UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

Corso di Laurea Magistrale in Ingegneria Informatica

Embedded Digital Control
Project Work Report

Authors:

Michele MARSICO
Mat. 0622702012
m.marsico10@studenti.unisa.it

Adamo ZITO
Mat. 0622702026
a.zito32@studenti.unisa.it

ANNO ACCADEMICO 2023/2024

CONTENTS

CHAPTER 1

INTRODUCTION

The DC motor is a commonly used actuator in industrial and robotic applications. Various control techniques are employed for the position control of brushed DC motors. The PID controller is the most popular and widely used in the industry due to its simplicity and reliability. Another option, based on the state-space representation of the system, is the Linear Quadratic Regulator (LQR), whose main advantage is to provide a systematic method for calculating the state feedback control gain matrix, minimizing the quadratic error of the states and the control effort.

1.1 Project Objective

The aim of this project is to develop digital controllers for a direct current (DC) motor. The main objective is to design and implement advanced control schemes, including state-feedback techniques, with a particular focus on Linear Quadratic (LQ) methods and standard regulators. The project also involves the integration and interfacing of a current sensor, the implementation of torque and position controls, and performance verification. Additionally, the implementation of code is required both through automatic generation (utilizing Simulink's external mode) and through direct coding. For the former approach, traditional design methodologies will not be employed; instead, the development process will follow the Model-Based Development (MBD) approach. In line with this approach, the validation of the implemented control schemes will be structured into the following phases:

- **Model-in-the-Loop (MIL)**: Validation of control models through simulation without hardware interfacing.
- **Software-in-the-Loop (SIL)**: Validation of control software integrated into the simulation model.
- **Processor-in-the-Loop (PIL)**: Validation of control software executed on a real processor, interfaced with the simulation model.

1.2 Hardware Setup

In Fig. 1.1, our fully assembled hardware setup is shown.

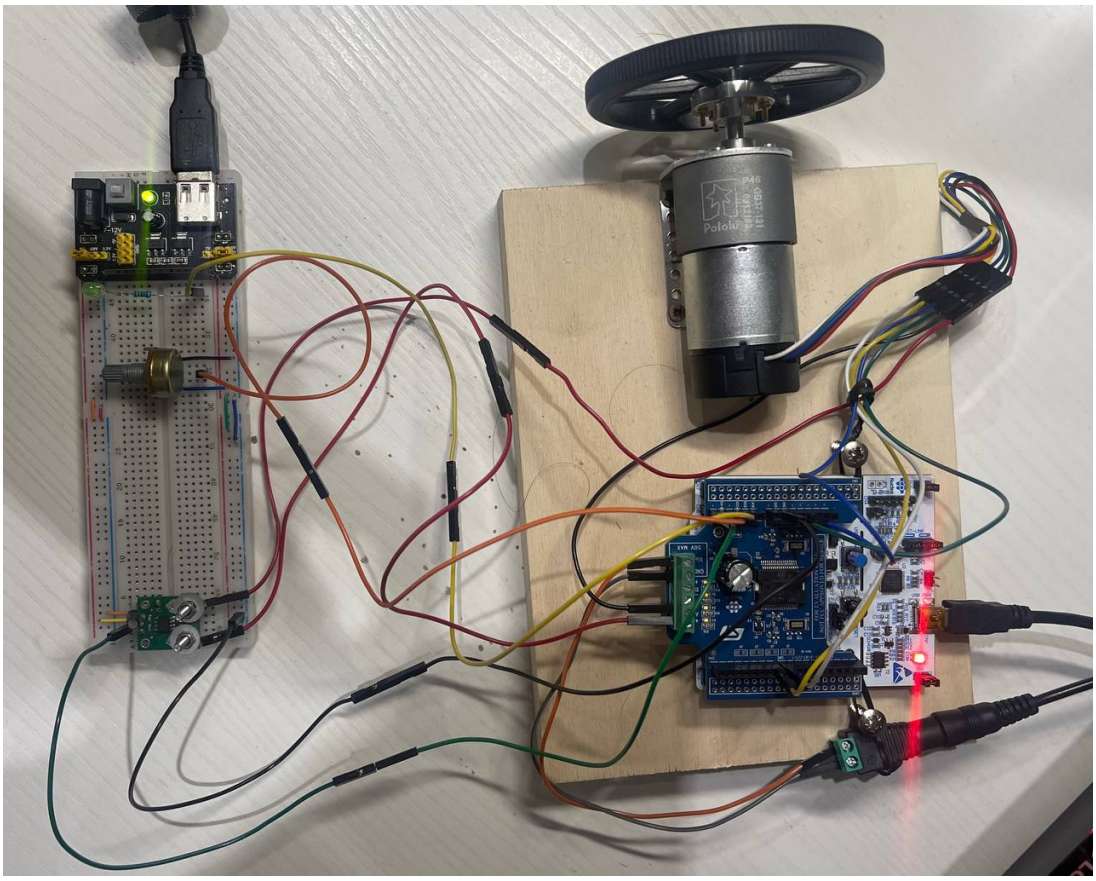

Figure 1.1: Hardware setup

- **STM32F401RE Nucleo Board**
  - **Description**: A development board based on the STM32F401RE microcontroller from STMicroelectronics. It features an ARM Cortex-M4 processor running at 84 MHz, with 512 KB of Flash memory and 96 KB of SRAM.
  - **Key Features**:
    - USB connectivity for programming and debugging.
    - Multiple I/O interfaces, including GPIO, ADC, PWM, I2C, SPI, and UART.
    - Compatibility with Arduino Uno R3 and ST morpho connectors for expansions.
- **Pololu 37D Gearmotor**
  - **Description**: A 12V DC motor with an integrated gearbox, produced by Pololu. It offers a good combination of speed, torque, and durability.
  - **Key Features**:
    - Nominal voltage: 12V.
    - Rotation speed: 80 RPM.

    - Torque: Typically around 0.5 Nm.
    - Output shaft: 6 mm diameter with flat key for secure attachment.

- **STM X-Nucleo IHM04A1 Driver**
  - **Description**: A DC motor driver board compatible with Nucleo boards like the STM32F401RE. It uses the L6206 motor driver, a high-power dual H-bridge.
  - **Key Features**:
    - Capable of controlling DC motors up to 48V and 5.6A.
    - Protection against over-voltage, over-current, and over-temperature.
    - Control interfaces: PWM for speed regulation, direction, and braking.
    - Compatibility with the STM32 ecosystem and ST software libraries.
- **ACS714 Current Sensor Carrier (-5A to +5A) Fig.**
  - **Description**: A current sensor based on the ACS714 from Allegro Microsystems, capable of measuring currents from -5A to +5A with high precision.
  - **Key Features**:
    - Measurement range: -5A to +5A.
    - Sensitivity: 185 mV/A.
    - Supply voltage: 5V.
    - Analog output proportional to the measured current, with offset at half of the supply voltage for zero current.
- **Potentiometer B10K**
  - **Description**: A 10 k Ω potentiometer used to provide a variable input in the circuit.
  - **Key Features**:
    - Resistance: 10 k Ω.
    - Mounting type: Panel mount.
    - Applications: Voltage adjustment and control of analog parameters.
- **Breadboard Power Module MB102 (3.3V and 5V) Fig. 1.2**
  - **Description**: A power module for breadboards capable of providing 3.3V and 5V for prototyping with Arduino and other DIY projects.
  - **Key Features**:
    - Output voltages: 3.3V and 5V.
    - Compatibility: Solderless breadboards.
    - Applications: Providing stable power supply to circuits on breadboards.

Figure 1.2: Breadboard Power Module MB102

- **12V Power Supply**
  - **Description**: A 12V power supply used to provide power to the DC motor.
  - **Key Features**:
    - Output voltage: 12V.
    - Current: Suitable for powering the Pololu 37D Gearmotor.
    - Applications: Powering motors and other devices requiring 12V.
- **3.3V Green LED**
  - **Description**: A green LED that will be used to indicate when the output matches the reference.
  - **Key Features**:
    - Forward voltage: 3.3V.
    - Color: Green.
    - Applications: Visual indication of output and reference alignment.

CHAPTER 2

POSITION CONTROL

2.1 Obtaining Model from Step Response

In the initial phase of the project, the focus was on identifying the model of the DC motor, a crucial step for the subsequent development of the digital controller. This section describes the methods used to obtain a mathematical representation of the system's dynamic behavior.

When the physical characteristics of the system are not known in advance, one of the most commonly used approaches to obtain a model, particularly a transfer function, is to exploit the system's step response. Such a system must have two fundamental requirements:

1. **Asymptotic stability.**
2. **Step response without overshoot (typical of systems with real poles).**

The goal is to derive a transfer function in the following form:

$$G(s) = \frac{\mu}{1 + Ts} e^{-\tau s} \tag{2.1}$$

This model is known as the First Order Plus Dead Time (FOPDT) model.

In real practice, it is sometimes very complex to design controllers for a real process even if its dynamics are known. The corresponding FOPDT model simplifies this task precisely due to the simplicity of the model that allows it to be obtained.

2.2 Model Identification Methods

Two approaches were evaluated for identifying the DC motor model:

- Tangent Method
- Area Method

These methods aim to derive the system's transfer function from its step response.

2.2.1 Tangent Method

The tangent method was considered for its conceptual simplicity in determining the point of maximum slope of the system's step response. However, its vulnerability to noise was identified as a significant limitation, as noise can compromise the accuracy of the point of maximum slope, leading to inaccurately high derivations. This method, although it has been one of the most used in the past, using incremental differentiation techniques, requires particular attention to high-frequency noise filtering. In the case under consideration, this could represent a strong criticality since the motor speed measurements made by the encoder are significantly affected by noise.

2.2.2 Areas Method

Similarly to the aforementioned method, the areas-based approach also allows deriving a transfer function from the system's step response. This method represents a more robust solution with respect to measurement noise, in contrast to the tangent method: this technique is based on calculating two areas through integration, and thanks to the presence of the integral, the effects of noise are compensated, allowing a more accurate estimation of the model parameters.

In light of the above, the area method was chosen. A simple formalization is illustrated Eqs. 2.2 and 2.3.

Consider the step response $\bar{u}$, the area between the response and the steady-state asymptote

$$S_1 = \mu\bar{u}\tau \int_0^\infty \mu\bar{u}e^{-\frac{t}{T}}dt = \mu\bar{u}(T+\tau) = \bar{y}(T+\tau). \tag{2.2}$$

Next, consider the area between the x-axis, the response, and the straight line that intersects the x-axis at $T+\tau$ and denote this area by S_2. The expression for S_2 is:

$$S_2 = \int_0^T \mu\bar{u}(1-e^{-\frac{t}{T}})dt = \frac{\mu\bar{u}T}{e} = \frac{\bar{y}T}{e}. \tag{2.3}$$

By solving the equations for S_1 and S_2, the unknown variables T and τ can be obtained.

2.2.3 Application of the Areas Method

To apply the chosen method, it has need to know the system's step response: in the case under consideration, since it is a 12V DC motor, the practical steps that were carried out are as follows:

- Provide a 12V voltage input to the motor.
- Record the motor's response for a sufficient time, where sufficient means until the system settles around the steady-state value Fig. 2.1.
- Verify the asymptotic stability of the system: being a speed-controlled generator, the system is stable as expected.
- Verify that the system's response does not overshoot.
- Apply the areas method to obtain T and τ in order to derive the system's input-output transfer function.

- Substitute T and τ into the general expression of $G(s)$ for FOPDT systems and obtain the identified transfer function.

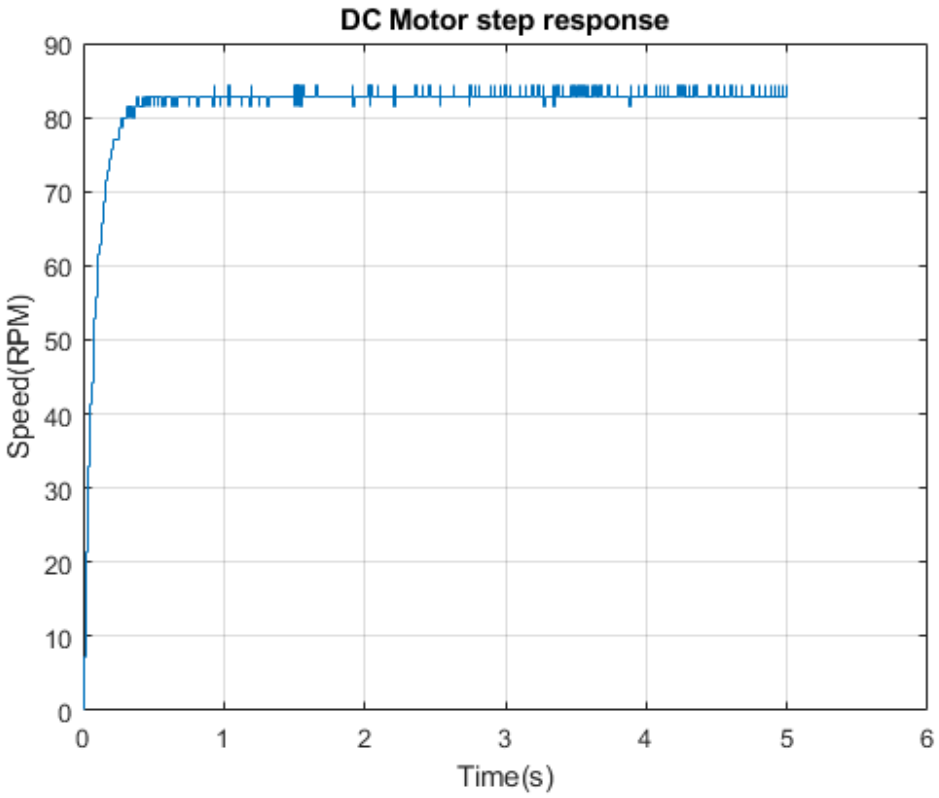

Figure 2.1: Step response of the real dc motor

2.2.4 Filtered Response VS Raw Response

The system's output is the speed expressed in RPM (revolutions per minute) and is measured using an encoder. Thus, this output is not "clean" but is affected by noise. It has been have already mentioned that one of the advantages of the area method is its robustness to noisy measurements. However, applying the aforementioned method, it was noticed that in some cases, the value of τ was negative: since this is the time constant characterizing the delay, it is obvious that such a result was not "realistic" but a consequence of the noisy output. Although this occurred only in some cases, it was decided to consider filtering y to "clean" the encoder measurements. Consequently, a moving average filter was employed, which operates by sliding a window of a certain size along the data and calculating the averages of the data within each window. The larger the window size, the smoother the filtered output, but this introduces a delay (not referring to τ, but to the filter delay), causing the filtered output to diverge from the true one. The goal was to find a moving average filter window size that was a good trade-off between output smoothness and the delay introduced by filtering. In practice, three significant ranges of window values were identified:

- In the range $2-3$, the value of τ was negative, thus presenting the same problem as using the original data.

- For values between 4 and 8, τ assumed positive values.
- For a window size of 9 or more, the difference between the original and filtered measurements became significant.

Our initial choice was to set the window size to 6, and the results showed that with this value, the filtered response almost perfectly matched the true one. Therefore, it was hypothesized that the filter could be used. The obtained values were:

$$T = 0.095\tau = 0.0066 \tag{2.4}$$

However, the decision to refer to the filtered response with the above-mentioned window size was later deemed inadequate: once the sampling time ($T_s = 0.005$) was chosen, it became apparent that the value of τ resulting from the method applied to the filtered response, though not negative, was no longer negligible compared to T_s (the choice of which is carefully detailed later into Ch. 2.7). This means that introducing the filter with $windowSize = 6$ would have led to designing the controller considering a "non-real" delay introduced only by the filtering. To better understand how the parameter influences the delay introduced by the filter, refer to Fig. 2.1.

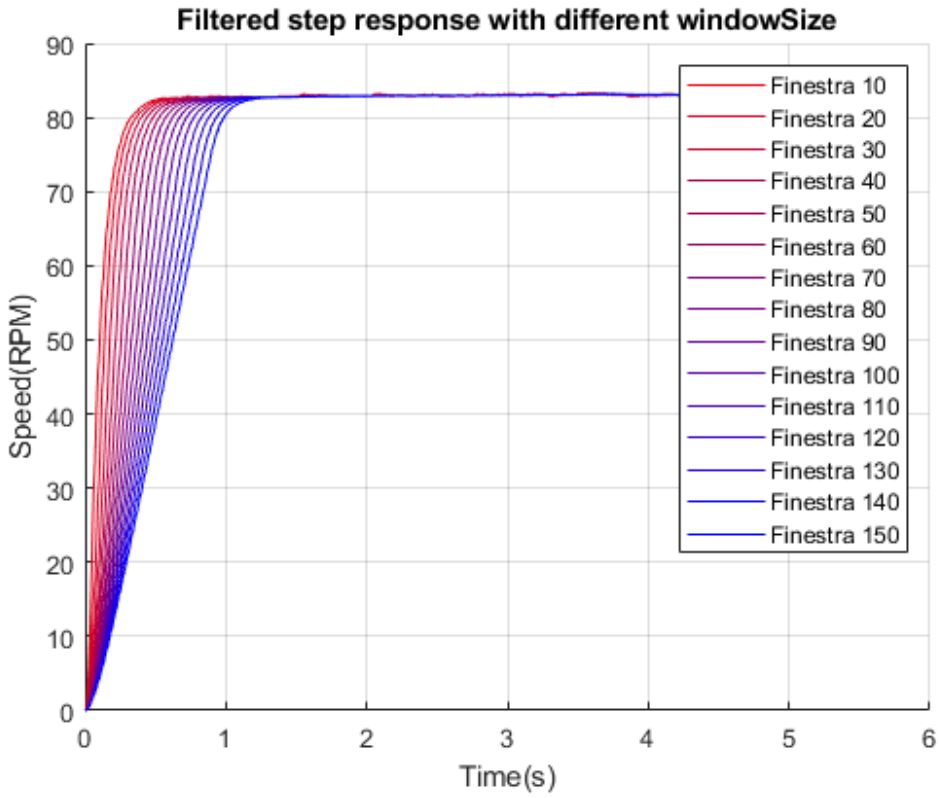

Figure 2.2: Filtered data with different window sizes

By fine-tuning the filter parameter (windowSize = 4, Fig. 2.3), it has been obtained a value of $\tau = 0.0004$, which is both positive and negligible compared to the sampling time, while $T = 0.094$.

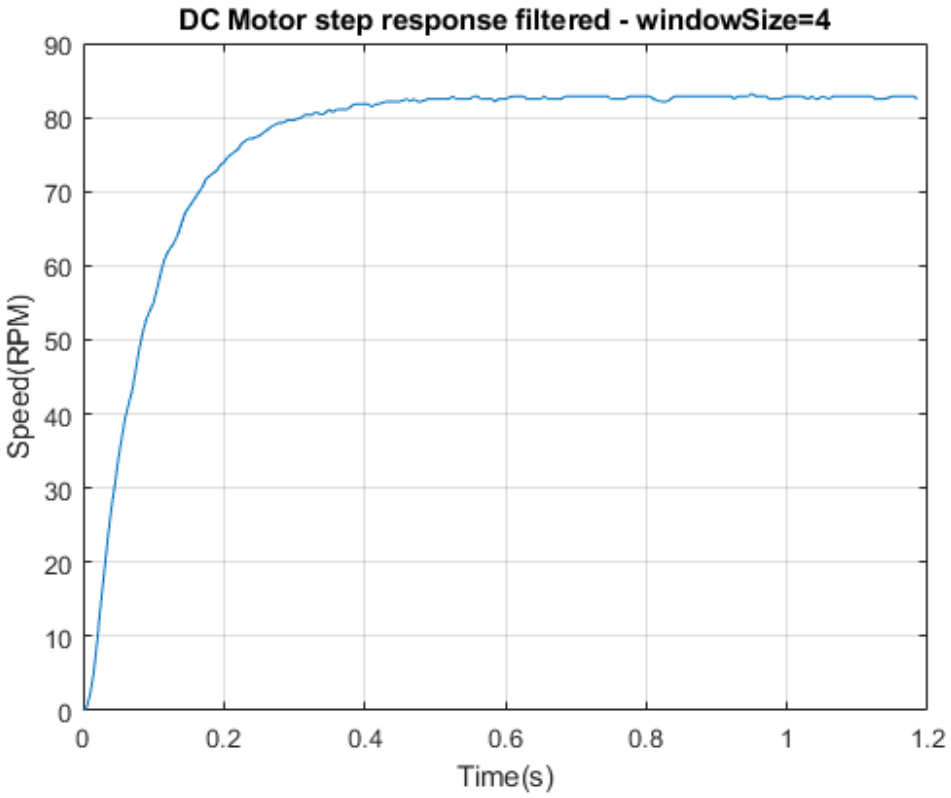

Figure 2.3: Filtered step response with *windowSize=4*

Using unfiltered data resulted in $\tau = -0.0042$ and $T = 0.090$. Given the values obtained with both filtered and original data, it was realized that the model was almost identical and that the time constant τ related to the FOPDT model delay could be assumed to be zero. The obtained transfer function is:

$$G_{vel}(s) = \frac{\frac{\bar{y}}{\bar{u}}}{1+Ts}e^{-\tau s} = \frac{\mu}{1+Ts}e^{-\tau s} = \frac{6.893}{1+0.094s} \tag{2.5}$$

The comparison between the step response of the real motor, the transfer function obtained using the original data, and the one derived from the filtered data can be appreciated Fig. 2.4:

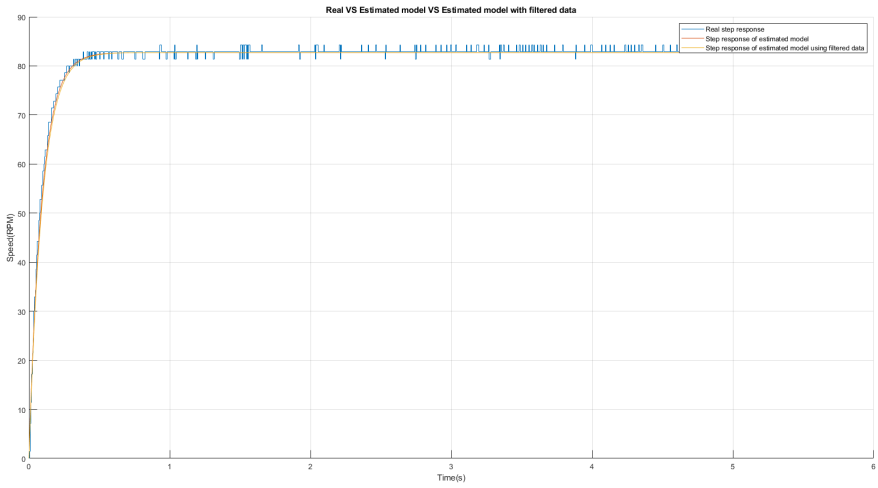

Figure 2.4: Step response of the real motor VS Step response of the estimated model VS Step response of the estimated model obtained with filtered data

Note that the obtained transfer function has the motor speed in RPM as its output. Since our interest is in position control, our transfer function will be:

$$G_{pos}(s) = \frac{6.893}{9.5493s(1 + 0.094s)} \tag{2.6}$$

where a pole at the origin is introduced and the multiplicative constant 9.5493 serves to convert the speed from RPM to rad/s.

2.3 State Space Model

For the implementation of a linear quadratic regulator, it need the state-space model. The first step is therefore to derive this representation from the transfer function obtained in the paragraph Methods of Areas 2.2.

The values of the matrices obtained in Matlab are:

$$A = \begin{bmatrix} -10.6383 & 0 \\ 1.0000 & 0 \end{bmatrix}; \quad B = \begin{bmatrix} 2 \\ 0 \end{bmatrix}; \quad C = \begin{bmatrix} 0 & 3.8395 \end{bmatrix}; \quad D = 0 \tag{2.7}$$

The resulting system is:

$$\dot{x}_1 = -10.6383x_1 + 2u \tag{2.8}$$

$$\dot{x}_2 = x_1 \tag{2.9}$$

$$y = 3.8395x_2 \tag{2.10}$$

The model in question is characterized by two state variables:

- x_1 represents the motor speed

- x_2 represents the motor position

It is also important to note that it was necessary to identify a state transformation T such that it obtain a matrix $C = [0\,1]$ while not changing the system dynamics (the eigenvalues of the matrix A must not vary). The transformation was obtained by defining the matrix T symbolically and solving a system of equations that imposed the desired values for A and C. The reason for this transformation is related to the fact that otherwise, in order to feedback the state in the Simulink scheme, it was necessary to scale the output by a factor equal to the coefficients of the original matrix C.

The matrices of the transformed system were obtained as follows:

$$A_t = T^{-1}AT \tag{2.11}$$

$$B_t = T^{-1}B \tag{2.12}$$

$$C_t = CT \tag{2.13}$$

$$D_t = 0 \tag{2.14}$$

The values of the transformed matrices are:

$$A_t = \begin{bmatrix} -10.6383 & 0 \\ 1.0000 & 0 \end{bmatrix}; \quad B_t = \begin{bmatrix} 7.6791 \\ 0 \end{bmatrix}; \quad C_t = \begin{bmatrix} 0 & 1 \end{bmatrix}; \quad D_t = 0 \tag{2.15}$$

Although the dynamic matrix of the transformed system coincides with that of the original system, and therefore its eigenvalues have not changed, it is considered appropriate, to substantiate the statement, to provide a comparison between the step responses of the two systems. As can be seen in Fig. 2.5, the two responses are perfectly matching:

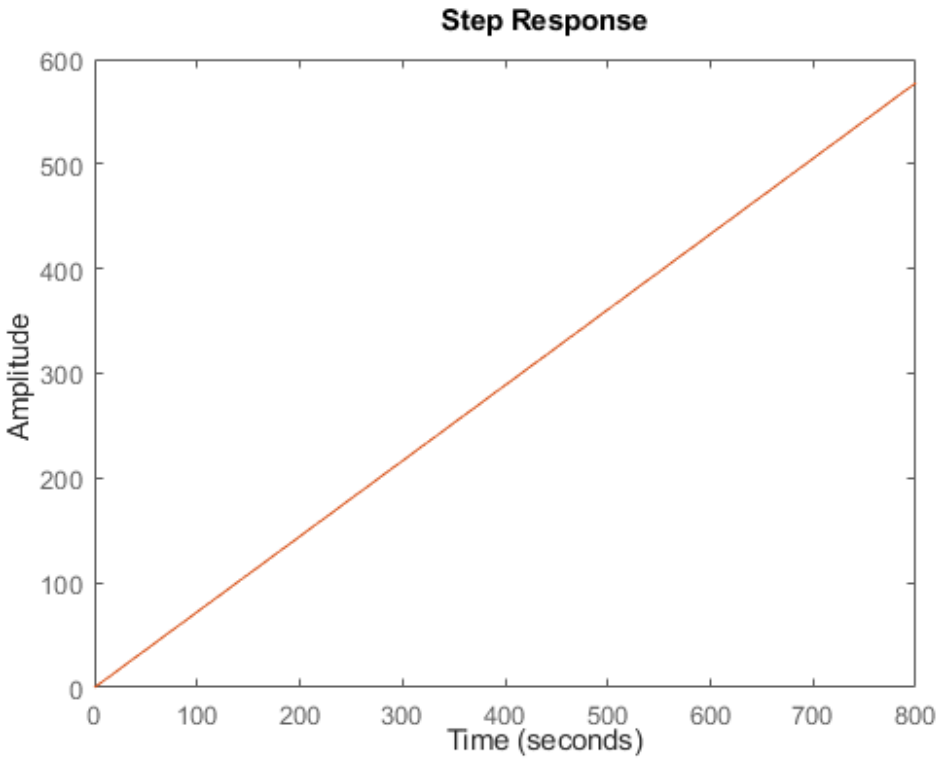

Figure 2.5: Step response of both original system and transformed system.

The continuous-time equations of the transformed system become:

$$\dot{x}_1 = -10.6383x_1 + 7.6791u \tag{2.16}$$

$$\dot{x}_2 = x_1 \tag{2.17}$$

$$y = x_2 \tag{2.18}$$

Once the dynamic equations of the model have been obtained, it was verified that the original system and, obviously, for completeness, also the transformed system were controllable, i.e., that the rank of the following matrices was equal to the order of the system, which in this case is 2:

$$W_r = [B \quad AB] = \begin{bmatrix} 2 & -21.2766 \\ 0 & 2 \end{bmatrix} \tag{2.19}$$

$$W_{r_t} = [B_t \quad A_t B_t] = \begin{bmatrix} 7.6791 & -81.6924 \\ 0 & 7.6791 \end{bmatrix} \tag{2.20}$$

Both systems are controllable, so it can proceed with the control design that will be addressed in the next paragraph. For further details, the reader is referred to the code.

2.4 Formalization of State-Feedback Control

Below is a brief formalization of the state feedback control law that it will implement.
Consider a linear dynamic system represented in state space by:

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{2.21}$$

$$y(t) = Cx(t) + Du(t) \tag{2.22}$$

where:

- $x(t)$ is the state vector
- $u(t)$ is the input vector
- $y(t)$ is the output vector
- A, B, C, D are the system matrices

In the state feedback control strategy, the control law $u(t)$ is determined based on the state vector $x(t)$ as follows:

$$u(t) = -Kx(t) + K_r r(t) \tag{2.23}$$

where:

- K is the state gain matrix
- $r(t)$ is the reference
- $K_r = \frac{-1}{C(A-BK)^{-1}B}$ is a term that serves to track the reference in steady state.

To reject disturbances, it need to insert an integral action in the control action:

$$u(k) = -Kx - K_I \int_0^T (y(\tau) - r)\, d\tau + k_r r \tag{2.24}$$

It defined another state of the system as:

$$\dot{z} = y - r \tag{2.25}$$

in order to rewrite the control law as:

$$u(k) = -Kx - K_I z + k_r r \tag{2.26}$$

Our extended system is now:

$$\begin{cases} \dot{x} = Ax + Bu \\ \dot{z} = y - r = Cx - r \end{cases} \tag{2.27}$$

In fact, the term $k_r r$ disappears since it is the integral action that guarantees steady-state performance. It can rewrite the system as:

$$\begin{bmatrix}\dot{x}\\ \dot{z}\end{bmatrix} = \begin{bmatrix}A & 0\\ C & 0\end{bmatrix}\begin{bmatrix}x\\ z\end{bmatrix} + \begin{bmatrix}B\\ 0\end{bmatrix}u + \begin{bmatrix}0\\ -1\end{bmatrix}r = \begin{bmatrix}A & 0\\ C & 0\end{bmatrix}\begin{bmatrix}x\\ z\end{bmatrix} + \begin{bmatrix}B & 0\\ 0 & -1\end{bmatrix}\begin{bmatrix}u\\ r\end{bmatrix} \tag{2.28}$$

In this case, the matrices of the extended system will be:

$$A_{ext} = \begin{bmatrix}-10.6383 & 0 & 0\\ 1.0000 & 0 & 0\\ 0 & -1 & 0\end{bmatrix}; \quad B_{ext} = \begin{bmatrix}7.6791\\ 0\\ 0\end{bmatrix}; \quad C_{ext} = \begin{bmatrix}0 & 1\end{bmatrix}; \quad D_{ext} = 0 \tag{2.29}$$

2.5 LQI Controller Design

For the implementation of the state-feedback control, it has chosen to use a control technique that solves a linear quadratic optimization problem and allows for disturbance rejection. This technique is called Linear Quadratic Integral (LQI) control. The reason for our choice is that it allows us to obtain optimal values for the gains K and K_I of the control law by tuning two weight matrices Q_z and Q_u, which impact the control task and the control effort, respectively. These matrices are chosen to be diagonal and positive definite (Q_z can also be positive semi-definite). The coefficients of the first matrix allow us to tune the "importance" it give to the various states: in our case, it is interested in position control and therefore "neglect" the velocity. Additionally, it must ensure that our goal is to track the reference in steady state and guarantee certain dynamic performance in the transient. Based on this, the coefficients of the Q_z matrix are chosen as follows:

- 0.015 for the first state of the system (velocity)
- 1.0 to weigh the position errors of the system more heavily
- 35.0 for the state associated with the integral action as it is responsible for ensuring $y = r$ in steady state

During the controller tuning process, it emerged that higher values for the coefficients related to velocity and position caused overshoot and a significant increase in the settling time, as can be seen in Figs. 2.7 and 2.8. As for the Q_u matrix, the chosen value is 0.001 because it was considered a good trade-off between control effort and system responsiveness.

The gain values obtained from the optimization process are:

$$K_1 = 4.2194 \tag{2.30}$$

$$K_2 = 55.6518 \tag{2.31}$$

$$K_I = -187.0829 \tag{2.32}$$

Note that the value of the gain K_I is negative simply because Matlab solves the LQ problem with respect to $r - y$ and not $y - r$ as required by the control law. Therefore, the control law will be:

$$u(t) = -K_1 x_1(t) - K_2 x_2(t) - K_I z(t) = -4.2194 x_1(t) - 55.6518 x_2(t) - 187.0829 z(t) \quad (2.33)$$

The closed-loop control system becomes the following:

$$\begin{bmatrix} \dot{x} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} A - BK & -BK_I \\ C & 0 \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} r \quad (2.34)$$

from which:

$$A_{cc} = \begin{bmatrix} -43.096 & -427.3544 & -1436.6 \\ 1.0000 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}; \quad B_{cc} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}; \quad C_{cc} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}; \quad D_{cc} = 0 \quad (2.35)$$

From this state space representation, it can obtain the closed-loop transfer function:

$$W(s) = \frac{-1437}{s^3 + 43.04 s^2 + 427.4 s + 1437} \quad (2.36)$$

In Fig. 2.6, the Bode diagrams of the closed-loop transfer function $W(s)$ can be appreciated.

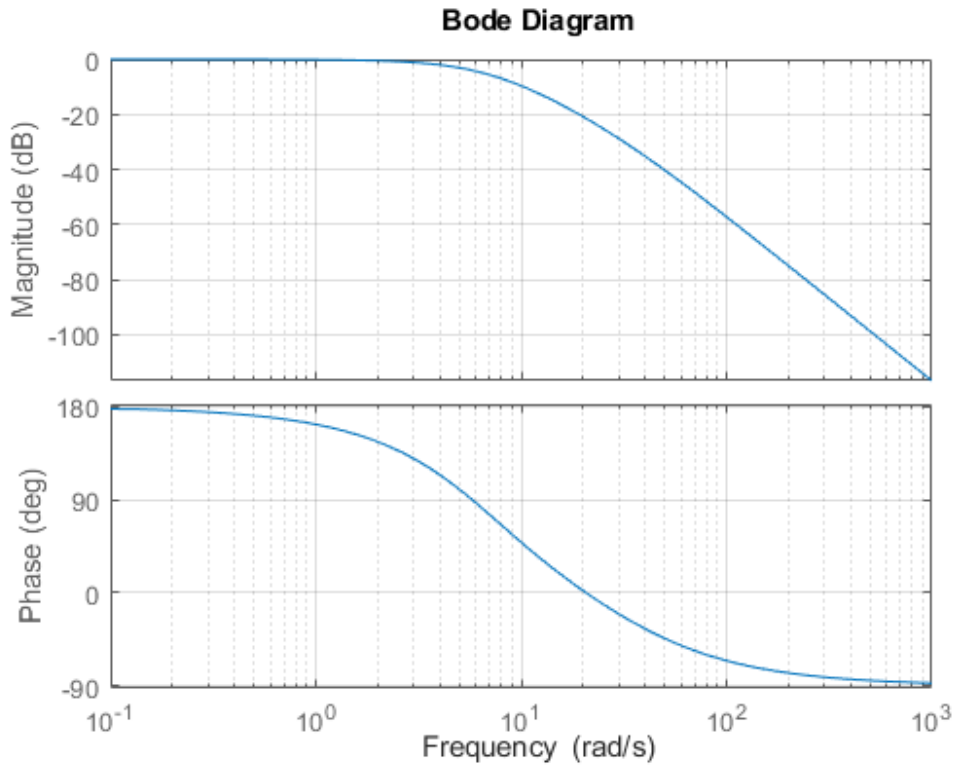

Figure 2.6: Bode diagrams of closed loop transfer function.

It is worth noting that the bandwidth of the closed-loop system is approximately $\simeq 5\,\text{rad/s}$.

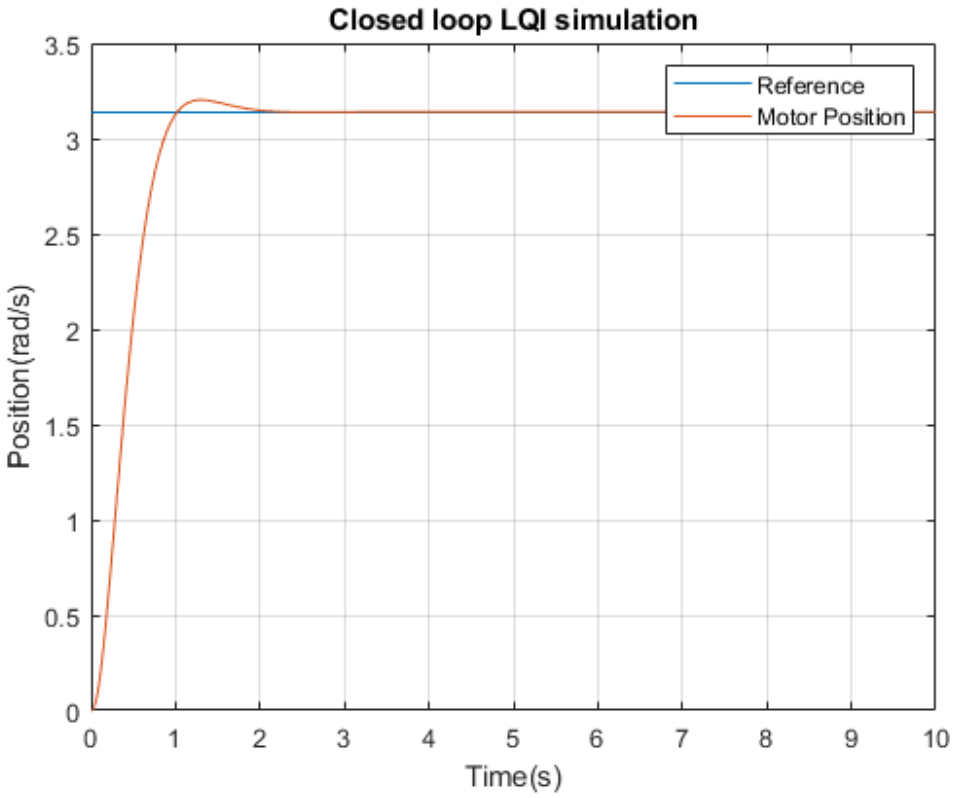

Figure 2.7: LQI performance when Qz(1,1) is 0.15

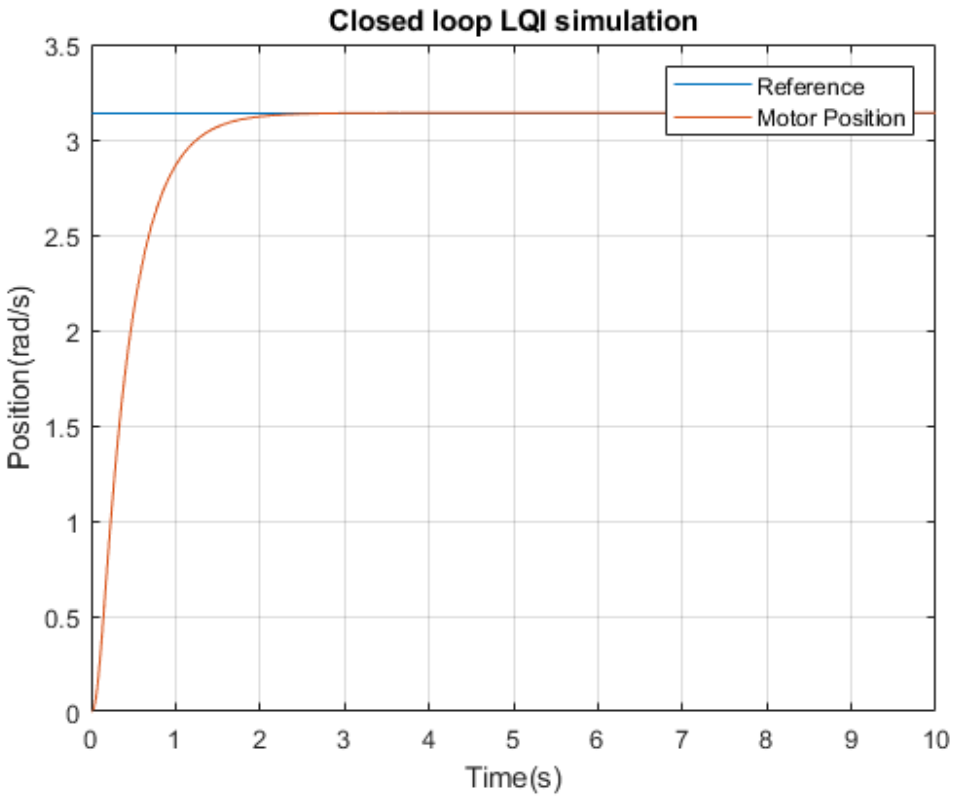

Figure 2.8: LQI performance when Qz(2,2) is 5

2.6 Incremental Algorithm

Following the LQI design phase, it was considered to implement an incremental control law. For a deeper understanding of the rationale behind this choice, please refer to Chs. 2.12.1 and 2.12.2. Given the significant role of the incremental algorithm in this project, it present a simplified formalization.

Consider the discrete control law:

$$u[k] = -Kx[k] - K_I z[k] \tag{2.37}$$

To express the control input not in absolute terms but in incremental form, it define the control increment $\Delta u[k]$ as:

$$\Delta u[k] = u[k] - u[k-1] \tag{2.38}$$

Substituting the discrete control law into the increment expression gives us:

$$\Delta u[k] = -K(x[k] - x[k-1]) - K_I(z[k] - z[k-1]) \tag{2.39}$$

From a practical standpoint, the implementation of the incremental algorithm in Simulink involves two key functions:

- `calculateDeltau`: This function takes as input the state values at time k and $k-1$, as well as the integral sum values for the same instances, along with the gains K (which includes K_I in this case). It computes the output increment Δu as the difference between the current and previous control inputs.
- `uFromDeltau`: This function acts as an integrator, updating the control input value based on the increment value received. When an increment is inputted, it is added to the current integral sum variable representing the integral action. It's important to note that the integral sum is subject to software saturation implemented within the function.

For further implementation details, please refer to the Simulink diagram and accompanying code.

2.7 Sampling Time Selection

The control law design has been conducted and tested in simulation in the continuous-time domain. However, for hardware deployment, a digital control law is necessary. Therefore, discretizing the control law (using the Tustin bilinear transformation) requires careful selection of the sampling period T_s.

Intuitively, one might think that a very small sampling period would allow the digital system to converge to the performance of its analog counterpart. In reality, this is not the case due to finite computation resolution in digital systems, which sets a lower limit on the sampling period.

Choosing too small of a sampling period introduces several issues:

- Error drift phenomenon
- Higher energy consumption

Conversely, setting the sampling period too high can destabilize the feedback loop, lead to information loss due to sampling effects, and compromise control algorithm accuracy.

It is crucial to strike a balance between response quality and required bit resolution. One initial consideration relates to the phenomenon of aliasing: according to the Nyquist-Shannon theorem, the sampling frequency must be at least twice the signal frequency. However, this limit is conservative, and additional constraints from the discretization technique must also be considered.

Initially, two empirical rules were considered for choosing the sampling period:

1. Dominant plant time constant T_p: Choosing T_s such that

$$T_s < \frac{T_p}{10} \tag{2.40}$$

   This rule is commonly used but can be risky under conditions where aggressive closed-loop performance is required on a system with poor open-loop performance.

2. Closed-loop performance requirements: If the closed-loop system needs to achieve a settling time T_{ss} or a natural frequency ω_n, it is reasonable to choose T_s such that

$$T_s < \frac{T_{ss}}{10} \quad \text{or} \quad \omega_s \geq \alpha\omega_n \tag{2.41}$$

   where $5 \leq \alpha \leq 20$ and $\omega_s = \frac{2\pi}{T_s}$.

In this case, the second empirical rule was adopted, selecting the sampling period based on desired closed-loop performance. Ideally, a settling time below 1 second was targeted, so initially, T_s was chosen as 0.05 seconds, dividing by 15 instead of 10 due to the heuristic suggesting a tighter constraint.

However, using this sampling period, it was observed that the system output oscillated around the steady-state value (see Fig. 2.9 and Fig. 2.10). Initial analysis suggested the need for a further reduction in the sampling period since, with T_s not being very low, it was unlikely to be an error drift phenomenon. To validate this hypothesis, the sampling period was reduced by 50%. The results demonstrated that the oscillations were almost entirely removed, but the system output did not precisely reach the desired steady-state value (see Fig. 2.11 and Fig. 2.12).

To further enhance the control system's response quality, the decision was made to lower the sampling period to 0.005 seconds, always keeping in mind the physical limits imposed by the STM32 NUCLEO F401RE board. As shown in Fig. 2.13, the performance significantly improved, with no oscillations observed and precise tracking of the reference achieved.

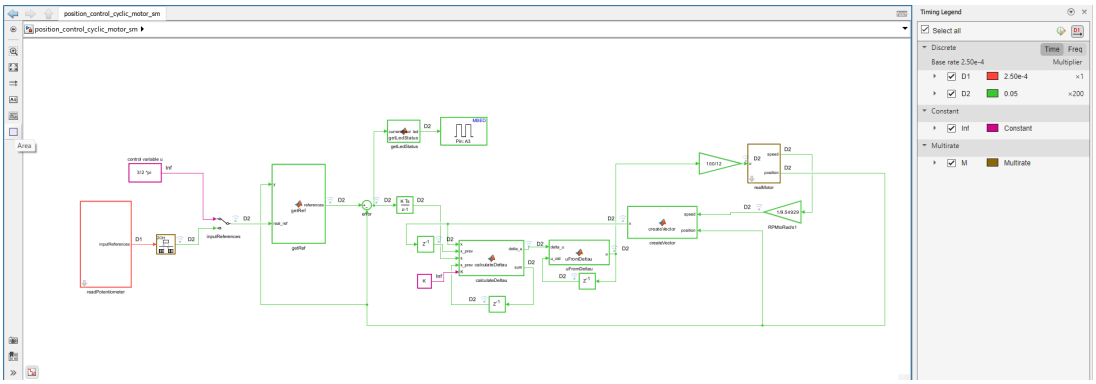

Figure 2.9: Timing legend of the Simulink scheme for $T_s = 0.05$ seconds.

Figure 2.10: Performance of the algorithm with $T_s = 0.05$ seconds.

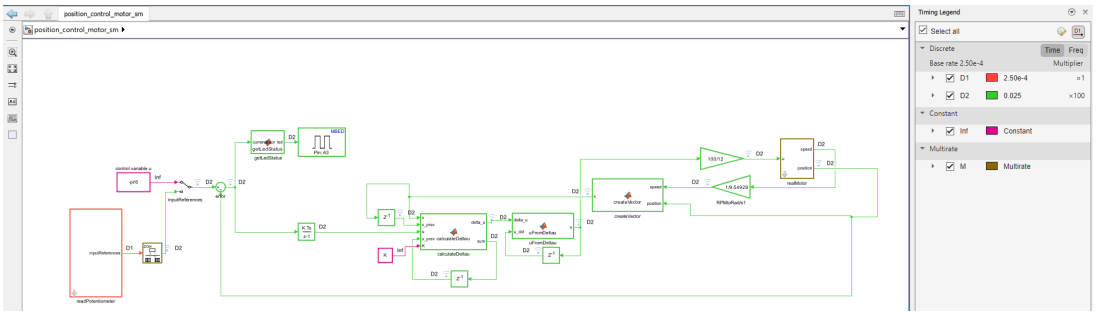

Figure 2.11: Timing legend of the Simulink scheme for $T_s = 0.025$ seconds.

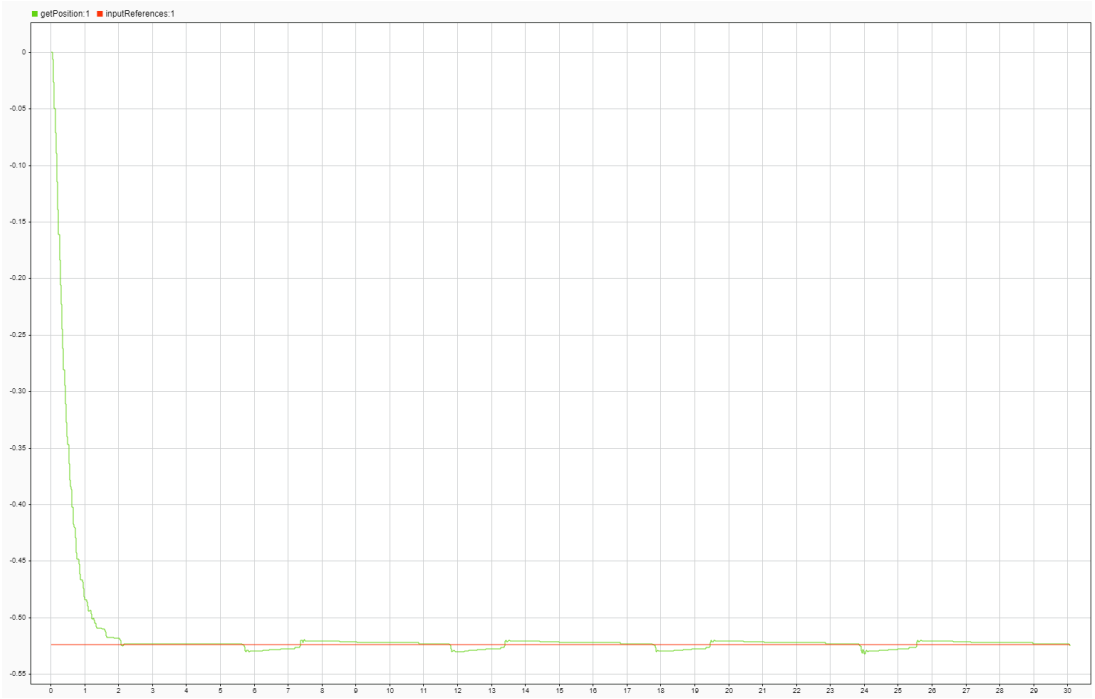

Figure 2.12: Performance improvement with $T_s = 0.025$ seconds, but some oscillations remain.

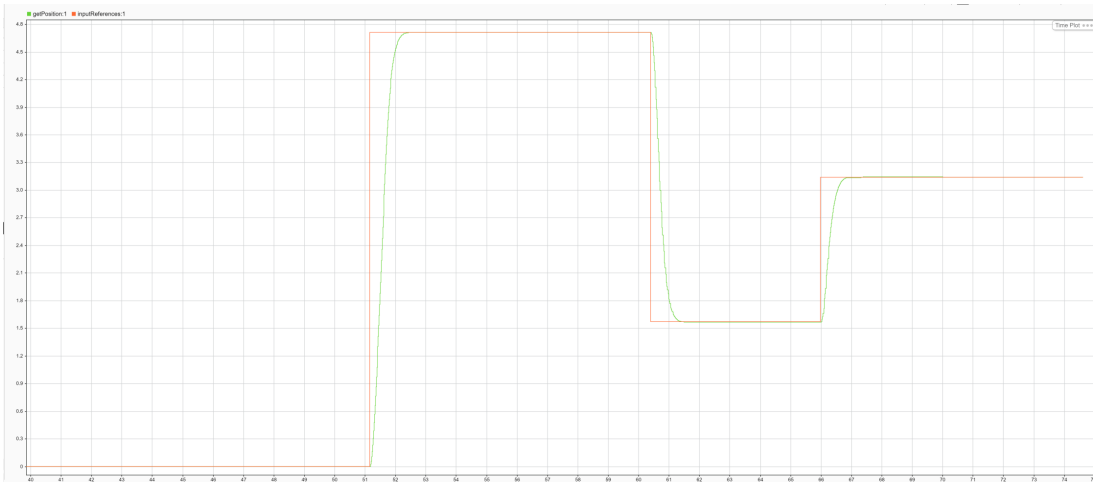

Figure 2.13: $T_s = 0.005$ seconds allows the system to precisely follow the reference.

2.8 Digital control law implementation

In Ch. 2.6, it was anticipated that for the implementation of the control law, an incremental control algorithm was chosen and a brief formalization was presented. In this paragraph, it will specialize the algorithm presented for the specific case at hand.

At each sampling period, the control input provided is given by:

$$u[k] = u[k-1] + \Delta u[k] \tag{2.42}$$

where

$$\Delta u[k] = -K_1(x_1[k] - x_1[k-1]) - K_2(x_2[k] - x_2[k-1]) - K_I(s[k] - s[k-1]) \tag{2.43}$$

In the calculation of the increment, note that $\Delta s[k] = s[k] - s[k-1]$. The term $s[k]$, which represents the integral sum of the error, is calculated as follows:

$$s[k] = s[k-1] + 0.0025e[k] + 0.0025e[k-1] \tag{2.44}$$

where $e[k] = y[k] - r[k]$ is the difference between the reference and the output at instant k. Substituting the gain coefficients gives us:

$$\Delta u[k] = -4.2194(x_1[k] - x_1[k-1]) - 55.6518(x_2[k] - x_2[k-1]) - 187.0829(s[k] - s[k-1]) \tag{2.45}$$

Since the range of values (expressed in Volts) that the motor accepts as input is $[-12, 12]$, when $u[k] > 12$, it set $u[k] = 12$, and when $u[k] < -12$, it set $u[k] = -12$. It is also noted to the reader that the expression for calculating $s[k]$ is the result of discretizing the integrator in the frequency domain, $\frac{1}{s}$, using the Tustin bilinear transformation with a sampling time $T_s = 0.005$s. In fact, by discretizing the continuous integrator as just described, in the z domain it obtain:

$$\frac{S(z)}{E(z)} = \frac{0.0025z + 0.0025}{z - 1} \tag{2.46}$$

from which it derive

$$zS(z) - S(z) = 0.0025zE(z) + 0.0025E(z) \tag{2.47}$$

Antitransforming the result gives us:

$$s(k+1) = s(k) + 0.0025e(k+1) + 0.0025e(k) \tag{2.48}$$

which, due to time invariance, can be rewritten as:

$$s(k) = s(k-1) + 0.0025e(k) + 0.0025e(k-1) \tag{2.49}$$

2.9 Model-in-the-Loop (MIL)

Model in the Loop (MIL) is a testing methodology used in embedded systems where the system behavior is simulated within a modeling framework without any physical hardware components. This approach allows for testing in the early stages of development, enabling the validation of requirements and gathering crucial information for subsequent project phases. During the MIL phase, setting the maximum achievable performance (steady-state error, response time) in simulation. The feasibility analysis of requirements allows determining the realizability of the project and defining requirements for the subsequent phases, which may vary from those initially requested by the user, depending on their feasibility.

The objective is to implement and verify a position control system for a DC motor using a Linear Quadratic Integrator (LQI) control implemented as an incremental algorithm.

2.9.1 Requirements

During the MIL phase, the system will be subjected to various test scenarios to validate the requirements and performance. The identified requirements for the model realization, which will be used to evaluate the model's effectiveness, are as follows:

1. **Asymptotic closed-loop stability**: This is a necessary and sufficient condition for the system's realization. It is mentioned for the sake of clarity, but it is almost trivial to emphasize.

2. **Steady-state error**: The system must ensure zero steady-state error with respect to the reference.

3. **Settling time**: The desired settling time is approximately 1-2 seconds.

4. **Rise time**: The rise time is not as critical as the settling time but should preferably be around 75-80% of the settling time.

5. **Overshoot**: Since our system is a DC motor, it is desirable to have no overshoot. A tolerance limit can be set around 3-4%.

6. **Disturbance rejection**: The system must be able to reject step disturbances.

2.9.2 Implemented Model

After defining the requirements, the model was developed Fig. 2.14. The control system model was built using Matlab/Simulink and includes the following main components:

1. **State-space representation of the motor**: Includes the dynamic equations describing the motor's behavior in terms of current, speed, and position.

2. **Incremental LQI Controller**: A Linear Quadratic Integrator (LQI) controller implemented as an incremental algorithm to regulate the motor's position.

3. **Position Reference**: A reference signal representing the desired motor position.

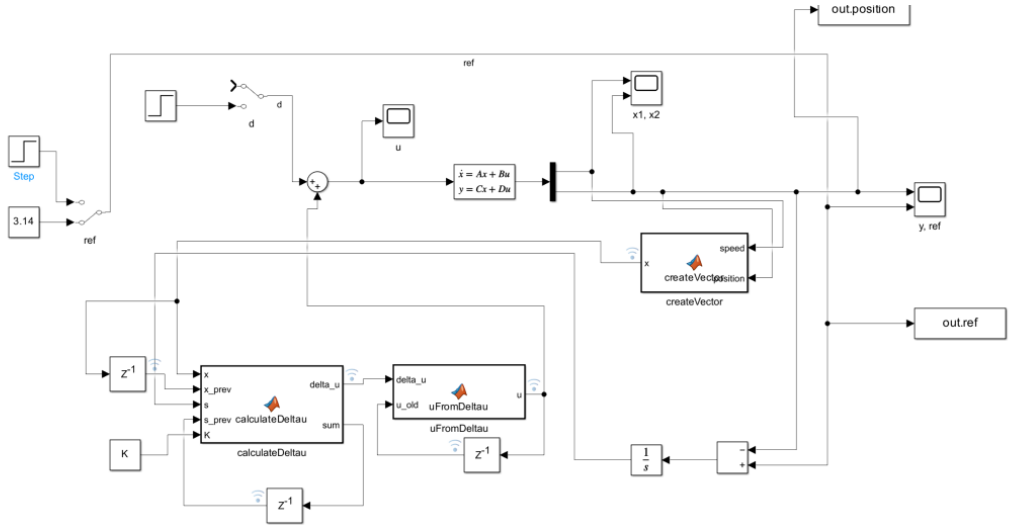

Figure 2.14: Scheme of position control MIL

2.9.3 Validation

This section presents and critically analyzes the results obtained during the validation phase. The model's effectiveness will be evaluated based on the compliance with the identified requirements. This phase is the first level of validation within the MBD paradigm and is crucial for identifying the model's issues.

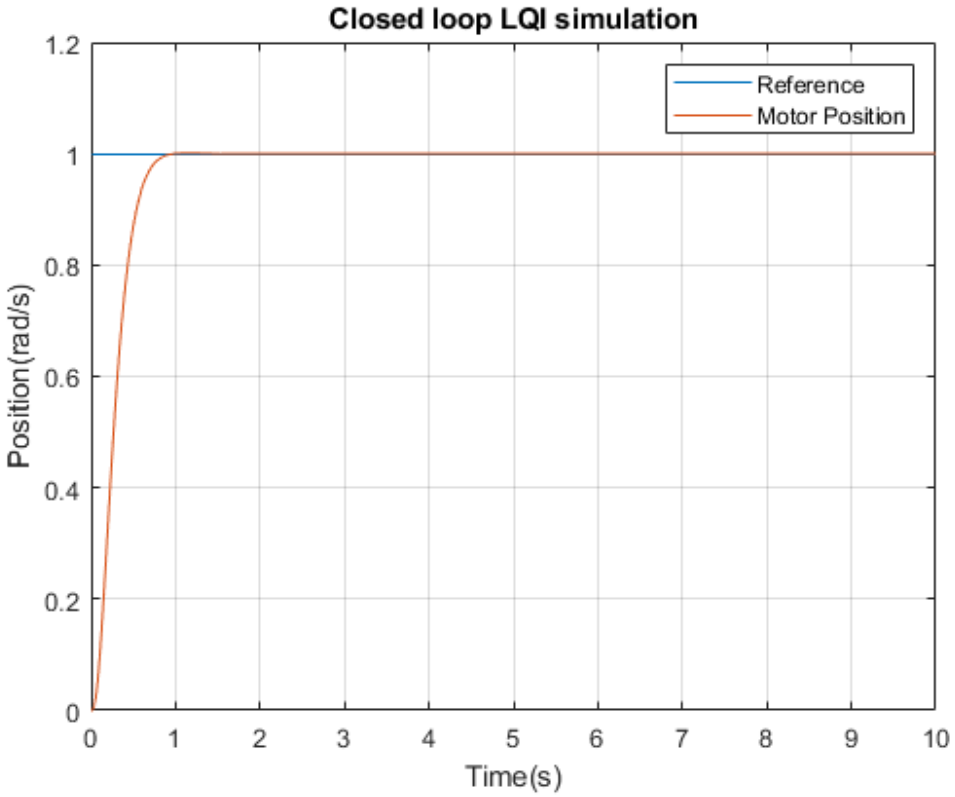

Figure 2.15: Step Response of the closed-loop system

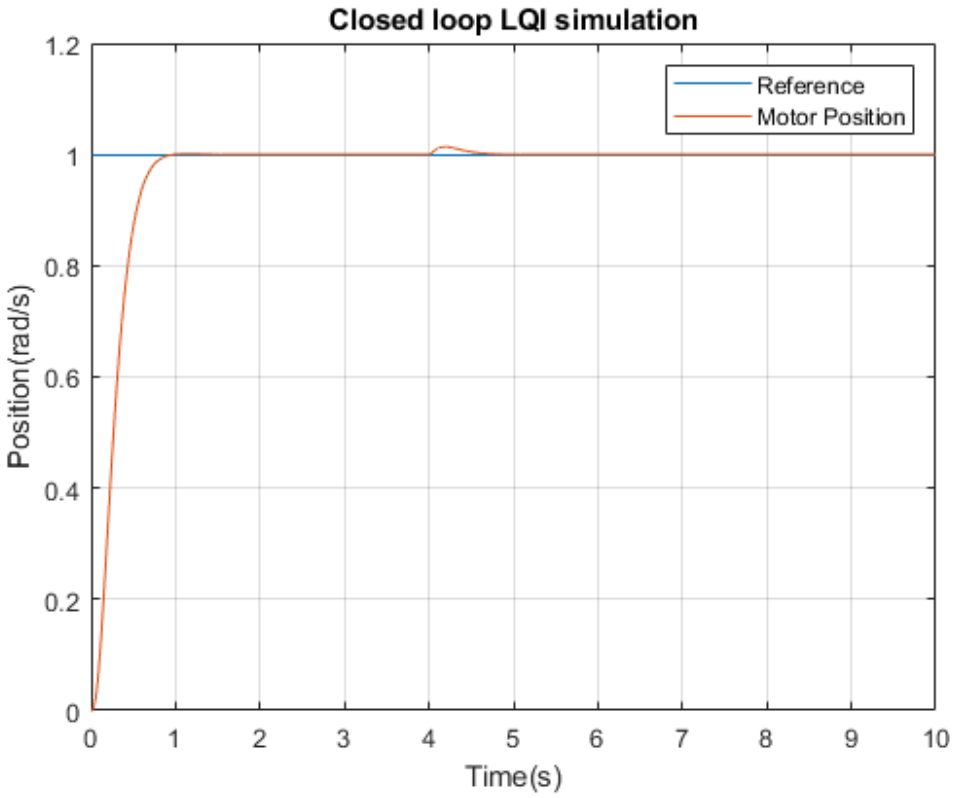

Figure 2.16: Effect of a step disturbance of amplitude 1

From the observed results (Figs. 2.15 and 2.16), it can be seen that:

1. The steady-state error is zero due to the integral action.
2. The settling time is approximately 1 second.
3. The rise time is 0.5 seconds.
4. The overshoot is less than 1%.
5. The system effectively rejects step disturbances. In the figure, the reference was 1, as was the disturbance amplitude occurring at $T = 4$ s.

In light of the conducted analysis, it is possible to affirm that the implemented model is well-aligned with the specified requirements, thus considered valid.

2.10 Software-in-the-Loop (SIL)

Software in the Loop (SIL) represents a crucial phase in the development process of embedded systems, where the controller code, both autogenerated and handwritten, is integrated and tested in a simulation environment. During the SIL phase, the controller code is executed on a simulated processor, different from the final one, allowing verification of the correctness and effectiveness of the code integration. This step helps identify and resolve issues that may not emerge when using only predefined Simulink blocks.

2.10.1 Model and Simulation

The control system model was developed using Matlab/Simulink Figs. 2.17 2.18, and the controller was implemented in C. The system included the following main components:

1. **DC Motor Model**: Included the dynamic equations describing the motor's behavior in terms of current, speed, and position.
2. **Incremental LQI Controller**: A Linear Quadratic Integrator (LQI) controller implemented as an incremental algorithm and coded in a Matlab function.
3. **Position Reference**: A reference signal representing the desired motor position.
4. **Autogenerated and Hand-written Code**: The code autogenerated by Matlab/Simulink was combined with handwritten code to verify its operation on a processor.

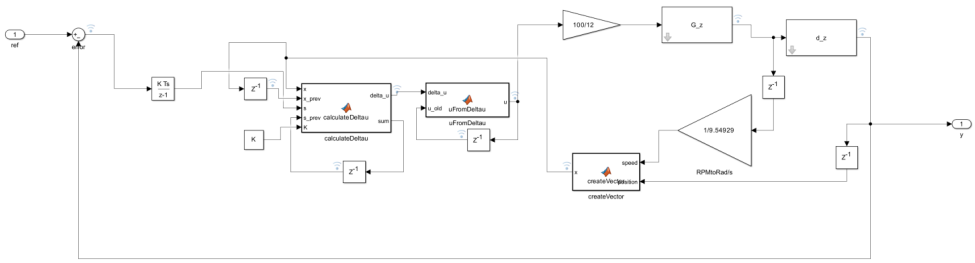

Figure 2.17: Scheme of position control z

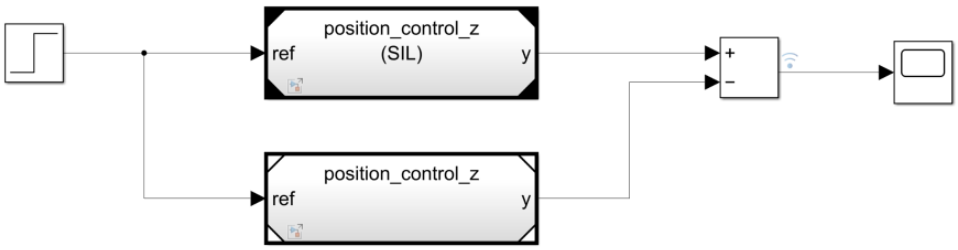

Figure 2.18: Scheme of position control SIL

2.10.2 Simulation Results

During the SIL phase, the system was subjected to various test scenarios to validate the requirements and performance. The obtained results were as follows Fig.2.19:

1. Verification of the requirements met during the MIL phase.

2. **Code Integration**: The integration between the handwritten and autogenerated code was successful, with all functionalities operating as expected.

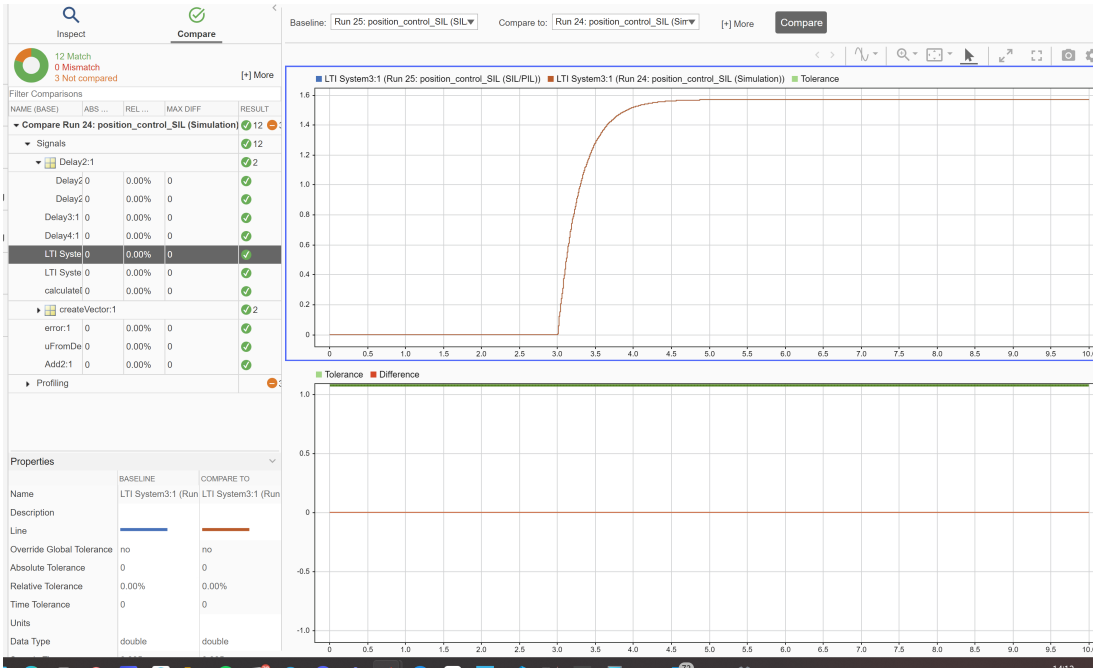

Figure 2.19: Output of position control SIL

2.11 Processor-in-the-Loop (PIL)

Processor in the Loop (PIL) represents an advanced phase in the development process of embedded systems, where the code obtained during the SIL phase is executed on the target machine, i.e., on the processor that will be actually used in the final system. In this phase, the autogenerated code is regenerated considering the target processor's specifications and executed in a simulation environment interfaced with the processor itself via USB connection. The main objective is to evaluate whether the processor is adequate for code execution, particularly in terms of execution time, by connecting the development environment with the target machine.

2.11.1 Model and Simulation

The model in this case included the same components as the SIL phase, except that in this case, Fig. 2.20, the target code is created and simulated for our board STM32F401RE.

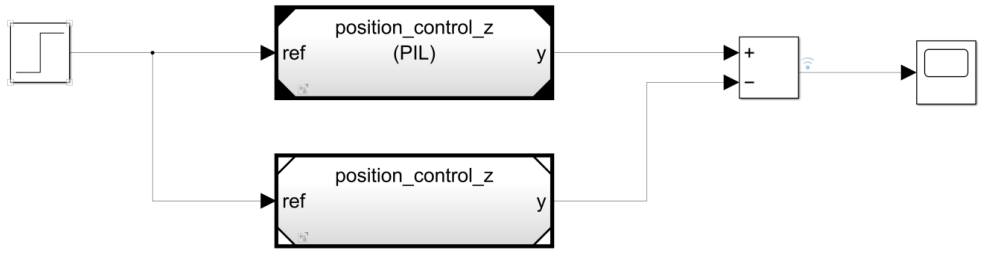

Figure 2.20: Scheme of position control PIL

2.11.2 Simulation Results

During the PIL phase, the system was subjected to various test scenarios to validate the requirements and performance. The obtained results are shown in Fig.2.21:

1. Verification of the requirements met during the SIL phase.

2. **Execution Time**: The total execution time of the algorithm was found to be 0.000036 seconds, Fig.2.22, corresponding to 0.72% of the sampling time of 0.005 seconds. This result is significantly below the 10% limit of the sampling time, and it is advisable to keep the execution time within 1-2% to ensure a wide safety margin, which is achieved as shown by the presented results.

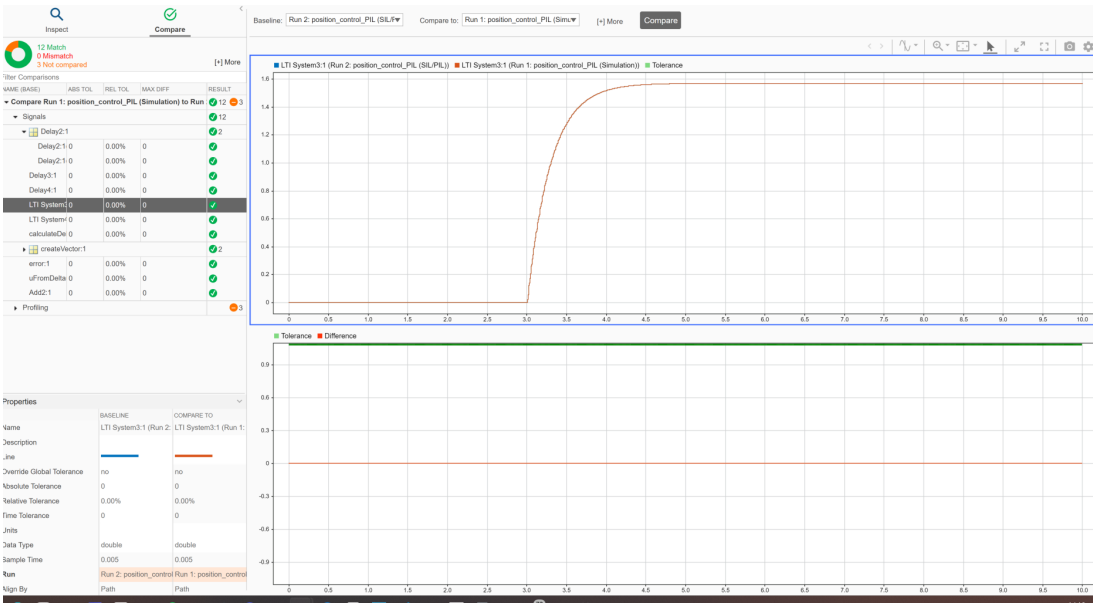

Figure 2.21: Output of position control PIL

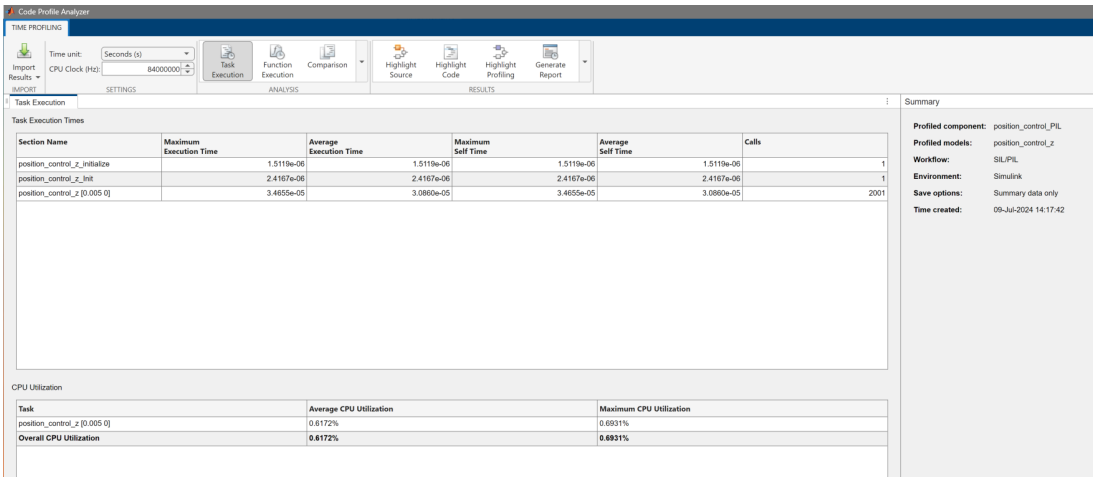

Figure 2.22: Execution time of position control PIL

2.12 Issues Encountered

In general, during the implementation of a digital controller, various types of issues need to be considered. However, the ones that have been particularly addressed in the context of this project are:

- Bumpless transfer
- Actuator limits
- Quantization

Regarding specific issues related to position control of our DC motor, the following have been encountered:

- Non-linearity introduced by the motor's dead zone
- Periodicity and error processing

2.12.1 Bumpless Transfer

In some cases, the process is started manually and operated by an operator during the transient phase. When the output approaches the desired state (near an equilibrium condition), the process is switched to automatic control. This can be due to reasons such as:

- The transient control might have surprises, requiring closer supervision.
- The system exhibits non-linearity during transient, making controller synthesis difficult.

However, manually operating the process in the equilibrium region and then switching to automatic control has a drawback: the integral term of the control law lacks information about the transient because it is not updated during manual mode. Since this integral term is the only memory element of our control law, transitioning to automatic mode without integrating

the difference between output and reference during manual mode leads to a jerk in the system's response. The solution considered for this issue was the implementation of an incremental control algorithm. One advantage of the incremental algorithm is its intrinsic bumplessness. By working with differences and providing input to the system not as the absolute input value but only the increment, when switching from automatic to manual, the increment is 0, eliminating the aforementioned issue. The challenge then shifts to integrating Δu: in our case, integration is performed via software.

Introducing a potentiometer in this context can be meaningful: it has been used not only to generate position references for LQI but also to directly provide the motor with an input voltage to manually bring the motor to around a desired position value and subsequently switch to automatic control. Thus, the potentiometer has been a key element in effectively validating the effectiveness of the incremental algorithm and in delineating performance differences between the incremental and "absolute" algorithms. In Fig.2.23, it observed how the potentiometer manually provides a control input to the motor to reach a certain position: after some time with no change in manual input, the switch to automatic mode occurs (highlighted in red), and it see a significant jerk in the control input, as expected, since the case shown does not implement an incremental control law. Note how the system's output also sharply moves away from the equilibrium zone it had reached.

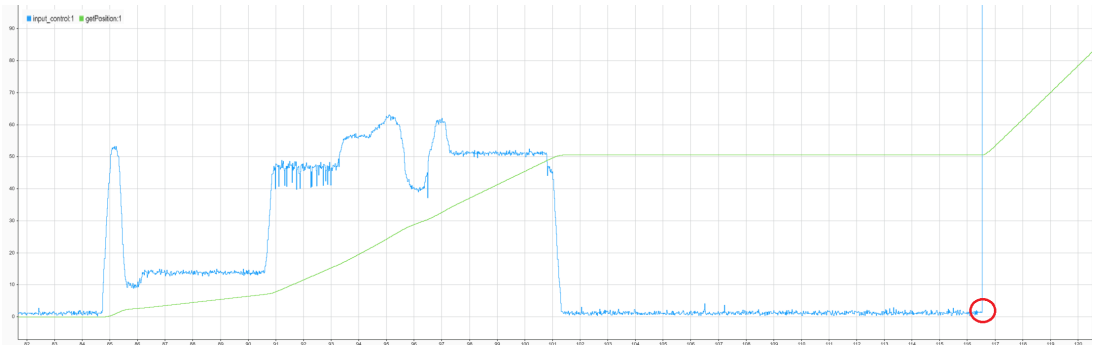

Figure 2.23: Bumpless transfer in position control of the motor

In Fig.2.24, it observed instead, by repeating the same procedure as described, that after the switch, the control input has a much smoother trend, and importantly, the system's output remains at the desired value.

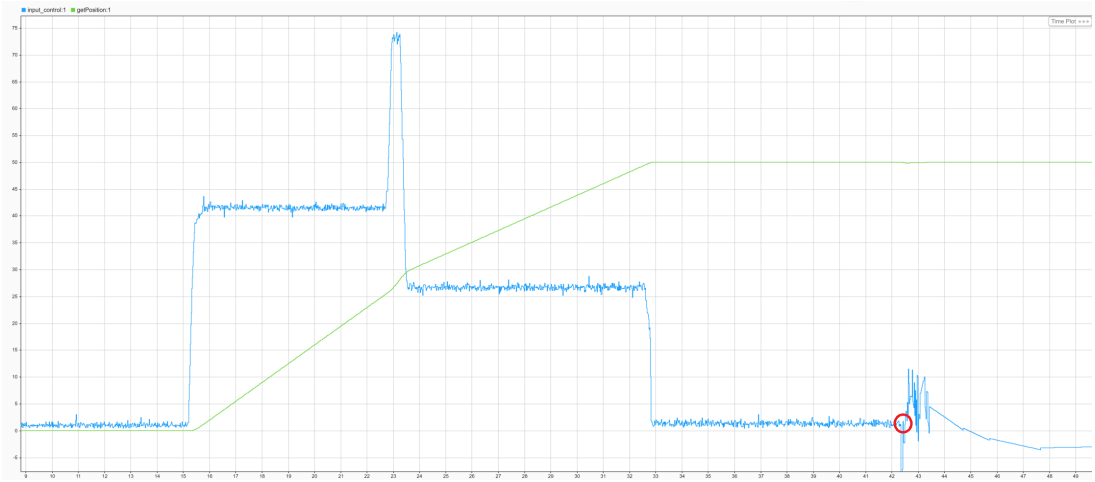

Figure 2.24: Resolved bumpless transfer in position control of the motor

2.12.2 Windup

Although the implemented control law is linear, there are some nonlinear phenomena that must be taken into consideration. These typically involve limitations in actuators: it is obvious that a motor has a limited speed. Since the motor operates over a wide range of conditions, it may happen that the control variable reaches the limits of the actuator. When this happens, the feedback loop is interrupted, and the system operates in open loop because the actuator remains at its limit regardless of the process output as long as the actuator remains saturated. The integral term also accumulates because the error is typically non-zero. Consequently, the integral term and the controller output can become very large. The control signal remains saturated even when the error changes, and it may take a long time before the integrator and the controller output return from the saturation range. The consequence is large transients. This situation is called integrator windup.

In the process of implementing the control law, it was considered that by implementing LQI, it would be possible to properly tune not only the integral action but also the control effort to limit this phenomenon. At the same time, from tuning the matrices, it was found that a higher weight on the integral action allowed significantly higher performance. Once again, the issue arose of finding a compromise between the two situations just illustrated. Although in most cases it was possible to track the reference without reaching actuator saturation and thus avoiding integrator windup, it was still decided to stress the control law to evaluate its performance even in more extreme situations. In practice, a step position reference with a significant amplitude was provided, and it was noted that actuator saturation, resulting in increased integral term, led to an increase in the transient time relative to degraded control law performance.

Although it is recognized that providing a step to the controller is not an optimal choice since the reference should vary slowly compared to the process and that with LQI it is possible to limit the phenomenon, it was still considered appropriate to address the windup problem. In the previous subsection, the incremental algorithm was mentioned. Among the advantages offered by the latter, in addition to bumpless transfer, one can include an increase in resolution (the increment requires fewer bits than an absolute input), but also the fact that it is inherently anti-windup. Below it explain the reason for this. When the incremental algorithm is used, two

paths open up:

- If integration is performed via hardware, it will be subject to default saturation. Therefore, when the aforementioned algorithm is adopted, the problem is structurally solved because at some point, the integrator stops.
- In the case where there is no hardware integrator, then the Δu is calculated and then a software implemented integral is used. In such a case, it is possible to insert a maximum limit beyond which integration is no longer performed, which has the same advantage as before.

The second solution is consistent with our case and is actually implemented by introducing saturation between $[-12, 12]$, which correspond to the extreme values that can be provided as input to the motor.

To demonstrate the effectiveness of the implemented solution, it will analyze the behavior of the system with and without the incremental algorithm.

Figure 2.25: Windup in position control motor

Fig. 2.25 shows how the motor follows the reference, which is initially set to 0. At a certain point, a significant step input is provided to the system, and it can be observed how motor saturation leads to windup.

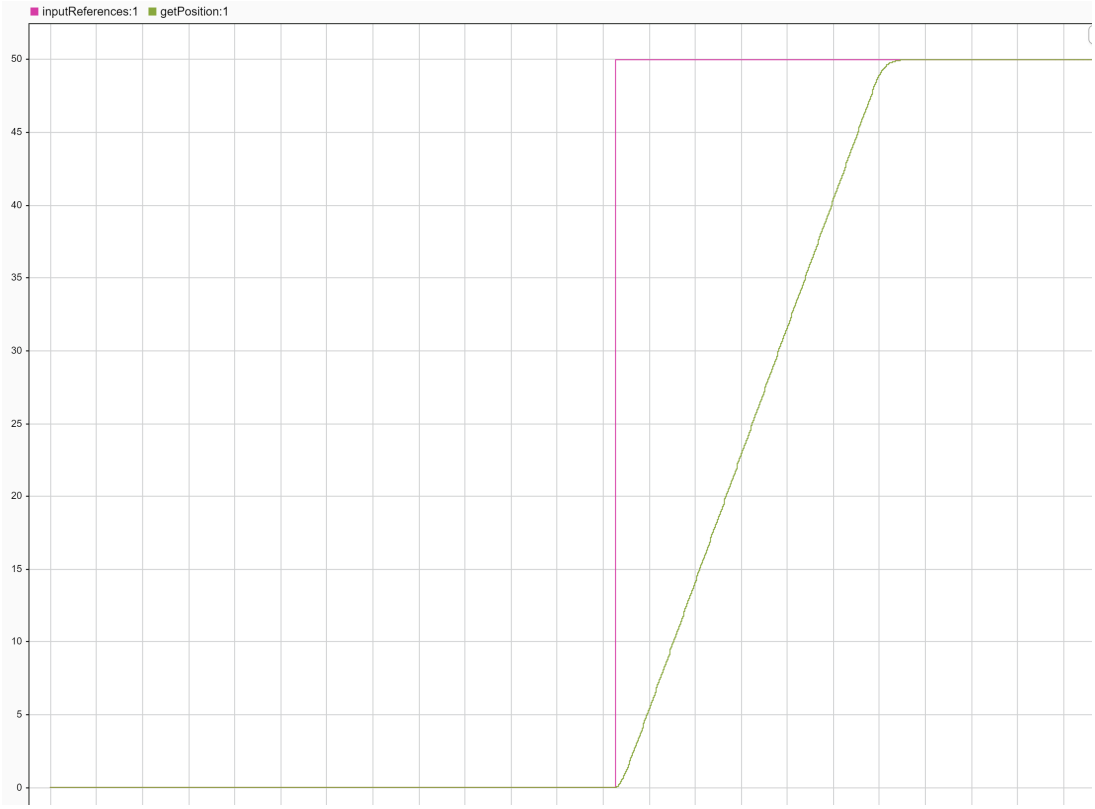

Figure 2.26: Resolved windup in position control motor

Fig. 2.26 illustrates how, replicating the scenario described above, the use of the incremental algorithm ensures that once the reference is reached, the motor follows it without overshooting as expected. The difference is so evident that it is trivial to underline the improvement achieved.

2.12.3 Quantization of the Integral Term

In general, another issue that may arise in the implementation of a digital controller is quantization, which is closely related to a crucial element of digital control: the sampling time. The effects of this problem are mainly twofold:

- **Approximation of Control Law Coefficients**: a type of error related to the precision with which numerical values can be represented in the digital system.
- **Error Drift Phenomenon**: error drift is closely related to the presence of the integral term.

Let's focus on the latter as the reason for its occurrence is somewhat subtle. Unlike continuous time, in discrete time a value equal to 0 might not truly be zero but rather very small. This means that the output actually does not reach the desired value precisely but arrives within a neighborhood of it. However, the real issue is that the sampling time can significantly impact this phenomenon: the lower the sampling time T_s, the wider the range of values that map to the "digital zero." The influence of T_s on quantization is also quantitatively motivated by [1]. This could even lead to sustained oscillations. Quantization, like Ch. 2.12.4 described dead zone of the motor, is a nonlinear element, and it can be shown that certain

nonlinear elements, including the aforementioned one, trigger limit cycles. However, unlike the motor's dead zone, this is not a mechanical phenomenon but rather a computational one.

Since the chosen sampling period T_s was significantly lower than suggested by the heuristic for the desired settling time, it raised the question of whether our control algorithm was affected by the problem described in this subsection.

The steps taken to verify this were as follows:

- **Output Analysis**: The system output was inspected to see if there were persistent oscillations that did not diminish over time. Quantization can introduce noise and oscillations that were not present in the analog system.
- **Comparison between Simulated and Real System Output**: A simulation of the control system was performed to obtain an ideal reference. The results of this simulation were then compared with those obtained from the real system.

Since no significant differences were found and there were no typical characteristics of quantization errors (e.g., discrete steps, oscillations at specific frequencies), it was concluded that our algorithm was not affected by the problem.

2.12.4 Dead Zone of the Motor

When controlling the position of a DC motor, the presence of a dead zone can significantly impact the performance of the control system. The dead zone is a region around the zero point of the control input where the motor does not respond. This nonlinear behavior can lead to the formation of a limit cycle, especially when the position error becomes small. Let's explore why and how this happens in detail.

Key points of our analysis revolve around three concepts:

- **Dead Zone**: The dead zone is a common feature in DC motors, caused by factors such as static friction and other mechanical or electrical imperfections. In this zone, if the control signal (e.g., the voltage applied to the motor) is too low, the motor does not move at all.
- **Effect on System Response**: When the position error is large, the control signal is sufficiently large to overcome the dead zone, and the motor responds correctly. However, when the error becomes small, the control signal may fall back into the dead zone, causing the motor to not respond.
- **Limit Cycle**: A limit cycle is a stable periodic oscillation that can occur in a nonlinear system. More formally, it is a closed trajectory in the phase plane that has the property that at least one other trajectory approaches it in a spiral manner as time tends to infinity or negative infinity.

In the case of a DC motor with a dead zone, the limit cycle can emerge as follows:

1. **Near the Reference Point**: As the system approaches the reference position, the error becomes small. If the error is so small that the control signal falls within the dead zone, the motor does not move. This leads to an accumulation of error.

2. **Error Accumulation**: Since the motor does not move, the error increases. Once the error exceeds a certain threshold, the control signal becomes large enough to exit the dead zone, causing the motor to move.

3. **Reaction and Overcorrection**: This movement may overcorrect the position, causing a reversal of the position error (i.e., crossing over the reference point).

4. **Cycle Repeats**: The cycle repeats as the error changes sign and decreases again, bringing the control signal back into the dead zone, and so on. This sequence of overcorrections and inactions leads to a stable oscillation around the reference point: the limit cycle.

Therefore, the limit cycle is a direct result of the nonlinearity introduced by the dead zone, making it difficult to accurately reach and maintain the desired reference position. One possible solution would have been to use prediction and limit cycle suppression techniques leveraging Particle Swarm Optimization as suggested in [2]. However, due to time constraints, this approach was not pursued.

2.12.5 Error Processing and Oscillations Around Zero

In position control, two particularly relevant problems need to be addressed:

- **Error Processing**
- **Oscillations Around Zero**

The motor position, as well as the system output, is measured in radians. In this specific case, it is considered periodic, meaning that at 2π the position restarts from 0. Due to the periodicity of the motor position, there is a challenge in processing the error to determine the smallest rotational angle at which the motor would reach the desired position.

Another issue encountered when testing the control algorithm on real hardware is that providing a position reference of either 0 or multiples of 2π sometimes led to loss of control of the system. This problem is actually related to the solution identified for the first problem: by mapping the error to a range between 0 and 2π, if the position measurement oscillates around 0 by $+\epsilon$ or $-\epsilon$, the error will be respectively ϵ and $2\pi - \epsilon$, potentially causing oscillations that destabilize the system. However, this problem would not have existed if the system under consideration had limit switches for obvious reasons.

To address the described problems, the ‘getRef‘ function was implemented. Although in the second case, the optimal solution would probably have been the use of quaternions, this will be discussed in Ch. 5.

The ‘getRef‘ function is designed to find the closest angular reference to the current motor position, determining whether it is more convenient to use the input reference or its dual. This is particularly useful in systems where the output must always be maintained within the range $[0, 2\pi]$, even if the input reference is not initially within this range. The business logic of the function is entirely centered around ensuring that it is interested only in the final position of the motor and how to reach it with the minimum rotation possible.

To simplify understanding, the key steps of the reference mapping process are illustrated below:

- **Normalization of Reference**: The assigned reference angle (‘$real_ref$‘) is normalized to the range $[0, 2\pi]$. This means that any reference angle, regardless of its initial value, is converted to its equivalent within a full cycle of 2π. This is useful in systems where the output must always be between 0 and 2π, regardless of the initial reference angle.
- **Normalization of Motor Position**: The current motor position (‘y‘) is also normalized to the range $[0, 2\pi]$ to ensure consistency in comparison.
- **Calculation of Comparison Values**: Three values are defined for comparison:
  - The normalized reference angle (‘a‘).
  - The normalized current motor position (‘b‘).
  - The dual of the normalized reference angle (‘c‘), calculated as $-(2\pi - a)$ if ‘a‘ is positive, or $2\pi + a$ if ‘a‘ is negative.
- **Ensuring Positivity**: All comparison values are brought into a positive range by adding multiples of 2π if necessary. This allows comparing distances without worrying about negative values.
- **Calculation of Distances**: The absolute differences between the comparison values are calculated to determine which is closest: - The distance between the normalized reference angle (‘a‘) and the current motor position (‘b‘). - The distance between the dual of the reference angle (‘c‘) and the current motor position (‘b‘).
- **Determination of Correct Reference**: The function determines which reference, between the original (‘a‘) and its dual (‘c‘), is closer to the current motor position. If the distance between the current motor position and the original reference angle is greater than the distance between the current motor position and the dual of the reference, the dual is chosen. Otherwise, the original reference is chosen.
- **Returning the Reference**: Finally, the function returns the correct reference, which can be either the original or the dual, depending on which is closer to the current motor position.

The operation of the implemented solution may not be immediately clear at first glance, which is why it will provide numerical examples.

In Fig. 2.27, the motor position is seen to be $-\frac{\pi}{2}$ at one point. At a later time, the reference changes to π. Since the dual of π is $-\pi$, and the latter is closer to the current motor position, ‘getRef()‘ will provide $-\pi$ as the reference.

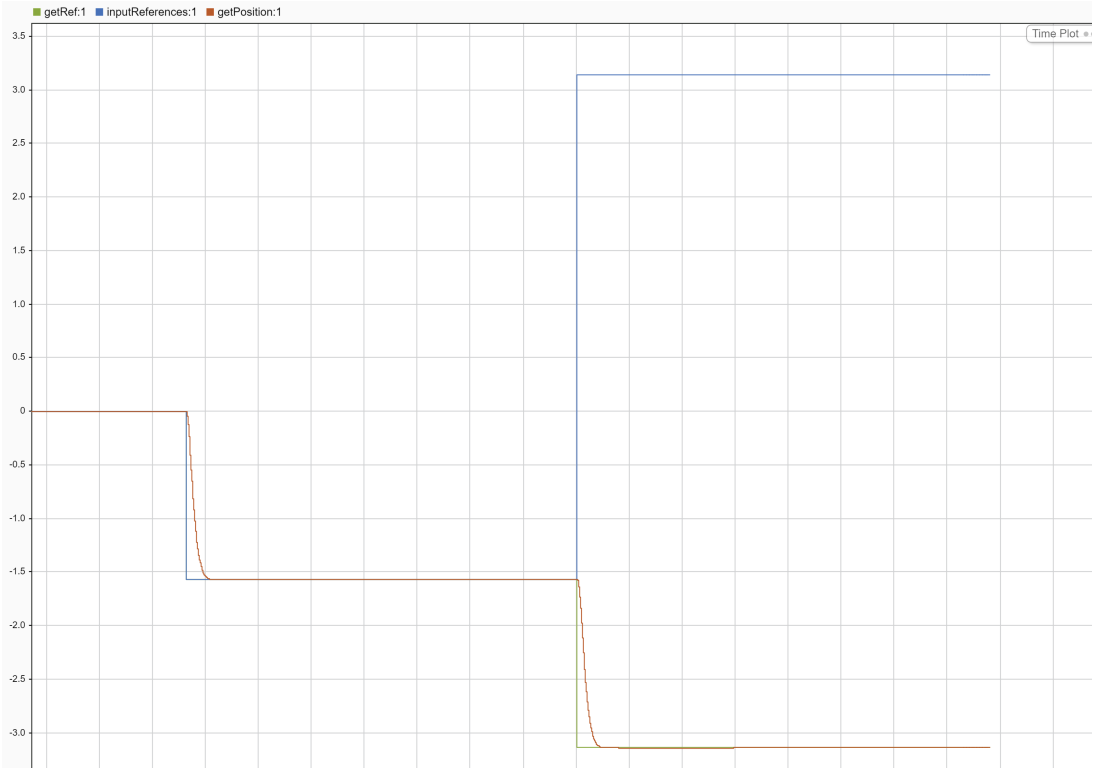

Figure 2.27: Position control cyclic with π reference

In Fig. 2.28, the motor position is π at a certain instant. At one point, the reference changes to $-\frac{\pi}{2}$. Since the dual of $-\frac{\pi}{2}$ is $\frac{3\pi}{2}$, and the latter is closer to the current motor position, 'getRef()' will provide $\frac{3\pi}{2}$ as the reference.

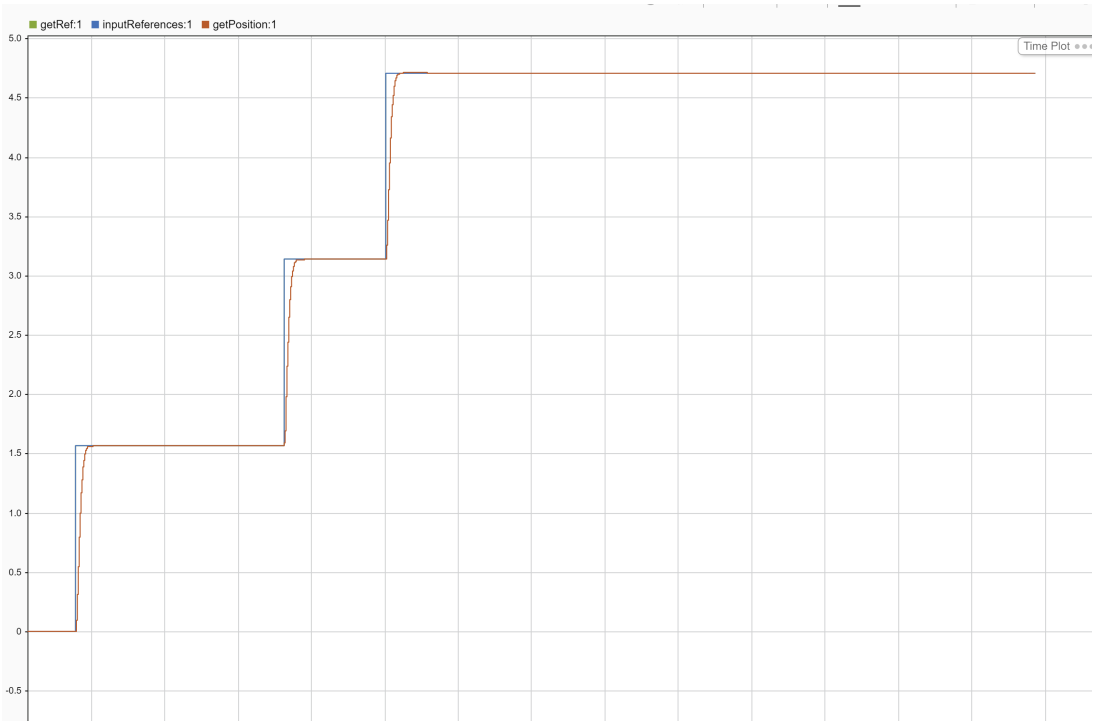

Figure 2.28: Position control cyclic with $\frac{\pi}{2}$ reference

In Fig. 2.29, the current motor position is 0, and the given reference is $\frac{3\pi}{2}$. Since the dual of $\frac{3\pi}{2}$ is $-\frac{\pi}{2}$, and the latter is closer to the current motor position, 'getRef()' will provide $-\frac{\pi}{2}$ as the reference.

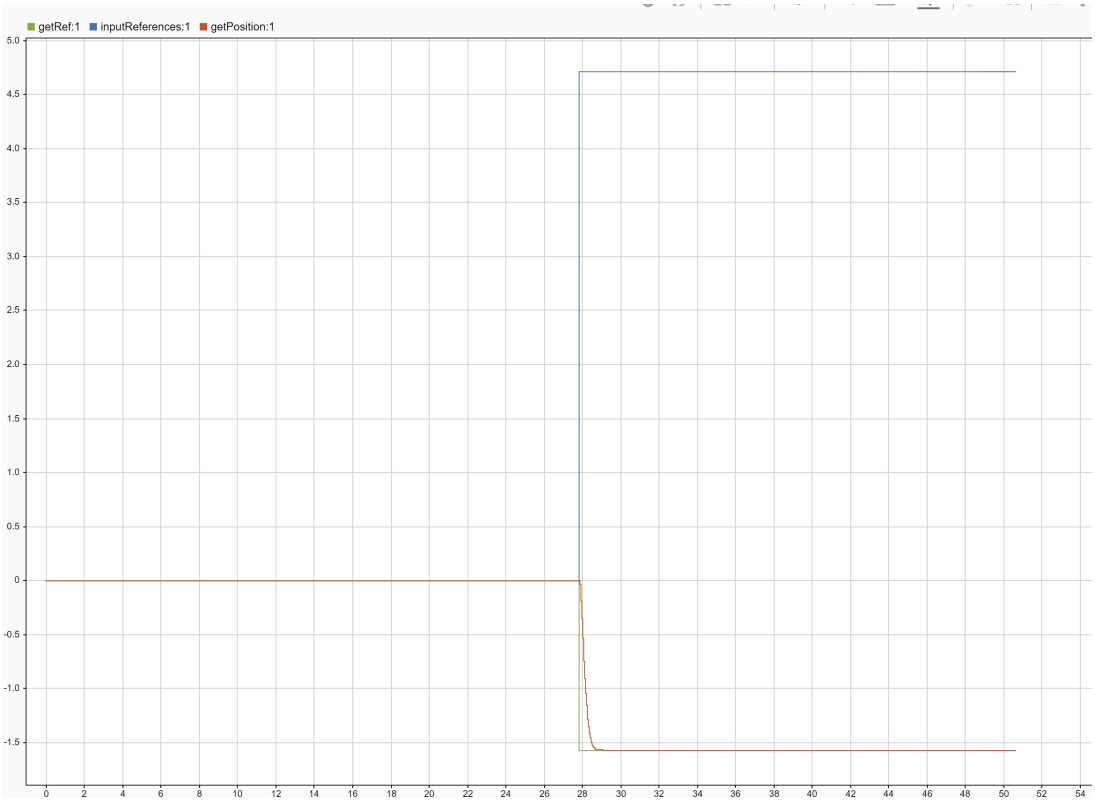

Figure 2.29: Position control cyclic with $\frac{3\pi}{2}$ reference

In Fig. 2.30, the current motor position is π, and the given reference is 10π. Since going to 10π is equivalent to going to 2π, and the latter is closer to the current motor position, 'getRef()' will provide 0 as the reference.

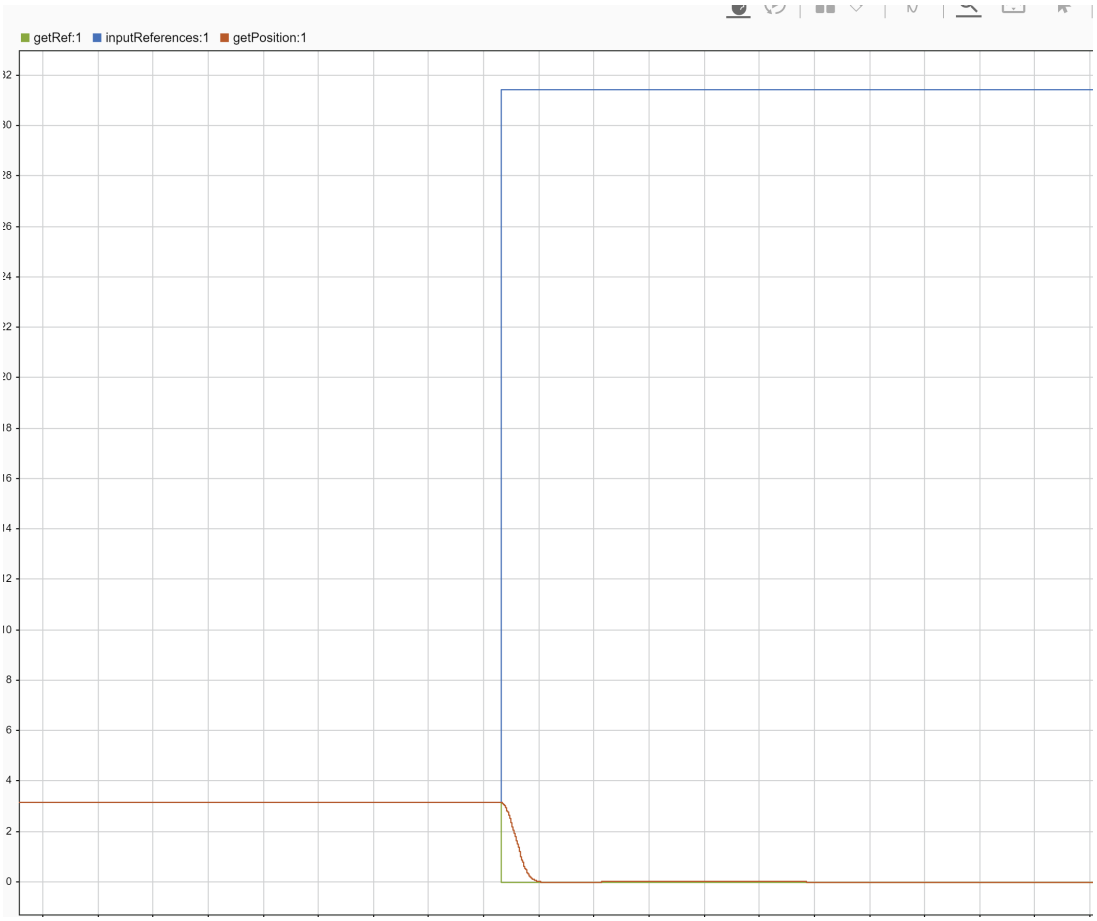

Figure 2.30: Position control cyclic with 10π reference

All that described is implemented by algorithm `getRef`:

```
function references = getRef(y, real_ref)
    real_ref_mod = rem(real_ref, 2*pi);
    y_mod = rem(y, 2*pi);

    b = y_mod;
    a = real_ref_mod;

    if a > 0
        c = - (2*pi - real_ref_mod);
    else
        c = 2*pi + real_ref_mod;
    end

    % Ensure a, b, and c are positive
    % Determine the smallest value and how many times it need to add 2*pi
    min_value = min([a, b, c]);
    num_additions = ceil(abs(min_value) / (2*pi));
```

```
    a = a + num_additions * 2*pi;
    b = b + num_additions * 2*pi;
    c = c + num_additions * 2*pi;

    if a > b
        ab = abs(a - b);
    else
        ab = abs(b - a);
    end

    if c > b
        bc = abs(c - b);
    else
        bc = abs(b - c);
    end

    if ab > bc
        if real_ref_mod > 0
            sign_part = - (2*pi - real_ref_mod);
        elseif real_ref_mod < 0
            sign_part = 2*pi + real_ref_mod;
        else
            sign_part = real_ref_mod;
        end
    else
        sign_part = real_ref_mod;
    end

    references = sign_part;
end
```

In reality, in some cases, it may not be of interest to always map the reference within the range $[0, 2\pi]$ or any other bounded range, but it may be more relevant for the application that the position reference is used for a different purpose: consider the case where it is important for the motor to arrive around position zero, but it does so by completing a certain number of full rotations. As illustrated earlier, if the motor was already at 0 rad, it would not move, whereas in this other case, giving, for example, a reference equal to 4π, it would be desirable for the motor to complete two rotations and then stop.

In light of these two different use cases, two different Simulink schemes have been implemented, depending on whether the reference is to be treated in a "periodic" manner or if there is a specific interest in jointly controlling the final position and the desired number of rotations. In particular, in the latter case, there is no complex error processing as in the case of `getRef`, but it is simply calculated as $e = r - y$. Also, for this mode, the system's output, namely the motor position, is considered without periodicity (as it is).

2.13 Hardware Deployment

Once all the issues discussed in Chapter 2.12 were addressed and discussed, the implementation of various Simulink schemes for hardware deployment commenced. It is highlighted that to enable this, blocks previously discussed and validated in the Model-in-the-Loop (MIL) phase (Chapter 2.9) and Software-in-the-Loop (SIL) phase (Chapter 2.10) were utilized. Additionally, the transfer function modeling the motor was replaced with blocks interfacing with the actual motor.

Specifically, Fig. 2.31 shows the scheme for deploying modes where the motor output remains within the range $[0, 2\pi]$. Note the presence of the 'getRef' function discussed in Chapter 2.12.5.

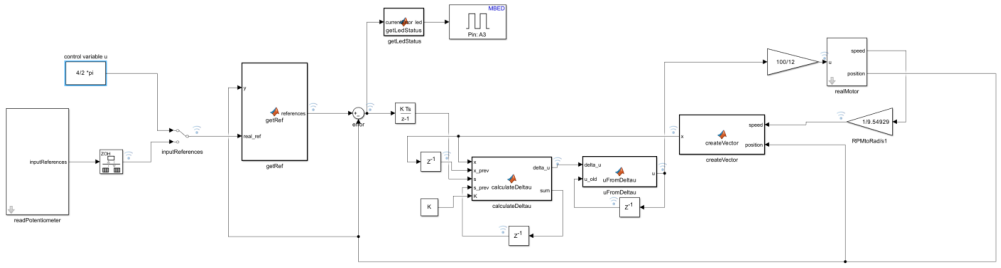

Figure 2.31: Position control scheme for periodic input

Furthermore, Fig. 2.32 displays results of the described operating mode.

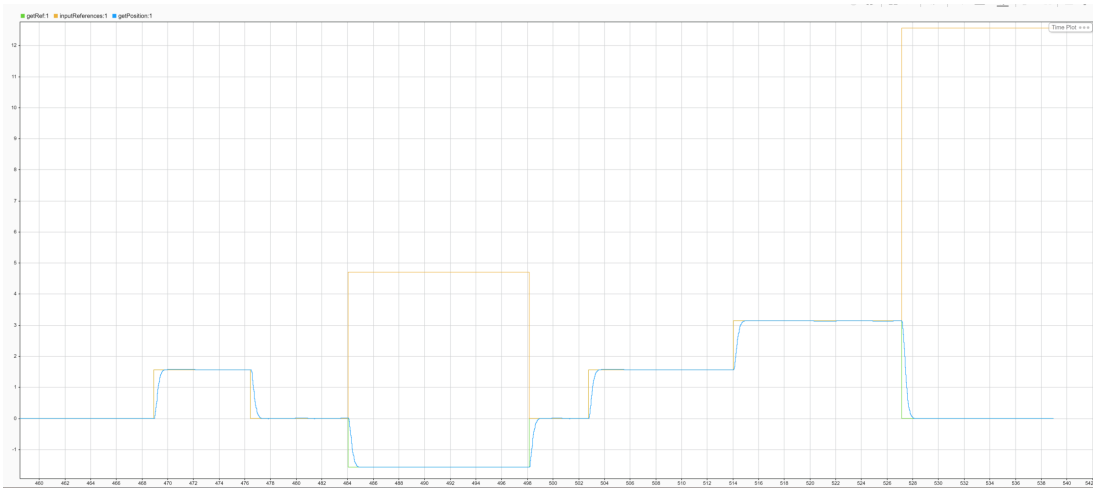

Figure 2.32: Output of cyclic position control

In Fig. 2.33, the operation scheme is shown for the mode where the motor output is not limited to $[0, 2\pi]$. Note the absence of the 'getRef' function in this case, and unlike the previous operating mode, the blocks interfacing with the real motor are configured to allow positions outside the aforementioned range.

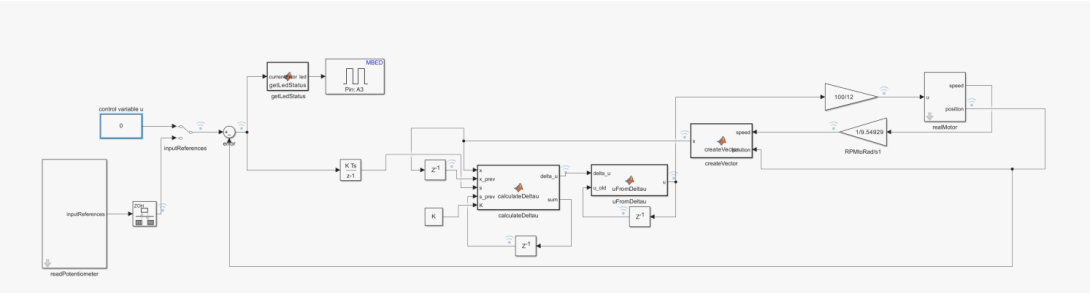

Figure 2.33: Scheme for real motor position control

Additionally, Fig. 2.34 displays results of the described operating mode.

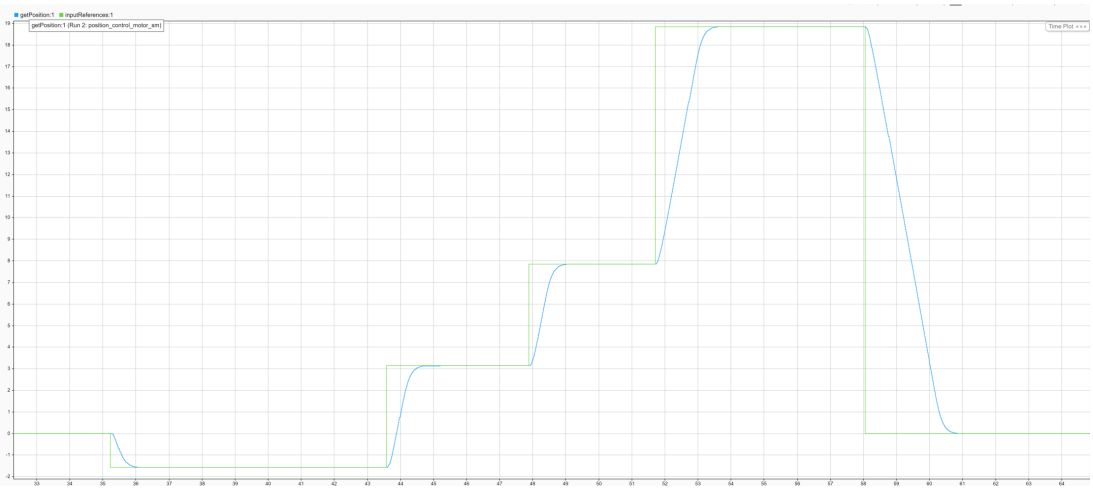

Figure 2.34: Output of position control

Both hardware deployment schemes described above include a ‘readPotenziometer‘ block used to read a potentiometer and utilize this reading as a position reference.

In Fig. 3.25, the internal structure of this block is shown. Note that a ‘firstOrderLagFilter‘ was used for reading to obtain a smoother signal. The implementation details of these blocks for potentiometer reading are similar to those used for current sensor reading, detailed in Chapter 3.5. In this case, the potentiometer range is set to $[0, 2\pi]$.

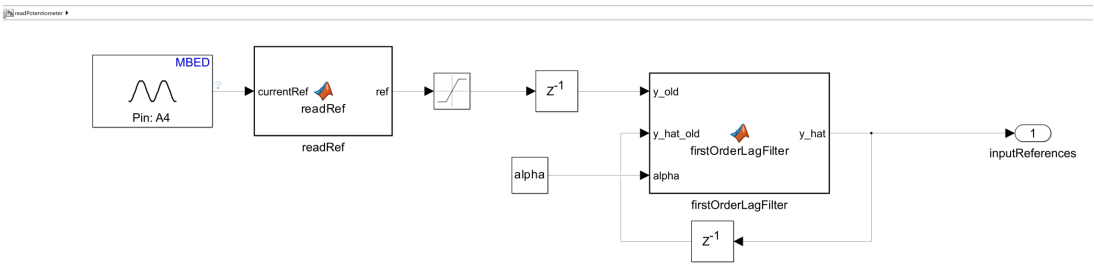

Figure 2.35: Read potentiometer

Furthermore, Fig. 2.36 shows an example of system operation when the reference is taken from the potentiometer. Additionally, a demo video has been created for this operating mode: Link to the Video Demo with potenziometer as reference.

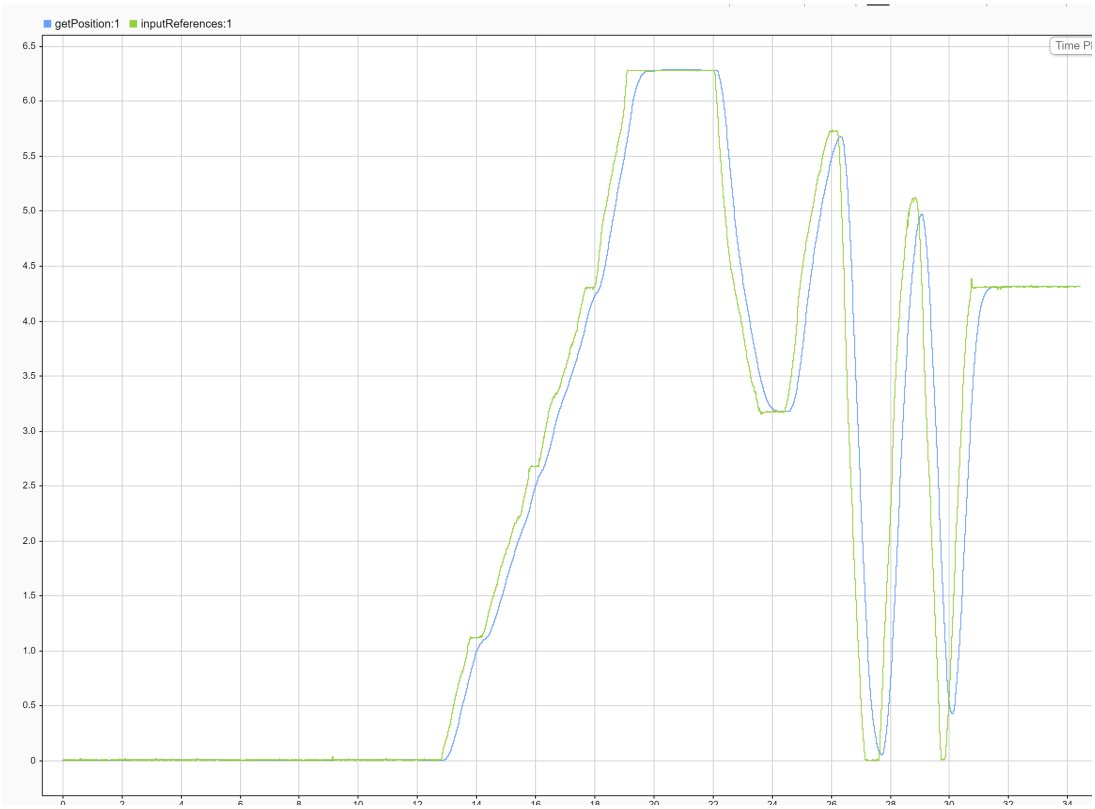

Figure 2.36: Reference by potentiometer position control

It is acknowledged that oscillations due to the potentiometer reading, although mitigated by the use of the autoregressive moving average filter, are not ideal for providing a reference since it should ideally be slowly variable. However, as previously emphasized, this choice was made to test the algorithm under non-ideal conditions, and based on what can be observed in Fig. 2.36, it can reasonably be stated that the control system offers satisfactory performance.

CHAPTER 3

TORQUE CONTROL

As for torque control, it refer equivalently to torque and current because these two measurements are equal up to a multiplicative constant K_t.

3.1 Torque Model

3.1.1 Torque control scheme

In the Laplace domain s, the electrical equilibrium of the armature circuit is described by the equations:

$$V_a = (R_a + sL_a)I_a + V_g \tag{3.1}$$

$$V_g = k_v \Omega_m \tag{3.2}$$

where V_a and I_a represent the voltage and current applied to the armature circuit with resistance R_a and inductance L_a, and V_g represents the back electromotive force proportional to the rotational speed Ω_m through the voltage constant k_v, which depends on the machine's construction characteristics and excitation flux.

The mechanical equilibrium is described by the equations:

$$C_m = (sI_m + F_m)\Omega_m + C_l \tag{3.3}$$

$$C_m = k_t I_a \tag{3.4}$$

where C_m and C_l represent the electromechanical torque and the load torque, I_m and F_m are the moment of inertia and viscous friction coefficient of the motor, and the torque constant k_t, when expressed in the MKS system, typically assumes the same numerical value as k_v.

Regarding the power amplifier, it is associated with an input-output relationship linking the control voltage V_c and the armature voltage V_a through the transfer function:

$$\frac{V_a}{V_c} = \frac{G_v}{1 + sT_v} \tag{3.5}$$

where G_v is the voltage gain and T_v is a time constant that can be considered negligible compared to other time constants in the control system. Hence, the power amplifier is considered a pure gain. Indeed, with modulation frequencies in the range of $(10, 100)$ kHz, the time constant of the amplifier falls within the range $(10^{-5}, 10^{-4})$ seconds.

Starting from the mathematical models described, it is possible to derive the block diagram in Fig. 3.1 for the torque control of the DC motor.

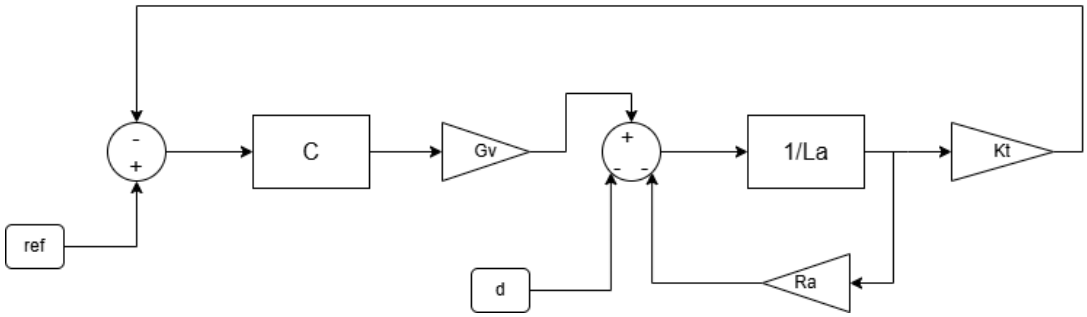

Figure 3.1: Block diagram for torque control

By not knowing I_m and F_m, the mechanical part of the motor dynamics is considered as additive disturbance d.

3.1.2 Parameters Identification

During the realization of this project, it delved into the identification or improvement of DC motor parameters, within the limits at hand.

Various techniques were explored to measure the inductance of a DC motor, but unfortunately, it was unable to perform the measurement due to the lack of necessary instruments and precautions to avoid motor damage. Here's a summary of the considered techniques:

1. **Direct Measurement with LCR Meter**: Uses an LCR meter to directly measure the motor's inductance. Lacking an LCR meter, it was unable to perform this measurement.

2. **Frequency Response Method**: Requires a signal generator and an oscilloscope to apply a sinusoidal signal and measure voltage and current at the motor terminals. Without an oscilloscope, it could not use this technique.

3. **Current Pulse Method**: Uses a DC power supply, a known external resistance, and an oscilloscope to apply a voltage pulse and measure the current response. Lack of an oscilloscope prevented us from proceeding with this method.

4. **Step Response Method**: Involves using a DC power supply, a known external resistance, and an oscilloscope to apply a voltage step and measure the current response. Once again, the absence of an oscilloscope hindered us from performing this measurement.

5. **Locked Rotor Impulse Response Method**: Requires physically locking the motor shaft, using a DC power supply, a known external resistance, and an oscilloscope. To avoid the risk of damaging the motor, it decided against using this method.

In summary, due to the lack of an LCR meter and an oscilloscope, and with the intention of avoiding motor damage, it was unable to measure the motor inductance using the described techniques.

Ultimately, for the inductance value determination, it referred to a Pololu forum where it found a thread where a Pololu engineer specifically indicated the inductance value of our motor. The thread is available at the following link: Link to the Forum Post. With this information, it was able to obtain the necessary inductance value for our motor.

Regarding the resistance R, however, the outcome was quite different.

Steps taken to measure the resistance of a DC motor with a multimeter were as follows:

1. The motor was completely powered off to avoid inaccurate readings.
2. The multimeter was set to the resistance measurement mode, denoted by the ohm symbol Ω.
3. The multimeter probes were connected to the motor terminals, ensuring good metal contact.
4. After correctly connecting the probes, the multimeter displayed a resistance value of 4.8 ohms (Ω).

The measured resistance value in this case was 4.8 Ohms (Fig. 3.2). It is important to note that the resistance of DC motors can vary based on their design and internal conditions, affecting aspects such as current and other operating characteristics of the motor.

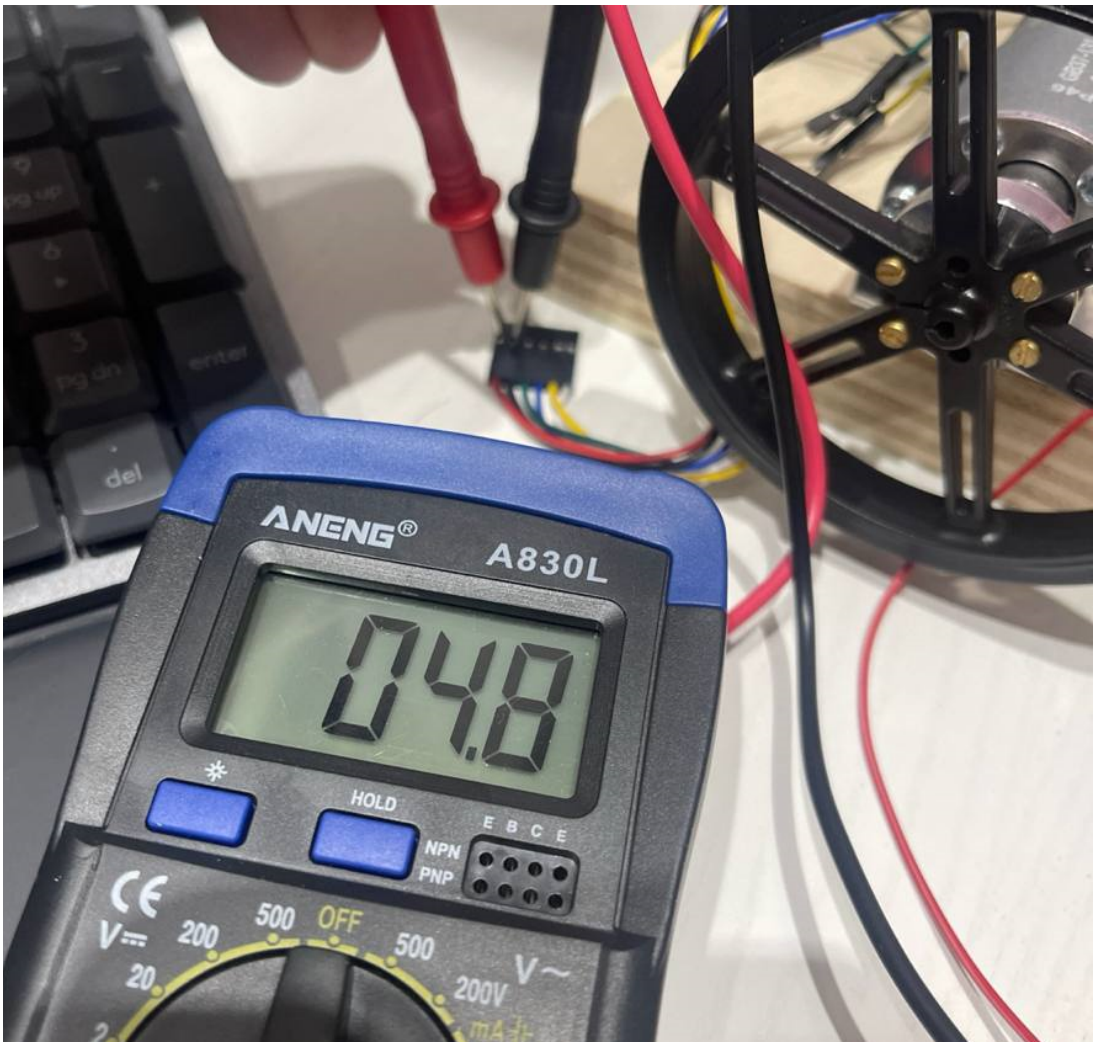

Figure 3.2: Measured resistance

3.2 Formalization of PI Control

The following is a formalization of the PI controller: since it is a standard controller and widely used, the treatment will be brief and provided solely for completeness. The transfer function of a PI controller in the Laplace domain is given by:

$$C(s) = \frac{k_I}{s}\left(1 + \frac{k_P}{k_I}s\right) \tag{3.6}$$

In addition to the integral action, there is a zero at the break frequency $\frac{k_I}{k_P} = \frac{1}{T_i}$.

On the other hand, the general form of a digital PI controller is:

$$u(k) = k_P e(k) + k_I s(k) \tag{3.7}$$

where $s(k) = s(k-1) + e(k)$ represents the integral term.

3.3 PI Controller Design

To design the PI controller for the system, it initially used MATLAB's PID Tuner as a starting point. This tool provides controller parameters based on specifications related to system

robustness and speed, which in this case were set to their maximum values. Subsequently, manual fine-tuning of the controller was performed. This was crucial to optimize the quality of the control system response.

The controller provided by the tool was:

$$C(s) = \frac{2.0811 \cdot (s + 91.93)}{s} \tag{3.8}$$

From the analysis of the Bode diagrams (Figs. 3.3 and 3.4) of the open-loop transfer function, it emerged that this controller could ensure a phase margin $m_\phi = 145°$ and an infinite gain margin m_a. The crossover frequency ω_c was approximately 275 rad/s. Observing the step response (Fig. 3.5), the following performance indices were deduced:

- Settling time: 0.07 s
- Rise time: 0.03 s
- No overshoot.

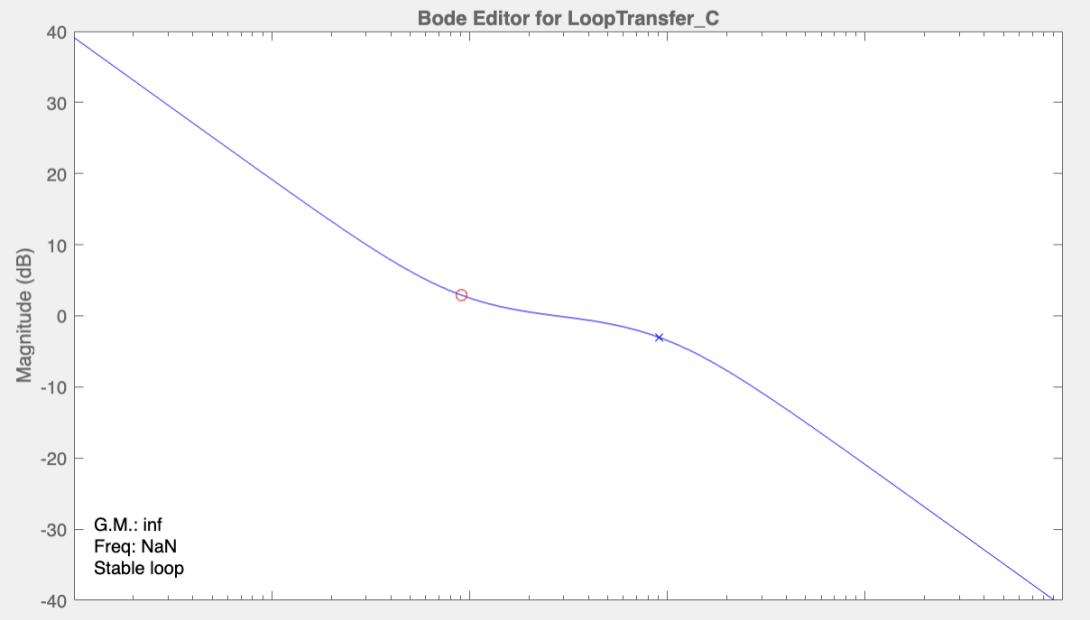

Figure 3.3: Bode module diagram of open-loop transfer function $F(s)$ relative to the PI designed with the PID tuner tool.

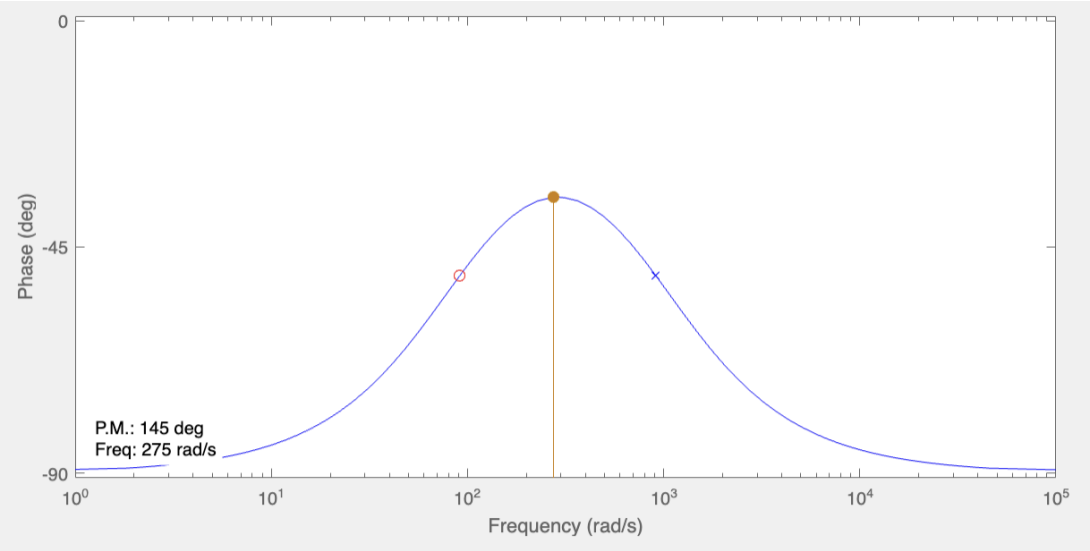

Figure 3.4: Bode phase diagram of open-loop transfer function $F(s)$ relative to the PI designed with the PID tuner tool.

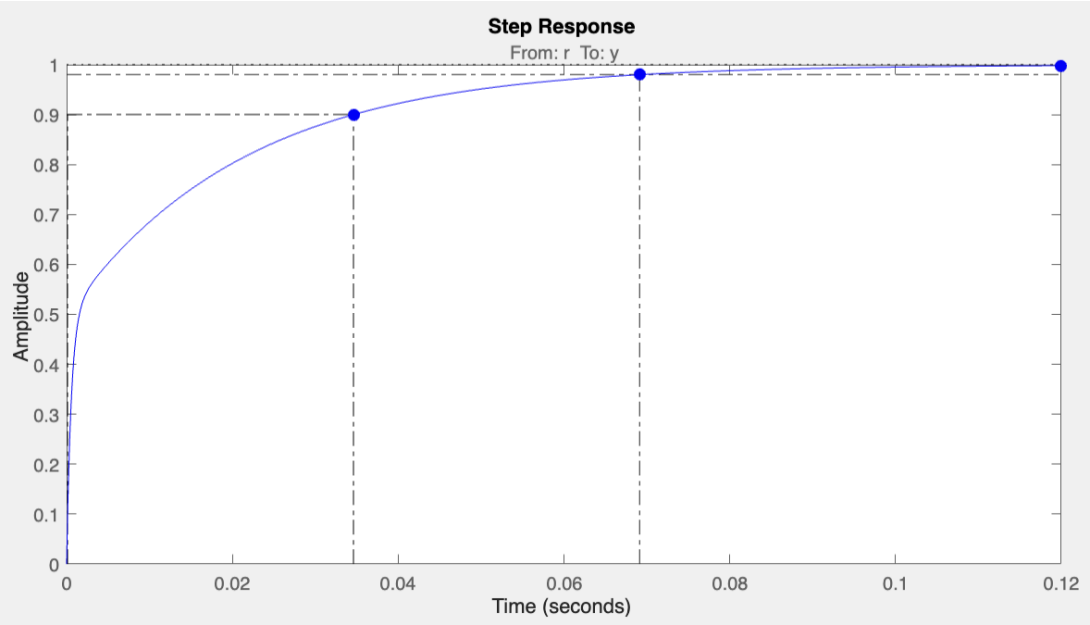

Figure 3.5: Step response relative to the PI designed with the PID tuner tool.

Given the achieved performance and robustness margins, one might ask why perform further fine-tuning of the controller since the results obtained could be considered sufficiently satisfactory. The answer, as with other challenges faced during the project, lies in the need for a digital controller, which in turn requires discretization with a specific sampling time. The choice of this sampling time is motivated in Ch. 3.4.

The discretized controller using the Tustin bilinear transformation with $T_s = 0.005$ was:

$$C(z) = \frac{2.556z - 1.606}{z - 1} \tag{3.9}$$

If it denote the degree of the numerator as m and the denominator as n, in this case, $n = m$, indicating the discrete controller is not strictly proper. Consequently, the control input at time

k depends on the error at the same instant, introducing a phase delay:

$$\text{Delay} = -\frac{3}{2}\omega T_s \rightarrow -\frac{3}{2} \cdot 275 \text{ rad/s} \cdot 0.005 = -2.0625 \text{ rad/s} = 118^\circ \tag{3.10}$$

Thus, the phase margin considering the delay becomes:

$$m_\phi = 145^\circ - 118^\circ = 27^\circ \tag{3.11}$$

It is now much simpler to understand the rationale behind modifying a controller that was initially performing so well. The new controller was obtained by appropriately adjusting the gain to find a compromise that ensured both performance and wider robustness margins. The frequency domain expression of the modified controller is:

$$C(s) = \frac{1.8 \cdot (s + 91.93)}{s} \tag{3.12}$$

The phase margin remains unchanged from the previous controller, with a gain margin of $m_\phi = 139^\circ$, as shown in Figs. 3.6 and 3.7. However, in this case, the crossover frequency is $\omega_c = 145$ rad/s. From the step response, it deduce:

- Settling time: 0.076 s
- Rise time: 0.039 s
- No overshoot.

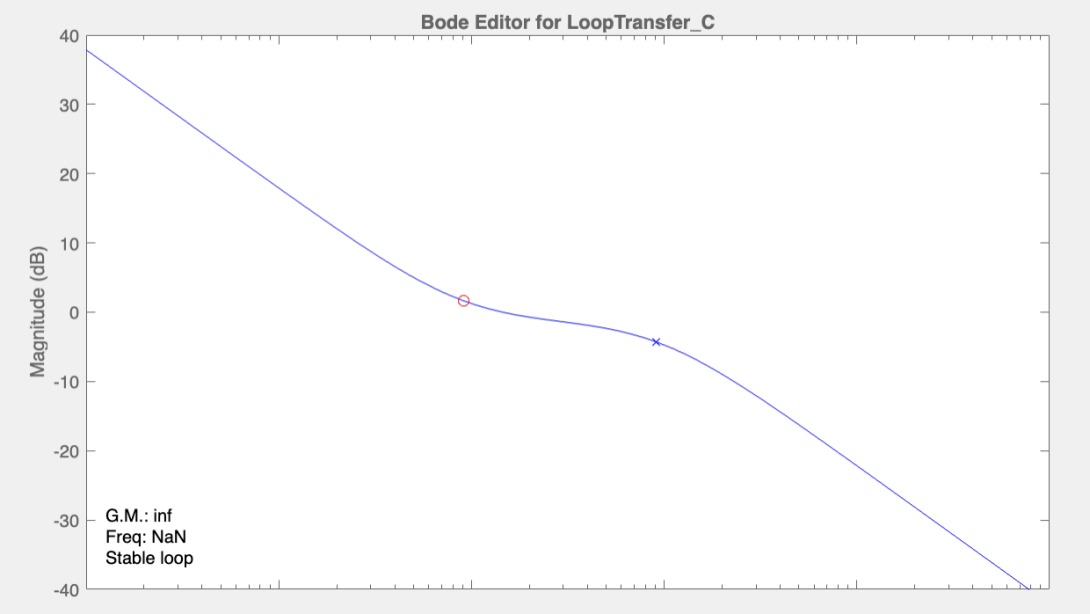

Figure 3.6: Bode module diagram of open-loop transfer function $F(s)$ relative to the modified PI controller.

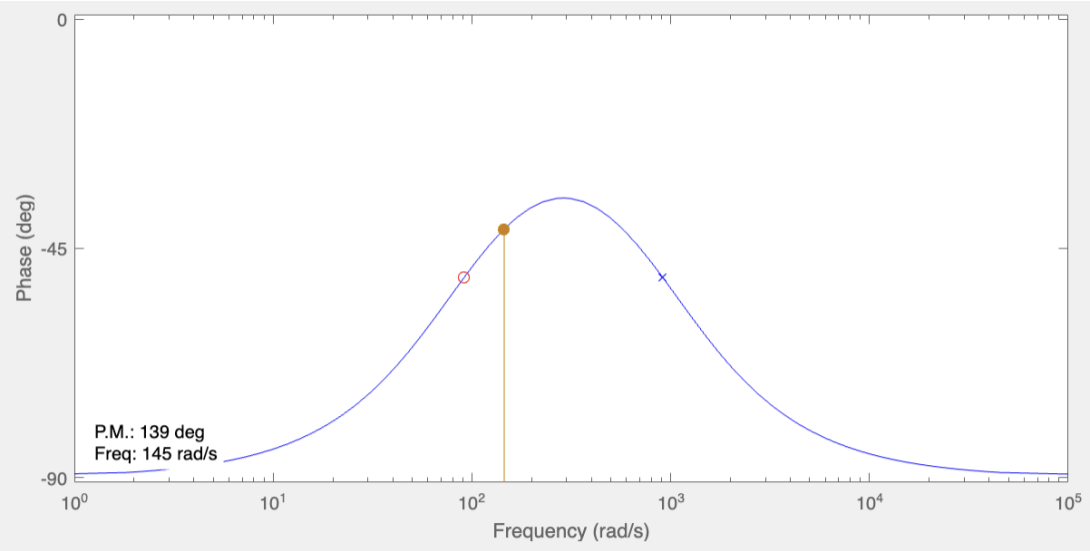

Figure 3.7: Bode phase diagram of open-loop transfer function $F(s)$ relative to the modified PI controller.

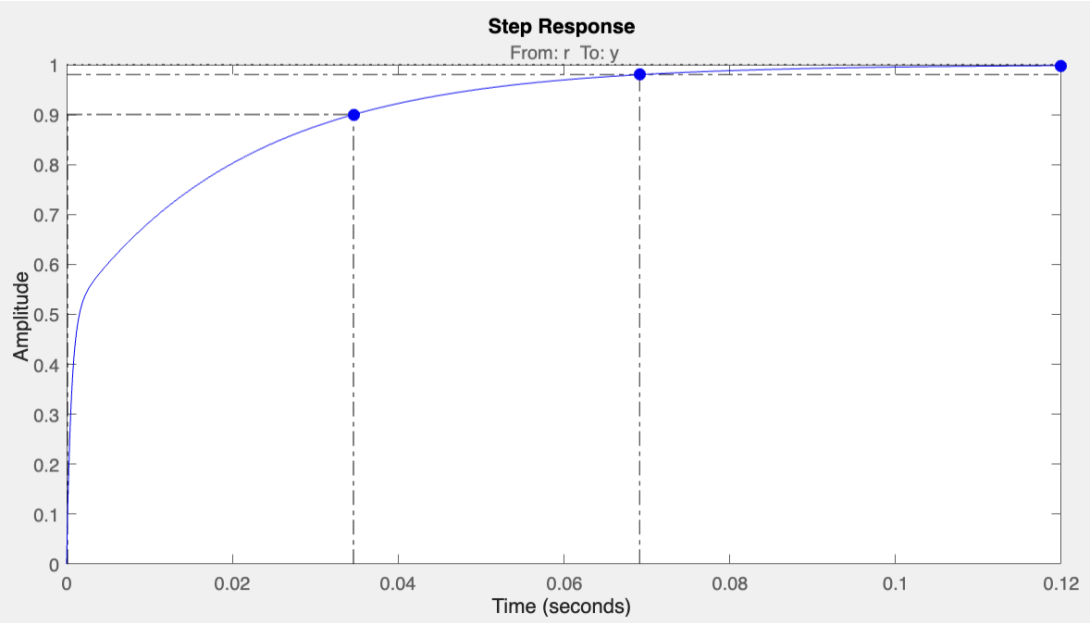

Figure 3.8: Step response relative to the modified PI.

It is not difficult to notice that the previous controller was slightly more performant compared to the selected one. However, it is equally noticeable, observing the crossover frequency, how in this case the delay introduced by discretization is significantly lower. The discrete-time controller is:

$$C(z) = \frac{2.211z - 1.389}{z - 1} \tag{3.13}$$

And although it is again not strictly proper, it note that:

$$\text{Delay} = -\frac{3}{2}\omega T_s \rightarrow -\frac{3}{2} \cdot 145 \text{ rad/s} \cdot 0.005 = -1.0875 \text{ rad/s} \approx 62^\circ \tag{3.14}$$

Consequently, considering the delay, the phase margin will be:

$$m_\phi = 139^\circ - 62^\circ = 77^\circ \tag{3.15}$$

It was considered more reasonable and conservative to use a slightly less performant controller (the differences are almost negligible) with a lower bandwidth but with a significantly larger phase margin.

3.4 Sampling Time Selection

To determine the sampling period for torque control, similar considerations were made as those for position control in Ch. 2.7. Given that the closed-loop bandwidth of the system is approximately 67 rad/s, to meet heuristics on closed-loop performance requirements, it chose:

$$\omega_s = 20 \cdot 67 \text{ rad/s} \tag{3.16}$$

Thus,

$$T_s = \frac{2\pi}{\omega_s} \approx 0.005 \text{ s} \rightarrow f_s = 200 \text{ Hz} \tag{3.17}$$

The same result could have been achieved by considering the closed-loop settling time and dividing it by 15:

$$T_s = \frac{T_{ss}}{15} \approx 0.005 \text{ s} \tag{3.18}$$

Where T_{ss} is $0.07s$ like in Fig. 3.8.

It should be noted that T_s should be strictly less than the closed-loop settling time divided by 10; therefore, to provide an additional margin, it decided to divide by 15.

3.5 Current Sensor Interfacing

The Pololu ACS714 Current Sensor Carrier is used to measure currents in the range from -5A to 5A. Here's how it operates:

1. **Power Supply**: The sensor must be powered with a 5V voltage.
2. **Zero Current Output**: When the measured current is 0A, the sensor output is 2.5V.
3. **Conversion Factor**: The sensor has a conversion factor of 0.185V/A. This means that each ampere of current causes a 0.185V change in the sensor output.

3.5.1 Description of the Reading and Filtering System in Simulink

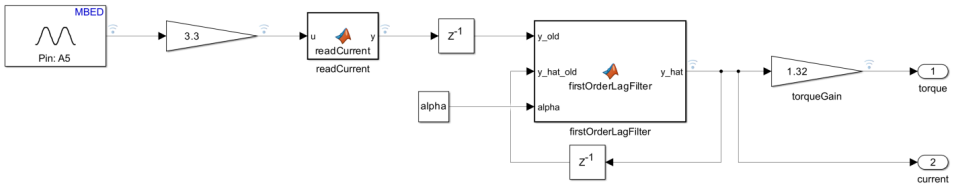

Figure 3.9: Read current scheme

In Fig.3.9 shows the signal reading and filtering system implemented in Simulink to improve the precision of the measurement used in torque control:

1. **Signal Acquisition**:

   The current signal is read from pin A5 of the STM32 Nucleo F402RE board.

2. **Amplification**:

   The acquired signal is amplified by a factor of 3.3.

3. **Signal Calibration**:

   To convert the sensor reading into a current measurement, the following MATLAB function `readCurrent` is used:

   $$y = \frac{u - 2.5 - 0.070}{0.185} \tag{3.19}$$

   Here, the offset -0.070 was added to correct for a phase lag observed at 0A current during sensor readings with a multimeter.

4. **Signal Filtering**:

   The current signal is filtered using a first-order lag filter, implemented in the function `firstOrderLagFilter`:

   $$y\_hat = \alpha \cdot y\_hat\_old + (1 - \alpha) \cdot y \tag{3.20}$$

   The parameter α is chosen as 0.9, based on the filter's sampling frequency relative to the control loop. This filter is equivalent to an autoregressive moving average (ARMA) filter when α is $[0, 1]$.

   To calculate α, the following formula is used:

   $$\alpha = e^{-\frac{T_{fs}}{T_f}} \tag{3.21}$$

   Where:

   - T_{fs} is the filter's sampling time, much smaller than the control loop sampling time.

- T_f is half the control loop sampling time.

In this case:

- $T_{fs} = \frac{0.005 \text{ s}}{20} = 0.00025 \text{ s}$
- $T_f = \frac{0.005 \text{ s}}{2} = 0.0025 \text{ s}$

Using these values:

$$\alpha = e^{-\frac{0.00025}{0.0025}} \approx 0.9 \tag{3.22}$$

This α value is then used in the first-order filter function to obtain the filtered signal.

5. **Sampling Frequency**:

   The filtering chain operates at a frequency 20 times higher than the control loop. If the control loop has a sampling time of 0.005s, the filter operates at 0.00025s. This is set in MATLAB to run the entire current sensor reading and filtering cycle. This can be seen in Fig. 3.10 where, thanks to analyze the displayed tool work.

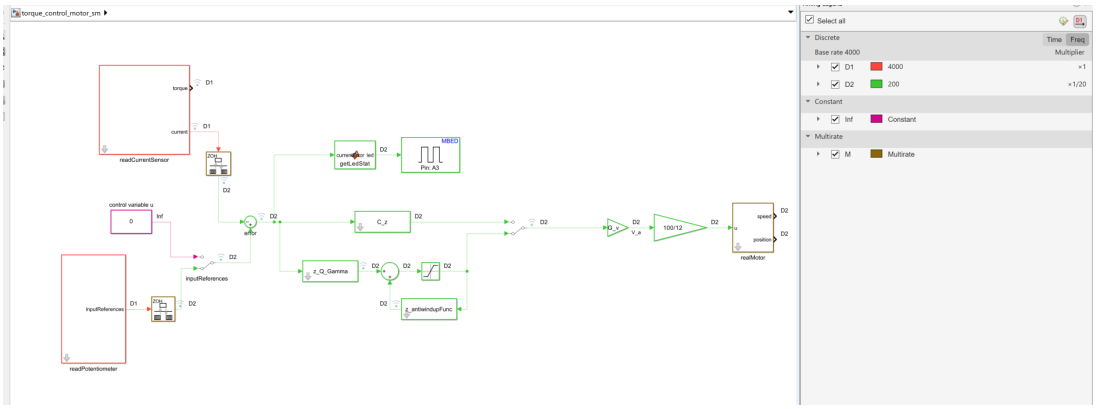

Figure 3.10: frequency torque control scheme

6. **Torque Calculation**: The filtered value is multiplied by the torque constant k_t to obtain the torque value.

7. **Frequency Transition**: To integrate the filtered value into the control loop, a rate transition block is used to adapt the frequency from 0.00025s to 0.005s. This block is highlighted in Fig. 3.11. The different coloring of the input and output arrows shows the frequency change performed by this block.

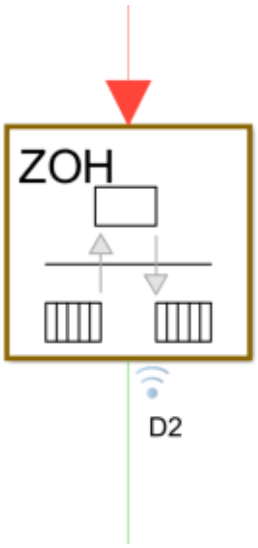

Figure 3.11: rate transition block

In Fig. 3.12, the difference between the filtered and unfiltered signals is shown. It can be observed how the amplitude of the oscillations is significantly reduced thanks to the use of the filter, which is crucial for achieving more precise control.

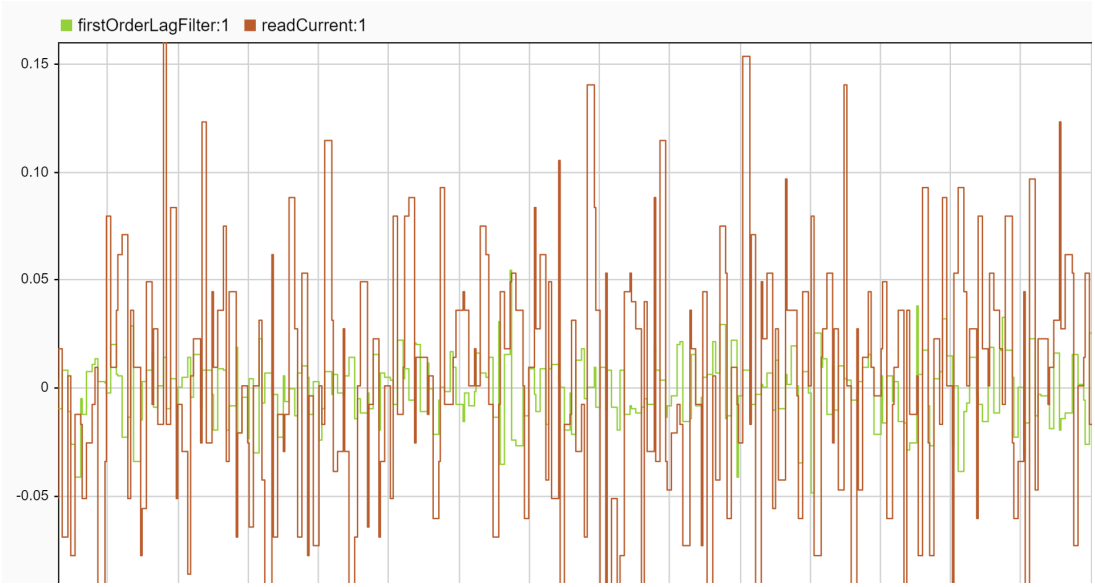

Figure 3.12: filtered vs not filtered current sensor

3.6 Digital control law implementation

In Ch. 3.3, it was previously mentioned that for the implementation of the control law, a PI controller was chosen. In this section, it will specialize the algorithm presented for our specific case, as the final control law is not simply obtained by discretizing the aforementioned controller but uses the back-calculation and tracking technique to solve the windup problem (and implicitly the bumpless transfer) discussed in Ch. 3.10. To implement this solution, it need to refer to two transfer functions in the discrete-time domain:

$$z_Q\_Gamma(z) = \frac{\text{num}(C(z))}{z} = \frac{2.211z - 1.389}{z} \tag{3.23}$$

$$z\_antiwindup(z) = \frac{z - \text{den}(C(z))}{z} = \frac{1}{z} \tag{3.24}$$

where $C(z) = \frac{2.211z-1.389}{z-1}$ is the digital PI controller designed in Section 3.3.

Consider $z_Q\_Gamma(z) = \frac{out\_z_q\_gamma(z)}{E(z)}$ where $out\_z_q\_gamma$ and $E(z)$ are respectively the output and input of the transfer function. it has:

$$\frac{out\_z_q\_gamma}{E(z)} = \frac{2.211z - 1.389}{z} \tag{3.25}$$

Multiplying both sides by z gives:

$$z \cdot out\_z_q\_gamma = E(z) \cdot (2.211z - 1.389) \tag{3.26}$$

Antitransforming yields:

$$out\_z_q\_gamma[k+1] = 2.211e[k+1] - 1.389e[k] \tag{3.27}$$

which simplifies to:

$$out\_z_q\_gamma = 2.211e[k] - 1.389e[k-1] \tag{3.28}$$

where $e[k]$ is the difference between the reference and the output at time k.

Now consider $z\_antiwindup(z)$. Following similar steps to obtain $z_q\_gamma[k]$, and considering $out\_z\_antiwindup(z)$ and $U(z)$ as the output and input of the transfer function, it has:

$$\frac{out\_z\_antiwindup(z)}{U(z)} = \frac{1}{z} \tag{3.29}$$

Multiplying by z gives:

$$z \cdot out\_z\_antiwindup(z) = U(z) \tag{3.30}$$

which simplifies to:

$$out\_z\_antiwindup[k] = u[k-1] \tag{3.31}$$

The control input for the motor is obtained as:

$$u[k] = out\_z_q\_gamma[k] + out\_z\_antiwindup[k] \tag{3.32}$$

Similar to position control, in this case, the saturation limits of $u[k]$ are within the range $[-12, 12]$.

3.7 Model-in-the-Loop (MIL)

Similar to the position control context, the first fundamental step in the validation phase is model testing.

3.7.1 Requirements

To validate the model, precise system requirements must be defined for torque control:

1. **Steady-State Error**: The system must ensure zero steady-state error with respect to the reference.
2. **Settling Time**: The desired settling time value is approximately 0.5 seconds.
3. **Rise Time**: The rise time is not as critical as settling time but ideally should be around 75%-80% of the settling time.
4. **Overshoot**: The system must not have a percentage overshoot greater than 3%-4%.
5. **Disturbance Rejection**: The system should be able to reject step disturbances.
6. **Robustness**: The system must have ample robustness margins to compensate for model uncertainties. Given the fast current process, the delay introduced by discretization could be particularly significant. A phase margin greater than 90° is required.

3.7.2 Implemented Model

The control system model was developed using Matlab/Simulink Fig. 3.13. The system includes the following main components:

1. **Process Transfer Function**: This is the transfer function relating the control input (voltage) to the output (motor torque).
2. **PI Controller**: The standard proportional-integral (PI) controller implemented for torque control.
3. **Torque Reference**: A reference signal representing the desired torque output by the motor.
4. **Disturbance**: Used to account for uncertainties in the mechanical and electrical parts of the model.

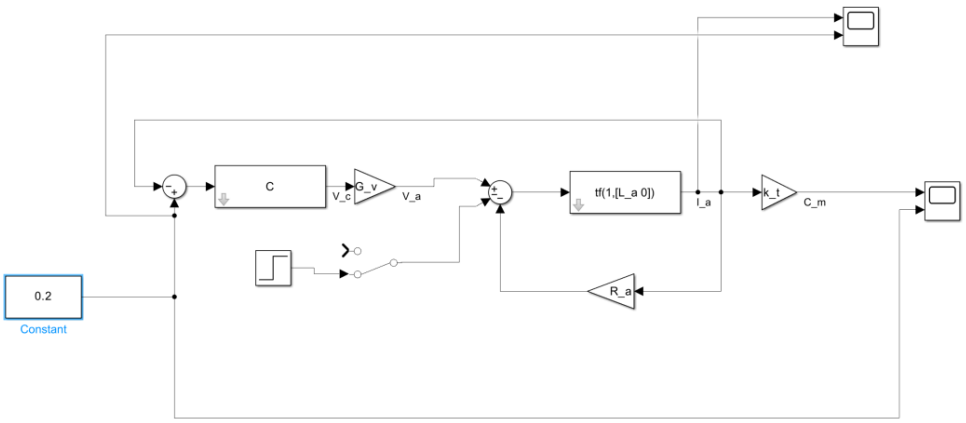

Figure 3.13: Torque Control scheme MIL

3.7.3 Validation

Following the model implementation, validation is conducted to ensure compliance with the specified requirements. As the settling time, rise time, robustness margins, and percentage overshoot have already been discussed in the Ch. 3.3, this section focuses on the system's ability to reject disturbances.

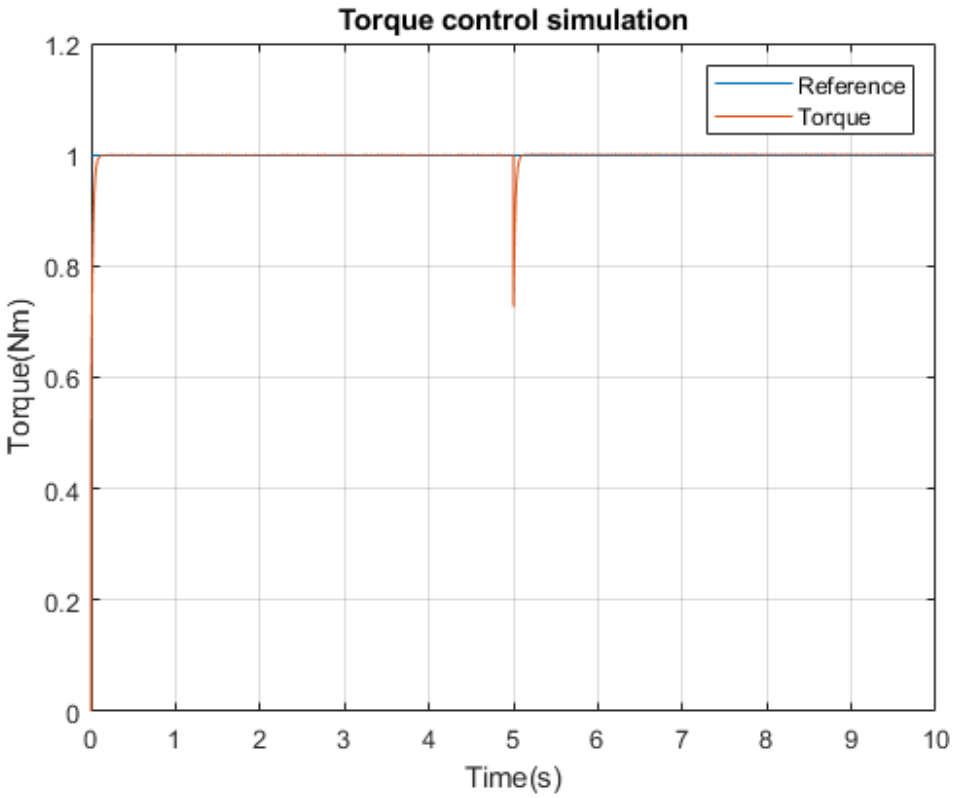

Figure 3.14: Torque Control MIL Output

From the observed results in Figs. 3.14, 3.6, 3.7, and 3.8, it is evident that:

1. The steady-state error is zero due to the presence of integral action.
2. The settling time is approximately 0.07 seconds.
3. The rise time is 0.03 seconds.
4. There is no overshoot.
5. The system effectively rejects step disturbances. In the Fig.3.14, the reference and disturbance amplitudes were both set to 1 at time $t = 5s$.
6. The phase margin is 139° and the gain margin is infinite.

Based on the analysis conducted, it can be affirmed that the implemented model is largely in line with the specified requirements and is therefore considered valid.

3.8 Software-in-the-Loop (SIL)

3.8.1 Model and Simulation

The control system model was developed using Matlab/Simulink (see Figs. 3.15 and 3.16), and the controller was implemented in C. The system included the following main components:

1. **DC Motor Model**: Included dynamic equations describing the motor behavior in terms of torque.
2. **PI Controller**: A PI controller for motor torque control.
3. **Torque Reference**: A reference signal representing the desired motor torque.
4. **Disturbance**: Used to account for uncertainties in the mechanical and electrical parts of the model.
5. **Generated and Hand-Written Code**: Matlab/Simulink generated code was combined with manually written code to address specific issues not manageable through predefined blocks.

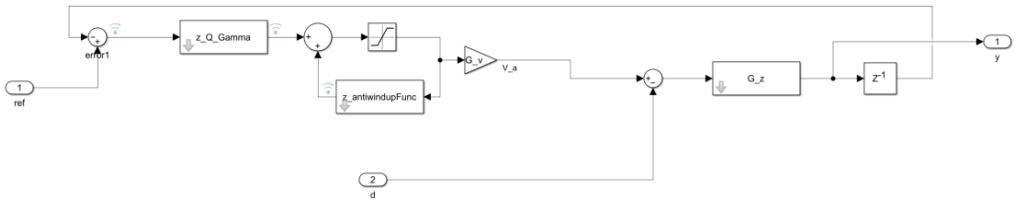

Figure 3.15: Torque Control scheme Z

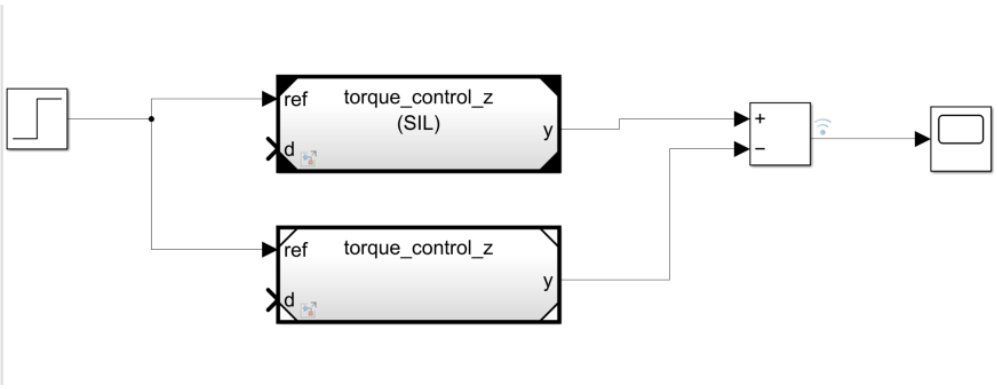

Figure 3.16: Torque Control scheme SIL

3.8.2 Simulation Results

During the Software-in-the-Loop (SIL) phase, the system underwent various test scenarios to validate requirements and performance. The obtained results (Fig. 3.17) were as follows:

1. Verification of requirements satisfied during the Model-in-the-Loop (MIL) phase.

2. **Code Integration**: The integration between hand-written and auto-generated code was successful, with all functionalities operating as expected.

Figure 3.17: Output of Torque Control SIL

3.9 Processor-in-the-Loop (PIL)

3.9.1 Model and Simulation

The model in this case included the same components as the SIL phase (Fig. 3.18), except that in this case, the target code is created and simulated for our STM32F401RE board.

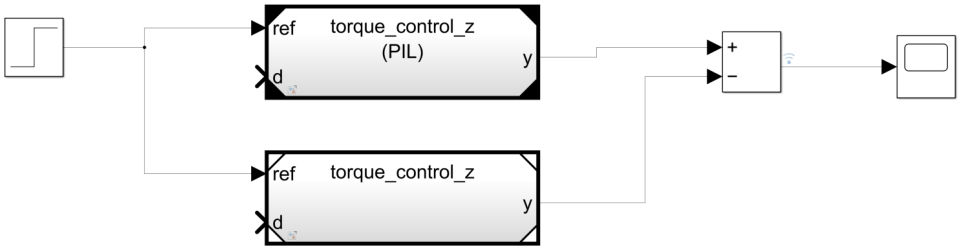

Figure 3.18: Torque Control scheme PIL

3.9.2 Simulation Results

During the Processor-in-the-Loop (PIL) phase, the system underwent various test scenarios to validate requirements and performance. The obtained results (Fig. 3.19) were as follows:

1. Verification of requirements satisfied during the SIL phase.

2. **Execution Time**: The total algorithm execution time was found to be 0.000021 seconds (Fig. 3.20), corresponding to 0.43% of the 0.005-second sampling time. This result is significantly below the 10% sampling time limit and is advisable to keep the execution time within 1-2% for a wide safety margin, as shown in the results.

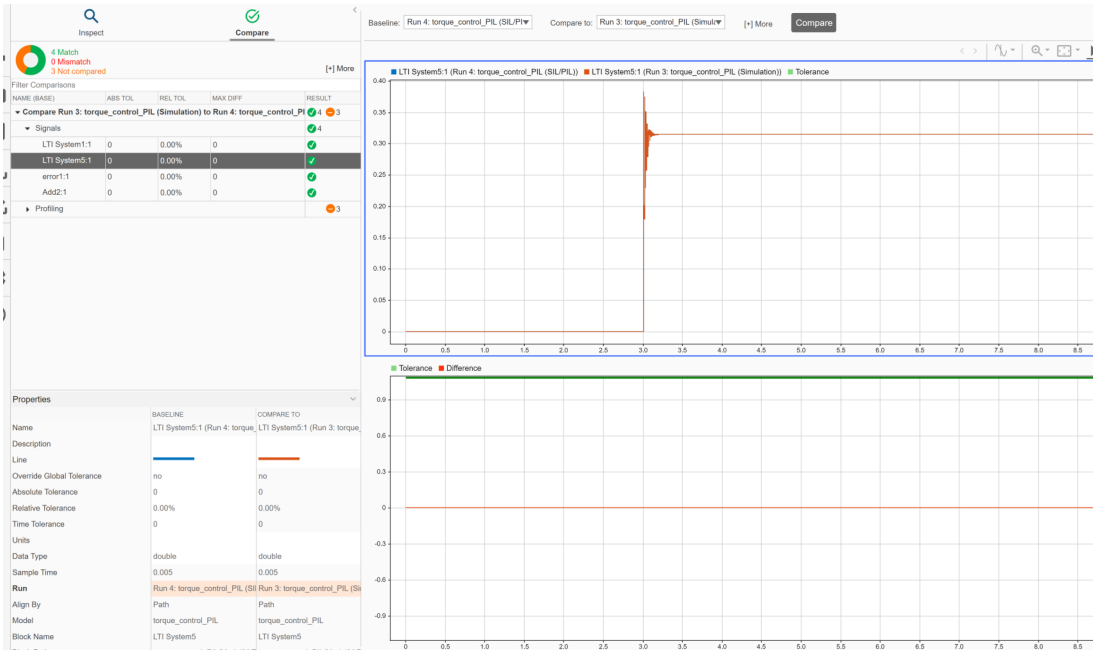

Figure 3.19: Output of Torque Control PIL

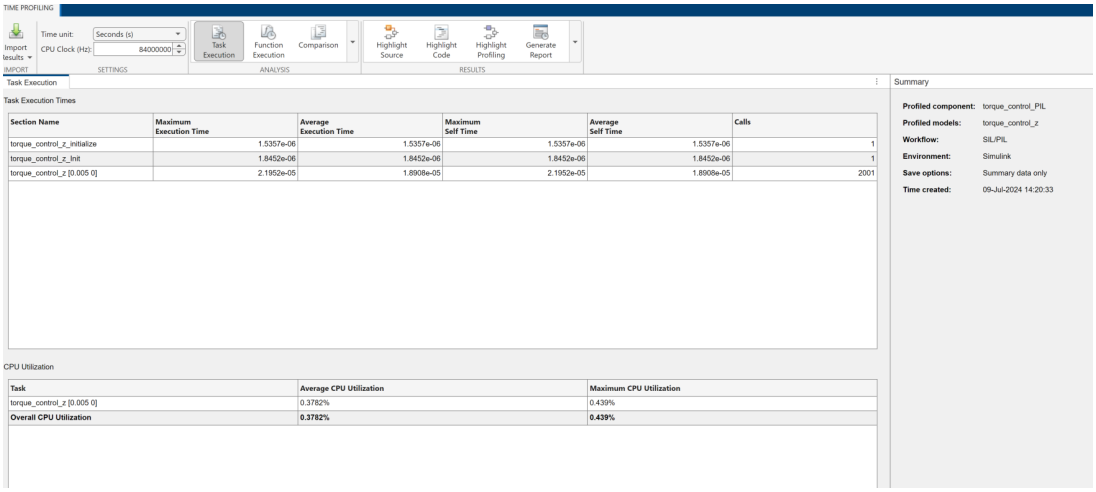

Figure 3.20: Execution Time of Torque Control PIL

3.10 Issues Encountered

This section describes the issues encountered during the design and subsequent implementation of torque control. The discussion here will be more concise since some problems, such as windup and bumpless transfer, have already been described in the case of position control.

The reasons behind the issues of windup and bumpless transfer are practically the same.

However, in this case, the technique used to solve them was different. The algorithm used is called back-calculation and tracking. It is referred to as tracking because it follow the u_M (the actual output of the actuator) which is fed back into the control loop, and back-calculation because the integral action is not explicitly calculated but through a positive feedback "backwards." Although it was initially developed for windup, this technique is implicitly bumpless because the positive feedback related to the integral term updates it even if the process were manually controlled. Figs. 3.21, 3.22, and 3.8 illustrate these concepts.

Another issue encountered was the noise related to current measurements. The solution adopted to address this issue was a first-order lag filter, which is an autoregressive moving average filter. The filter was appropriately tuned using the only configurable parameter α, as described in Ch. 3.5.1.

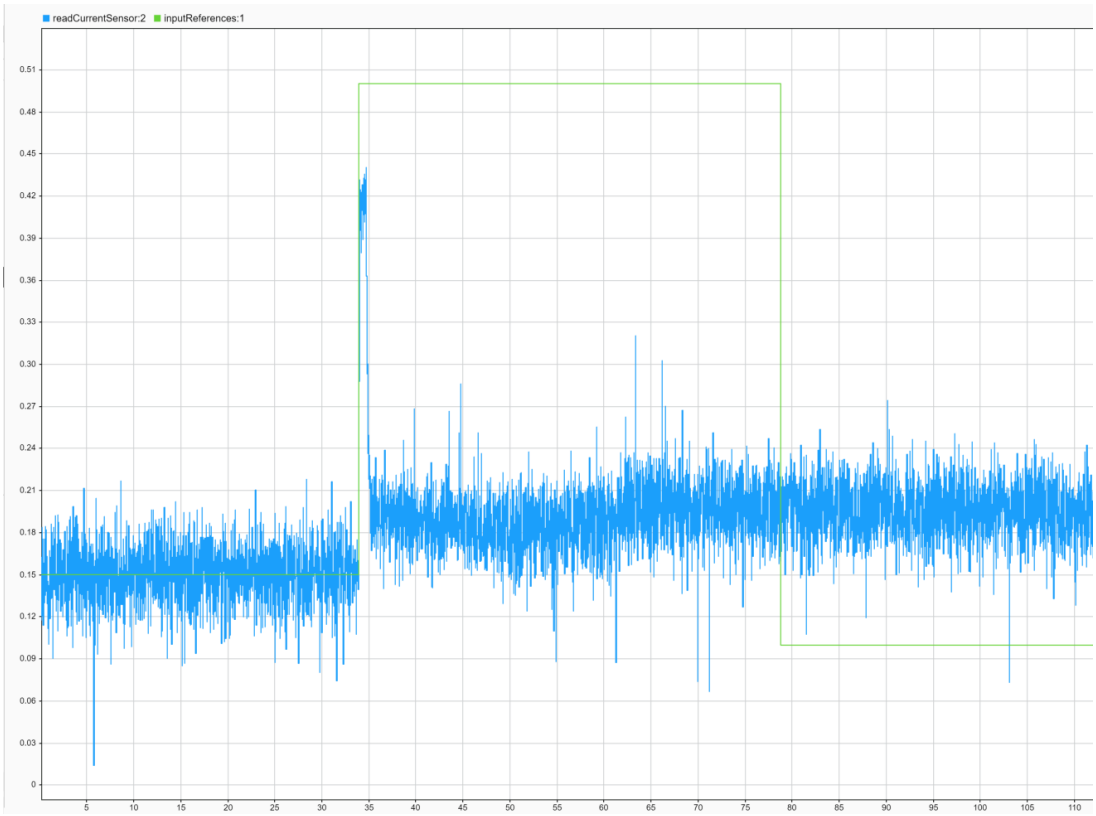

Figure 3.21: Windup torque control

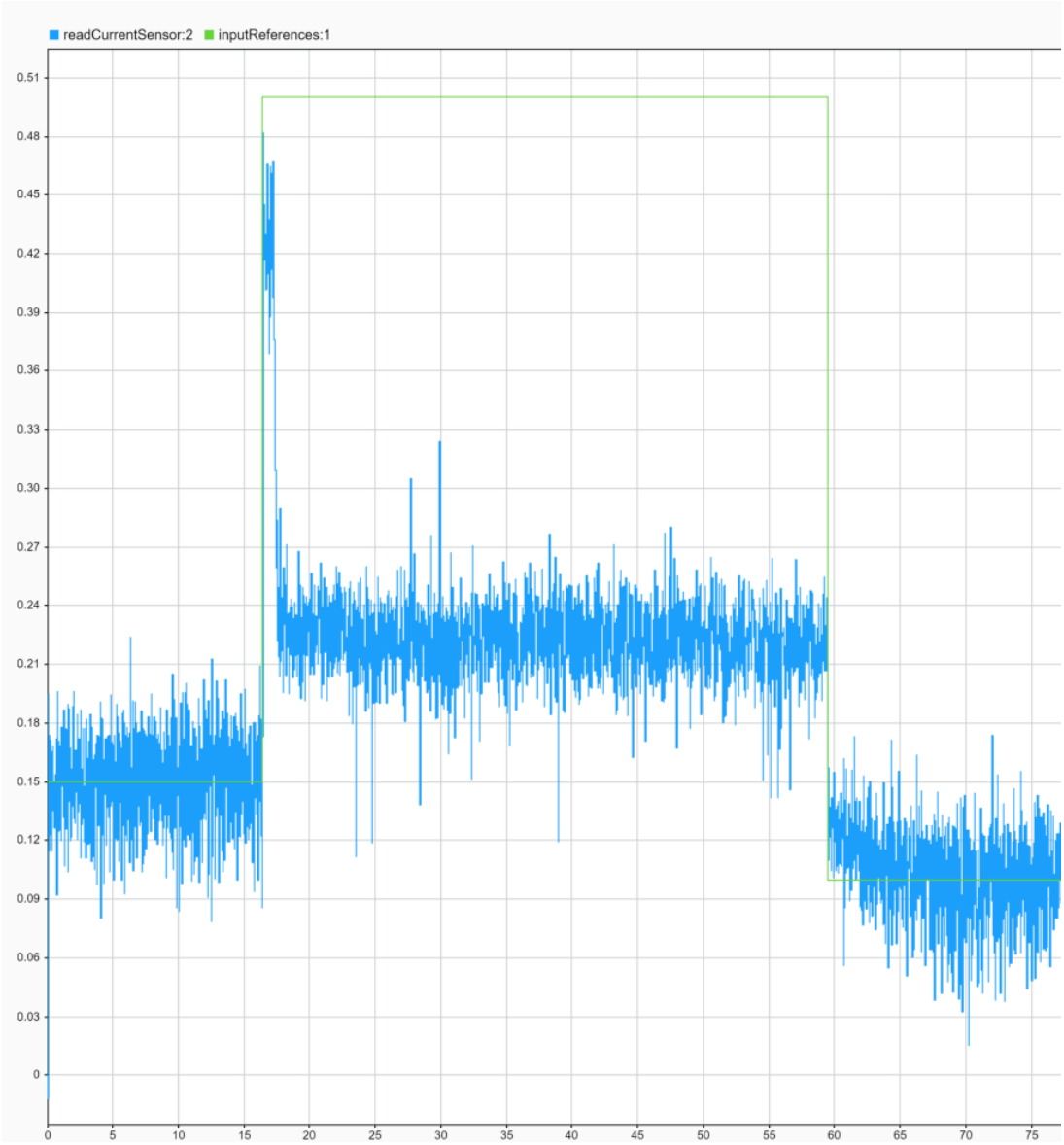

Figure 3.22: Anti-windup torque control

3.11 Hardware Deployment

Once all the previously discussed issues in Ch. 3.10 were addressed and discussed, it proceeded with the implementation of various Simulink schemes for deployment on real hardware. To enable this, the blocks previously discussed and validated during the MIL in Ch. 3.7 and SIL in Ch. 3.8 phases were utilized. Additionally, the transfer function modeling the motor was replaced with blocks interfacing with the real motor, while the current or torque feedback (numerically identical except for the torque constant k_t) present in the simulation schema was replaced with measurements from the current sensor as discussed in Ch. 3.5.

In detail, Fig. 3.23 depicts the schema for deployment on the real motor. Note the presence of controllers with and without anti-windup, chosen for educational purposes to visualize the differences between the two.

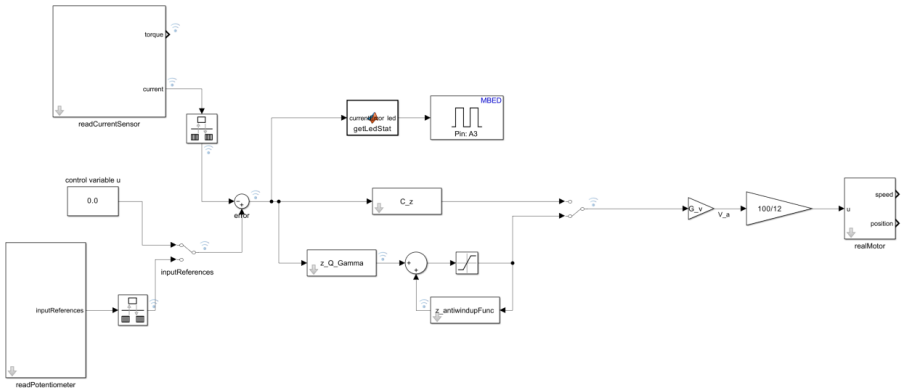

Figure 3.23: scheme torque control real motor

In Fig. 3.24, the results obtained on the real motor are shown.

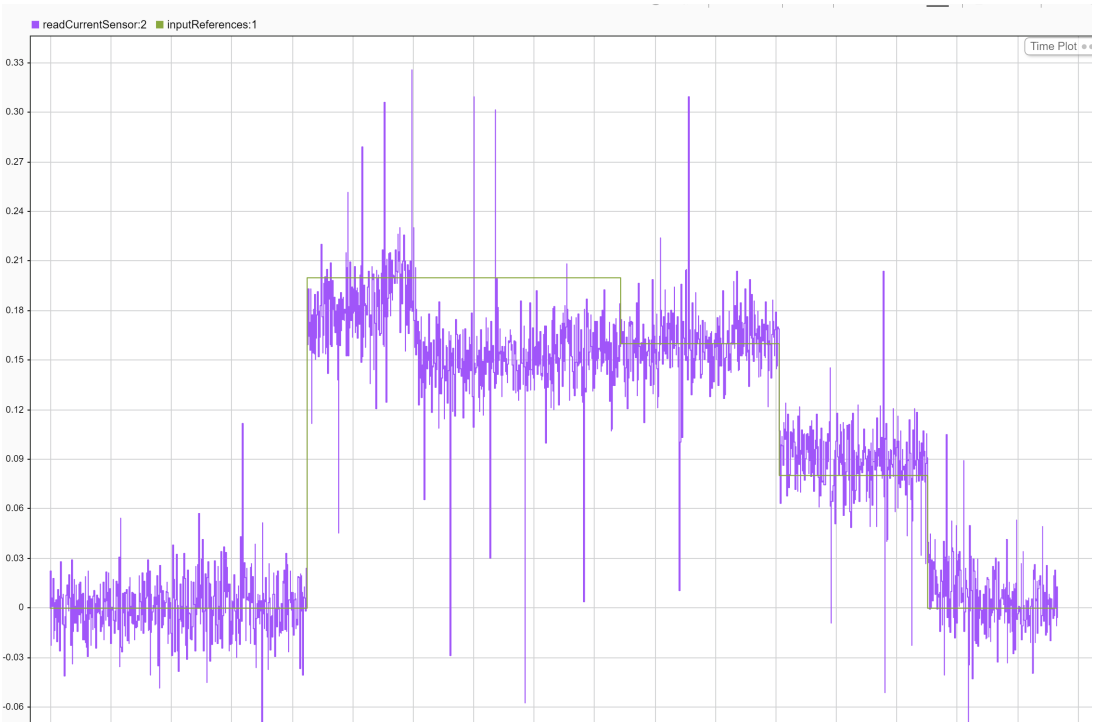

Figure 3.24: out torque control motor

Similar to the position control in Ch. 2.13, there is also a readPotenziometer block present in this schema, used to read a potentiometer and use this reading as the torque reference in this case.

In Fig. 3.25, the internal schema of this block is shown. Note that a firstOrderLagFilter was used for a cleaner reading. The implementation details of these blocks for reading the potentiometer are almost identical to those used for reading the current sensor; for further details, please refer to Ch. 3.5. In this case, the potentiometer range is set to $[0A, 0.25A]$.

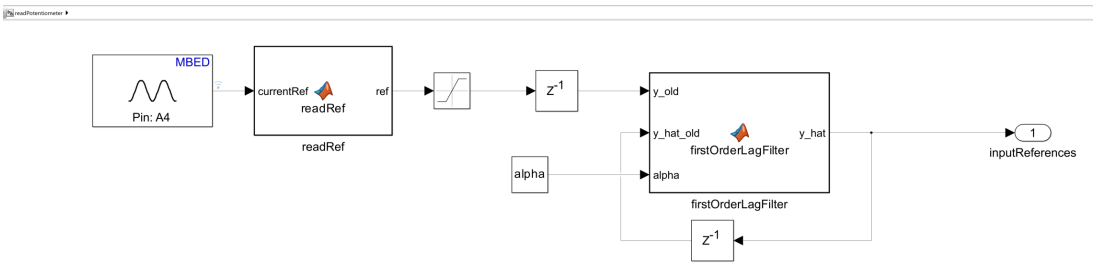

Figure 3.25: read potenziometer

Below, in Fig. 3.26, an example of the system's operation when the reference is taken from the potentiometer is shown.

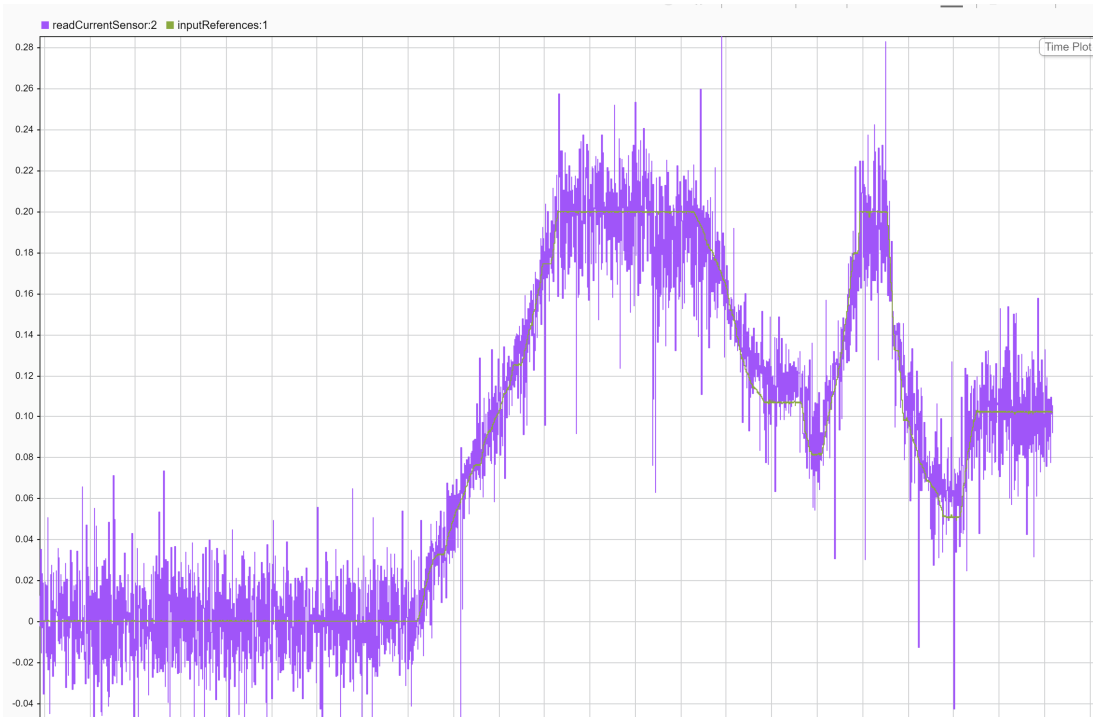

Figure 3.26: potenziometer torque control motor

It is highlighted how using the potentiometer as a system reference is not an optimal choice, as the reference should be slowly variable compared to the process. While this was already true in the case of position control, it is even more true in the case of current control, as the current process is faster than the position process. However, as previously emphasized, this choice was made to test the algorithm under non-ideal conditions. Based on what can be observed in Fig. 3.26, it can be reasonably stated that the control system offers satisfactory performance.

CHAPTER 4

DIRECT CODING

4.1 The motivation of Direct Coding

One of the main advantages of direct coding over automatic code generation is significant cost savings. Consider the following aspects:

- **Cost of MATLAB License**: Purchasing a MATLAB license, required for Model-Based Development (MBD), can cost a company between 10,000 and 15,000 euros. This cost includes partnerships that MathWorks has with board manufacturers, offering an integrated solution for automatic code generation.
- **Cost of Direct Coding**: By using direct coding, the only cost is that of the hardware board, such as the STM32F401RE Nucleo Board, which costs around 20 euros. Additionally, the STM IDE is free, eliminating further software expenses.

Therefore, implementing direct coding can reduce costs from tens of thousands of euros to just a few dozen euros. This represents a significant saving for companies, especially small to medium-sized enterprises or projects with limited budgets.

Direct coding also offers greater control and flexibility in the development process:

- **Customization**: Engineers can customize the code to fit the specific project needs, without being limited by predefined libraries or software generation constraints.
- **Optimization**: It's possible to manually optimize the code to improve performance, reduce memory consumption, and increase system energy efficiency.

Both direct coding and automatic code generation have their own advantages and disadvantages. The choice depends on the specific needs and resources of the company:

- **Direct Coding**: Offers significant cost savings, greater control and customization, but requires skilled engineers and longer development time.
- **Automatic Code Generation**: Involves high costs for licenses, but allows for rapid development and high productivity with fewer specialized skills required.

Ultimately, maximizing productivity from each method depends on the company's business context and available resources.

4.2 Direct Coding Implementation

In the context of developing controllers through direct coding, there are several challenges to address.

4.2.1 Control Loop Synchronization

Synchronization is key to correctly implementing any controller, such as a PI controller. The transfer function (TF) of the controller must be translated into a discrete-time control law that is evaluated exactly every T_s seconds, neither before nor after. This is crucial to ensure that the system behavior matches the theoretical model.

The main tools to achieve this synchronization are:

1. **Internal Interrupts**: By using internal timers that generate interrupts, it can ensure that the control law is executed accurately at the predetermined time T_s. This approach interrupts current tasks at the right moment to execute the control law and then returns to the main task for other operations. This solution is particularly useful when managing other asynchronous tasks alongside the main control, such as UART communication or other system functionalities.
2. **Syscall (System Call)**: An alternative method is based on system calls that provide the current time of the board. This method, although simple, may be optimal for less complex systems where accuracy to the microsecond is sufficient, such as an oven. However, it requires careful handling to avoid additional delays caused by machine instructions.

In our case, it as been exploited the STM32F401RE's capability to manage internal interrupt, as they offer greater precision.

4.2.2 Manual Configuration

In direct coding, it need to manually configure pins and timers either through the .ioc file or directly in the code. This includes configuring PWM for the H-bridge and managing sensors.

H-bridge Configuration

- it allocated a timer and configured pins PB4 and PB5 for PWM.
- it used HAL (Hardware Abstraction Layer) to activate the H-bridge, for example, using `GPIO_PIN_SET` to turn on the bridge.
- it implemented logic to decide which channel of the H-bridge to use based on the sign of the control variable u.

Encoder Configuration

- it set timers in encoder mode and manually handled the reading and processing of received signals.

Current Sensor

- it manually configured pin PC0 for reading the signal from the ACS714 current sensor.
- Using HAL, it implemented reading the signal from the current sensor, necessary for precise motor control.

Current Control

- it created a new timer, TIM2, with a frequency 20 times higher than the main control cycle TIM4.
- TIM2 is configured to perform current sensor readings and apply filtering at a much higher sampling frequency of 4000Hz.
- This setup allows for reading the current sensor with high precision and applying necessary filters for efficient current control.

4.2.3 Implementation of Digital Control Law

A critical aspect of direct coding is translating the control law from its representation in the z-domain to its implementation as a discrete algorithm in the k-domain. In MATLAB/Simulink, this operation is automated, but in direct coding, it must handle it manually.

When developing a controller like a PI or PID, it start from its transfer function in the z-domain:

$$C(z) = \frac{U(z)}{E(z)}$$

where:

$U(z)$ is the z-transform of the controller output.

$E(z)$ is the z-transform of the control error.

To transform this transfer function into the k-domain, it apply the transformation:

$$G(k) = G(z)\Big|_{z=e^{sT}}$$

Where s is a complex variable and T is the sampling period.

Sampling Period

Regarding the implementation of the digital control law, obviously the sampling period chosen in the paragraph Chs. 2.7 and 3.4 plays a key role in this area. Although the observed performance was considered satisfactory, it attempted to lower T_s to verify if this could lead to a significant improvement in response quality. By pursuing this path, it noticed two main issues:

- Serial port baud rate saturation
- Frequency required for filtering

Regarding the first issue, having the ability, in the case of direct coding, to record timestamps of control inputs sent and considering that the serial port baud rate is 115200 bits/s, it checked if a lower sampling time did not exceed the limits of serial communication. It was found that even for a sampling time of 0.004 s, sometimes the control algorithm sent control inputs at non-uniform time intervals. This phenomenon occurs more frequently as the value of T_s decreases. Therefore, it concluded that lowering the sampling time was not feasible because the data log during the execution of the control algorithm saturated the baud rate. Removing the logging part would probably have prevented this phenomenon within a certain range of T_s values, but it would have made it impossible to graphically validate the system's performance.

To demonstrate this, an ad-hoc Matlab script was used Cod.4.1 to control the time between the timestamps associated with the control inputs sent from the controller to the process.

Listing 4.1: Checking timestamps of control input

```
%% Checking timestamps of control input
% Load data
data = load('wrong_timestamps_demostration_data')
timestamps = data.timestamps
n=1;
for i=2:size(timestamps);
    if timestamps(i)-timestamps(i-1) > 0.00401
        expired(n) = i;
        n=n+1;
        fprintf("The difference between %d-th timestamp and
               the %d-th timestamp is greater than Ts:
               difference is %f\n",i+1,i,
               timestamps(i)-timestamps(i-1))
    end
end
```

It is a simple for loop that checks the difference between successive timestamps and saves in an array those whose difference exceeds T_s, which in this case is set to 0.004s, equivalent to a frequency of $250Hz$. As can be seen from the timestamps data structure in Figs. 4.1a 4.1b and the output of the script 4.2, the difference between the 525th and the 524th element is as much as 4 seconds, as is the case between the 1053rd and the 1054th element.

timestamps

1301x1 double

	1
530	6.2070
531	6.2110
532	6.2150
533	6.2190
534	6.2230
535	6.2270
536	6.2310
537	6.2350
538	6.2390
539	6.2430
540	6.2470
541	6.2510
542	6.2550
543	6.2590
544	6.2630
545	6.2670
546	6.2710

(a) Timestamps 1

timestamps

1301x1 double

	1
1048	8.2790
1049	8.2830
1050	8.2870
1051	8.2910
1052	8.2950
1053	8.2990
1054	12.3030
1055	12.3070
1056	12.3110
1057	12.3150
1058	12.3190
1059	12.3230
1060	12.3270
1061	12.3310
1062	12.3350
1063	12.3390
1064	12.3430
1065	12.3470

(b) Timestamps 2

Figure 4.1: Timestamps data structure

```
>> Logging_Sampling_time

data =

  struct with fields:

    timestamps: [1301×1 double]

The difference between 526-th timestamp and the 525-th timestamp is greater than Ts: difference is 4.004000
The difference between 1055-th timestamp and the 1054-th timestamp is greater than Ts: difference is 4.004000
```

Figure 4.2: Output of the Matlab script for checking timestamps.

Another issue encountered concerns the filtering of current measurements. For these measurements, a first-order lag filter was employed due to noise. For effective filtering, it is necessary that the filter's operating frequency be much higher than that of the signal under consideration. Since readings from the sensor are taken every T_s, this would have resulted in a particularly high operating frequency for the filter, which could have exceeded the computational limits of the board. It chose not to go into further detail because from the PIL (Process In the Loop) validation phase, it emerged that with $T_s = 0.005$, the algorithm's computation time was approximately $1 - 2\%$ of the sampling period and therefore it was not reasonable to further

reduce the latter.

4.2.4 Position Direct Coding Implementation

The incremental control algorithm is implemented within the HAL_TIM_PeriodElapsedCallback callback function, which is executed every time the TIM4 timer completes a period. Timer TIM4 is configured with a frequency of 200Hz, so the callback is triggered every 5 milliseconds (0.005 seconds).

$$\text{Period} = \frac{1}{\text{Frequency}} = \frac{1}{200 \text{ Hz}} = 0.005s$$

Execution Time Management: The ticControlStep and tocControlStep times are calculated using HAL_GetTick(), which provides the elapsed time in milliseconds since system startup.

```
k_controller = k_controller + 1;
if (k_controller == 0) {
ticControlStep = HAL_GetTick();
}
tocControlStep = HAL_GetTick();
```

Speed and Position Calculation: The speed (*speed*) and position (*position*) are calculated using specific functions that consider the difference between encoder pulse counts (currentTicks and lastTicks).

```
currentTicks = (double) __HAL_TIM_GET_COUNTER(&htim1);
double speed = getSpeedByDelta(
    getTicksDelta(currentTicks, lastTicks, Ts),
    Ts) / 9.54929;
double position = getPosition(
    getTicksDelta(currentTicks, lastTicks, Ts),
    &ticks_star);
```

Reference Acquisition and Error Calculation: The reference value (*reference*) is read, and the error (e) is calculated as the difference between the reference value and the current position.

```
double reference = referenceVal;
e = reference - position;
```

State Increment Calculation: The increment of the control action related to the state increments (*delta_pos* and *delta_vel*) is computed.

```
double delta_pos = position - pos_prev;
double delta_vel = speed - vel_prev;
double a = (-k_vel * delta_vel) + (-k_pos * delta_pos);
```

Integral Increment Calculation: The increment of s, which is the integral sum, is calculated using the current and previous errors (e and *e_prev*) and *s_prev*.

```
s = s_prev + 0.0025 * e + 0.0025 * e_prev;
double delta_s = s - s_prev;
double b = delta_s * -k_integral;
```

Control Calculation: The control value (u) is calculated based on *delta_u* and constrained within a specified range.

```
double delta_u = a + b;
u = u_prev + delta_u;
if (u > 12) {
    u = 12;
}
if (u < -12) {
    u = -12;
}
```

The output value u is set using the setPulseFromDutyValue() function, which determines the PWM duty cycle to be applied through timer *htim3*.

```
setPulseFromDutyValue(u * 100 / 12);
```

Variable Updates: Update variables (*u_prev*, *e_prev*, *s_prev*, *pos_prev*, *vel_prev*) with current values for the next execution cycle.

```
u_prev = u;
e_prev = e;
s_prev = s;
pos_prev = position;
vel_prev = speed;
```

Logging Data Structure Filling: Fill the data structure *record* for logging purposes.

```
record r;
r.current_u = u;
r.current_current = y_hat_local;
r.current_error = e;
r.current_ref = reference;
r.cycleCoreDuration = controlComputationDuration;
r.cycleBeginDelay = tocControlStep - ticControlStep - (k_controller * Ts * 1000);
r.currentTimestamp = HAL_GetTick();
r.current_pos = position;
r.current_vel = speed;
```

4.2.5 Torque Direct Coding Implementation

For reading the current value and applying the first-order lag filter, it used the callback function `HAL_TIM_PeriodElapsedCallback`, which is executed every time the TIM2 timer completes a period, operating at a frequency of 4000Hz. This means the callback is triggered every 0.25 milliseconds (0.00025 seconds), as:

$$\text{Period} = \frac{1}{\text{Frequency}} = \frac{1}{4000 \text{ Hz}} = 0.00025$$

The reason for this choice is to perform filtering at a frequency 20 times higher than that of the control algorithm. The `HAL_TIM_PeriodElapsedCallback` function for the TIM2 timer performs the following operations:

- **ADC Value Acquisition:** ADC conversion is started using `HAL_ADC_Start`, and then it wait for the conversion to complete using `HAL_ADC_PollForConversion`. Once the conversion is done, the raw ADC value is obtained using `HAL_ADC_GetValue` and stored in the variable `raw`.
- **ADC Data Processing:** The raw ADC value (`raw`) is processed by the function `process_adc_data`, which returns a value `y_hat`.

```
if (htim == &htim2) {
uint32_t raw;
HAL_ADC_Start(&hadc1);
HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
raw = HAL_ADC_GetValue(&hadc1);
y_hat = process_adc_data(raw);
}
```

The function **`process_adc_data`**, compute several operations:

- **ADC Conversion:** The raw ADC value `raw` is converted into a digital value (`digital_current_value`) using the `getDigitalSensorValue` function.
- **Current Calculation:** The digital value is then converted into current using the `getAmpere` function.
- **First-Order Filtering:** The current value undergoes first-order lag filtering with the parameter `alpha`, returning `y_hat_local_process_adc_data` as the filtered value.

```
double process_adc_data(uint32_t raw) {
    double y_hat_local_process_adc_data = 0;
    double digital_current_value = getDigitalSensorValue(raw);
    double ampere_value = getAmpere(digital_current_value);
    y_hat_local_process_adc_data = firstOrderLagFilter(y_old, y_hat_old, alpha);
    y_old = ampere_value;
    y_hat_old = y_hat_local_process_adc_data;
    return y_hat_local_process_adc_data;
}
```

Functions Details:

- `getDigitalSensorValue:` Converts the raw ADC value to a digital voltage using the formula:

$$\text{digital\_value} = \text{raw\_value} \times \frac{3.3}{4095}$$

```
double getDigitalSensorValue(double raw_value) {
double digital_value = raw_value * 3.3 / 4095;
return digital_value;
}
```

- `getAmpere:` Calculates the current from the digital voltage, using a formula based on the known reference voltage and sensor characteristics.

```
double getAmpere(double voltage_value) {
return ((voltage_value - 2.5 - 0.070) / 0.185);
}
```

- `firstOrderLagFilter:` Implements a first-order lag filter that combines the filtered value `y_filtered` based on the current (`y_hat_old`), previous (`y_old`), and a filter coefficient (`alpha`).

```
double firstOrderLagFilter(double y_old, double y_hat_old, double alpha) {
double y_filtered = (alpha * y_hat_old) + ((1 - alpha) * y_old);
return y_filtered;
}
```

The torque control algorithm is implemented within the `HAL_TIM_PeriodElapsedCallback` function, which is executed every time the TIM4 timer completes a period, operating at a frequency of 200Hz. This means the callback is triggered every 5 milliseconds (0.005 seconds), as:

$$\text{Period} = \frac{1}{\text{Frequency}} = \frac{1}{200 \text{ Hz}} = 0.005$$

Execution Time Management: The times `ticControlStep` and `tocControlStep` are calculated using the `HAL_GetTick()` function, which returns the elapsed time in milliseconds since the system started, used to calculate the algorithm's execution time.

```
if (k_controller == 0) {
    ticControlStep = HAL_GetTick();
}
tocControlStep = HAL_GetTick();
```

A local value of the measured current (`y_hat`) is stored, as this value is updated at a higher frequency than the control algorithm. Additionally, the reference value is retrieved.

```
double y_hat_local = y_hat;
double reference = referenceVal;
```

Error Calculation: The error `e` is calculated as the difference between the reference value `reference` and the current position `position`.

```
e = reference - position;
```

Speed and Position Calculation: Speed (`speed`) and position (`position`) are calculated using specific functions involving the difference in encoder counts (`currentTicks` and `lastTicks`).

```
currentTicks = (double) __HAL_TIM_GET_COUNTER(&htim1);
double speed = getSpeedByDelta(
    getTicksDelta(currentTicks, lastTicks, Ts),
    Ts) / 9.54929;
double position = getPosition(
    getTicksDelta(currentTicks, lastTicks, Ts),
    &ticks_star);
```

PI Controller Implementation with Antiwindup: The control value (`u`) is calculated based on the error (`e`) between the desired reference and the current controlled value (`y_hat`). The antiwindup term (`z_antiwindFunc`) manages the controller output saturation problem, limiting it within a specified range.

```
z_q_gamma = + 2.556  e - 1.606  e_last
z_antiwindFunc = u_last
u = z_q_gamma + z_antiwindFunc
if (u > 12) {
    u = 12;
}
if (u < -12) {
    u = -12;
}
```

The output value `u` is set using the `setPulseFromDutyValue()` function, which adjusts the PWM duty cycle applied through timer `htim3`.

```
setPulseFromDutyValue(u * 100 / 12);
```

Variable Updates: Variables (`u_prev`, `e_prev`, `s_prev`, `pos_prev`, `vel_prev`) are updated with current values for the next execution cycle.

```
u_prev = u;
e_prev = e;
s_prev = s;
pos_prev = position;
vel_prev = speed;
```

Logging Data Structure Filling: Populate the `record` data structure for logging purposes.

```
record r;
r.current_u = u;
r.current_current = y_hat_local;
r.current_error = e;
r.current_ref = reference;
r.cycleCoreDuration = controlComputationDuration;
r.cycleBeginDelay = tocControlStep - ticControlStep - (k_controller * Ts * 1000);
r.currentTimestamp = HAL_GetTick();
r.current_pos = position;
r.current_vel = speed;
```

4.3 Results And Discussion

4.3.1 Direct Coding Position Results

The results obtained through direct coding in Fig. 4.3 and Fig. 4.4 confirm the validity of the reasoning discussed in the previous sections of Ch. 2 of the project.

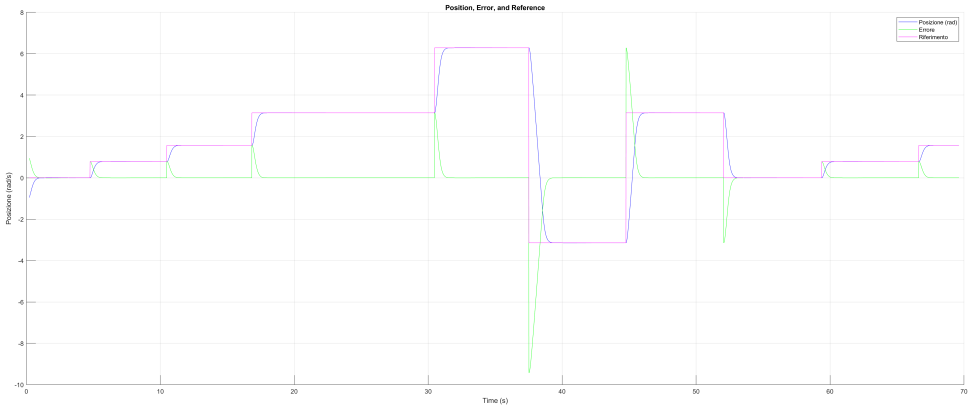

Figure 4.3: Position control direct coding

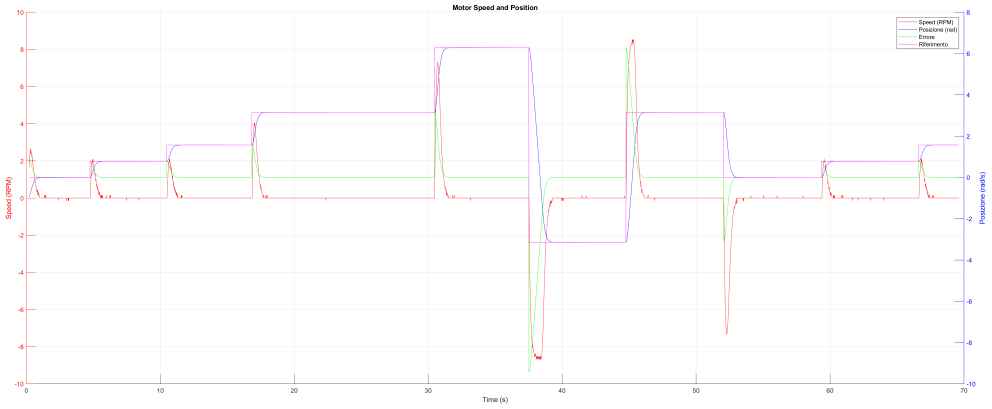

Figure 4.4: Position control 2 direct coding

4.3.2 Direct Coding Torque Results

The results obtained through direct coding in Fig. 4.5 confirm the validity of the reasoning discussed in the previous sections of Ch. 3 of the project.

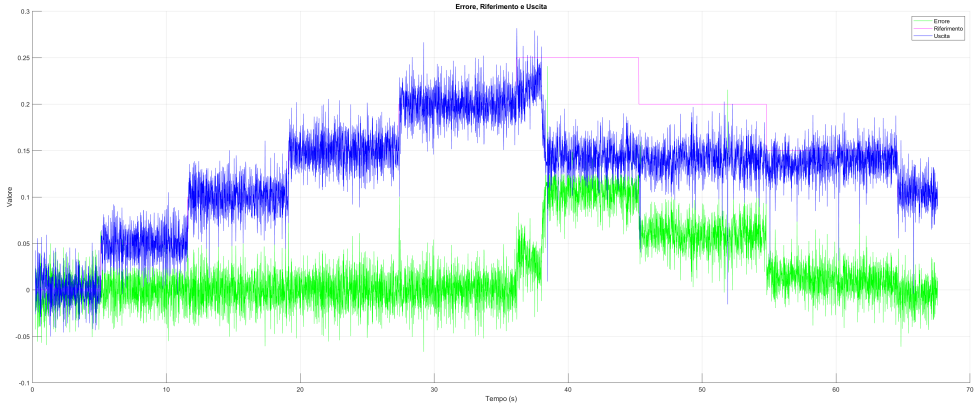

Figure 4.5: Torque control direct coding

CHAPTER 5

RESULT DISCUSSION AND FUTURE DEVELOPMENTS

5.1 Result Discussion

The project, as extensively described, focused on the implementation of position and torque (current) control of a DC motor. In the preliminary phase, the method of areas was adopted to obtain a FOPDT (First Order Plus Dead Time) model of the motor. To improve the accuracy of the model, a moving average filter was used to clean the encoder speed measurements. Attempts were also made to identify the motor parameters, sometimes successfully, as in the case of the armature resistance, and sometimes unsuccessfully, due to both the limitations of the lack of adequate instrumentation and the potentially harmful impact that some identification procedures could have on the motor itself, such as in the case of inductance.

From a strictly hardware perspective, the entire setup of the physical system consisting of the motor, the motor driver, the current sensor, and the potentiometer, as well as their interfacing with the microcontroller and Matlab/Simulink, was handled. The integration of a potentiometer allowed for manual reference inputs for both position and torque, enabling more realistic bumpless transfer tests. Furthermore, the use of the current sensor not only made it possible to implement torque control but also provided an opportunity to truly understand all the issues related to measurement noise and disturbances that are not appreciable in simulation: indeed, the current measurements showed very significant oscillations, and merely bringing the sensor slightly closer to the computer provided tangible evidence of electromagnetic interference. For the current measurements and the values provided by the potentiometer, a first-order lag filter was used to ensure cleaner and more accurate measurements. In fact, a significant portion of the project's effort was dedicated to signal filtering and the related issues, such as the need for the filter to operate at frequencies much higher than the signal, which can sometimes clash with the microcontroller's limitations.

Regarding the design and testing of the controllers, the Model-Based Development paradigm was followed, adhering to the Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), and

Processor-in-the-Loop (PIL) phases. This approach ensured robust design and accurate verification of the controllers' performance at different stages of the development cycle.

The implementation of position control using an incremental LQI regulator proved effective in avoiding issues such as windup and bumpless transfer. This solution allowed the system to maintain stable performance and ensure a smoother and more precise response even in the presence of setpoint variations or external disturbances.

In parallel, the torque control was developed using a PI regulator with the back-calculation and tracking technique. This strategy avoided the aforementioned windup and bumpless transfer phenomena by employing a different solution from the incremental algorithm, but it proved to be equally effective.

During development, various issues that could affect system performance were addressed and, as much as possible, resolved. In addition to the already mentioned problems, difficulties related to the management of the "periodic" error, non-linearities due to the motor's dead zone, quantization, and board limitations related to sampling time were encountered. It was also realized that baud rate limitations imposed barriers when using Simulink in external mode, especially when attempting to push the sampling time below certain thresholds.

In summary, the results obtained demonstrate the validity of the implemented techniques and show that the achieved performances can reasonably be considered satisfactory.

5.2 Future Developments

Despite the positive results obtained, there are several areas where the project can be further improved and developed:

- **Integration of more precise sensors**: Using more advanced sensors and feedback systems, such as high-resolution encoders or precision current sensors, could improve control quality and system response.
- **More powerful hardware**: In the context of current control, having a microcontroller with higher resolution and performance would have allowed for a lower sampling time, improving response quality and enabling higher system speed, thus increasing closed-loop bandwidth.
- **Motor parameters**: With adequate instrumentation, motor parameters could be improved, allowing for a more accurate model.
- **Use of quaternions for error management**: Concerning the problem described in Ch. 2.12.5, an "ad-hoc" solution was employed; however, it is acknowledged that using quaternions, which do not have representation singularities, would have been a more valid solution.

The following are the potential future developments envisioned:

- **Synchronization of Two Motors**: A possible future development could involve the synchronization of two motors. This phase would allow exploring and solving issues related to coordination and synchronization between two drive units, thus improving system performance in more complex applications.

- **Modeling the System as a Segway**: Subsequently, the system could be modeled as an inverted pendulum, also modeling the wheels. This would allow the development of advanced control to keep the system in the upright position, i.e., the equilibrium point $[\pi, 0]$.

The suggested improvements and directions for future developments offer opportunities and insights to further refine the control system, thereby increasing its robustness, precision, and applicability in real-world contexts.

LIST OF FIGURES

BIBLIOGRAPHY

[1] J. Slaughter. "Quantization errors in digital control systems". In: *IEEE Transactions on Automatic Control* 9.1 (1964), pp. 70–74. DOI: 10.1109/TAC.1964.1105624.

[2] Biresh Kumar Dakua and Bibhuti Bhusan Pati. "Prediction and Suppression of Limit Cycle Oscillation for a Plant with Time Delay and Backlash Nonlinearity". In: *2020 IEEE International Symposium on Sustainable Energy, Signal Processing and Cyber Security (iSSSC)*. 2020, pp. 1–5. DOI: 10.1109/iSSSC50941.2020.9358900.