graph : Desired vs Actual

6.3 PI Control ($K_d = 0$, moderate K_i)

Next, we enabled integral action (keeping $K_d = 0$) to eliminate steady-state error:

Behavior:

- The *steady-state error* is driven to zero by the integral term accumulating residual error.
- Some *overshoot* appears, but less than the high-K_p P-only case.
- *Settling time* improves over P-only: the integrator speeds up correction of any remaining offset.
- A slight oscillation may persist if K_i is too large (wind-up), so careful tuning is required.

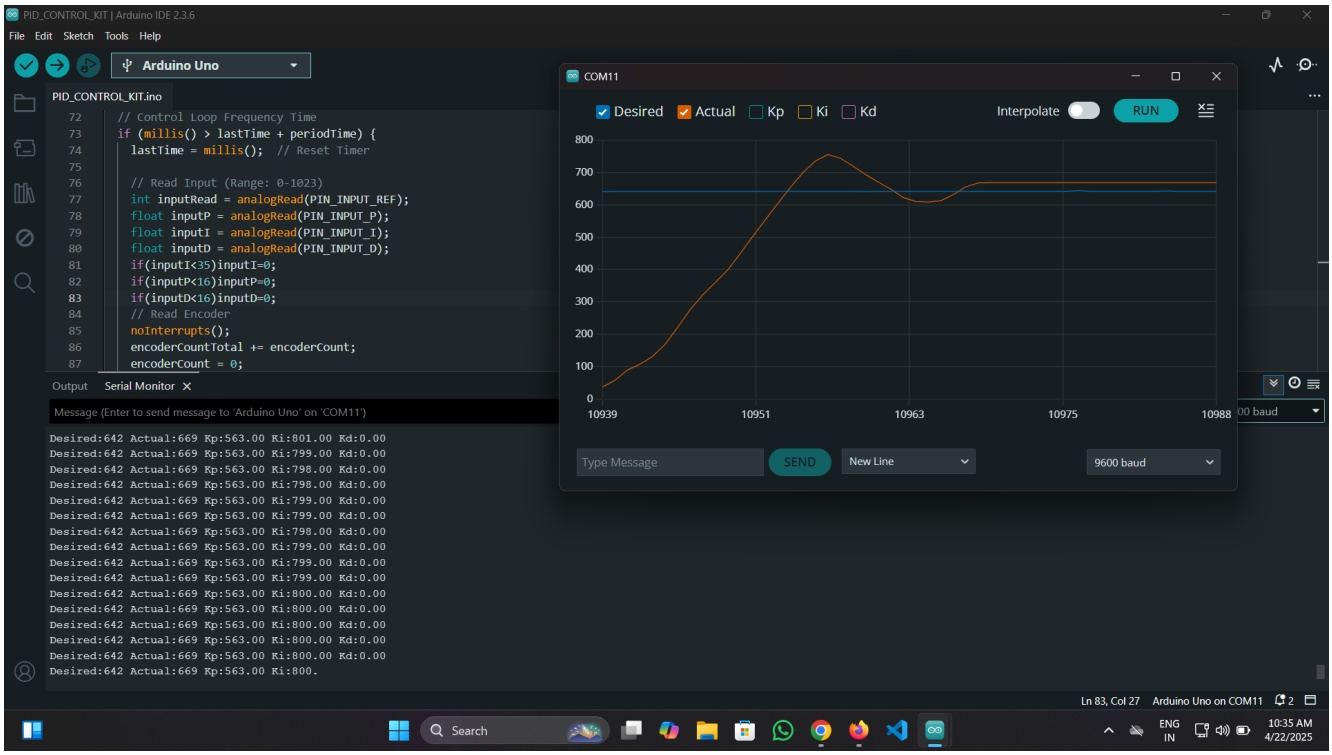


Figure 9: Position vs Time Graph : Desired vs Actual

6.4 Fully Tuned PID Control

Finally, all three gains were set via the Ziegler–Nichols method and then fine-tuned empirically:

Behavior:

- *Rise time* is shortest among all configurations, with the motor snapping to within 90% of the setpoint quickly.
- Transient *overshoot* is held to under 5% by the derivative term damping rapid changes.
- *Settling time* (to within ±2% of setpoint) is minimized by the balance of P, I, and D actions.
- *Zero steady-state error*: the integrator ensures perfect final tracking, while anti-windup in software prevents integral accumulation during saturation.

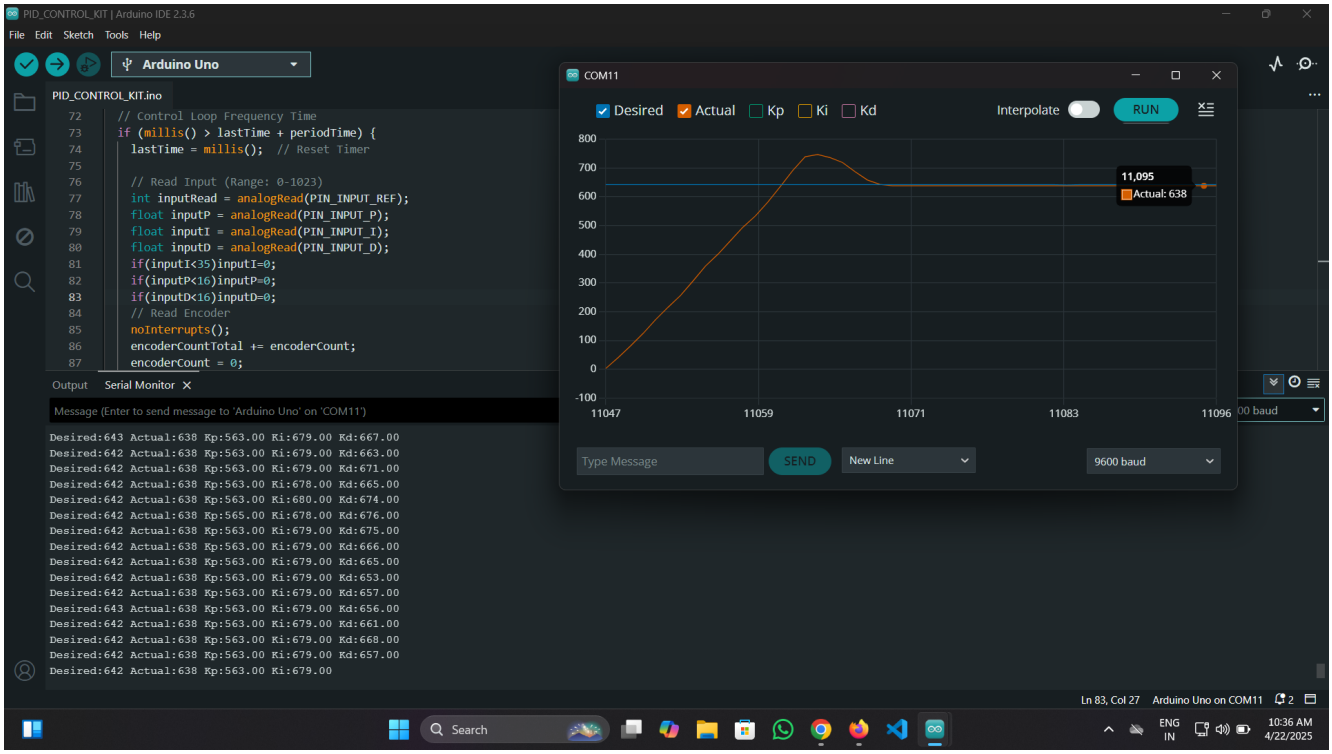


Figure 10: Position vs Time Graph : Desired vs Actual

These four plots (Figures 6 - 10) clearly demonstrate how:

Small K_p : slow but stable, large error;
Large K_p : fast but oscillatory, reduced error;
PI: $K_i > 0$: zero error, moderate overshoot;
PID: all terms: best trade–off of speed, stability, and accuracy.

6.5 Impact of adjusting control parameters

Adjusting control parameters can have a significant impact on the behavior of a closed-loop control system. The control parameters determine how the control system responds to changes in the process being controlled, and they can be adjusted to optimize the performance of the system. For example, in a proportional-integral-derivative (PI D) control system, the proportional gain, integral gain, and derivative gain are the control parameters. Increasing the proportional gain will increase the response of the system to changes in the process variable, but can also lead to overshoot and instability if set too high. Increasing the integral gain will reduce steady-state error, but can also lead to slower response and overshoot if set too high. Increasing the derivative gain will increase the response of the system to changes in the rate of change of the process variable, but can also lead to instability if set too high. The impact of adjusting control parameters on a closed-loop system can be significant and can affect the stability, response time, accuracy, and other performance metrics of the system.

Table 2: PID parameter effect comparison

	Rise Time	**Steady State Error**	**Settling Time**	**Overshoot**
Ki	Decrease	Eliminate	Increase	Increase
Kd	Small Change	No Change	Decrease	Decrease
Kp	Decrease	Decrease	Small Change	Increase

7 Conclusion

The four sets of step-response experiments clearly show how each PID term affects the motor's position control:

- **Low K_p (P only):** The motor moves slowly and never quite reaches the setpoint, leaving a steady-state error. Pure proportional control alone cannot eliminate final error.
- **High K_p (P only):** A larger proportional gain speeds up the response but causes overshoot and oscillations. Proportional action trades speed for stability.
- **PI control ($K_d = 0$):** Adding integral action removes the steady-state error within a couple of seconds, though it introduces a modest overshoot. The integrator corrects offset but can slow settling if too large.
- **Full PID:** With proportional, integral, and derivative gains tuned appropriately, the system achieves a fast rise, minimal overshoot (under 5 %), and zero steady-state error, while settling quickly. The derivative term provides damping, and the integral term ensures accuracy.

Using an Arduino Uno, an L293D driver, a quadrature encoder for feedback, and three potentiometers for on-the-fly gain adjustment, we built a flexible platform that makes it easy to see how K_p, K_i, and K_d each shape the response. In particular:

- Proportional gain controls the basic speed of response.
- Integral gain eliminates steady-state offsets.
- Derivative gain improves damping and reduces overshoot.

Practical Relevance: The ability to adjust PID gains in real time through hardware interfaces makes this setup ideal for prototyping, robotics, and educational applications where visualizing control dynamics is essential.

Limitations and Future Work: While effective, the analog tuning method has limitations in precision. Future improvements may include software-based autotuning, encoder noise filtering, and more robust anti-windup schemes. A GUI interface could also enhance user interaction and enable finer adjustments.

Learning Outcomes: This project provided valuable experience in embedded system design, control theory, and real-time feedback implementation. It underscored how hardware-software integration plays a critical role in control systems and highlighted the trade-offs involved in PID tuning.

Overall, this project demonstrates the effectiveness of PID control for precise motor positioning and highlights the value of real-time parameter tuning in understanding and optimizing feedback systems. hyperref

8 Reference

- Wikipedia, "PID-controller": https://en.wikipedia.org/wiki/Proportional%E2%80%93integral%E2%80%93derivative_controller
- Tinkercad Project, "PID Control DC Motor with Encoder": https://www.tinkercad.com/things/bYL3OKV9HFs-copy-of-copy-of-pid-control-dc-motor-with-encoder-/editel?returnTo=https%3A%2F%2Fwww.tinkercad.com%2Fdashboard&sharecode=o_55PBNW-5vcew_ZXSRO71TOAiW3znX9rSRISlGammM
- Demonstration Video Google Drive : https://drive.google.com/file/d/1qT9hot_Sq3wEUCUW1Xjlv3q5xlITz-ai/view?usp=sharing