ZEWAIL CITY OF SCIENCE AND TECHNOLOGY

Faculty of Engineering

Motor Speed Controller

Advanced Control System Design with Integral Action

Project Team

Name:	Mohamed Saad Beltagy	202301436
	Mohamed Mashal	202201119
	Walid Sherif	202200702
	Abdelrahman Elfeky	202100560
Instructor:	Dr. Abdel Elshafaei	

Abstract

This report presents the design and experimental validation of a closed-loop DC motor speed controller using state-space control with integral action. The motor is driven through an Arduino-based PWM interface and an H-bridge driver, while speed feedback is obtained from an incremental encoder and conditioned using a low-pass filter. A first-order transfer function model relating PWM input to motor speed is identified using MATLAB System Identification Toolbox from several experimental trials. Based on the selected model, state-feedback and integral gains are computed via pole placement and implemented in real time using Simulink Connected I/O. Experimental results on the real motor demonstrate stable tracking of a square-wave reference corresponding to approximately 70% of the rated speed (about 130 rpm), with near-zero steady-state error and repeatable transient behavior. Practical limitations such as actuator saturation and integrator windup are also discussed, and anti-windup protection is proposed to improve robustness under PWM constraints.

Contents

1 Introduction

DC motor speed regulation is a fundamental problem in motion control because both steady-state speed and transient behavior vary with supply voltage, load torque, and friction. In practice, open-loop PWM actuation cannot guarantee accurate tracking of a desired speed profile, especially when the motor is subjected to load variations or unmodeled disturbances. For this reason, feedback control is required to maintain performance and suppress steady-state offset.

This project focuses on the design and real-time implementation of a closed-loop DC motor speed controller using a state-space approach with integral action. The controller is implemented on an Arduino-based platform and validated experimentally using encoder-based speed feedback. The objective is twofold: (i) improve transient behavior compared to open-loop operation (e.g., reduced settling time and better damping), and (ii) achieve robust tracking of reference speed commands with near-zero steady-state error.

1.1 Project Objectives and Performance Requirements

The target system is a low-power DC motor driven by PWM through an H-bridge. The required controller must track a square-wave reference speed while maintaining stable and well-damped transient behavior. The project requirements can be summarized as:

- Develop an experimental setup for PWM actuation and encoder-based speed measurement.
- Acquire input–output data and obtain a mathematical model using system identification.
- Assuming a first-order model between PWM input and motor speed, design a state-feedback controller with integral action.
- Implement the controller in real time with a fixed sampling period and evaluate tracking performance experimentally under practical constraints (e.g., PWM saturation).

1.2 Report Organization

Section 2 describes the hardware platform and the Simulink/Arduino real-time configuration. Section 3 presents the encoder-based speed estimation method and the filtering used to obtain a reliable feedback signal. Section 4 introduces the controller design methodology using state feedback with integral action and discusses its discrete-time realization for implementation. Section 5 then details the system identification procedure, including the trial-based datasets and the final first-order model used to compute the implemented gains. Section 6 documents the Simulink model structure and explains the main subsystems used for real-time control. Section 7 presents the experimental tracking results on the real motor, followed by a discussion of actuator saturation and integrator windup and the proposed anti-windup remedy. The report concludes with a summary of findings and recommendations for improvement.

2 Experimental Setup and Real-Time Implementation

The experimental platform consists of an Arduino Uno microcontroller, a DC motor equipped with an incremental encoder, and an L298N H-bridge driver. The H-bridge enables bidirectional motor actuation and provides a PWM-enabled input for speed control. Two digital lines control the direction of rotation, while a PWM line controls the effective motor voltage.

2.1 Hardware Connections

The motor driver is interfaced to the Arduino through three control lines: two digital pins for direction selection and one PWM pin for speed actuation. The encoder provides two quadrature signals (channels A and B) connected to the Arduino interrupt-capable pins to enable reliable counting at runtime. All grounds (Arduino, encoder, and driver) are tied together to ensure a common reference and reduce measurement errors.

2.2 Simulink Connected I/O Configuration

Real-time execution is achieved using Simulink Connected I/O. The model is configured for a fixed-step solver with a sampling period T_s and includes a real-time synchronization block to pace the execution to wall-clock time. A pulse generator is used during identification/verification experiments to apply a repeatable PWM excitation signal, which supports consistent data collection for modeling.

3 Speed Measurement and Signal Conditioning

Accurate feedback is critical for high-quality speed regulation. The encoder output is processed in discrete time to estimate speed from sampled position/count measurements. Let $N[k]$ denote the encoder count (or accumulated pulses) at sample k. The incremental count is

$$\Delta N[k] = N[k] - N[k-1]. \tag{1}$$

Dividing by the sampling interval produces a count-rate measurement. After accounting for the encoder resolution and gearbox ratio, the motor speed is converted to revolutions per second and then to rpm. In practice, this discrete differentiation amplifies quantization and measurement noise; therefore, the raw speed estimate is filtered before it is used for feedback.

3.1 Low-Pass Filtering

A first-order low-pass filter is used to suppress noise while preserving the dominant speed dynamics. The continuous-time filter is chosen as

$$G_f(s) = \frac{1}{Ts+1}, \tag{2}$$

where T is the filter time constant selected experimentally as a compromise between smoothness and response speed. The filter is discretized using a bilinear (Tustin) transformation to obtain a stable, causal discrete-time transfer function suitable for real-time implementation. Filter tuning is performed by comparing the raw and filtered speed signals and selecting T that provides adequate noise reduction without excessive phase lag.

4 Controller Design

The control objective is accurate speed tracking with improved transient performance and elimination of steady-state error. A state-feedback controller is selected due to its direct pole-placement capability and its compatibility with augmented integral action.

4.1 State Feedback (Regulator Concept)

For a state-space model $\dot{x} = Ax + Bu$, a state-feedback law $u = -Kx$ places the closed-loop poles at desired stable locations by shaping the eigenvalues of $(A - BK)$. Pole selection is a trade-off: faster poles typically reduce settling time but increase sensitivity to noise and unmodeled dynamics.

4.2 Integral Action for Zero Steady-State Error

To guarantee zero steady-state error for constant reference commands (and to reduce offset under constant disturbances), an integrator is added on the tracking error:

$$\dot{\xi}(t) = r(t) - y(t), \tag{3}$$

where $r(t)$ is the speed reference and $y(t)$ is the measured speed. Defining the augmented state

$$X_a(t) = \begin{bmatrix} x(t) \\ \xi(t) \end{bmatrix},$$

the augmented dynamics become

$$\dot{X}_a(t) = \underbrace{\begin{bmatrix} A & 0 \\ -C & 0 \end{bmatrix}}_{A_a} X_a(t) + \underbrace{\begin{bmatrix} B \\ 0 \end{bmatrix}}_{B_a} u(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} r(t). \tag{4}$$

The augmented feedback law is chosen as

$$u(t) = -K_a X_a(t) = -\begin{bmatrix} K & k_I \end{bmatrix} \begin{bmatrix} x(t) \\ \xi(t) \end{bmatrix}. \tag{5}$$

The gain vector $K_a = [K \quad k_I]$ is computed using pole placement on (A_a, B_a) to satisfy the settling-time requirement and to ensure a well-damped response. The selected closed-loop poles and the final gains are:

$$p_{cl} = \text{(your selected poles)}, \qquad K = \text{(value)}, \qquad k_I = \text{(value)}.$$

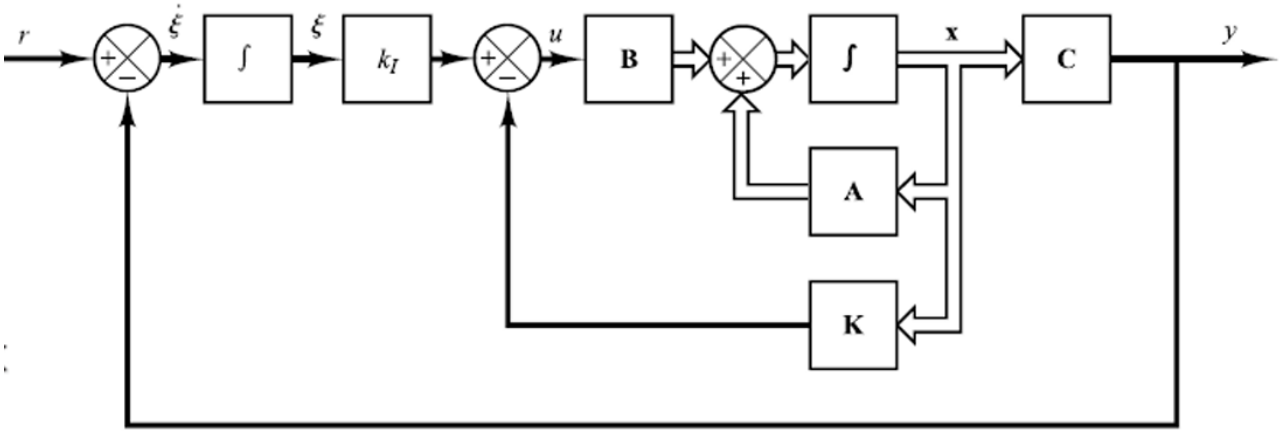

Figure 1: controller with integral action plant

4.3 Discrete-Time Realization

Because the controller runs on a digital platform with sampling period T_s, the implementation is executed in discrete time. The continuous-time integrator can be realized digitally using a backward (Euler) transformation, which maps

$$\frac{1}{s} \quad \Rightarrow \quad \frac{T_s z}{z-1}.$$

This yields a simple recursive update for the integral state:

$$\xi[k] = \xi[k-1] + T_s\Big(r[k] - y[k]\Big).$$

The final discrete-time control law is computed at each sampling instant using the measured (filtered) speed and the updated integrator state, then applied through the PWM output to the H-bridge.

5 System Identification and Modeling

5.1 Experimental setup and measured signals

The motor is driven through an L298N H-bridge. Two digital pins are used for direction control and one PWM pin is used to apply the speed command. Motor speed is measured using an incremental encoder connected to Arduino interrupt pins, allowing reliable pulse counting during real-time execution. The Simulink model is executed with a fixed sampling period

$$T_s = 0.1 \text{ s},$$

which is used consistently for data logging, speed estimation, and later controller implementation.

5.2 Speed estimation and measurement conditioning

The encoder provides a pulse count signal $N[k]$ sampled every T_s seconds. Speed is estimated from pulse differences:

$$\Delta N[k] = N[k] - N[k-1]. \tag{6}$$

This is converted to rotational speed using the encoder resolution. In rpm form:

$$\omega_{\text{rpm}}[k] = \frac{60}{(\text{PPR})\,T_s}\,\Delta N[k] \quad \text{(and include gearbox ratio if applicable).} \tag{7}$$

Because this estimate is based on a discrete difference, it exhibits quantization ripple and high-frequency noise (especially near steady-state). For identification and control, the measured speed is therefore filtered using a low-pass filter to reduce ripple while keeping phase lag acceptable.

A first-order low-pass filter is used:

$$G_f(s) = \frac{1}{T_f s + 1}, \tag{8}$$

then discretized for implementation. The selected T_f value used in experiments is reported with the results.

5.3 Identification goal and model structure

The identification objective is to obtain a simple input–output model relating PWM command $u(t)$ to motor speed $y(t)$ that is accurate enough for controller design and validation. In this work, and consistent with the project assumption, the motor is modeled as a first-order transfer function:

$$G_m(s) = \frac{K}{\tau s + 1}, \tag{9}$$

where K is the steady-state gain (rpm per PWM unit, depending on scaling) and τ is the dominant time constant.

5.4 Trial 1: Multi-level step experiments, merging, and validation

5.4.1 Collected experiments

In Trial 1, multiple step-response datasets were collected at different PWM levels to capture the dominant motor speed dynamics over a practical operating range. Each experiment starts from rest, then a constant PWM command is applied and the speed response is recorded until steady state is reached. The estimation (training) experiments were:

$$\text{PWM} \in \{120,\ 150,\ 180,\ 220,\ 255\}.$$

Two additional datasets were reserved for validation:

$$\text{PWM} = 200\ (\texttt{PWM200\_Validation}), \qquad \text{PWM} = 135\ (\texttt{Val\_PWM135}).$$

Figure 2 shows the recorded input–output data. The top subplot presents the measured speed responses, while the bottom subplot shows the corresponding PWM step inputs. As the PWM level increases, the steady-state speed increases accordingly. A small steady-state ripple is visible in the measured speed, which is attributed to encoder quantization and sampling effects.

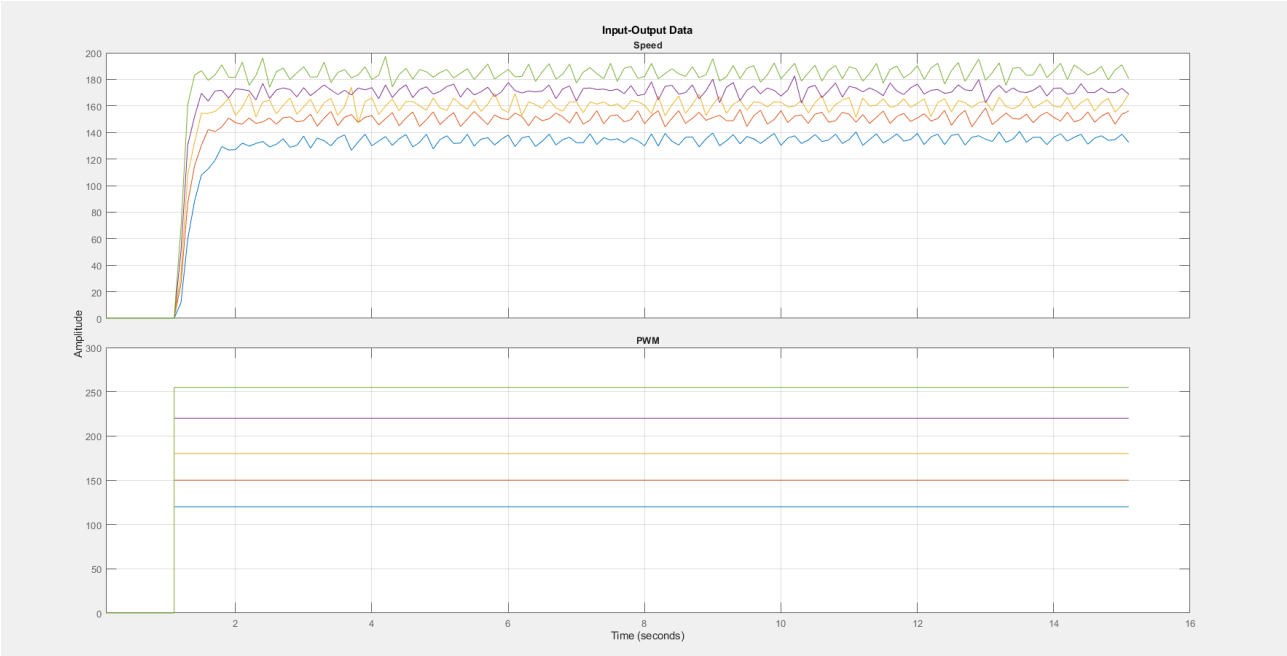

Figure 2: Trial 1 input–output datasets. Top: measured speed responses for multiple PWM step levels. Bottom: corresponding constant PWM inputs.

5.4.2 System Identification App workflow and merging

All datasets were imported into the MATLAB *System Identification* app as separate experiments (PWM120, PWM150, PWM180, PWM220, PWM255), along with the two validation experiments. After import, the signals were checked for consistent sampling time and step alignment. Any necessary preprocessing (e.g., removing initial idle samples before the step) was applied so the estimation focuses on the motor's dynamic response rather than non-informative segments.

A key step in Trial 1 was combining all estimation experiments into a single dataset using the app's *Merge* operation. The five estimation experiments were merged into one working dataset (`Merged_data`). The motivation is that merging increases the information content available to the estimator and encourages the resulting model to represent the dominant dynamics across the tested PWM range rather than fitting only one operating point.

5.4.3 Estimated models (first-order transfer functions)

Following the project assumption of a first-order relationship between PWM input and motor speed, first-order transfer functions were estimated for:

- each individual dataset: `tf_PWM120`, `tf_PWM150`, `tf_PWM180`, `tf_PWM220`, `tf_PWM255`, and
- the merged dataset: `tf_merged`.

5.4.4 Validation on unseen data (PWM200) and model selection

To select the nominal model, all estimated transfer functions were compared on the unseen validation dataset `PWM200_Validation` using the app's *Model Output* comparison view. Figure 3 shows the measured validation speed signal overlaid with simulated outputs from the candidate models, and it reports a fit percentage for each model. The merged model achieved the best agreement with the validation dataset by a clear margin:

- **`tf_merged`: 89.07%** (best)
- `tf_PWM220`: 76.44%
- `tf_PWM180`: 69.89%
- `tf_PWM255`: 54.74%
- `tf_PWM150`: 23.08%
- `tf_PWM120`: -25.23%

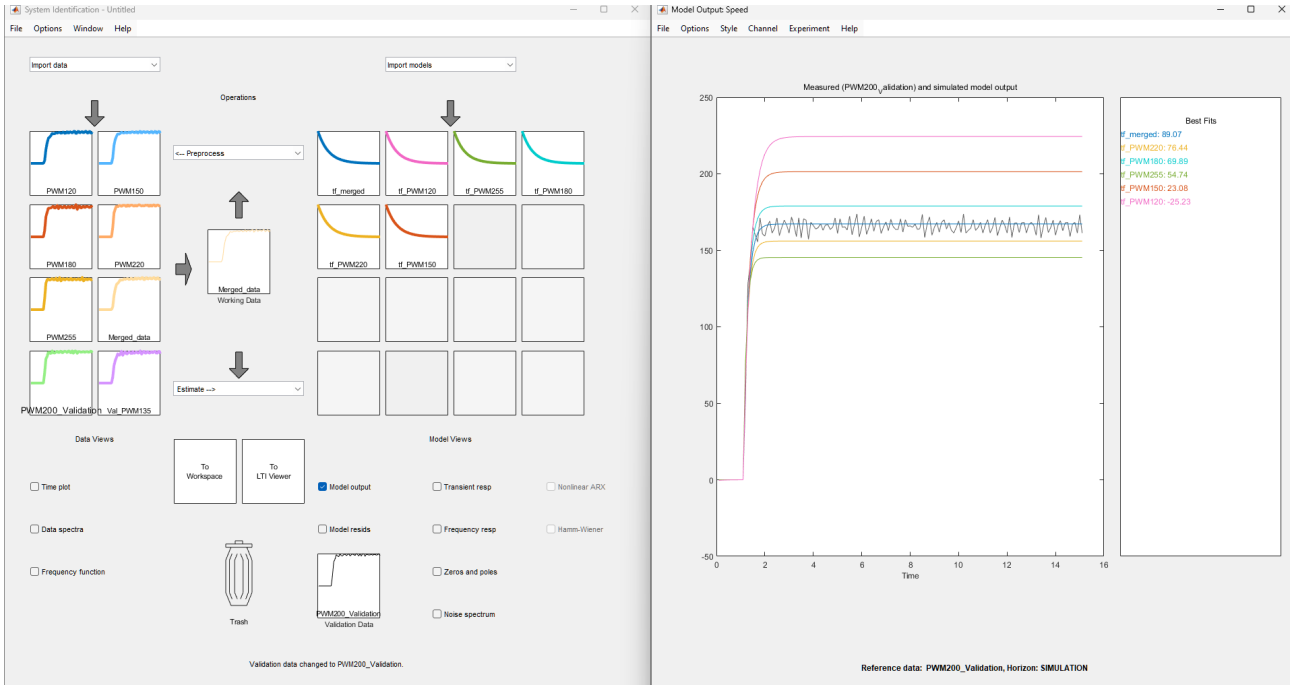

Figure 3: System Identification App results for Trial 1. Left: imported experiments and merged dataset creation. Right: model output comparison on `PWM200_Validation`, showing best fit achieved by `tf_merged` (89.07%).

5.4.5 Interpretation and notes

The superior performance of `tf_merged` indicates that combining multiple operating points into one estimation dataset improves generalization to unseen PWM values. In contrast, models estimated from a single low-PWM experiment (notably PWM120) tend to generalize poorly, which is consistent with practical nonlinearities such as static friction, PWM dead-zone, and limited excitation at low speeds.

5.4.6 Selected nominal model from merged data

Based on the validation comparison, the merged first-order model (`tf_merged`) was selected as the nominal plant for Trial 1.

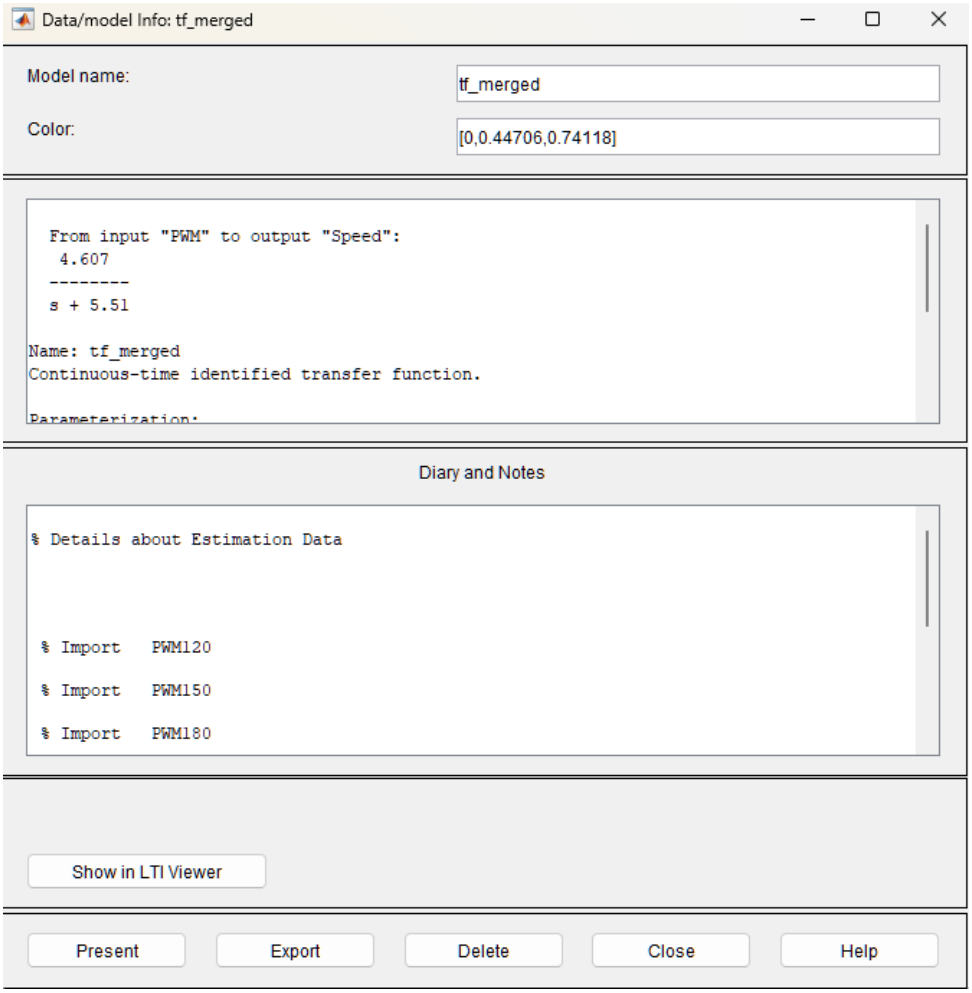

Figure 4: Identified continuous-time transfer function from input `PWM` to output `Speed` for the merged dataset (`tf_merged`).

Expressing (10) in the standard first-order form $G_m(s) = \frac{K}{\tau s+1}$ gives:

$$\tau = \frac{1}{5.51} = 0.1815 \text{ s}, \qquad K = \frac{4.607}{5.51} = 0.8361.$$

This model is used in the controller design section for Trial 1.

```
Transfer Function G(s):

G =

   4.607
  --------
  s + 5.51

Continuous-time transfer function.
Model Properties

================ OPEN-LOOP DYNAMICS ================
Open-loop pole              : -5.5100
Open-loop time constant     : 0.1815 s
Open-loop settling time (2%): 0.7260 s

================ CLOSED-LOOP SPECIFICATIONS ================
Desired settling time (2%)  : 0.5808 s
Dominant closed-loop pole   : -6.8875
Faster closed-loop pole     : -34.4375
Desired pole vector         : [-6.8875  -34.4375]

================ AUGMENTED SYSTEM (INTEGRAL ACTION) ================
Augmented state matrix A_aug:
   -5.5100         0
   -4.6070         0

Augmented input matrix B_aug:
     1
     0


================ CONTROLLER GAINS ================
State feedback gain Kx : 35.815000
Integral gain Ki       : 51.484324
```

Figure 5: controller design

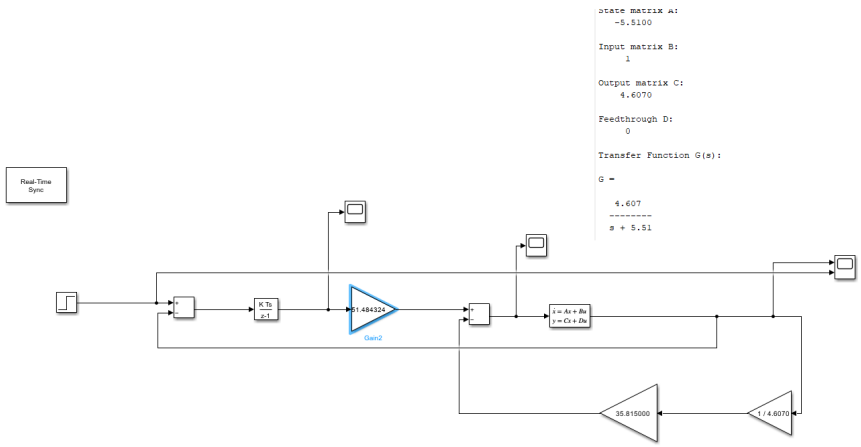

Figure 6: plant

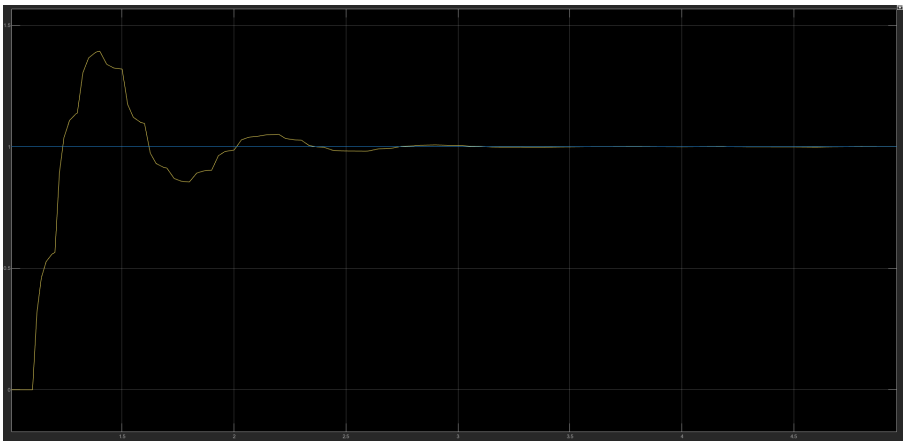

Figure 7: plant response with controller

The identified model is:

$$G_m(s) = \frac{4.607}{s + 5.51}. \tag{10}$$

5.5 Trial 2: Single-experiment identification using pulse excitation

5.5.1 Collected dataset (single experiment)

In Trial 2, system identification was performed using a *single* experiment dataset rather than multiple PWM step experiments. The recorded dataset (labeled `Pulse_ignoreFirst`) was obtained by exciting the motor using a pulse-like PWM input, which forces repeated acceleration and deceleration and therefore provides informative dynamics for estimating a low-order model. As indicated by the dataset name, the initial portion of the record was excluded to avoid start-up artifacts and non-representative initial behavior.

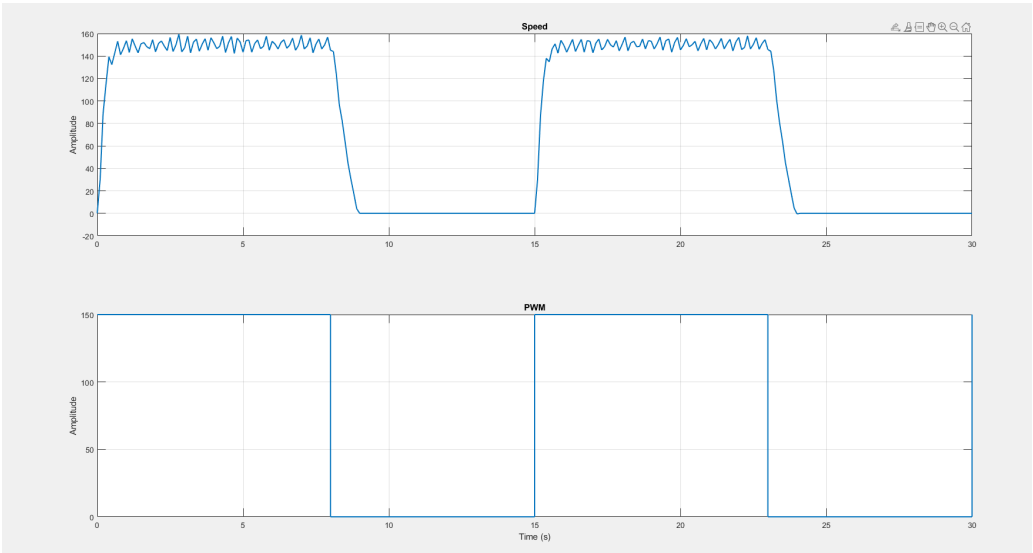

Figure 8: Trial 2 input–output data (to be added): PWM pulse excitation (input) and the corresponding measured motor speed response (output) used for identification (`Pulse_ignoreFirst`).

5.5.2 Model output comparison (fit)

The model quality was evaluated using the *Model Output* comparison view. Figure 9 overlays the measured speed signal from `Pulse_ignoreFirst` with the simulated output of the identified model and reports the fit metric. The estimated model achieved a fit of **88.71%**, indicating strong agreement with the measured response for this experiment.

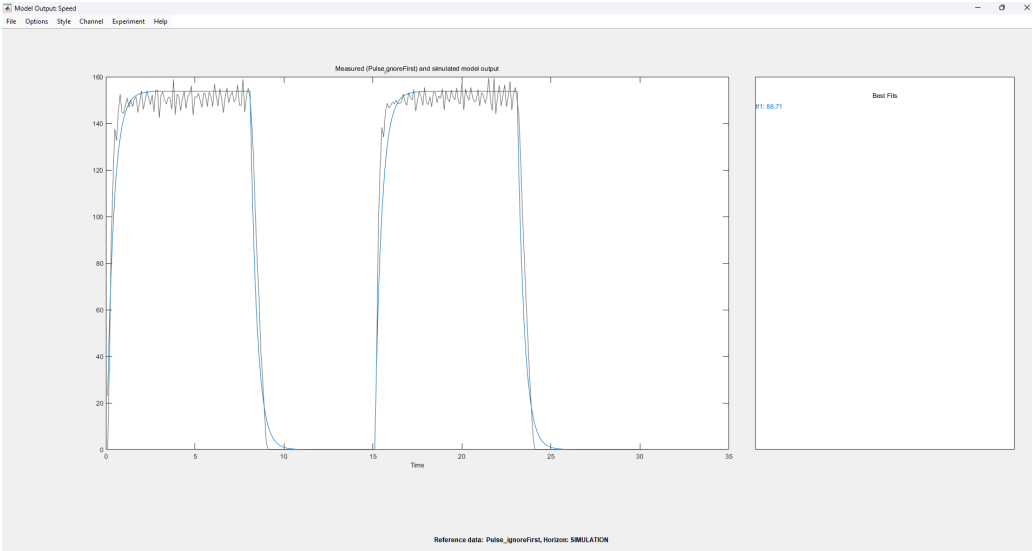

Figure 9: Trial 2 model output comparison for `Pulse_ignoreFirst`. The identified first-order model achieves a fit of 88.71%.

Outcome of Trial 2. The transfer function in (11) (Fig. 10) is taken as the nominal plant model for Trial 2 and will be used in the controller design section to compute the state-feedback and integral gains under the same methodology applied across trials.

5.5.3 Model estimation (first-order transfer function)

Using the System Identification App, a continuous-time first-order transfer function (one pole, zero zeros) was estimated from input `PWM` to output `Speed`. The resulting model was initially stored as `tf1` and then renamed to `tf_pulses` for clarity.

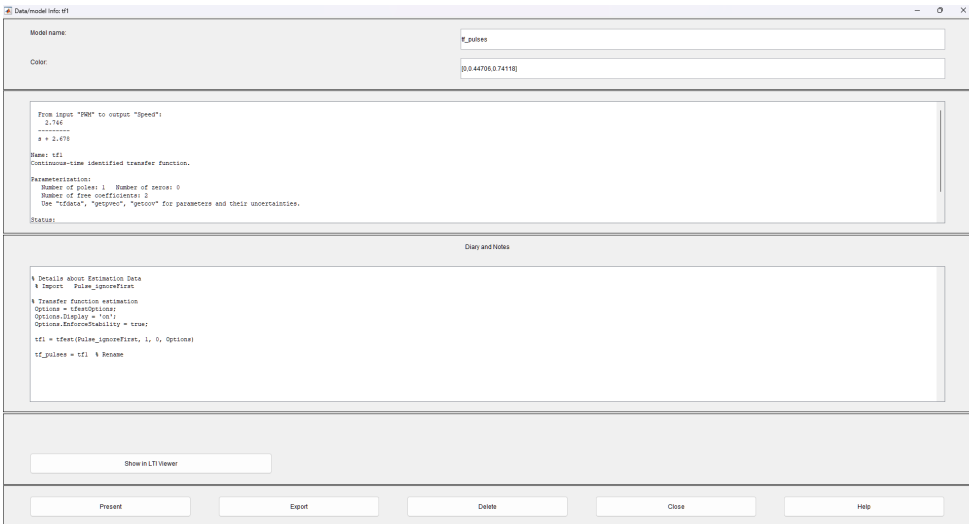

Figure 10: Trial 2 identified transfer function from input `PWM` to output `Speed`. The model was estimated as `tf1` and renamed to `tf_pulses`.

The identified model for Trial 2 is:

$$G_{m,2}(s) = \frac{2.746}{s + 2.678}. \tag{11}$$

```
     1

Output matrix C:
    2.7460

Feedthrough D:
     0

Transfer Function G(s):

G =

    2.746
  ---------
  s + 2.678

Continuous-time transfer function.
Model Properties

================ OPEN-LOOP DYNAMICS ================
Open-loop pole              : -2.6780
Open-loop time constant     : 0.3734 s
Open-loop settling time (2%): 1.4937 s

================ CLOSED-LOOP SPECIFICATIONS ================
Desired settling time (2%)  : 1.1949 s
Dominant closed-loop pole   : -3.3475
Faster closed-loop pole     : -16.7375
Desired pole vector         : [-3.3475  -16.7375]

================ AUGMENTED SYSTEM (INTEGRAL ACTION) ================
Augmented state matrix A_aug:
   -2.6780         0
   -2.7460         0

Augmented input matrix B_aug:
     1
     0


================ CONTROLLER GAINS ================
State feedback gain Kx : 17.407000
Integral gain Ki       : 20.403780
>>
```

Figure 11: controller design

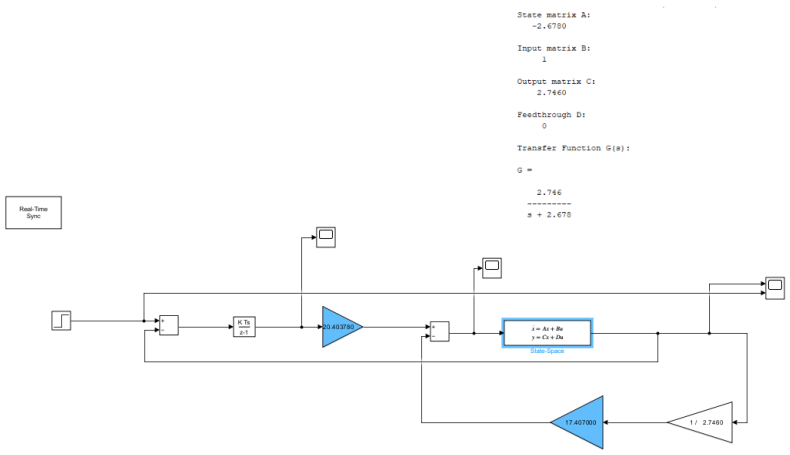

Figure 12: controller design

Figure 13: plant response with controller

5.6 Trial 3 (final): Pulse experiment with filtered speed signal

5.6.1 Collected dataset (input–output record)

Trial 3 follows the same pulse-excitation idea used previously, but with an additional refinement: the measured speed signal is processed/filtered to reduce ripple and improve the consistency of the identified dynamics. The input is a PWM pulse sequence (high/low switching), and the output is the corresponding measured motor speed response. Figure 14 shows the input–output record used for identification, where the motor accelerates to a repeatable steady-state speed during each “PWM-high” interval and decelerates during the “PWM-low” interval.

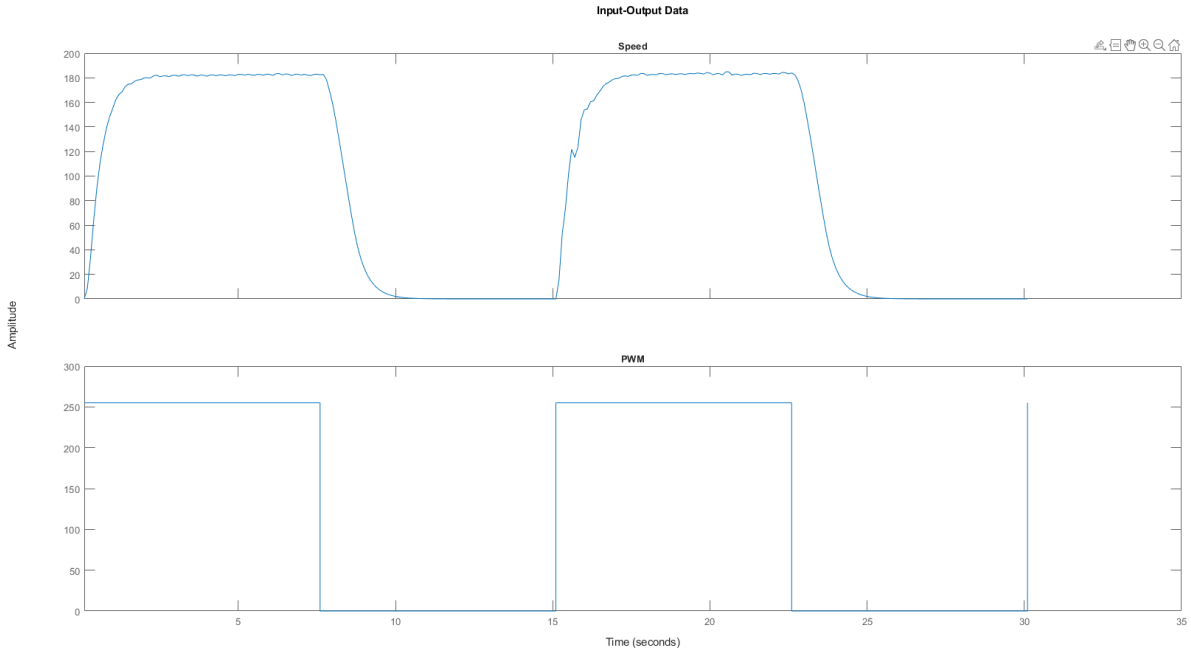

Figure 14: Trial 3 input–output dataset used for identification. Top: measured motor speed response under pulse excitation. Bottom: PWM pulse command applied to the motor driver.

5.6.2 Model estimation (first-order transfer function)

Using the System Identification App, a continuous-time first-order transfer function (one pole, zero zeros) was estimated from input `PWM` to output `Speed`. The resulting model was saved and renamed (as shown in the model information window) to reflect that it corresponds to the filtered pulse dataset.

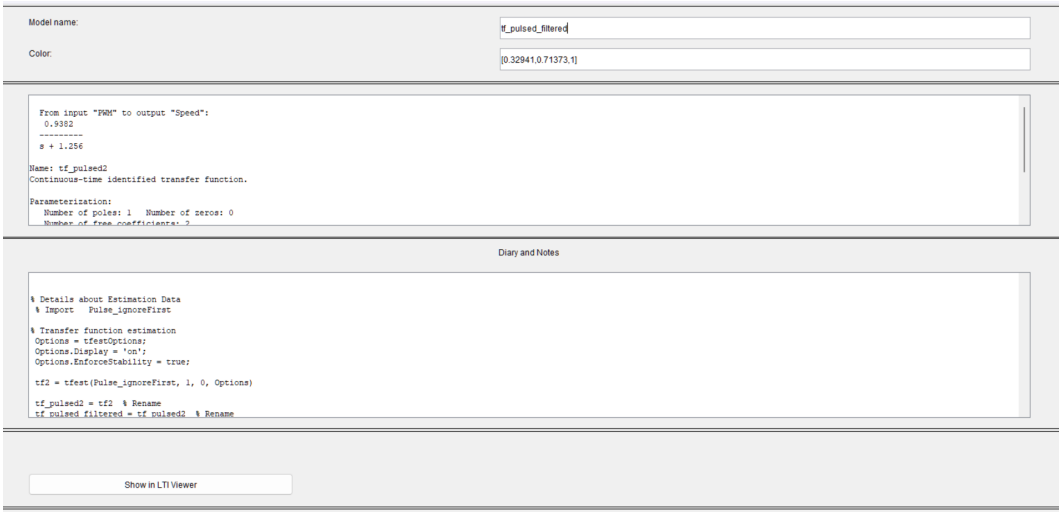

Figure 15: Trial 3 identified transfer function from input `PWM` to output `Speed` for the filtered pulse dataset (model information window).

5.6.3 Model-output comparison (fit)

The model quality was assessed using the *Model Output* comparison view, which overlays the measured speed response with the simulated output of the identified model. Figure 16 shows the comparison and the reported fit value. The estimated model achieved a fit of **86.57%**, indicating that the first-order structure captures the dominant rise and decay dynamics of the pulse response with good accuracy.

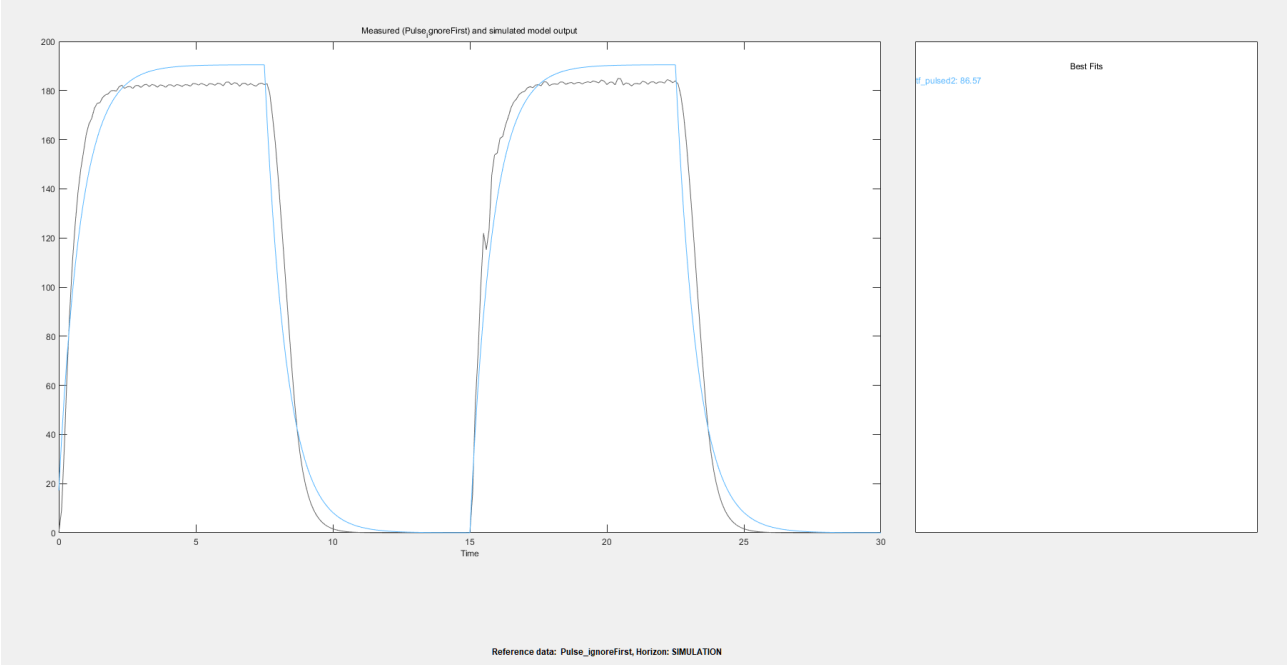

Figure 16: Trial 3 model output comparison on the pulse dataset. The identified first-order model achieves a fit of 86.57%.

Outcome of Trial 3. The transfer function in (12) (Fig. 15) is selected as the nominal plant model for Trial 3. This model is referenced in the controller design section when computing the state-feedback and integral gains and when discussing practical limitations (e.g., actuator saturation and windup) under real PWM constraints.

The identified model for Trial 3 is:

$$G_{m,3}(s) = \frac{0.9382}{s + 1.256}. \tag{12}$$

```
     1

Output matrix C:
    0.9382

Feedthrough D:
     0

Transfer Function G(s):

G =

   0.9382
  ---------
  s + 1.256

Continuous-time transfer function.
Model Properties

================ OPEN-LOOP DYNAMICS ================
Open-loop pole              : -1.2560
Open-loop time constant     : 0.7962 s
Open-loop settling time (2%): 3.1847 s

================ CLOSED-LOOP SPECIFICATIONS ================
Desired settling time (2%)  : 2.5478 s
Dominant closed-loop pole   : -1.5700
Faster closed-loop pole     : -7.8500
Desired pole vector         : [-1.5700  -7.8500]

================ AUGMENTED SYSTEM (INTEGRAL ACTION) ================
Augmented state matrix A_aug:
   -1.2560         0
   -0.9382         0

Augmented input matrix B_aug:
     1
     0


================ CONTROLLER GAINS ================
State feedback gain Kx : 8.164000
Integral gain Ki       : 13.136325
```

Figure 17: controller design(with faster pole)

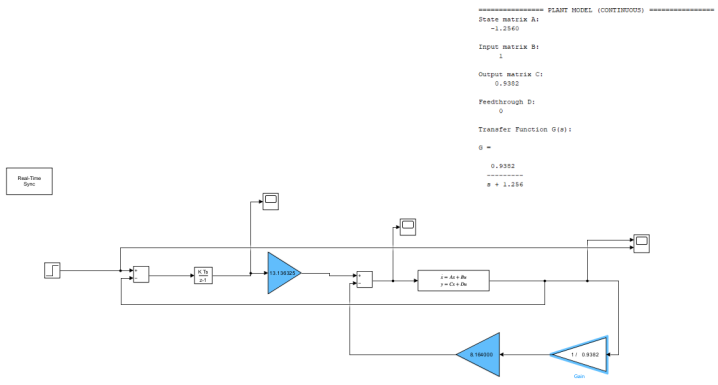

Figure 18: plant(with faster pole)

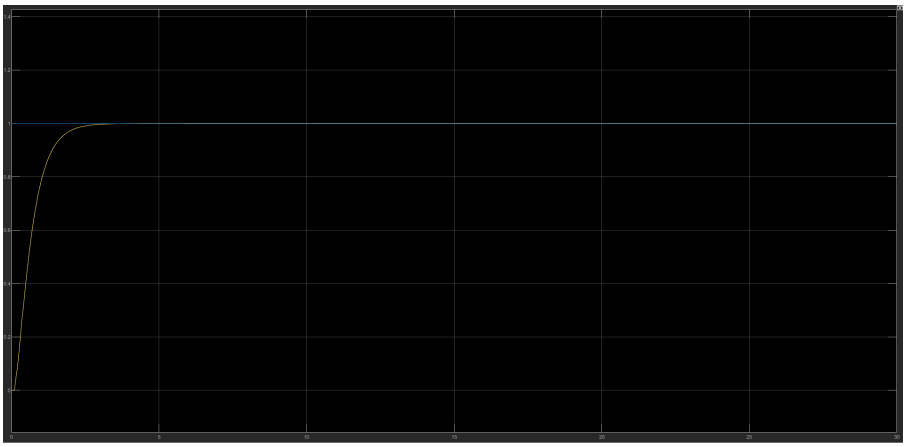

Figure 19: plant response with controller(with faster pole)

```
     RiseTime: 1.7492
TransientTime: 3.1147
 SettlingTime: 3.1147
  SettlingMin: 0.6756
  SettlingMax: 0.7465
    Overshoot: 0
   Undershoot: 0
         Peak: 0.7465
     PeakTime: 5.8298

1.2560   -1.6814
```

Figure 20: controller design(with repeated poles)

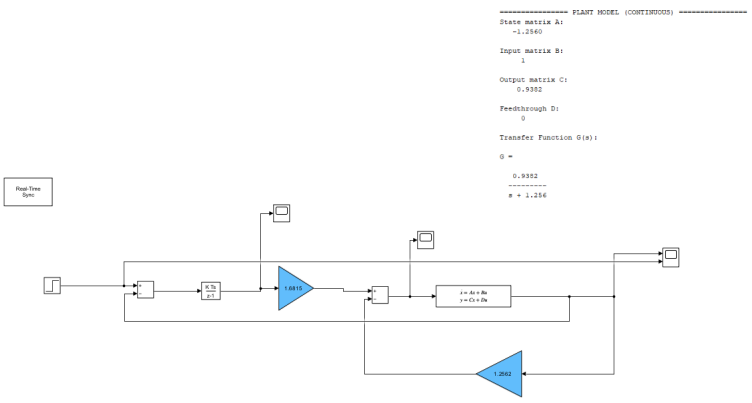

Figure 21: plant(with repeated pole)

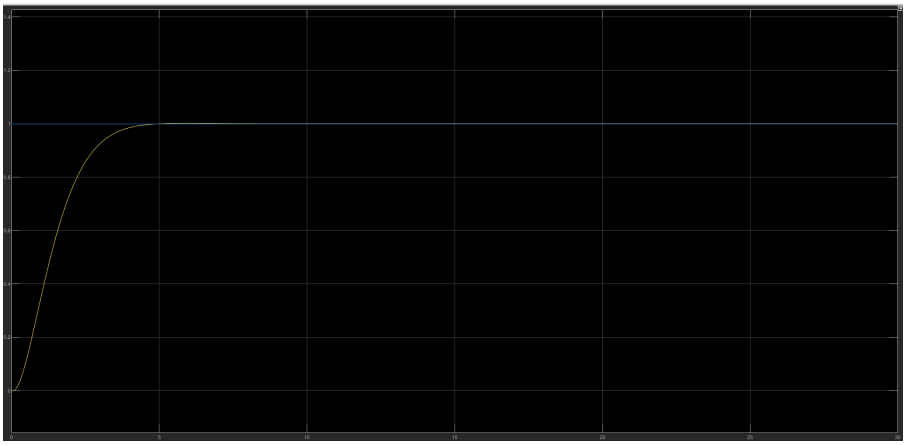

Figure 22: plant response with controller(with faster pole)

6 Simulink Model and Real-Time Implementation

This project is implemented and tested in real time using Simulink Connected I/O. The complete model is organized into two main subsystems: (i) a state-space controller with integral action that computes the actuator command, and (ii) a DC motor hardware interface subsystem that applies PWM through the Arduino and measures speed from the encoder. Figure 23 shows the overall closed-loop signal flow, while Figures 24 and 25 show the internal structure of the controller and the motor interface, respectively.

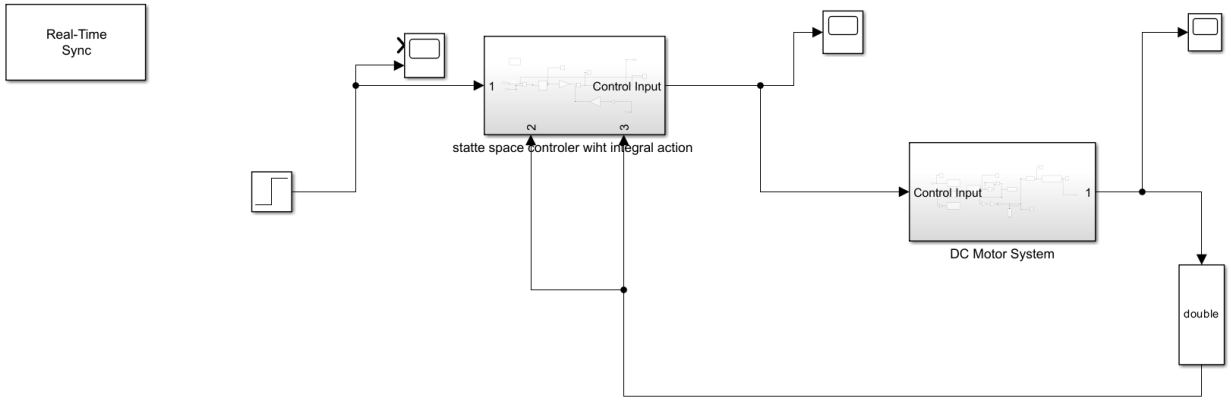

Figure 23: Top-level Simulink model used for real-time closed-loop motor speed control.

6.1 Top-level model description (Figure 23)

The top-level model implements a standard feedback loop:

- **Real-Time Sync:** Paces the Simulink execution to real time so that each sample is executed with the intended sampling period during Connected I/O operation.

- **Ref. Speed (Constant block):** Generates the reference command. The numeric value (e.g., 120) represents the desired speed in the same units produced by the speed-estimation chain.
- **Sum block** (+/−)**:** Computes the tracking error:

$$e[k] = r[k] - y[k],$$

  where $r[k]$ is the reference speed and $y[k]$ is the measured (filtered) motor speed returned from the DC motor subsystem.
- **Subsystem: *state space controller with integral action*:** Receives the error (and the measured speed feedback) and computes the control command (PWM-equivalent command) using integral action and state feedback (details in Figure 24).
- **Subsystem: *DC Motor System*:** Applies the actuator command to the Arduino/H-bridge and returns the measured speed computed from encoder pulses (details in Figure 25).
- **Data Type Conversion (`double`):** Converts the measured feedback signal to `double` precision. This avoids type-mismatch issues (common when Arduino blocks output integer types) and ensures correct arithmetic inside the controller.
- **Scopes/Displays:** Used throughout the model for live monitoring of key signals such as reference, measured speed, and the control input.

6.2 Controller subsystem (Figure 24)

The controller subsystem implements integral action (to eliminate steady-state error) combined with state feedback (to shape transient response). Figure 24 shows the exact block-level implementation.

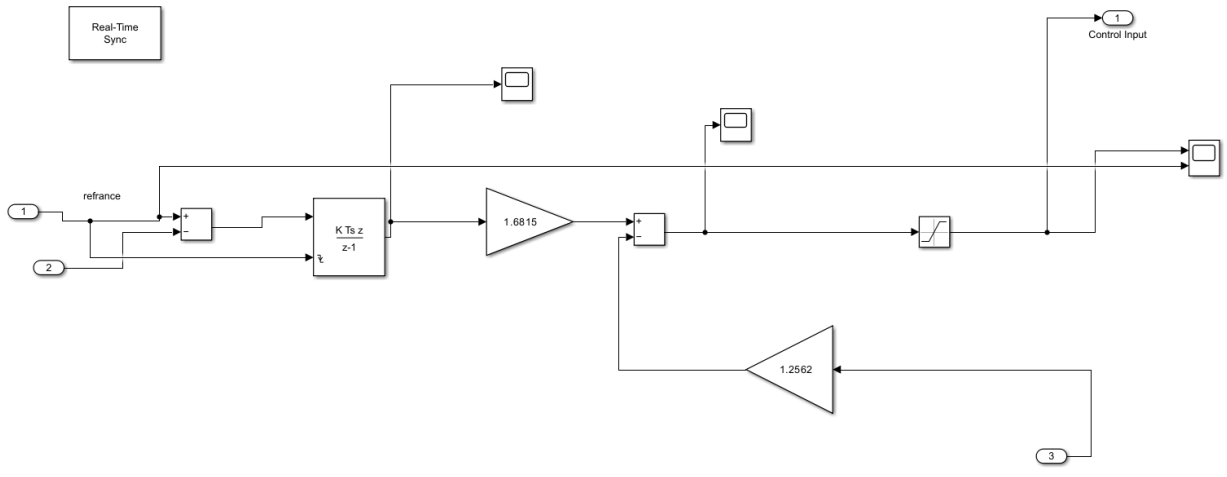

Figure 24: Controller subsystem: state-space controller with integral action and actuator saturation.

Block-by-block explanation

- **Inport 1 (reference):** Reference speed command $r[k]$.

- **Inport 2 (measured speed for error):** Measured (filtered) speed $y[k]$ used for computing the tracking error.
- **Sum block** $(+/-)$**:** Forms the error signal $e[k] = r[k] - y[k]$.
- **Discrete integrator block** $\left(\frac{KT_s z}{z-1}\right)$**:** Implements integral action on the error. This is the discrete-time realization of
$$\xi[k] = \xi[k-1] + T_s\, e[k],$$
(with internal scaling consistent with the chosen implementation). The integrator state $\xi[k]$ is what forces steady-state error toward zero.
- **Gain block (1.6815):** Scales the integrator contribution. Practically, this is the integral feedback gain applied to $\xi[k]$.
- **Inport 3 (measured speed for state feedback):** Feedback signal used in the proportional/state-feedback path. (In many implementations this is the same measured speed signal; a separate input is often used simply to keep wiring and routing clear.)
- **Gain block (1.2562):** Scales the state-feedback term (proportional feedback on the measured speed).
- **Sum block** $(+/-)$ **after gains:** Combines the integral and state-feedback actions to form the commanded control:
$$u_{\text{cmd}}[k] = (\text{integral term}) - (\text{state feedback term}).$$
- **Saturation block:** Limits the actuator command to the allowable PWM range of the hardware. This is essential because the driver/Arduino cannot output beyond fixed bounds; it also prevents unrealistic commands in real-time tests.
- **Outport (Control Input):** The final saturated command $u[k]$ that is sent to the DC Motor System subsystem.
- **Scopes:** Monitor internal controller signals (e.g., pre-saturation command and saturated command) to diagnose saturation and windup-related behavior.

6.3 DC Motor System subsystem (Figure 25)

This subsystem contains the Arduino I/O, the encoder processing chain, and the speed filtering. The subsystem takes the controller command and outputs a clean speed measurement for feedback.

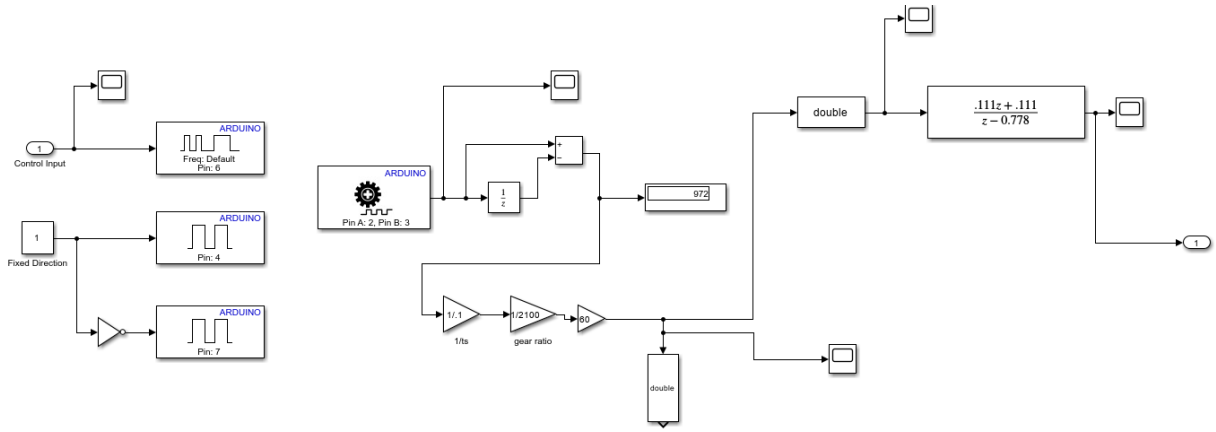

Figure 25: DC Motor System subsystem: Arduino PWM + direction outputs, encoder acquisition, speed computation, and low-pass filtering.

Block-by-block explanation

- **Inport (Control Input):** Receives the controller output $u[k]$.
- **Arduino PWM block (Freq: Default, Pin 6):** Writes the command to the PWM-enabled pin connected to the motor driver enable (speed actuation).
- **Fixed Direction (Constant = 1):** Sets a fixed rotation direction for the experiment (forward). This simplifies testing and matches the identification assumptions.
- **Arduino Digital Output (Pin 4):** Direction control line 1 to the H-bridge.
- **NOT (logic inverter):** Generates the complementary direction signal.
- **Arduino Digital Output (Pin 7):** Direction control line 2 to the H-bridge (complement of Pin 4), ensuring a valid direction command pair.
- **Arduino Encoder block (Pin A: 2, Pin B: 3):** Reads the quadrature encoder channels and outputs a pulse count/position signal $N[k]$.
- **Unit Delay ($1/z$):** Stores the previous encoder reading $N[k-1]$.
- **Sum block ($+/-$):** Computes the incremental pulse count:

$$\Delta N[k] = N[k] - N[k-1],$$

  which is the core step for speed estimation.
- **Display block:** Shows the instantaneous value (e.g., pulse difference / count) during runtime for quick debugging.
- **Gain block ($60/206.8$):** Converts the incremental pulse count to a speed quantity using the encoder scaling and sampling time (a calibrated conversion constant).
- **Absolute value block $|u|$:** Ensures the speed magnitude is positive (useful when direction is fixed and the control objective is speed magnitude).

- **Discrete Transfer Function block** $\left(\frac{0.111z+0.111}{z-0.778}\right)$**:** Implements a first-order discrete low-pass filter to smooth the measured speed and suppress quantization ripple/noise before feedback.
- **Outport:** Outputs the filtered speed $y[k]$ back to the top-level feedback loop.
- **Scopes:** Monitor intermediate signals such as raw speed estimate and filtered speed for verification and tuning.

7 Experimental Results and Hardware Validation

This section presents the measured closed-loop performance of the implemented controller on the real DC motor setup (Arduino + L298N driver + encoder feedback). The controller gains were computed using the identified first-order plant model selected in the identification stage (Trial 3), and then deployed in the real-time Simulink Connected I/O implementation.

7.1 Tracking requirement and reference selection

A key project requirement is to track a commanded speed equal to 70% of the rated/open-loop speed. The rated speed measured for the motor in our setup was approximately

$$\omega_{\text{rated}} \approx 185 \text{ rpm}.$$

Therefore, the required reference for closed-loop tracking is:

$$r_{\text{req}} = 0.7\,\omega_{\text{rated}} = 0.7 \times 185 = 129.5 \approx 130 \text{ rpm}. \tag{13}$$

Accordingly, the controller was tested using a square-wave reference that switches between 0 rpm and approximately 130 rpm to evaluate repeated acceleration, settling, and deceleration behavior under realistic conditions.

7.2 Measured closed-loop response on the real motor

Figure 26 shows the measured speed response obtained from the real motor during repeated reference steps. The response demonstrates consistent tracking across multiple cycles: the motor accelerates toward the commanded speed level and returns to near zero speed when the reference drops, with repeatable behavior over the experiment duration.

Two signals are shown in the same plot. The more oscillatory staircase-like trace corresponds to the raw/less-conditioned measured speed (encoder quantization and sampling ripple are visible), while the smoother trace corresponds to the conditioned/converted feedback signal used by the controller (after data type conversion and filtering in the Simulink acquisition chain). Importantly, both traces confirm that the achieved steady-state speed level is close to the required reference value in (13).

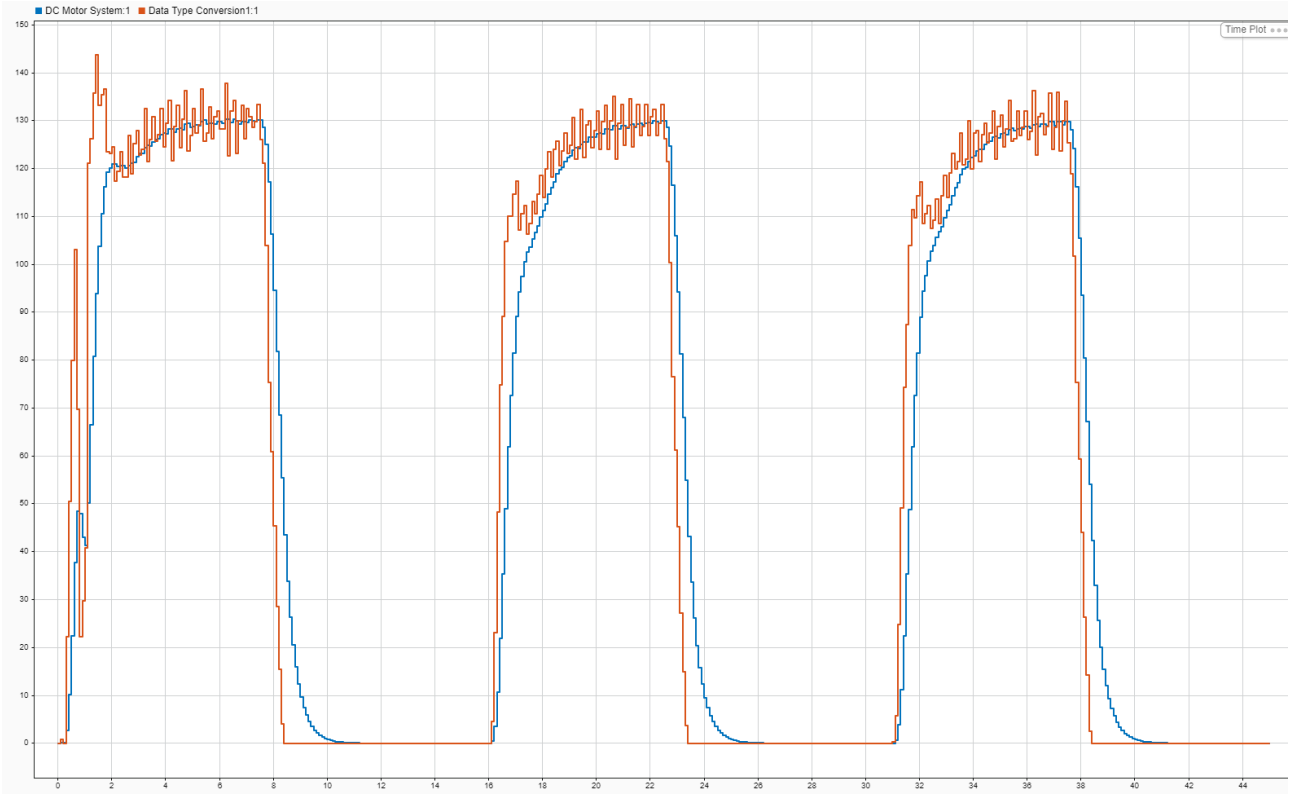

Figure 26: Experimental closed-loop response on the real motor for a square-wave reference of approximately 130 rpm (70% of rated speed). The plot shows the measured speed signals used for monitoring/feedback during real-time execution.

7.3 Controller gains used (implementation values)

The controller was implemented using the gains computed from the selected identified model (Trial 3). The implemented control law (as realized in Simulink) combines integral action on the tracking error with proportional/state feedback on the measured speed:

$$u[k] = K_I\,\xi[k] - K\,y[k], \qquad \xi[k] = \xi[k-1] + T_s\Big(r[k] - y[k]\Big).$$

The numerical gains used in the real-time experiment were:

$$K = 1.2560, \qquad K_I = 1.6814.$$

7.4 Discussion

The experimental results confirm that the controller achieves the required tracking level (approximately 130 rpm) in steady state and maintains repeatable response across multiple reference transitions. The remaining ripple in the raw measured speed is primarily due to encoder quantization and sampling effects; the filtering/conditioning path reduces this ripple and provides a cleaner feedback signal for control. These hardware results validate the effectiveness of the designed integral action for eliminating steady-state error and demonstrate that the controller remains functional under real PWM constraints.

8 Actuator Saturation and Integrator Windup Problem

8.1 Observed issue after implementing integral action

After implementing the state-feedback controller with integral action, the closed-loop response exhibited an undesired behavior: the motor speed tended to rise to (and remain near) the rated/maximum speed regardless of the applied reference command. This indicated that the control signal was frequently driven to its saturation limit (PWM/duty-cycle constraints), causing the plant to operate in a nonlinear regime not captured by the linear design model.

Experimental evidence of saturation and windup

Figure 27 provides direct experimental evidence of actuator saturation. The PWM command reaches the maximum allowable value and remains saturated for an extended interval, while the motor speed climbs toward the rated speed and does not follow the reference step. This behavior is consistent with integrator windup: once the actuator saturates, the tracking error persists and the integral term continues to accumulate, forcing the controller to remain saturated even when the output is already beyond the desired reference.

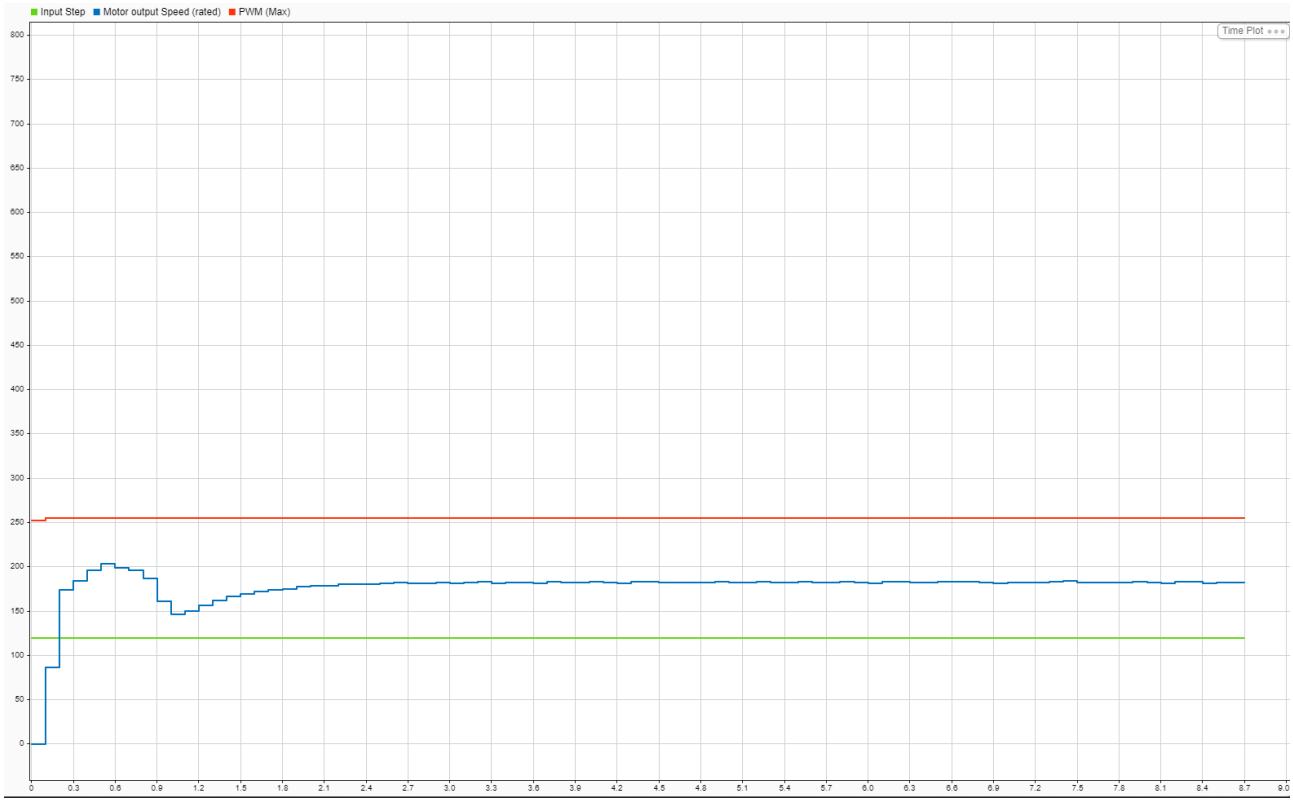

Figure 27: Hardware evidence of actuator saturation and windup. The PWM command reaches the maximum limit and remains saturated, while the measured speed rises toward the rated speed and becomes insensitive to the reference command.

Figure 28 shows the controller implementation used during this condition, including the integrator path, gain blocks, and the saturation block that limits the actuator command. The presence of saturation together with integral action motivates the anti-windup protection described later in this section.

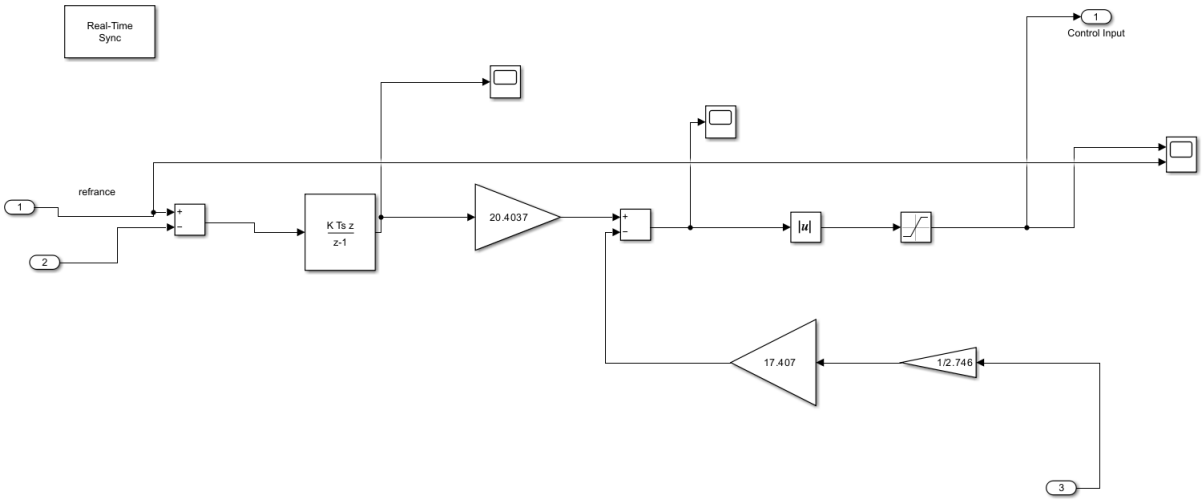

Figure 28: Controller subsystem during the windup condition, showing the implemented integral-action structure, the actuator saturation block, and the numerical gain values applied in this test (gains obtained from Trial 2 in the System Identification section).

8.2 Root cause: integrator windup under saturation

The implemented tracking controller with integral action can be represented as

$$\dot{\zeta}(t) = r(t) - y(t), \tag{14}$$

$$u_{\text{cmd}}(t) = -Kx(t) - K_i\zeta(t), \tag{15}$$

$$u(t) = \text{sat}(u_{\text{cmd}}(t)), \qquad u \in [u_{\min}, u_{\max}]. \tag{16}$$

When the actuator saturates, the applied input $u(t)$ can no longer follow the commanded input $u_{\text{cmd}}(t)$. As a result, the tracking error $e(t) = r(t) - y(t)$ may persist for an extended period, and the integral state $\zeta(t) = \int e(t)\,dt$ continues to accumulate. This phenomenon, known as *integrator windup*, can force the controller to remain saturated and may produce large overshoot, long settling times, and the appearance that the output is insensitive to the reference.

8.3 Proposed mitigation: anti-windup protection

To restore consistent tracking performance under realistic actuator limits, an anti-windup mechanism is introduced to prevent the integrator state from accumulating when saturation makes additional control action ineffective. Two widely used anti-windup schemes are summarized below.

8.3.1 Method A: conditional integration (integrator clamping)

In the clamping approach, the integrator update is disabled whenever the actuator is saturated *and* the instantaneous error would push the control signal further into saturation:

$$\dot{\zeta}(t) = \begin{cases} r(t) - y(t), & \text{if } u_{\min} < u(t) < u_{\max}, \\ 0, & \text{if } \Big(u(t) = u_{\max} \ \wedge \ (r(t) - y(t)) > 0\Big), \\ 0, & \text{if } \Big(u(t) = u_{\min} \ \wedge \ (r(t) - y(t)) < 0\Big). \end{cases} \tag{17}$$

This logic prevents the integral term from accumulating when the actuator cannot apply further effort in the required direction, thereby reducing overshoot and improving settling time.

8.3.2 Method B: back-calculation (tracking anti-windup)

Back-calculation introduces an additional feedback term proportional to the saturation mismatch:

$$\dot{\zeta}(t) = (r(t) - y(t)) + K_{aw}\Big(u(t) - u_{\text{cmd}}(t)\Big), \tag{18}$$

where $K_{aw} > 0$ is the anti-windup gain. During saturation, $u(t) \neq u_{\text{cmd}}(t)$, so the correction term drives $\zeta(t)$ toward a value consistent with the saturated actuator, enabling faster recovery once the output approaches the reference.

8.3.3 Implementation note and project limitation

As part of the mitigation effort, a back-calculation (tracking anti-windup) scheme was implemented in Simulink by feeding back the saturation mismatch term $\Big(u(t) - u_{\text{cmd}}(t)\Big)$ into the integrator dynamics as in (18). However, due to the limited project timeline, there was insufficient time to fully tune the anti-windup gain K_{aw}, validate the design under different operating conditions (e.g., varying references and load disturbances), and perform comprehensive experimental testing on the hardware.

9 Conclusions

This project presented the complete development cycle of a DC motor speed controller, from experimental setup and measurement conditioning to model identification, controller design, and real-time hardware validation. A first-order transfer function model relating PWM input to motor speed was obtained using MATLAB System Identification across multiple trials, and the selected model was used to compute state-feedback and integral gains via pole placement. The controller was implemented in discrete time using Simulink Connected I/O with Arduino, and experimental results confirmed stable and repeatable tracking of the required reference level (approximately 130 rpm, i.e., 70% of the rated speed) with near-zero steady-state error. Practical non-idealities observed during implementation—most notably actuator saturation and integrator windup—highlighted the importance of incorporating anti-windup protection for robust operation under PWM limits. Overall, the work demonstrates that state-space control with integral action provides effective speed regulation on low-cost embedded hardware when supported by careful signal conditioning, appropriate model selection, and consideration of real actuator constraints.

.1 MATLAB Implementation

```
clc; clear; close all;

%% 1) Plant Definition (TF & State-Space)
num = 0.9382;
den = [1 1.256];

G = tf(num, den);
[A,B,C,D] = tf2ss(num, den);

fprintf('================ PLANT MODEL (CONTINUOUS) ================\n');
disp('A ='); disp(A);
disp('B ='); disp(B);
disp('C ='); disp(C);
disp('D ='); disp(D);
disp('G(s) ='); disp(G);

%% 2) Open-Loop Dynamics
pole_ol = A;
tau_ol  = 1/abs(pole_ol);
Ts_ol   = 4*tau_ol;

fprintf('\n================ OPEN-LOOP DYNAMICS ================\n');
fprintf('Open-loop pole              : %.4f\n', pole_ol);
fprintf('Open-loop time constant     : %.4f s\n', tau_ol);
fprintf('Open-loop settling time (2%%): %.4f s\n', Ts_ol);

%% 3) Closed-Loop Specs and Desired Poles
Ts_cl = 0.8*Ts_ol;

p_d = -4/Ts_cl;
p_f = 5*p_d;
poles_desired = [p_d p_f];

fprintf('\n================ CLOSED-LOOP SPECIFICATIONS ================\n');
fprintf('Desired settling time (2%%)  : %.4f s\n', Ts_cl);
fprintf('Dominant closed-loop pole   : %.4f\n', p_d);
fprintf('Faster closed-loop pole     : %.4f\n', p_f);
fprintf('Desired pole vector         : [%.4f  %.4f]\n', poles_desired(1), poles_desired(2));

%% 4) Augmented System (Integral Action)
% xa = [x; zeta], with zeta_dot = r - y = r - Cx
A_aug = [ A   0;
         -C   0];
B_aug = [ B;
          0];

```

```
fprintf('\n================ AUGMENTED SYSTEM (INTEGRAL ACTION)
    ================\n');
disp('A_aug ='); disp(A_aug);
disp('B_aug ='); disp(B_aug);

%% 5) Gains via Pole Placement
K_aug = place(A_aug, B_aug, poles_desired);

Kx = K_aug(1);
Ki = -K_aug(2);    % using u = -Kx*x + Ki*zeta

fprintf('\n================ CONTROLLER GAINS ================\n');
fprintf('Kx (state feedback) : %.6f\n', Kx);
fprintf('Ki (integral gain)  : %.6f\n', Ki);

%% 6) Verification
Acl = A_aug - B_aug*K_aug;
fprintf('\nClosed-loop poles (A_aug - B_aug*K_aug):\n');
disp(eig(Acl));
```

Listing 1: Pole placement with integral action with faster pole (Controller Design).

```
clc;
close all;
num = 0.9382;
den = [1 1.256];

% Transfer function model
G = tf(num, den);

info = stepinfo(G);
disp(info);
[A, B, C, D] = ssdata(G);

Aa =[A 0;-C 0];
Ba =[B ;0];

%Desired poles;

s1 = -1.2560;
s2 = -1.2560;
P = [s1 s2];

K = acker(Aa, Ba, P);

disp(K)
```

Listing 2: Pole placement with integral action with 2 repeated poles (Controller Design).