Module-03:
Speed Control of a DC Motor Using PID Controller

A: Design of PID Controller for DC Motor Speed Control

In this module, the touchless dustbin system is upgraded with a PID-based speed control mechanism for the DC motor. Because the speed at which the motor operates directly affects how quickly the lid opens and closes, precise speed regulation is crucial for consistent and reliable performance. You will use the same DC motor from the previous setup, removing it from the dustbin to focus on implementing closed-loop speed control of the DC motor. The motor's rotational speed (RPM) will be measured using a quadrature encoder and actively maintained at a user-selected reference value, chosen via a DIP switch. Both the target RPM and the real-time measured RPM will be monitored and displayed using the Arduino IDE serial monitor, ensuring clear feedback and control throughout the process.
This experiment will help you understand:

- How to connect and utilize a DC motor with a built-in quadrature encoder for real-time speed and direction sensing.
- Reading DIP switch or digital input states to specify user-defined speed setpoints.
- The principles and practical implementation of PID controllers on embedded hardware.
- The role of PPR (Pulses Per Revolution), gear ratio of a DC motor and how these influence accurate speed and position measurement with integrated encoders.
- Real-time monitoring and interpretation of feedback and control signals using the Arduino serial monitor and plotter.

Objectives: The objectives of this module are listed below. The methodology to complete the steps are given in Annexure-A.

1. Accurately measure the DC motor's rotational speed (RPM) and direction with the attached encoder.
2. Select desired RPM setpoints using a DIP switch, as specified:

DIP Switch State	Reference Speed
0	90% of `r`
1	80% of `r`
2	70% of `r`
3	60% of `r`

Here, `r` is the maximum RPM at rated voltage for the supplied motor, determined by experiment.

3. Display real-time current and target RPMs in the following serial format:

```
Des: ..... and Cur: .....
```

4. Design, implement, and tune a PID controller to achieve setpoint tracking. Begin with $K_I = K_D = 0$, tune K_P, then incrementally introduce K_I and K_D to optimize settling time, overshoot, and error. Expected overshoot is less than 20% and settling time is less than 4 second.

Pre-Lab Work:

1. Study DC motor speed control via PWM and the L293D driver
   `https://microcontrollerslab.com/dc-motor-l293d-motor-driver-ic-arduino-tutorial/`
2. Learn to use Arduino Serial Monitor and Serial Plotter for data visualization
   `https://docs.arduino.cc/software/ide-v2/tutorials/ide-v2-serial-plotter`
3. Review DIP switch operation and digital input reading in Arduino.
4. Understand quadrature encoder fundamentals: channels, PPR, direction, gear ratio, and speed calculation.

Module Theory Highlights:

- **Encoders and Direction Sensing:**
  Quadrature encoders output pulses on channels A (C_1) and B (C_2) with a 90° phase shift. Using interrupts on channel A, read channel B in the ISR to decode both pulse count and rotation direction.
- **Pulse per Revolution (PPR):**
  PPR is the number of pulses generated per motor shaft revolution. Determine PPR experimentally by tracking pulse count for a full shaft rotation. Accurate knowledge of PPR is critical for converting measured pulse rates to RPM and for all feedback calculations.

- **Speed and Gear Ratio:**
  Use the measured PPR value to calculate the true speed of the motor shaft. Speed (in rev/sec) is given by $\frac{\text{PPS}}{\text{PPR}}$, where PPS (pulses per second) is determined by counting encoder pulses in fixed time steps.

- **Filtering:**
  Apply a low-pass filter to encoder speed measurements to reduce noise and improve control performance.

- **PID Control Logic:**
  Implement the PID law by calculating the error $e = v_{\text{ref}} - v_{\text{measured}}$ and summing the proportional, integral (with anti-windup), and derivative terms to generate the actuator command. Tune gains to achieve desired transient and steady-state metrics.

Report Guidelines:

1. Experimental objectives
2. Neatly drawn circuit and block diagrams showing all connections and signal flow
3. Flowchart or pseudo code for the implemented control algorithm
4. Key observations and screenshots/plots clearly labeled, including:
   - Serial monitor/plotter outputs for various controller settings:
     - Impact of K_P ($K_I = K_D = 0$)
     - Impact of K_I ($K_P = K_D = 0$)
     - Impact of K_D ($K_P = K_I = 0$)
     - Final tuned system performance with all gains (`show settling bands if possible`)
   - Raw and filtered encoder signals (with plots)
   - Evidence and explanation of PPR measurement and direction sensing
   - Demonstration of anti-windup effect (if implemented)
   - Gear ratio determination method and results
5. Clear answers to the following technical questions

Answer the following:

1. Draw and explain the block diagram of your closed-loop system. Where is the setpoint, and how is feedback incorporated?
2. Why is closed-loop (feedback) control required for this application? Why wouldn't open-loop PWM control suffice?

3. How did you determine the PPR and gear ratio? Discuss their importance in your project.
4. How was rotation direction detected and validated using the encoder channels?
5. What maximum RPM (`r`) did you experimentally determine for your motor?
6. Based on discussions on DC motor in EE250, what is the order of the transfer function you expect the DC motor to have. Sketch the approximate root locus of the DC motor.
7. Describe the PID tuning approach used - Closed loop Zieglar-Nichols or trial-and-error. Justify
8. Explain the effect of increasing each individual gain (K_P, K_I, K_D), using both time-domain results and control theory concepts.
9. What filtering scheme was used on the encoder data, and how did it affect control performance?
10. What is the final PID values that you settle with? Report the response of the DC motor for each of the references.

Annexure - A

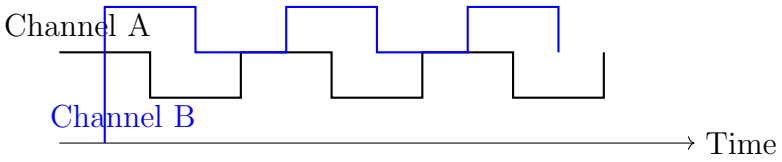
Understanding Encoder Channels A and B

Rotary encoders use two output channels, **A** and **B**, to generate digital pulses as the shaft rotates. These signals are offset by 90° (*quadrature encoding*), allowing both rotation count and direction detection.

Quadrature Encoding Principle

In quadrature encoding, channels A and B produce pulse trains with transitions occurring at different times. By observing which channel transitions first, the direction of shaft rotation is determined.

Timing Diagram



Interpretation:

- Channel A and B pulses are phase-shifted.
- The leading signal identifies rotation direction; e.g., if A leads B, the shaft rotates one way; if B leads A, the other way.

Direction Sensing

Detecting a rising edge on channel A and reading channel B at that instant:

- **B HIGH:** Increase count (e.g., counter-clockwise).
- **B LOW:** Decrease count (e.g., clockwise).

This enables efficient position and direction tracking, typically by attaching interrupts to channel A and sampling channel B inside the ISR.

Refer: `https://www.youtube.com/watch?v=1PJOzrXAlcg&t=1032s`

Counting Pulses Per Revolution (PPR) & Speed Measurement

What is PPR and Why is it Important?

Pulse Per Revolution (PPR) indicates how many electrical pulses an encoder produces per full shaft rotation. Higher PPR increases measurement resolution and enables accurate calculation of speed or position, which are critical for closed-loop motor control applications.

PPR enables:

- Fine position and speed measurements for precise feedback.
- Conversion of pulse counts into real-world quantities (rotation, speed, angle).
- Proper function of control algorithms such as PID.

Procedure for Determining PPR and Calculating Speed

1. **Power the encoder:** Connect the VCC and GND pins of the N20 DC motor's encoder to a 5V supply; this ensures the encoder receives power. The M1 and M2 pins of the N20 DC motor should be connected to the L293D motor driver, which provides power to the motor itself—typically 3V for the motors supplied in this experiment.

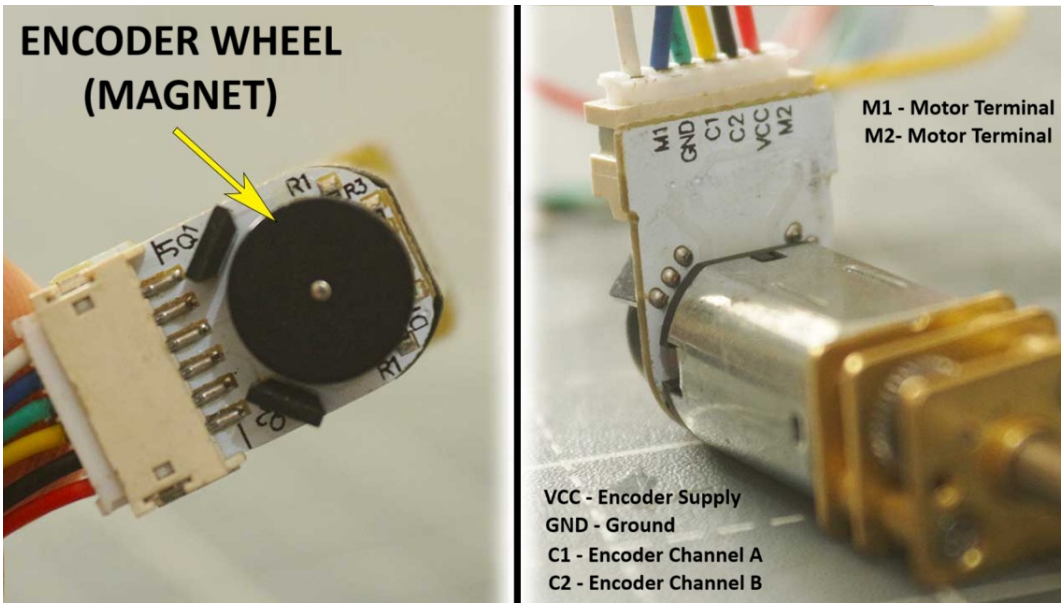


Figure 1: N20 DC motor magnet

Note: While determining the PPR of the encoder, you do not need to supply 3V to the M1 and M2 motor terminals. Instead, manually rotate the encoder wheel or magnet to generate encoder pulses as mentioned in Step-4.

2. **Pulse Counting:** Use interrupts to detect the rising edge on channel A (C_1), with direction determined by the state of channel B (C_2).

```
attachInterrupt(digitalPinToInterrupt(encoderChA), encoderA_SR, RISING);
void encoderA_SR() {
 if (digitalRead(encoderChB)==HIGH) {
  en_pulse_count++; // One direction
 } else {
  en_pulse_count--; // Opposite direction
 }
}
```

3. **Monitor Encoder Count:** Print `en_pulse_count` to the serial monitor for real-time tracking.

4. **Manual Rotation:** Spin the magnet/encoder wheel (see Figure 1) manually until the output shaft completes a full revolution. You will see the encoder count increase in the serial monitor.

5. **Record PPR:** The final count on the serial monitor after one shaft turn is your encoder's PPR.

6. **Find Gear Ratio:** Dividing shaft revolutions by encoder wheel revolutions yields the motor's gear ratio.

7. **Measure PPS:** Count encoder pulses over a known time step; divide by elapsed time to compute PPS.

8. **Speed Calculation:** Compute speed using:

$$\text{Speed (rev/sec)} = \frac{\text{PPS}}{\text{PPR}}$$

where PPS is pulses measured in one second.

9. **Note:** Keep motor IN1 and IN2 pins LOW as direction is not essential for finding the PPR.

PID Control for Speed Regulation

Implementation Steps

1. **Setup:** - Configure all relevant motor and encoder pins. - Initialize PID parameters: K_p, K_i, K_d.

2. **Set-Point:** - Define and update the desired speed (v_{ref}).

3. **Encoder Reading:** - Use ISR for pulse count, calculate measured speed; optionally filter velocity.

4. **PID Control Logic:** - Compute error: $e = v_{\text{ref}} - v_{\text{measured}}$. - Calculate Proportional, Integral, and Derivative terms and combine them:

$$u = K_p e + K_i \cdot \text{integral} + K_d \cdot \text{derivative}$$

   - Apply anti-windup by capping integral accumulation if output saturates.

5. **Motor Command:** - Set direction and PWM values as per control output, within hardware bounds.

6. **Monitoring:** - Output key variables for tuning and diagnostics.

7. **Safety:** - Include any necessary failsafes or delays in control loop.

PID Pseudo Code Example

```
START
INITIALIZE: set pins, interrupts, PID parameters, set-point
LOOP:
Read encoder count and velocity (safe access with interrupts)
Filter velocity (optional)
Update set-point if required
error = set-point - measured velocity
error_proportional = Kp * error
error_integral += error * elapsed_time (with anti-windup)
error_derivative = Kd * (error - previous_error) / elapsed_time
previous_error = error
control_output = error_proportional + Ki * error_integral + error_derivative
Set motor direction and clamp output
Apply anti-windup if needed
Issue motor command (direction, PWM)
Log variables for debugging
Wait small delay
END LOOP
```

OBSERVATION SHEET - MODULE 3A

`This sheet must be attached to the report (print on both sides).`

1. Show encoder signals (filtered and unfiltered) to the TA with screenshots comparing both (attach with report).

TA Signature with Date:

2. Demonstrate how the value of PPR was determined experimentally, including screenshots or serial monitor outputs for one full motor shaft rotation (attach with report).

TA Signature with Date:

3. Find the gear ratio (Show computation in report also).

TA Signature with Date:

4. Show measurement of pulses per second (PPS) for a given time step, and subsequent calculation of RPM using the determined PPR value (Show computation in report also).

TA Signature with Date:

5. What is the maximum rpm of the DC motor at rated voltage? (Show computation in

report also)

TA Signature with Date:

6. Demonstrate direction detection: show that pulse count increases or decreases with changes in shaft rotation direction. Provide screenshots or recorded observations for both directions.

TA Signature with Date:

7. Demonstrate proper interfacing of the motor and DIP switch with Arduino UNO. Show that current and desired RPM values are printed on the serial monitor.

TA Signature with Date:

8. Show output plots for different values of K_P, keeping $K_I = K_D = 0$. Confirm screenshots were taken (Attach with report).

TA Signature with Date:

9. Show output plots for increasing K_I with $K_P = K_D = 0$ (Attach with report).

TA Signature with Date:

10. Show output plots for increasing K_D with $K_P = K_I = 0$ (Attach with report).

TA Signature with Date:

11. Show final tuned system and verify that desired RPM is met (within 95%) for all DIP switch states. Provide settling bands and overshoot for each case (Attach with report).

TA Signature with Date:

`You can start Module-3B only after you complete Module-3A.`

B: Transfer Function Estimation of DC Motor

This module guides you through interfacing Arduino hardware with MATLAB and Simulink to acquire encoder data and estimate the DC motor's transfer function using the System Identification Toolbox.

Objectives

1. Configure MATLAB and Simulink to interface with Arduino for encoder pulse acquisition.
2. Calculate motor speed from encoder pulses using MATLAB.
3. Estimate the motor transfer function from collected input-output data in MATLAB.
4. Validate estimated transfer functions by comparison with experimental data.

Pre-Lab Preparation

- Install MATLAB, Simulink, and the Arduino hardware support packages.
- Understand shaft encoders, pulses per revolution (PPR), and basic system identification concepts.
- Connect Arduino and verify COM port connection.

Methodology

The steps to be performed for the module are enumerated below. Details of each steps can be found in Annexure - B.

Step 1: Interfacing Arduino and Motor

1. Connect motor, encoder, and Arduino hardware.
2. Install required MATLAB/Simulink support packages.
3. Open Simulink, create model, add Encoder and Scope blocks.
4. Configure fixed-step solver and hardware COM port.

Step 2: Find Speed

1. Acquire encoder pulses and determine PPR by rotating the shaft one full turn.
2. Measure pulses per second (PPS) during operation.
3. Calculate speed (RPS) using:
$$\text{Speed (RPS)} = \frac{\text{PPS}}{\text{PPR}}$$
4. Log speed data into MATLAB workspace.

Step 3: Find and Validate Transfer Function

1. Log input voltage and output speed data during open-loop operation.
2. Import data into the System Identification Toolbox.
3. Estimate three transfer function models:
   - (i) Two poles
   - (ii) One zero and two poles
   - (iii) Three poles and one zero
4. Apply a 2 V step input to both hardware and each TF model.
5. Plot and compare outputs; select the best fitting transfer function based on time-domain response and fit percentage.
   If y_{model} is the simulated output and y_{measured} is the measured output, then
$$\text{Fit (\%)} = \left(1 - \frac{\|y_{\text{measured}} - y_{\text{model}}\|}{\|y_{\text{measured}} - \bar{y}_{\text{measured}}\|}\right) \times 100$$

Questions for Report

Apart from the questions in the observation sheet, answer the following

- How does the encoder pulse count relate to shaft rotation and gear ratio?
- How was the motor speed calculated and validated?
- Present all three estimated transfer functions. Which one best represents the motor dynamics? Justify.

Annexure - B

Steps for Interfacing Arduino with MATLAB

1. **Install the required hardware support packages for MATLAB and Simulink**

   (a) Go to add-ons in the MATLAB workspace home section. This should open the add-on explorer window.

   (b) Search and install the following packages:

      i. *Simulink support package for Arduino hardware* by MathWorks.

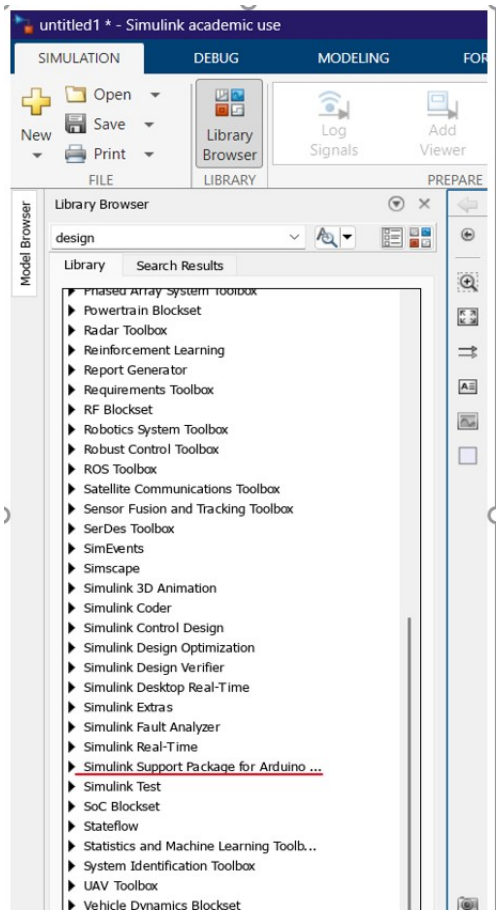
      ii. *MATLAB support package for Arduino Hardware* by MathWorks.

      iii. *Legacy MATLAB and Simulink Support for Arduino* by Giampiero Campa.

   (c) After installation, restart your MATLAB.

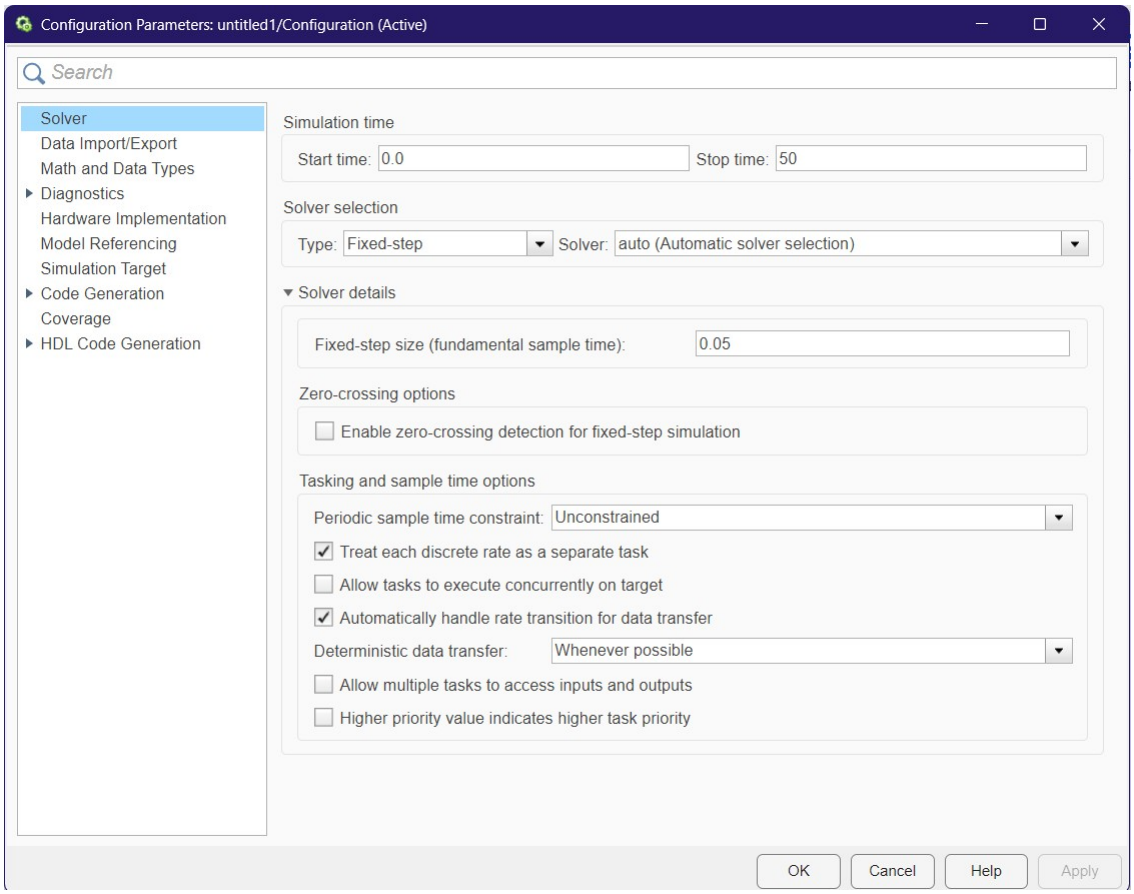
   (d) Open a blank Simulink page.

   (e) Go to the Library explorer section. You should find *Simulink support package for Arduino*. You can use the blocks available in this section for interfacing with your Arduino device.



2. **Setup Simulink to communicate with hardware**

(a) Go to *Model configuration parameters* or press **Ctrl+E** on the blank Simulink page. This shall open up the configuration parameter dialog box.



(b) Set solver type to *fixed step* and set fixed step sampling time to 0.05.

(c) Go to *Hardware Implementation* section in the dialog box.

i. Select the corresponding hardware device (e.g. Arduino Uno, Arduino Mega, etc.) in the *Hardware board*.

ii. Go to *Target hardware resources* in the Hardware board settings section which will appear after selection of the board.

iii. Go to *Host-Board connection*.

iv. Set the *host COM port* and *Baud Rate* manually.

v. COM port in which the Arduino device is connected can be seen from *Device Manager* → *Ports and COMs* and the baud rate for Arduino Uno is 9600.

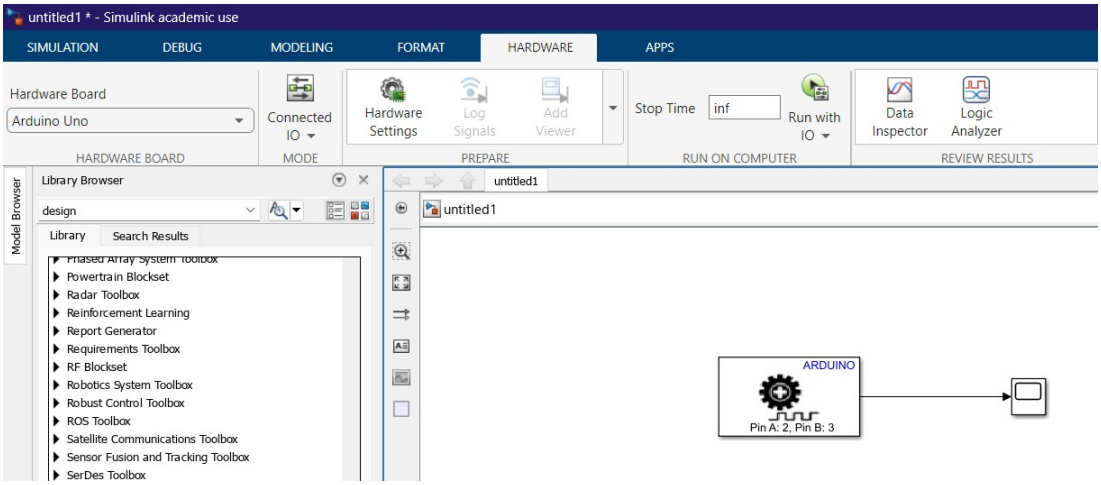
(d) After setting up the Hardware implementation part, you can see the Hardware section in the Ribbon on the Simulink page.

(e) Set the Mode to *connected IO*. Now, the Simulink page is ready for interfacing with the Arduino Board.

Steps for Reading Encoder Data and Counting PPR

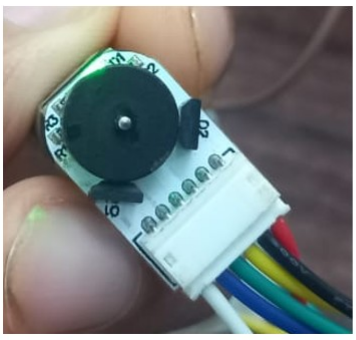
1. Insert the *Encoder* block from the library explorer (*Simulink support package for Arduino* $\rightarrow$ *Sensors* $\rightarrow$ *Encoder*) and a scope.

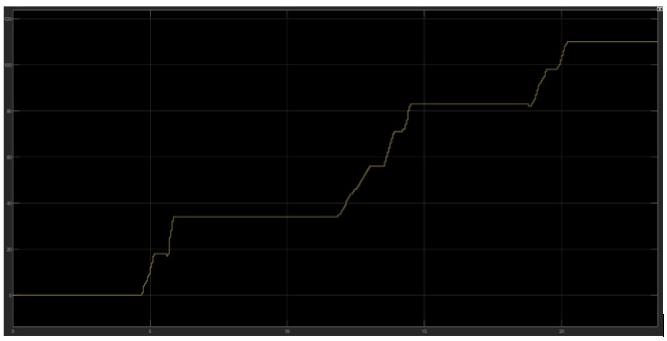
2. Connect the scope to the encoder block and set the stop time to *inf* to ensure infinite time running.



3. Run the interface by clicking *Run with IO*.

4. Spin the encoder wheel on the DC motor encoder and open the scope. You shall see the encoder count increase in the scope.





5. Spin the encoder wheel repeatedly until the motor shaft spins for one complete rotation.

6. Stop running the interface and note the final reading of the encoder count from the scope. This gives the **Pulse per Revolution (PPR)** of the encoder.

7. The number of rotations done in the encoder wheel for one complete rotation of the shaft gives the **gear ratio** of the DC motor.

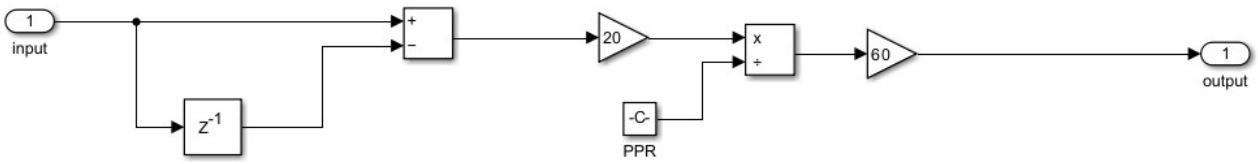
Steps for Speed Calculation and I/O Data Collection

1. Once we have the PPR, it can be used to calculate the speed of the DC motor using the following formula:

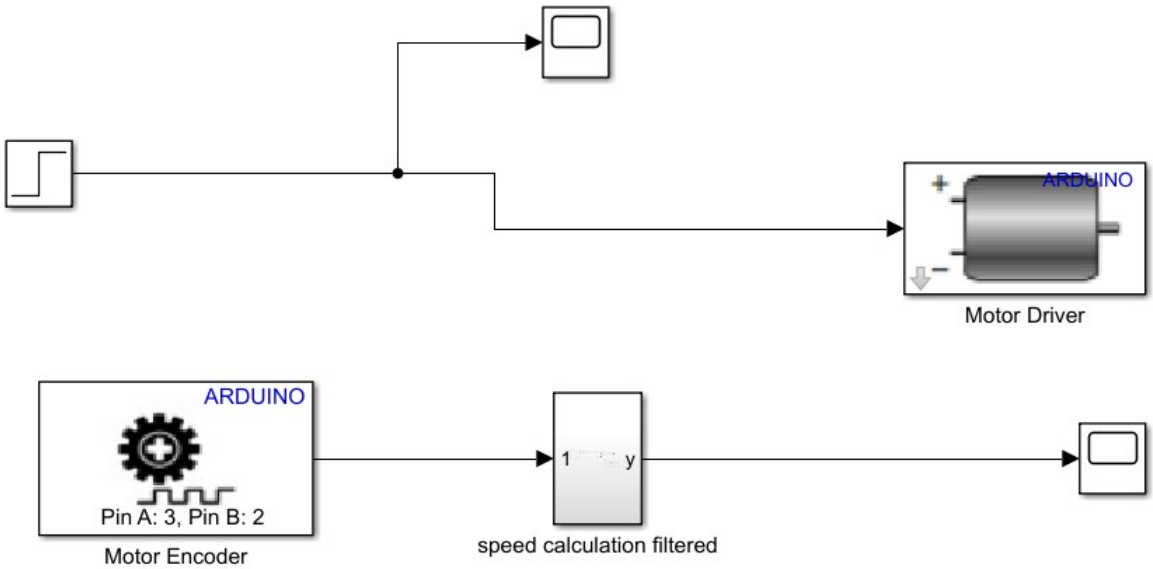
$$\text{Speed (RPS)} = \frac{\text{PPS}}{\text{PPR}}$$

where **PPS** is the pulse per second (number of encoder pulses measured in a second), and **PPR** is the pulses per revolution.

2. The speed calculation formula can be constructed in Simulink as follows:



3. The motor connection block diagram is shown below:



4. Before running the interface, make sure to log the scope data to the workspace.

   (a) Select the input scope.

   (b) Go to *configuration properties* of the scope.

   (c) Go to the *logging* section.

(d) Check the *Log data to workspace* box.

(e) Provide a suitable variable name for the input data (e.g., `input`).

(f) Change the *save format* to array.

(g) Perform the same operation for the output scope with a different variable name.

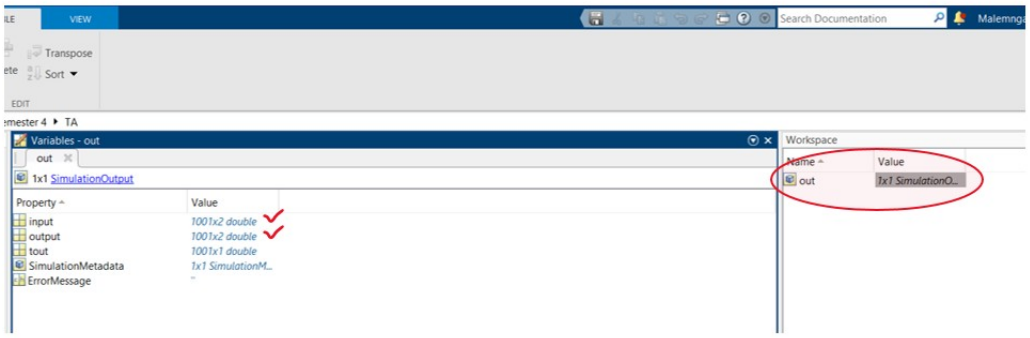
5. Set the stop time to a finite value (e.g., 50).

6. Connect the hardware to the appropriate Arduino pins and run the interface.

Steps for Transfer Function Estimation

1. After running the interface, go to the MATLAB workspace.

2. The dataset of the current run will be available under the name `out` in the workspace.



3. The `input` and `output` data arrays will be available inside the data under the designated variable names (e.g., `input` and `output`, to avoid confusion).

4. Extract the logged data to workspace variables using the following commands in the command window:

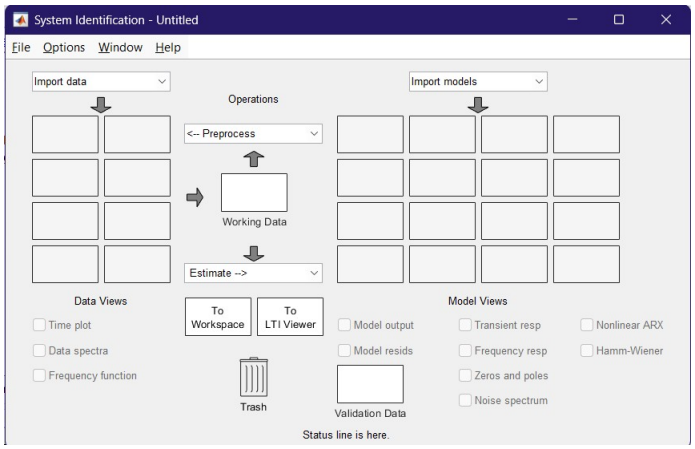
   (a) `(variable name) = out.input(:,2)` {this shall extract the input data to your variable of choice}

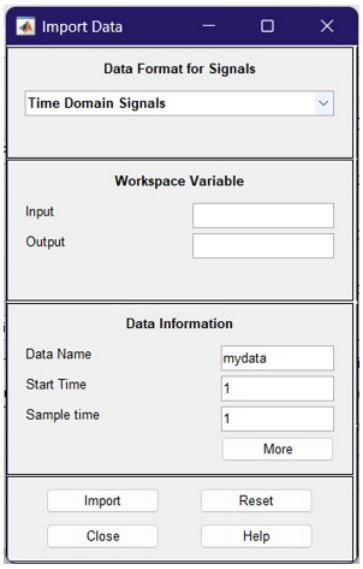
   (b) `(variable name) = out.output(:,2)` {this shall extract the output data to your variable of choice}

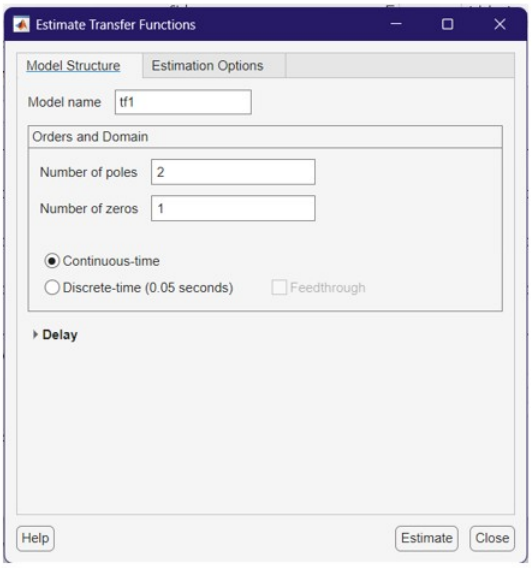
   (c) e.g. `xin = out.input(:,2)`, `yout = out.output(:,2)`

5. After extracting the data, open the *System Identification Toolbox* from the apps section of MATLAB.

6. Import extracted data for transfer function estimation:







(a) Click on *Import Data*

(b) Select *Time domain data*

(c) Enter the corresponding input and output variable names in the spaces provided, set the sampling time to `0.05` (or the value being used), and click *Import*

(d) Click on *Estimates* and select *Transfer function models*

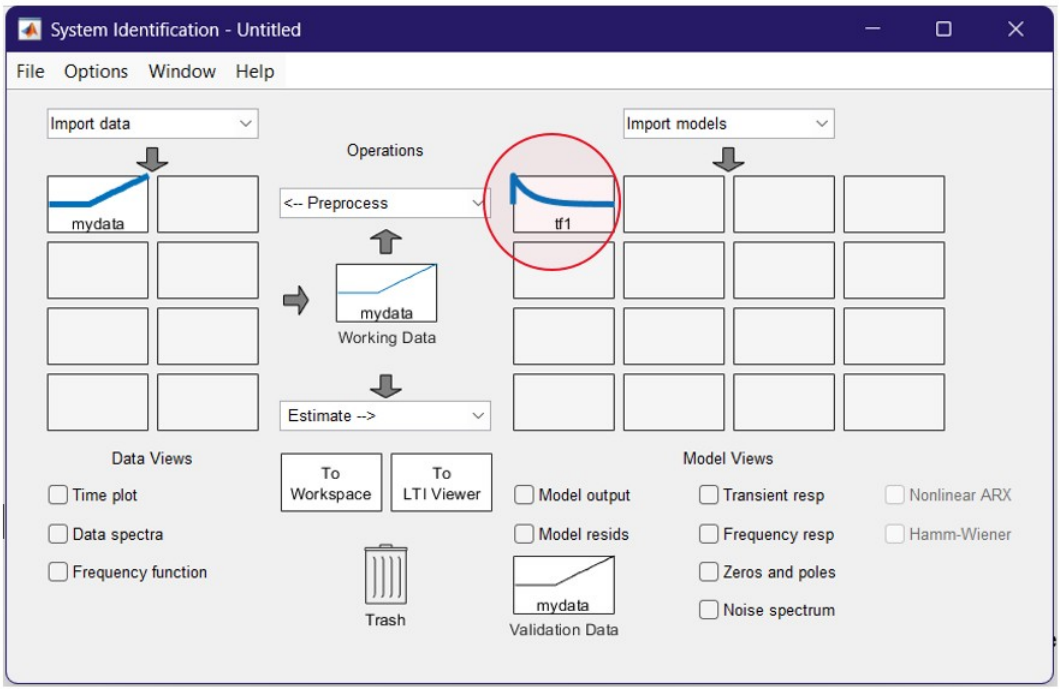
(e) In the *Estimate transfer function* dialog box, select the appropriate parameters for the transfer function (e.g., number of poles and zeros) and click *Estimate*

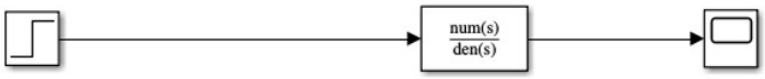
(f) After the estimation is complete, the estimated model will be available in the model section

(g) Right click on the tf model to reveal the transfer function

7. Use this transfer function and create an open loop block in Simulink as shown below:

(a) Provide the same input as given to the hardware and log the output data to the workspace, as done previously





(b) Plot the input, hardware output, and TF output in a single figure to compare the readings and validate the accuracy of the estimated TF.

OBSERVATION SHEET - MODULE 3B

`This sheet must be attached to the report (print on both sides).`

1. Demonstrate Arduino and MATLAB Simulink interfacing with Encoder and scope screenshots (Attach with report).

TA Signature with Date:

2. Show encoder pulse count increase for one rotation with screenshots or data logs (Attach with report).

TA Signature with Date:

3. Present Pulses Per Revolution (PPR) calculated from data (Show computation in report).

TA Signature with Date:

4. Show sample speed calculation with relevant plots/screenshots. (Show computation in report)

TA Signature with Date:

5. Demonstrate data logging of input/output variables to MATLAB workspace with screenshots (Attach with report).

TA Signature with Date:

6. Show MATLAB commands for extracting workspace data.

TA Signature with Date:

7. Show import and transfer function estimation steps in System Identification Toolbox with screenshots (Write in report).

TA Signature with Date:

8. Present the three estimated transfer functions and fit percentages in the table below, justify your selection (Write in the report).

Case	**Transfer Function**	**Fit %**	**Justification**
Two poles			
One zero, Two poles			
One zero, Three poles			

TA Signature with Date:

9. Validate transfer functions by comparing step responses (2 V input) between hardware and models — provide plots and conclusion (Attach with report and explain in details).

TA Signature with Date:

`You can start Module-3C only after you complete Module-3B.`

C: PID Controller Design for DC Motor Speed Control using MATLAB

This module focuses on designing and tuning a PID controller in Simulink using the transfer function identified in Module-3B, including implementation and validation on Arduino hardware.

Objectives

1. Construct a closed-loop control system using the estimated motor transfer function.
2. Design and tune a PID controller to meet performance specifications.
3. Implement and validate the PID controller on Arduino hardware.
4. Compare PID control performance with hit-and-trial tuning methods from Module-3A. (Consider each of the four reference speeds given in Module-3A for comparison.)

Pre-Lab Preparation

- Ensure completion of Module-3B transfer function estimation.
- Understand PID control fundamentals and tuning principles.
- Prepare Simulink environment for real-time hardware-in-the-loop control.

Methodology

The steps to be performed for the module are enumerated below. Details of each steps can be found in Annexure - C.

Step 1: Construct Closed-Loop Simulink Model

1. Using the finalized transfer function from Module-3B, build a closed-loop control system with a PID controller block.

Step 2: Define Performance Criteria

- Overshoot $\leq 20\%$
- Settling time < 4 seconds

Step 3: PID Tuning

1. Use MATLAB's PID Tuner app to automatically tune and adjust PID gains to meet criteria.

2. Simulate the closed-loop response; analyze overshoot, settling time, and rise time.
3. Log the simulated results for comparison.

Step 4: Hardware Implementation and Validation

1. Deploy the tuned PID controller to the Arduino hardware.
2. Run the system to acquire real-time speed control response.
3. Log output data for analysis.

Step 5: Compare with "Hit-and-Trial" PID from Module-3A

1. Implement the hit-and-trial PID gains used in prior modules both in simulation and hardware.
2. Compare closed-loop outputs (simulation and hardware) for both PID strategies.
3. Analyze and discuss differences in system performance and response criteria compliance.

Questions for Report

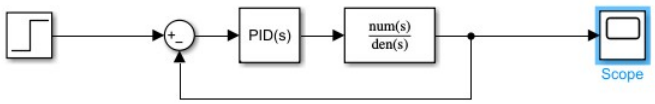
Apart from the questions in the observation sheet, answer the following

- How do PID tuning parameters affect overshoot and settling time?
- How well does the tuned PID controller meet the performance criteria?
- How does the tuned PID compare to the hit-and-trial PID in simulation and hardware?

Annexure - C

Steps for PID Tuning and Hardware Implementation

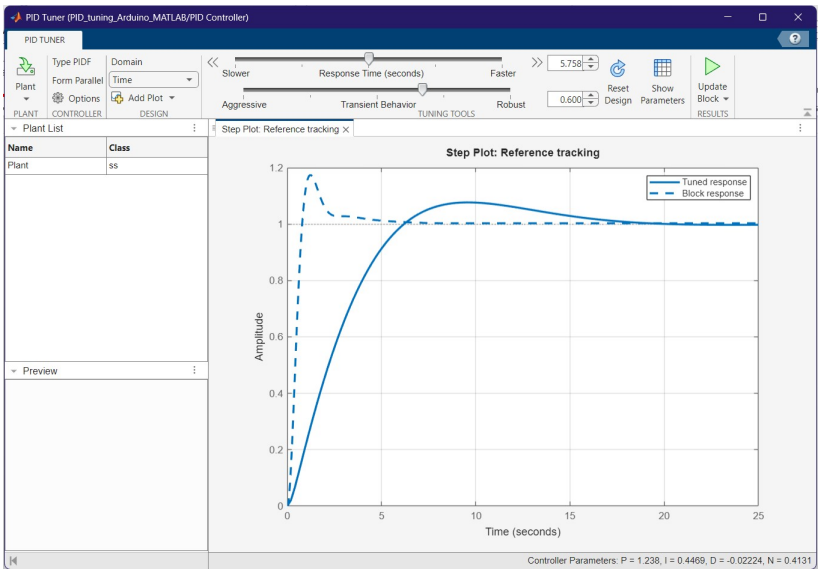
1. Design a closed-loop negative feedback system with a PID controller using the estimated transfer function in Simulink.



2. Run the simulation with random PID gain values and check the output.

3. Tune the PID to get the desired response parameters (overshoot, rise time, settling time, etc.):

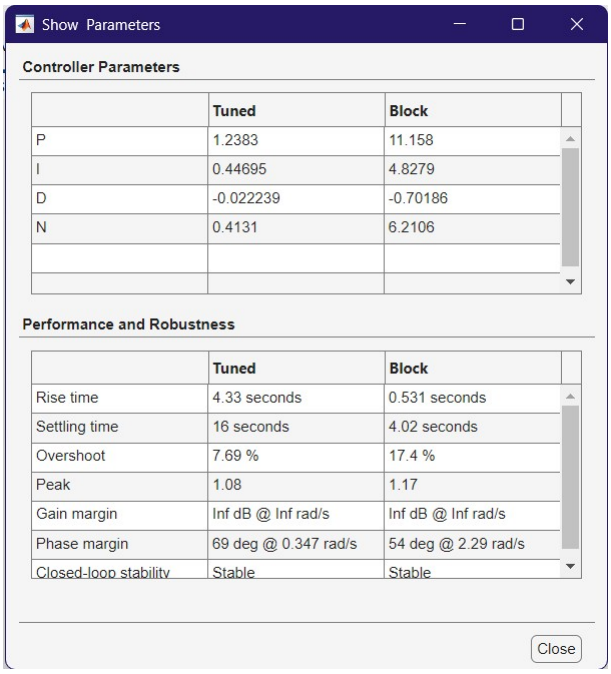
   (a) Double click the PID block and select the *Tune* option in the automated tuning section of the dialog box. This opens the PID Tuner application of MATLAB.

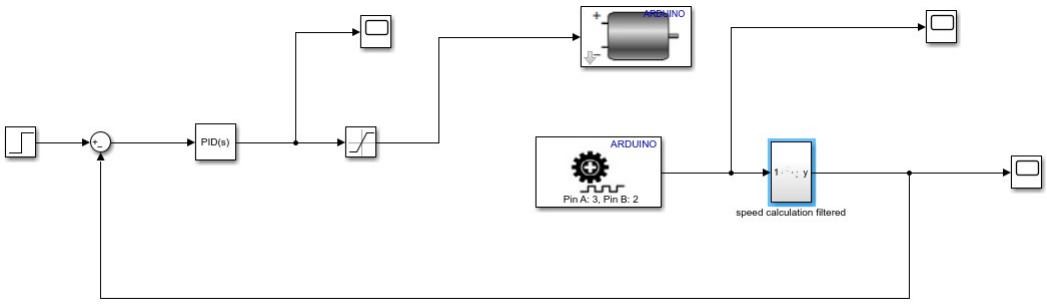


   (b) Click on the *System Parameters* button to see the tuned response parameters.

   (c) Adjust the *Response Time* and *Transient Behaviour* sliders to achieve the desired system response parameters and click on *Update Block*.

   (d) Run the simulation and check if the response is satisfactory, then log the output to the workspace.





4. Once the PID tuning is done with the estimated transfer function model, copy the tuned PID block and insert it in the closed-loop connection of the interface.

5. Run the interface and log the output data to the workspace.

6. Plot the input, hardware output, and simulation output of the closed-loop system together to verify the working of the tuned PID controller.

OBSERVATION SHEET - MODULE 3C

`This sheet must be attached to the report (print on both sides).`

1. Demonstrate construction of the closed-loop Simulink model with the PID controller. Include screenshots of the model and PID Tuner app. (Attach with the report)

TA Signature with Date:

2. Show plots from PID tuning steps and key system performance parameters. (Attach with the report)

TA Signature with Date:

3. Present simulation results before and after tuning including overshoot and settling time plots. (Attach with the report)

TA Signature with Date:

4. Demonstrate implementation of tuned PID controller on Arduino hardware. Show real-time response data [scope outputs or serial monitor]. (Attach with the report)

TA Signature with Date:

5. Show response results from hit-and-trial PID controller on both simulation and hardware. (Attach with the report)

TA Signature with Date:

6. Compare and discuss the performance between hit-and-trial and optimized PID controllers. (Explain in the report)

TA Signature with Date: