UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA


Corso di Laurea Magistrale in Ingegneria Informatica

# Embedded Digital Controllers Project

Lecturers:

**Ch.mo Prof.**

**Francesco BASILE**

**fbasile@unisa.it**

Authors:

**Davide RISI**

**Mat. 0622702013**

**d.risi2@studenti.unisa.it**

**Giovanni INTONTI**

**Mat. 0622701994**

**g.intonti@studenti.unisa.it**

ANNO ACCADEMICO 2023/2024

# CONTENTS

# CHAPTER 1

# INTRODUCTION AND SETUP OF THE PROJECT

## 1.1 Introduction

The objective of this project is to design modern control schemes using the state-feedback technique for a control system of a DC motor. Specifically, it is required to use the kit provided in the lecture and to design, realize and implement at least one linear quadratic controller.

### 1.1.1 Setup provided

The kit provided consists of:

- STM32F401RE Nucleo-board: a high-performance programmable board based on ARM Cortex-M4 microcontroller whose clock reaches 84 MHz. The board is equipped with a USB interface, via Universal Synchronous-Asynchronous Receiver Transmitter (USART), for communication with the PC and an ST-Link debug interface, 11 hardware timers that can be configured as PWM generators or time counters.

- Pololu 37D Gearmotor: it consists of a high power, 12 V brushed DC motor. It is equipped with a non-separable gearbox that translates a complete inner revolution to a fraction $\frac{1}{131.25}$ of a complete outer revolution. Thus, the gear ratio is $131.25 : 1$. The motor also provides a Hall effect quadrature encoder to sense rotation of the motor shaft. It generates 64 Counts Per Inner Revolution (CPR), thus generating $131.24 * 64 = 8400$ Counts Per outer Revolution.

- STM X-Nucleo-IHM04A1 driver: this module is a dual brush DC motor drive expansion board. It allows to drive up to two bipolar motors (A and B). The driver implements an H-Bridge to set the rotation direction of the motor.

- Kit wiring: allows us to provide the desired power supply to the motor, the maximum power supply is 12V.
- Pololu Wheel Kit: it consists of a wheel, an aluminum hub and an L-bracket Pair that can be used to mount the motor to the chassis.
- Current sensor ACS714: it is a Hall effect-based linear current sensor that can be used to measure the current flowing through the motor.

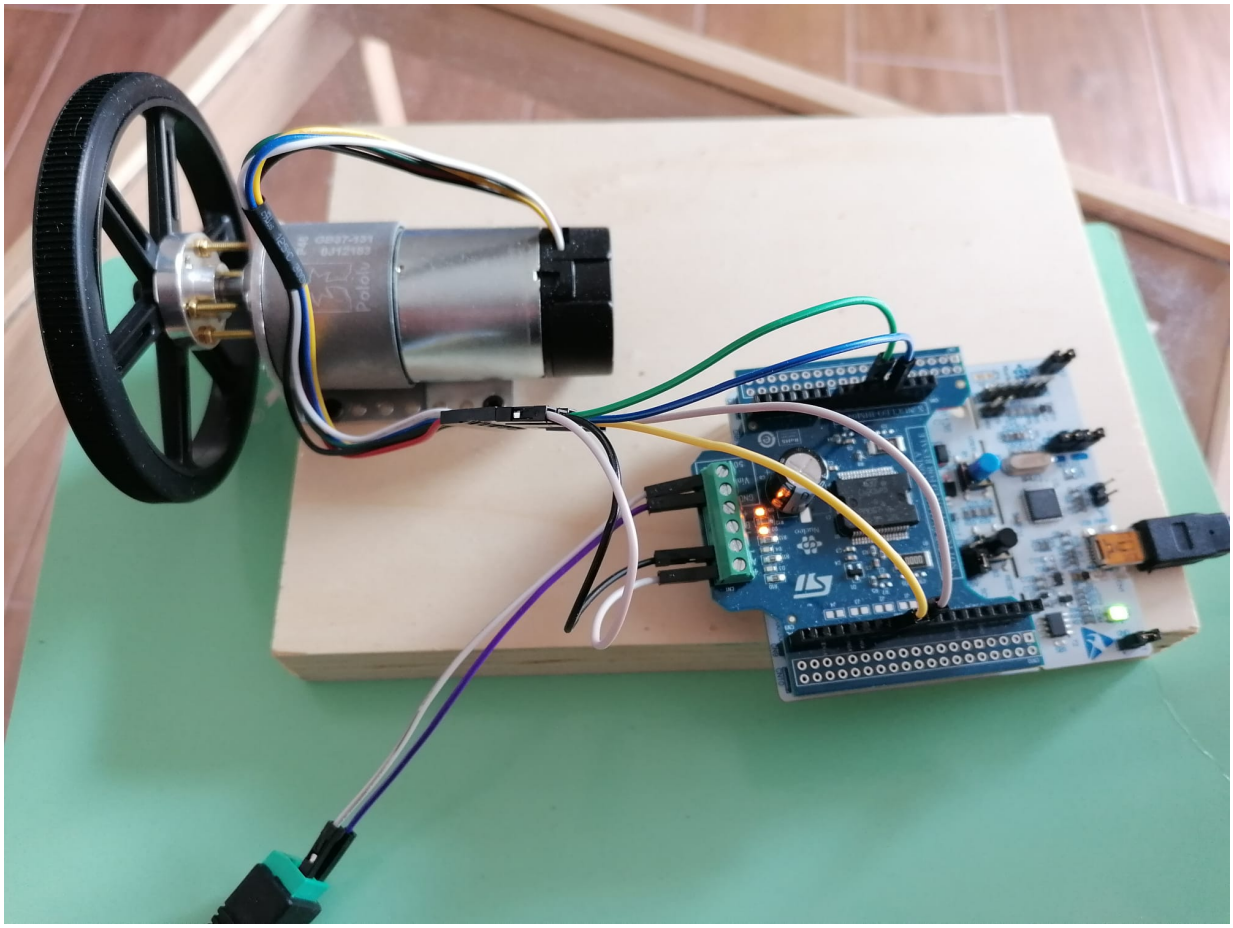

Figure 1.1: Kit provided and mounted for position control

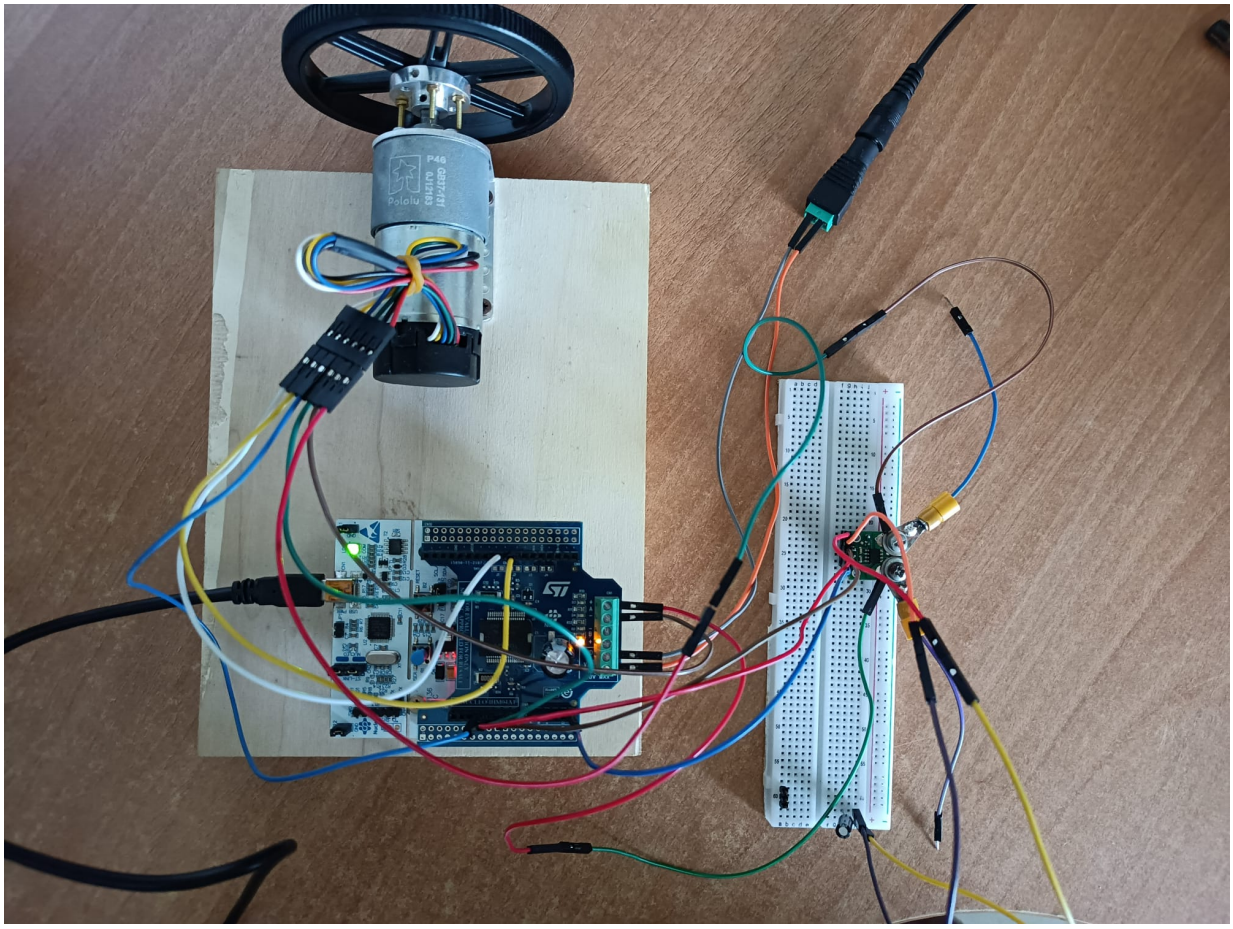

Figure 1.2: Kit provided and mounted for torque control

# CHAPTER 2

# MODEL

## 2.1 System Model

Any DC motor is described by the equations:

$$\begin{cases} V = L\frac{di}{dt} + Ri + K\omega \\ J\frac{d\omega}{dt} = Ki - b\omega \end{cases} \tag{2.1}$$

where: V is the voltage applied to the motor, L is the inductance of the motor, R is the resistance of the motor, K is the torque constant, $\omega$ is the angular velocity of the motor, J is the inertia of the motor, B is the viscous friction of the motor, and i is the motor current.

Neglectting the inductance L because the time constant of the electrical side $\frac{L}{R}$ is much smaller than the one of mechanical side $\frac{I}{b}$. By defining the time constant of the motor as $\tau_m = \frac{IR}{k^2}$ and the motor gain as $k_m = \frac{1}{k}$, the system can be described in state form as:

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx \end{cases} \tag{2.2}$$

where: A = $\begin{bmatrix} 0 & 1 \\ 0 & \frac{-1}{\tau_m} \end{bmatrix}$, B = $\begin{bmatrix} 0 \\ \frac{k_m}{\tau_m} \end{bmatrix}$, C = $\begin{bmatrix} 1 & 0 \end{bmatrix}$, x = $\begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}$, and u = V.

To derive the values of the system parameters, it would be necessary to make experimental measurements. However, in our project, we decided to approximate the system model through one of the techniques seen in class: the method of areas, although for the design of the torque controller, measurements were taken to calculate the motor parameters.

## 2.2 Moving average filter

. Before proceeding with system identification, it is necessary to filter the collected data, as the measured angular velocity had a very high noise. This phenomenon is caused by the nature of the encoder, which provides a discrete measurement of the angular velocity, and as a result the algorithm could measure the velocity wrong by at most one tick, causing, in this way a quantization error that generates noise.
For this kind of errors, a moving-average filter is preferred to other kinds of filters. In this filter a window of size N is scrolled over the data, and the average of the data in the window is calculated and the $i^{th}$ element is replaced with the calculated average.
Formally if $f[m]$ is the signal, w the window width and $w_0 = (w-1)/2$, (with w odd), then the filtered signal $y[m]$ is given by:

$$y[m] = \frac{1}{w} \sum_{m'=-w_0}^{m'=w_0} f[m-m'] \tag{2.3}$$

This filter was implemented in MATLAB, thanks to the *filter* function and applied to the angular velocity signal of the motor. Applying the filter, with a window size of 29, resulted in a cleaner signal, which allowed the system identification to proceed:

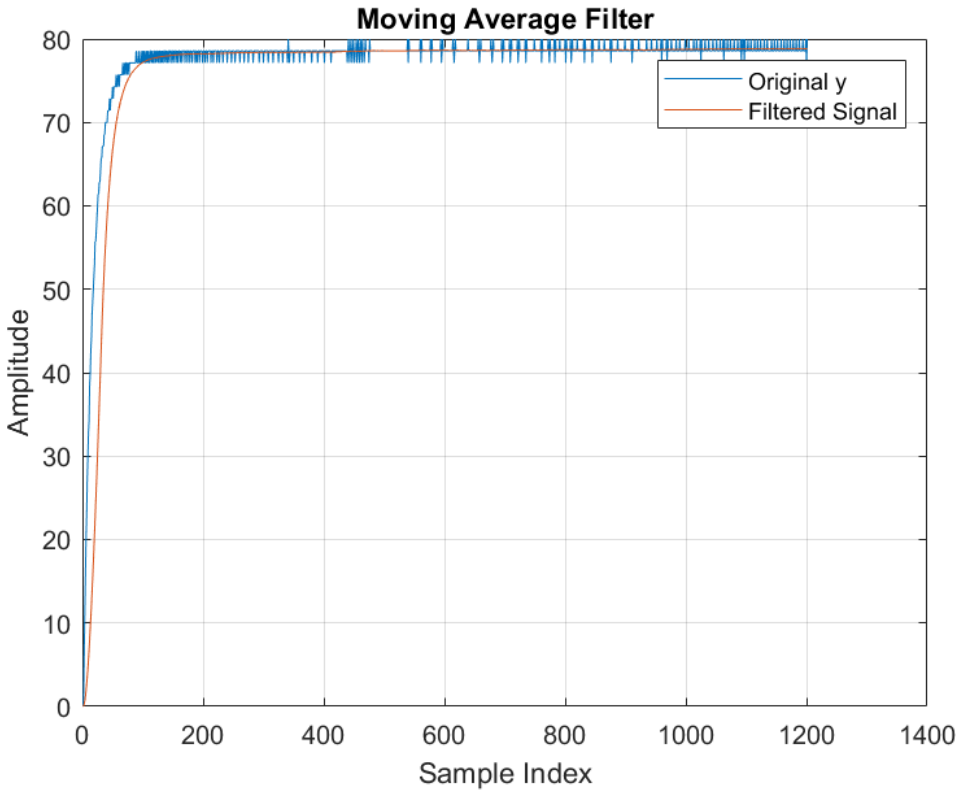


Figure 2.1: Filtered motor angular speed

## 2.3 Areas approach method

The areas approach method is a widely used technique for identifying the parameters of a dynamic system. It involves observing the behavior of the system following a step input, resulting in a measurement of the

system's output, and through the analysis of the areas underlying the system's response, the parameters of the system can be derived.

In particular, the goal is to approximate the model with a transfer function equal to $G(s) = \frac{\mu}{1+Ts}e^{-\tau s}$, where $\mu$ is the static gain, T is the time constant of the pole, and $\tau$ is the delay. To derive the unknown parameters, i.e., $\mu$, T and $\tau$, it is necessary to calculate the area subtended by the system response, called $S_2$ and the area between the system response and the asymptote of the steady state, denoted $S_1$. The latter area is given by:

$$S_1 = \mu\overline{u}\tau + \int_0^\infty \mu\overline{u}e^{-\frac{t}{T}}dt = \mu\overline{u}(T+\tau) = \overline{y}(T+\tau) \tag{2.4}$$

where $\overline{u}$ is the value of the input, while $\overline{y}$ is the steady state value of the output.

The area $S_2$ is given by:

$$S_2 = \int_0^\infty \mu\overline{u}(1-e^{-\frac{t}{T}})dt = \frac{\mu\overline{u}T}{e} = \frac{\overline{y}T}{e} \tag{2.5}$$

So, the parameters of the system can be derived as:

$$\begin{cases} \mu = \frac{\overline{y}}{\overline{u}} \\ T = \frac{S_2 e}{\overline{y}} \\ \tau = \frac{S1}{\overline{y}} - T \end{cases} \tag{2.6}$$

As for our system, we performed an experiment in which we applied a step voltage to the motor, equal to 12V, and measured the angular velocity of the motor output.

After filtering the signal, as explained in the previous section, we calculated the area subtended by the system response through the use of the function *trapz* of MATLAB. In the same way, we calculated the area between the system response and the asymptote of the regime.

Finally, we calculated the parameters of the system through the above formulas obtaining, as the transfer function:

$$G(s) = \frac{6.732}{1+0.1206}e^{-0.0568s} \tag{2.7}$$

The same process was repeated for the second motor used for the torque control, obtaining the transfer function:

$$G(s) = \frac{6.572}{1+0.1292}e^{-0.0496s} \tag{2.8}$$

For the next steps, we will consider the transfer function of the first motor, as the second motor has similar parameters.

## 2.4 State-space representation

Once the transfer function of the system has been obtained, it is possible to derive the state-form representation of the system.

To do this, first, keep in mind that the transfer function we just derived, links the input u, which in our

case is the voltage applied to the motor, to the output y, which in our case is the angular speed of the motor in *rpm*. However, the required control is in position, i.e., the reference is the angular position of the motor, and not the angular speed, consequently it is necessary to integrate the angular speed to obtain the angular position, and since it is preferable to measure it in radians, it is necessary to divide by a conversion constant of 9.5493.

Further step is to handle the delay present in the transfer function. It is possible, in this case, to pursue three ways:

- Use the Padè approximation of the first order, that is, $e^{-\tau s} \approx \frac{1-0.5\tau s}{1+0.5\tau s}$, and replace the delay with its approximation and then proceed with the transformation to state form.
- Use the state-form representation of the transfer function without the delay and handle it by inserting it into the state-form system and applying Padè's approximation directly to the system.
- Neglect the delay and proceed with the transformation to state form.

In our case, we decided to proceed by the second way that is equivalent to the first. The third way, instead, was not considered because we wanted to take into account the delay in the system as a design choice.

Once we described the system by means of the delay-free transfer function, we derived the state-form representation of the system, obtaining:

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx \end{cases} \tag{2.9}$$

where: $\mathrm{A} = \begin{bmatrix} -8.2891 & 0 \\ 1 & 0 \end{bmatrix}$, $\mathrm{B} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\mathrm{C} = \begin{bmatrix} 0 & 5.8438 \end{bmatrix}$.

Notice that that in this case the states of the system are the angular velocity ($x_1$) and the angular position ($x_2$) of the motor.

Having obtained the model of the system in state form, it is possible to consider the delay present in the transfer function, inserting it into the system in state form, and identifying it as InputDelay. Applying the MATLAB function *pade* to obtain the Padè approximation, and specifying the degree of the desired approximation, which in our case is 1, we obtain the system in state form with the approximation:

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx \end{cases} \tag{2.10}$$

where: $\mathrm{A} = \begin{bmatrix} -8.2891 & 0 & 8.8101 \\ 1 & 0 & 0 \\ 0 & 0 & -35.2405 \end{bmatrix}$, $\mathrm{B} = \begin{bmatrix} -1 \\ 0 \\ 8 \end{bmatrix}$ and $\mathrm{C} = \begin{bmatrix} 0 & 5.8438 & 0 \end{bmatrix}$. Notice that in this case the states of the system are the angular velocity ($x_1$), the angular position ($x_2$) of the motor, and the new state that represent the approximation of the delayed input ($x_3$).

# CHAPTER 3

# POSITION CONTROL

## 3.1 System reachability

Before proceeding with the design of the controller, it is necessary to check the reachability of the system to ensure that the system is controllable.

Given a linear system, it is known whether or not it is controllable by calculating a simple matrix (reachability matrix).

Considering the system in state form 2.4, the reachability matrix is given by:

$$\mathcal{W}_{\mathcal{R}} = \begin{bmatrix} B & AB & A^2B \end{bmatrix} \tag{3.1}$$

A linear system with a single input is reachable if and only if the determinant of $\mathcal{W}_{\mathcal{R}}$ is different from zero. This is generalizable to systems with multiple inputs, where the system is reachable if and only if the rank of $\mathcal{W}_{\mathcal{R}}$ is equal to $n$, where $n$ is the number of states of the system.

In our case, the determinant of $\mathcal{W}_{\mathcal{R}}$ is equal to 3, so the system is controllable.

## 3.2 Controller selection

Once the reachability of the system has been verified, it is possible to proceed with the design of the regulator.

The regulator is designed using the LQR control method, or Linear Quadratic Regulator.

This is a mature technology that involves the placement of the eigenvalues of the matrix of the closed-loop system with the aim of achieving performance objectives, namely minimization of a quadratic cost function.

The generic formulation of an LQR problem is given by:

$$\min_{u(t)} \tilde{J} \;\; s.t. \begin{cases} \dot{x} = Ax + Bu \\ x_0 \; given \end{cases} \tag{3.2}$$

where $\tilde{J}$ is the performance index, a cumulative index that in continuous time results in an integral over a time horizon $T$:

$$\tilde{J} = \int_0^T (x^T Q x + u^T R u) dt \tag{3.3}$$

Operationally in the LQR control we have $T = \infty$.

Once the problem is defined, conditions must be imposed on the matrices, namely $Q$ and $R$, which represent the design parameters of the controller; in fact, the solution, i.e., the optimal control law, is given by:

$$u(t) = -Kx(t) \tag{3.4}$$

where $K$ is the vector of the controller gains, which in the LQR problem are directly related to $Q$ and $R$.

For the problem to have solutions, both matrices must be symmetric and positive definite.

Actually, $Q$ would suffice to be positive semidefinite, but to ensure the convergence of the problem, we prefer to impose the positive definition.

From the control point of view, the first term of the performance index, i.e., $x^T Q x$, represents the cost associated with the system state, while the second term, i.e., $u^T R u$, represents the cost associated with the system input.

In general, $Q$ and $R$ are diagonal matrices, since we want to weight the individual states and inputs of the system differently.

## 3.3 Introduction of the integral action

To improve the performance of the controller, an integral action term can be introduced by expanding the state matrix of the system, by adding a new state, $\dot{z} = y - r$, where $y$ is the system output, $r$ is the reference, and $z$ is the integral action state.

With the addition of the new state, the $Q$ and $R$ matrices increase in size, since we need to weight the new state as well.

Notice that, the recheability matrix of the system now is bigger, but the system is still reachable as shown in the MATLAB files provided.

In this project, the $Q$ and $R$ matrices were calibrated manually, so as to obtain a controller that follows the reference accurately and quickly. Specifically, $Q$ was chosen as:

$$Q = \begin{bmatrix} 0.01 & 0 & 0 & 0 \\ 0 & 15 & 0 & 0 \\ 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0.2 \end{bmatrix} \tag{3.5}$$

while $R$ was chosen as:

$$R = 0.0002 \tag{3.6}$$

obtaining the vector of regulator gains equal to:

$$K = \begin{bmatrix} 26.7249 & 306.6733 & 8.7822 & -31.6228 \end{bmatrix} \tag{3.7}$$

through the use of MATLAB's *lqi* function, which calculates the controller gains from from the system in state form and the $Q$ and $R$ matrices.

In addition, it performs the expansion of the system in state form by adding the integral action term.

It should also be noted that the gain relative to the state of the integral action is of opposite sign, since the new state introduced by the function *lqi* is $z = r - y$, unlike what we've said before, i.e. $z = y - r$.

In this way, we have obtained a regulator that stabilizes the system around its equilibrium point, while thanks to the addition of the integral action we are not only able to reject constant disturbances, but we are also able to follow the reference.

In addition to this, an additional control term is introduced, equal to $k_r$ which allows tracking of the reference.

The term $k_r$ was calculated as:

$$k_r = \frac{1}{C(A - BK)^{-1}B} \tag{3.8}$$

where $A$ and $B$ refer to the system in non-extended state form, i.e., without the integral action, while $K$ is the vector of controller gains without the integral action.

Note that the term $k_r$ is optional and can be omitted, since the controller is able to follow the reference even without it. However, this is only valid if there are no constraints on the input of the system, and since, as explained in the next section, the anti-windup action will be introduced, it is necessary to introduce the term $k_r$.

Before continuing with the discussion, it is highlighted that for the choice of the $Q$ and $R$ matrices, it was necessary to carry out an analysis of the system performance, in particular it was necessary to evaluate the settling time, overshoot and step response of the system, considering the saturation of the system input. These evaluations were carried out through simulation of the system in Simulink, specifically in the MIL phase, in which it was possible to evaluate the performance of the system in response to different choices of the $Q$ and $R$ matrices, also evaluating the effect of disturbance.

## 3.4 Choice of sampling time

Before proceeding with the discretization of the system, which is necessary for the proper modeling of the predicted SIL/PIL stages, it is necessary to choose the sampling time of the system.

To do this, it is necessary to keep in mind that the sampling time must be small enough to ensure the stability of the system, but not too small to cause computational problems.

First of all, the choice of sampling time is linked to the desired settling time, as the sampling time must be at least 10 times shorter than the settling time of the system. This consideration will be respected for both position control and torque control. In addition, it should be noted that the problem of quantization of the integral term also comes into play: in discrete time, the null error, that the controller reads, includes everything that is assimilated to zero for the target machine. This implies that for small errors, the integral term does not act with a variation of the control input; this phenomenon is amplified by the decrease in sampling time, consequently, it may be inconvenient to choose a sampling time even smaller than the one chosen.

Considering, also, that the power supply to the motor is given by a PWM signal, it is necessary that the sampling time be small enough to ensure that the square wave oscillation can be considered as a constant signal.

For these reasons, in accordance with what was seen at the course, a a sampling time of 0.005 seconds, which by far guarantees that the PWM signal can be considered as a constant signal, in fact having a band

$$w_3 = 15.3 rad/s$$

the chosen sampling time is small enough to ensure that the sampling frequency is at least 20 times the system bandwidth, in this case the sampling rate is 200 Hz, which is about 82 times greater than the system bandwidth.

## 3.5 Discretization of the controller

Once the controller is obtained, it is necessary to discretize the system in order to implement it on the provided microcontroller and to test it in SIL/PIL mode. Tustin's discretization method is used here, which approximates the continuous system with a discrete system through the formula:

$$\frac{1}{s} \approx \frac{2}{T_s} \frac{z-1}{z+1} \tag{3.9}$$

Notably, the only term that needs to be discretized is the integral action term.

Applying Tustin discretization to the integral action term $\alpha(z)$, we obtain:

$$\alpha(z) = \frac{0.0791z + 0.0791}{z-1} \tag{3.10}$$

## 3.6 Anti windup

One problem that can occur with the integral action is the phenomenon of windup, caused by saturation of the control action.

This is due to the fact that the integral, even after saturating, continues to accumulate the error, becoming

a ramp.

Due to this phenomenon when the error changes sign, the integral action must dispose of the accumulated ramp, causing a delay in the system response.

To solve this problem, several solutions can be introduced, in our case we chose to follow one of the most common techniques seen during the course, namely back-calculation and tracking.

Before introducing the term of anti-windup, it should be noted that since the controller is state-feedback, the anti-windup technique acts only on the action integral, since the terms that weight the states of the system and the reference are not subject to this phenomenon.

Considering the back-calculation and tracking feedback control scheme:

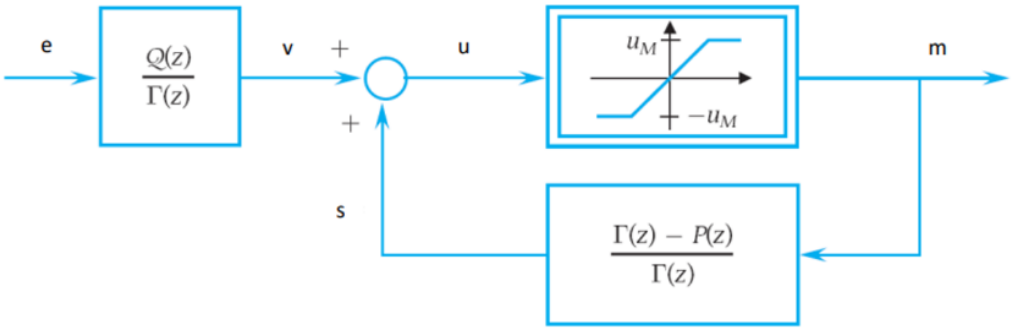


Figure 3.1: Scheme of feedback control with anti-windup

and considering the discretized control action, the various terms are calculated, from the integral action term 3.5, as:

$$\Gamma(z) = \frac{1}{z} \tag{3.11}$$

while the terms $Q(z)$ and $P(z)$ represent numerator and denominator of the discretized integral action transfer function.

## 3.7 Observability of the system

Given the system in the state space described above, notice that that, in the simulated system it is possible to backtrack both the angular position of the motor and the angular velocity of the motor, moreover it is also possible to measure the new state introduced by Padé's approximation, namely the state related to the delay.

However it is not possible to directly measure the state relative to the delay, consequently it is necessary the introduction of a state observer, capable of estimating the state relative to the delay.

To do this, it is first necessary to verify that the system is observable, that is, that the observability matrix $W_0$ is maximum rank.

Considering the system in state form with delay 2.4, the observability matrix is given by:

$$\mathcal{W}_{\mathcal{O}} = \begin{bmatrix} C \\ CA \\ CA^2 \end{bmatrix} \tag{3.12}$$

A linear system with a single state is observable if and only if the determinant of $\mathcal{W}_{\mathcal{O}}$ is non-zero.
This is generalizable to systems with multiple states, where the system is observable if and only if the rank of $\mathcal{W}_{\mathcal{O}}$ is equal to $n$, where $n$ is the number of states in the system.
In our case, the determinant of $\mathcal{W}_{\mathcal{O}}$ is equal to 3, so the system is observable.

## 3.8 Observer Design

Once the observability of the system has been verified, it is possible to proceed with the design of the state observer.
Note that the design of the controller and the state observer are two separate problems; in fact, it is possible to design them independently of each other.
The only condition that must be met is that the dynamic observer system is faster than the observed system to ensure that the observer converges in a finite time.
In this project, since the system to be observed has three poles, the state observer must also have three poles. To position them, you can use the method of pole placement based on the equivalence of polynomials, specifically between the characteristic polynomial of the system and the characteristic polynomial of the desired one.
To carry out this operation, since we do not have 2 poles, but 3, it is possible to rely on the approximation of the dominant poles of the system, placing the third pole at a greater distance than the first two.
In this way, we try to ensure that the observer converges according to the required specifications.
Notice that that the observer system is a MIMO system, since the input to the observer system is the control action and the output of the motor, while the output of the observer system are the estimates of the system states. Consequently, verifing that the observer system meets the required specifications through MATLAB's *step* command is impossible, since the two inputs are coupled and can not be indipendently chosen.
In fact, given that the step command results in two steps in the two inputs, without considering the coupling between the two inputs, it is not possible to verify that the observer system meets the required specifications.
For example given a step to the input related to the control action, would result in a ramp in the input related to the output of the motor. Vice versa, given a step to the input related to the output of the motor, would result in a control action that first increases and then decreases to zero. For these reasons, it is necessary to simulate the system in its entirety in Simulink, particularly in the MIL stage, where it is

possible to verify that the observer system meets the required specification .

## 3.9 Observer discretization

Once the state observer has been designed, it is necessary to discretize the observer system in order to implement it on the provided microcontroller and to be able to test it in the simulation stages. In this case, the Tustin discretization method 3.5 is used, resulting in the discretized observer system:

$$\begin{cases} x_{k+1} = A_d x_k + B_d u_K \\ y_k = C_d x_k + D_d u_k \end{cases} \tag{3.13}$$

where the matrices are given by:

$$A_d = \begin{bmatrix} 0.9316 & -11.35 & 0.0391 \\ 0.002742 & 0.1195 & 5.55e-05 \\ -0.007094 & -2.897 & 0.8379 \end{bmatrix} \tag{3.14}$$

$$B_d = \begin{bmatrix} -0.004047 & 1.942 \\ -5.745e-06 & 0.1507 \\ 0.03678 & 0.4957 \end{bmatrix} \tag{3.15}$$

$$C_d = \begin{bmatrix} 0.9658 & -5.673 & 0.01955 \\ 0.001371 & 0.5597 & 2.775e-05 \\ -0.003547 & -1.448 & 0.919 \end{bmatrix} \tag{3.16}$$

$$D_d = \begin{bmatrix} -0.002024 & 0.9708 \\ -2.872e-06 & 0.07534 \\ 0.01839 & 0.2478 \end{bmatrix} \tag{3.17}$$

## 3.10 MIL

For the development of the project, the V-Model methodology will be used, in which each design phase is accompanied by a verification phase, so as to ensure that the system meets the required specifications, the phases of verification involve the use of simulation methods, particularly MIL, SIL and PIL, until the system is implemented on the provided microcontroller and the system is tested in real mode.

To evaluate the performance of the controller, the MIL method, or Model In the Loop, can be used.

It involves simulating the system in a simulation environment. The purpose is to verify that the controller meets the imposed specifications. By modeling the system using MATLAB's state-space function, and implementing the controller and observer, it is possible to verify that the system meets the required specifications.

Starting from the Simulink diagram below:

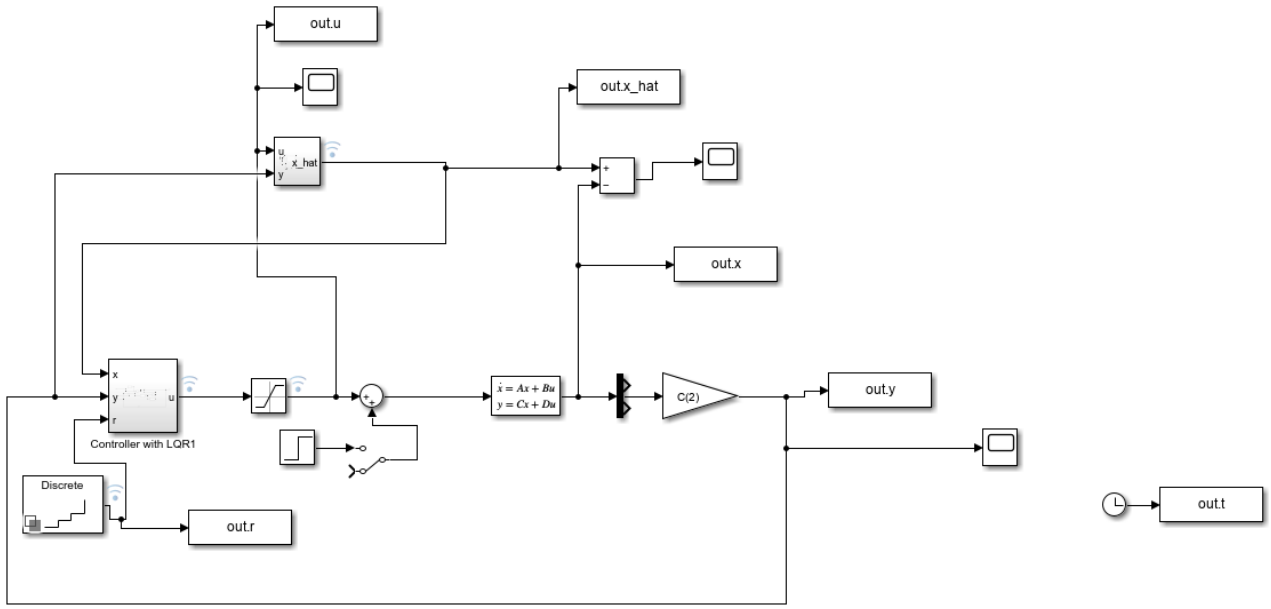


Figure 3.2: Simulink diagram for MIL simulationL

in which we point out the addition of the saturation of the system input and a gain term due to the fact that the system output is 5.84 times the $x_2$ state of the system.

In fact, in the engine simulation block, the matrix $C$ was not preserved but was set to

$$C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.18}$$

so that the state of the simulated system can be measured and compared with the state estimated from the observer system.

The control scheme consists of:

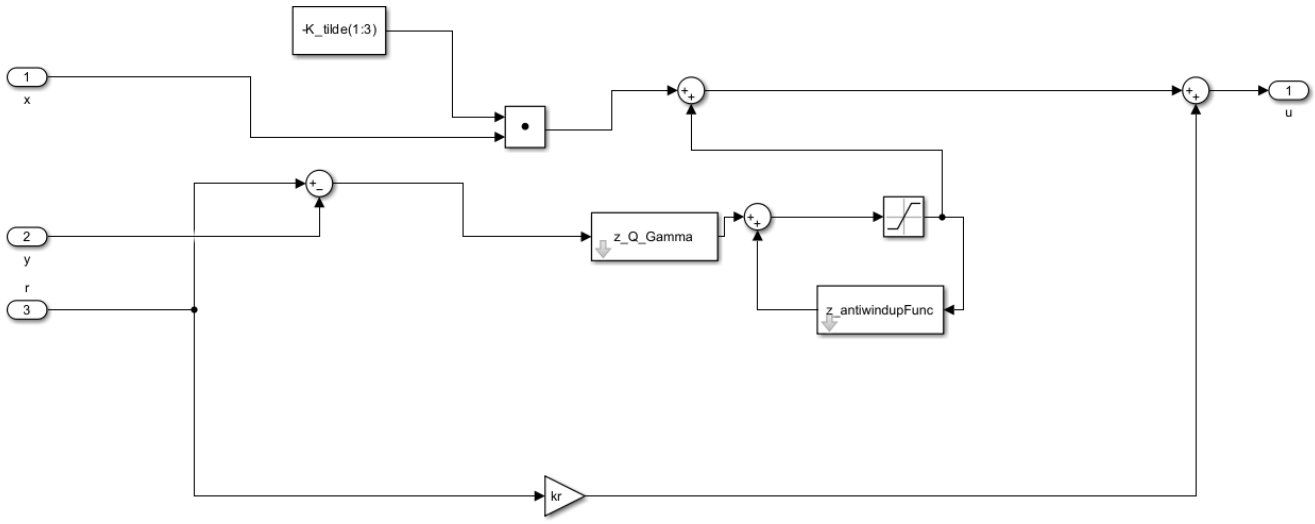


Figure 3.3: Scheme of the controller for MIL simulation

in which the control action, the anti-windup action, and the reference tracking action can be visualized.

In addition, the control scheme also consists of the state observer, specifically:

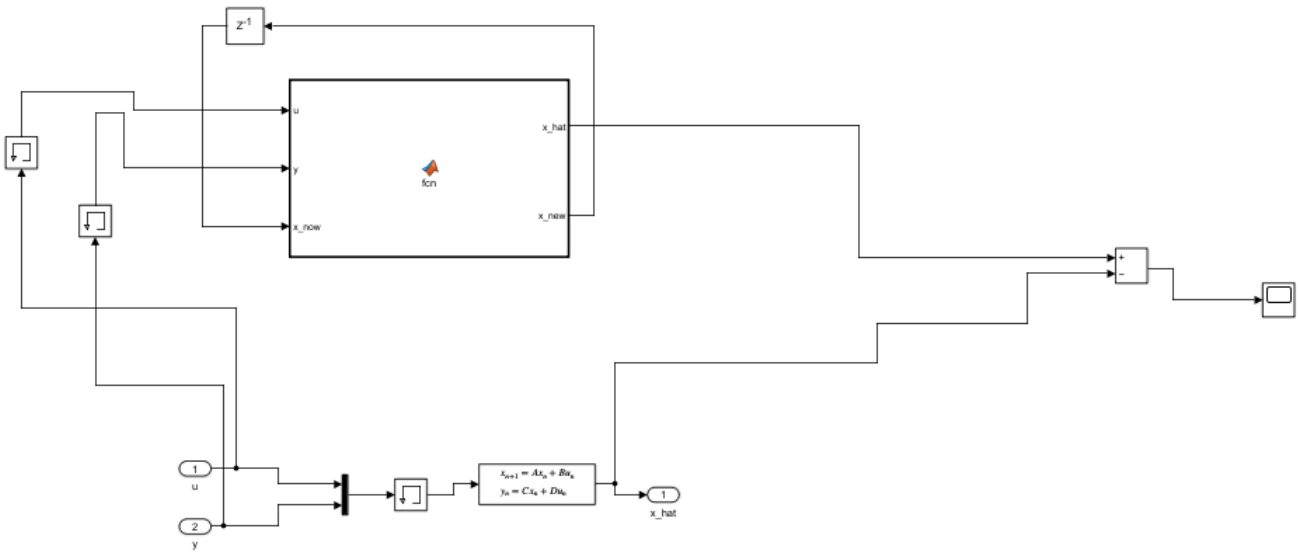


Figure 3.4: State observer diagram for MIL simulation

Note the addition of a memory term, which is necessary to avoid problems caused by the algebraic loop of the state observer, which in order to estimate the state, needs the control input, which in turn needs the estimated state to be computed, so a memory term must be introduced to initialize the system to the initial conditions.

Two different blocks with the same function were included in the observer block, one of which was introduced to verify the correct implementation of the state observer in code C in the next stage of direct coding on the microcontroller 5.1.

From the complete simulink scheme, it is possible to simulate the system under varying reference and disturbance.

Specifically, with this step, we are interested in verifying:

- The settling time of the system
- The overshoot of the system
- The system's response to disturbance
- The rate of convergence of the state observer with respect to the observed system

### 3.10.1 MIL results - without disturbance

In particular, for the evaluation of the system performance, we considered a stair reference with values of 0 ,$\frac{\pi}{2}$,$\pi$ and $2\pi$:

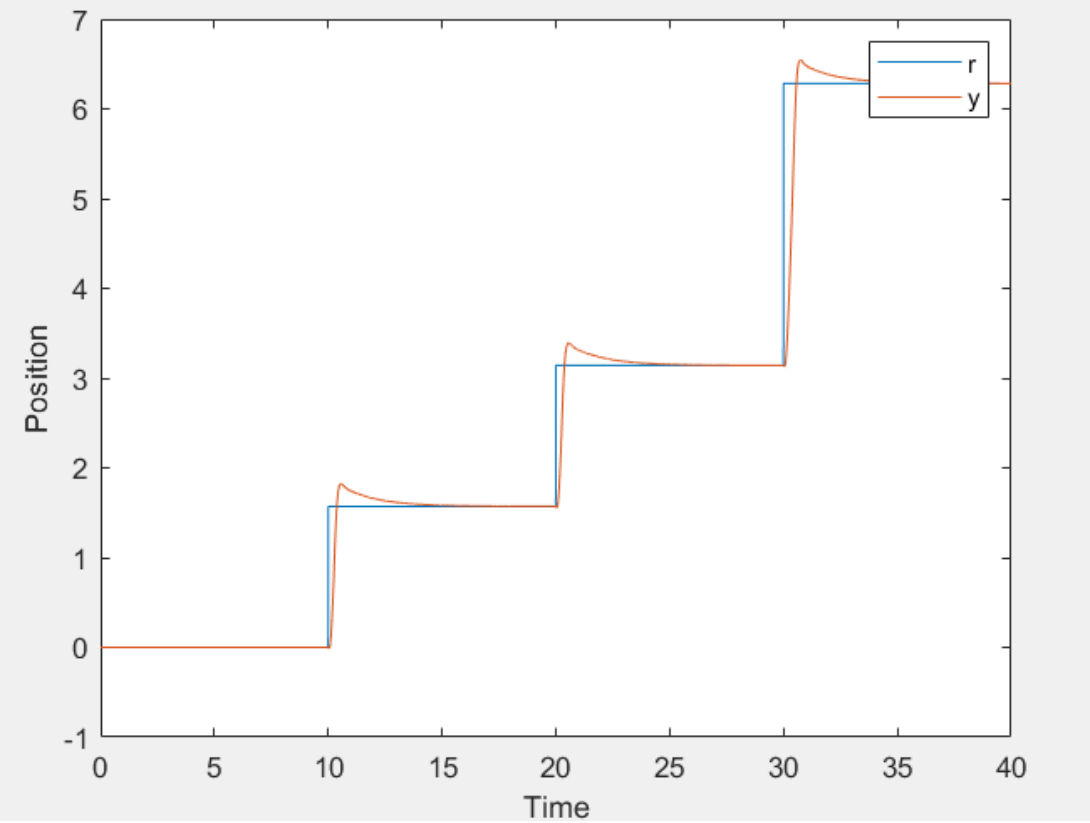


Figure 3.5: Response to the stair reference

As can be observed, the reference is correctly followed by the system, presenting a settling time equal to $\approx 3s$.

It is also possible to visualize the trend of the control action:

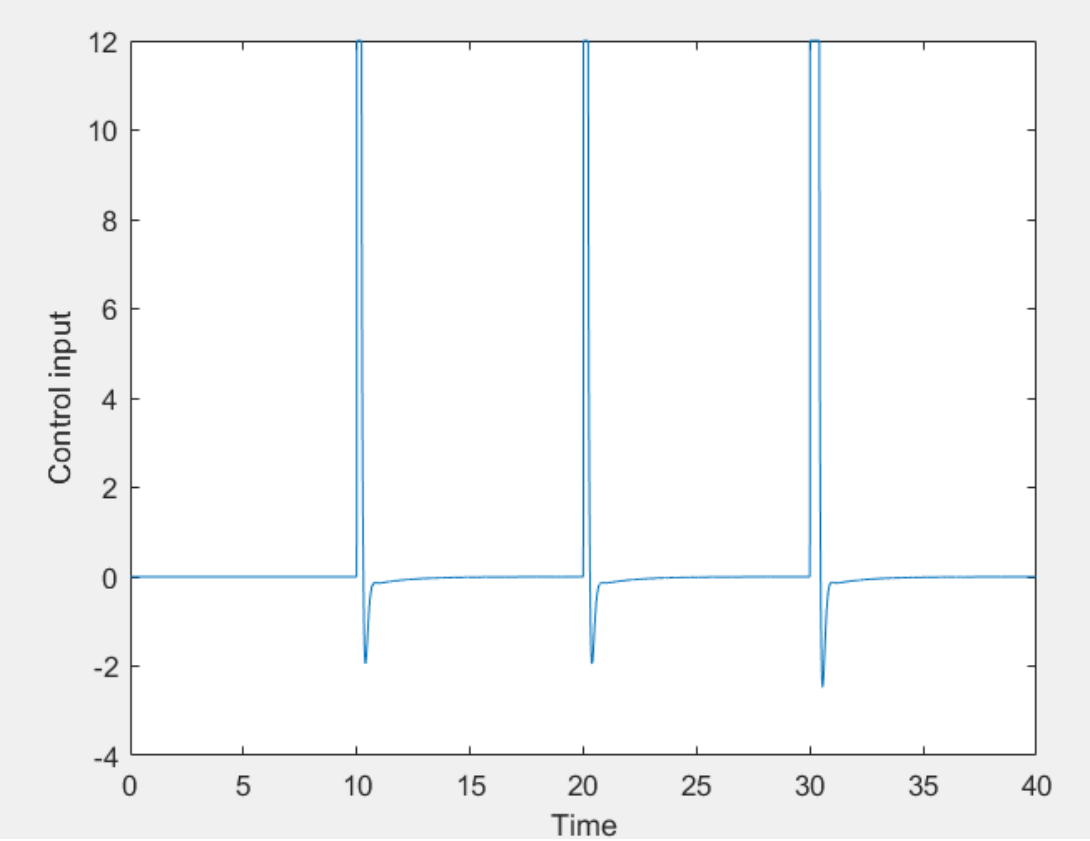


Figure 3.6: Control action

Which, thanks to the saturation of the input, turns out to be limited between the values $-12V$ and $12V$. Recurring to the discussion of 3.8, it is possible to verify that the observer system meets the required specifications, in particular, it is possible to visualize the error trend between the estimated state and the actual state:

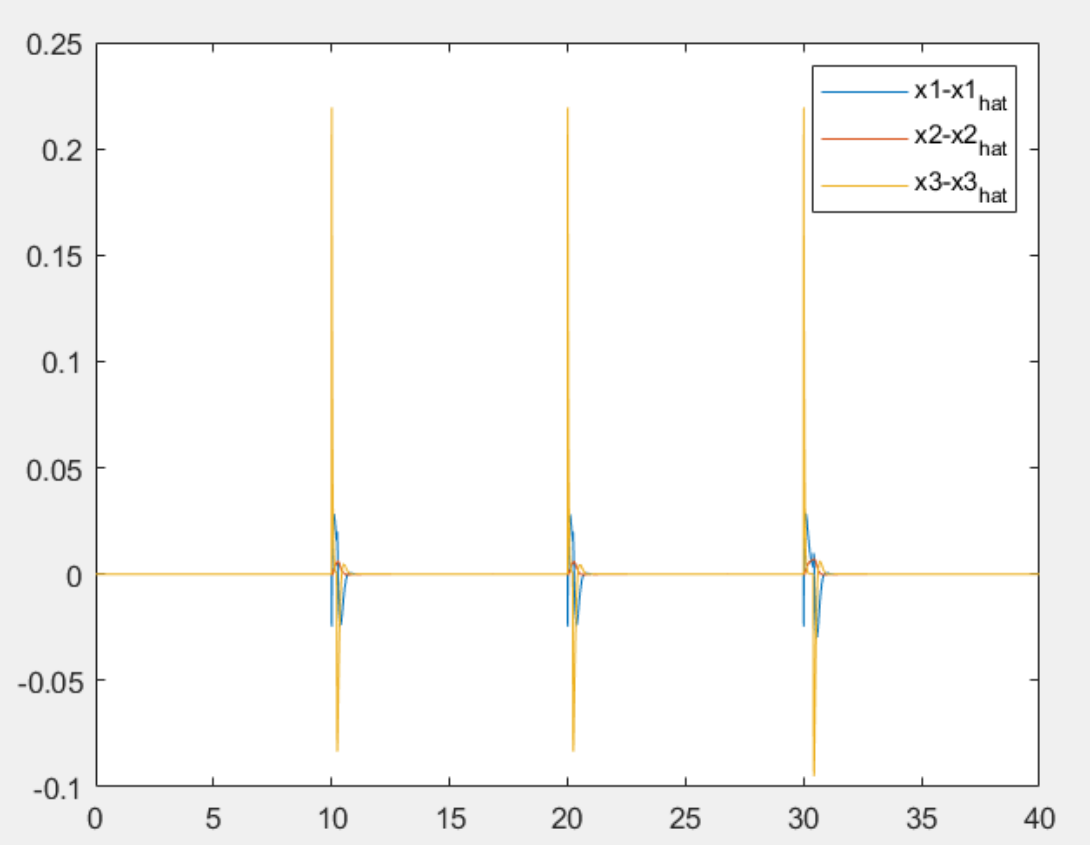


Figure 3.7: Error between estimated state and actual state

Notice that that the difference between the estimated and actual states is imperceptible except during the transient, but at steady state the error is practically zero.
It is also shown that the performance of the system is satisfing and in line with what has been required during the observer design phase.

### 3.10.2 MIL results - with disturbance

The same considerations made for the system without disturbance are also valid for the system with disturbance; in particular, it is possible to visualize the response pattern of the system with the addition of a disturbance step acting at $t = 15$ seconds and of amplitude 6:

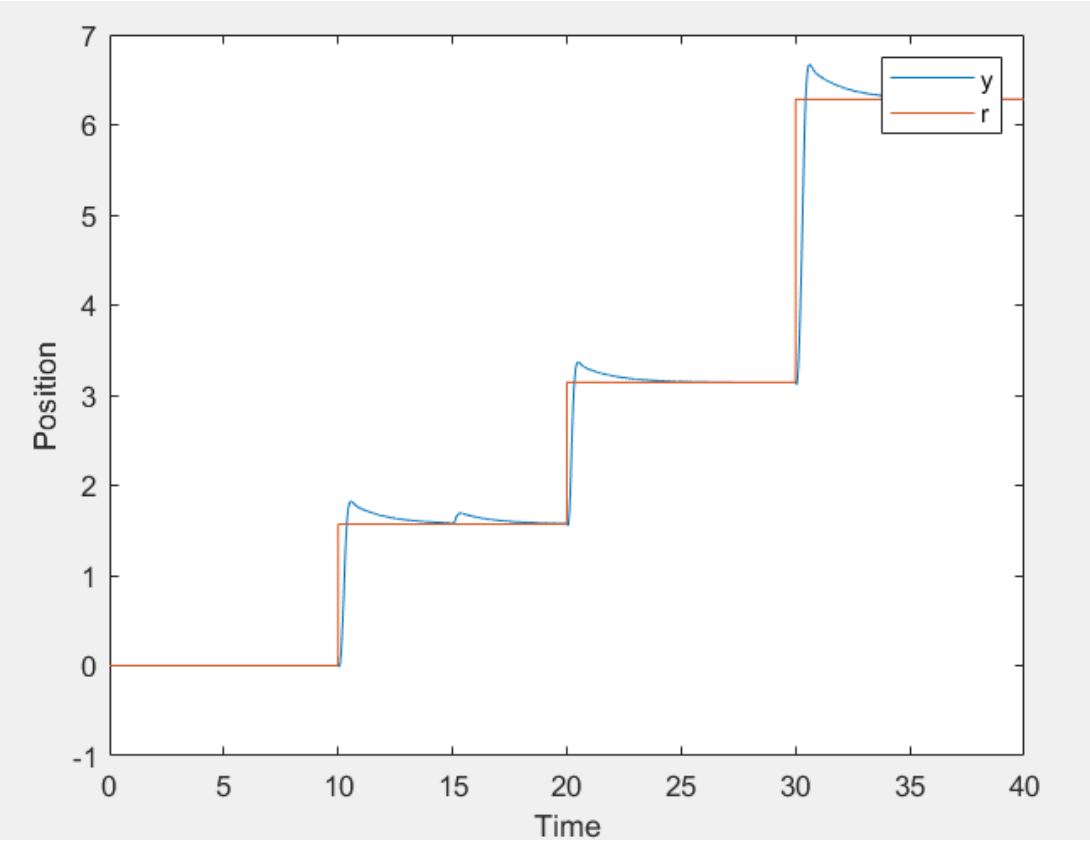


Figure 3.8: System response with disturbance

The trend of the control action turns out to be:

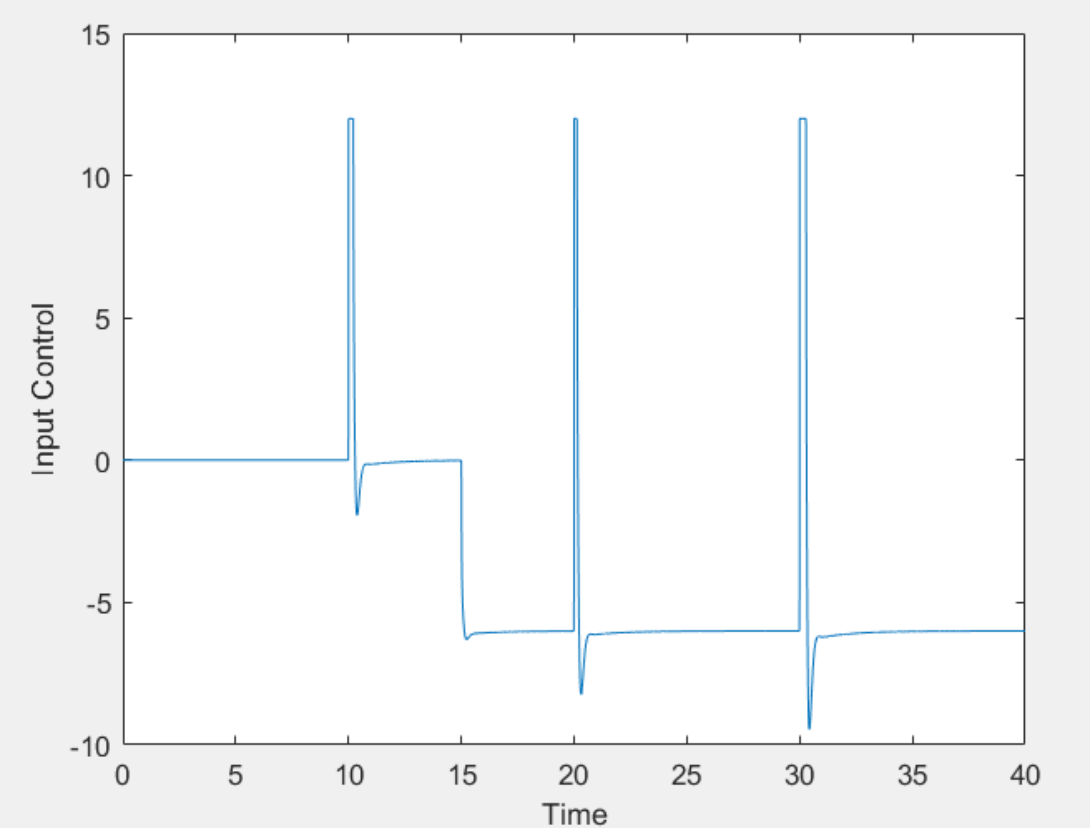


Figure 3.9: Control action with disturbance

Notice that the control action is able to compensate for the disturbance, and the system turns out to be

stable and able to follow the reference.

As for the error between the estimated state and the actual state, we have:

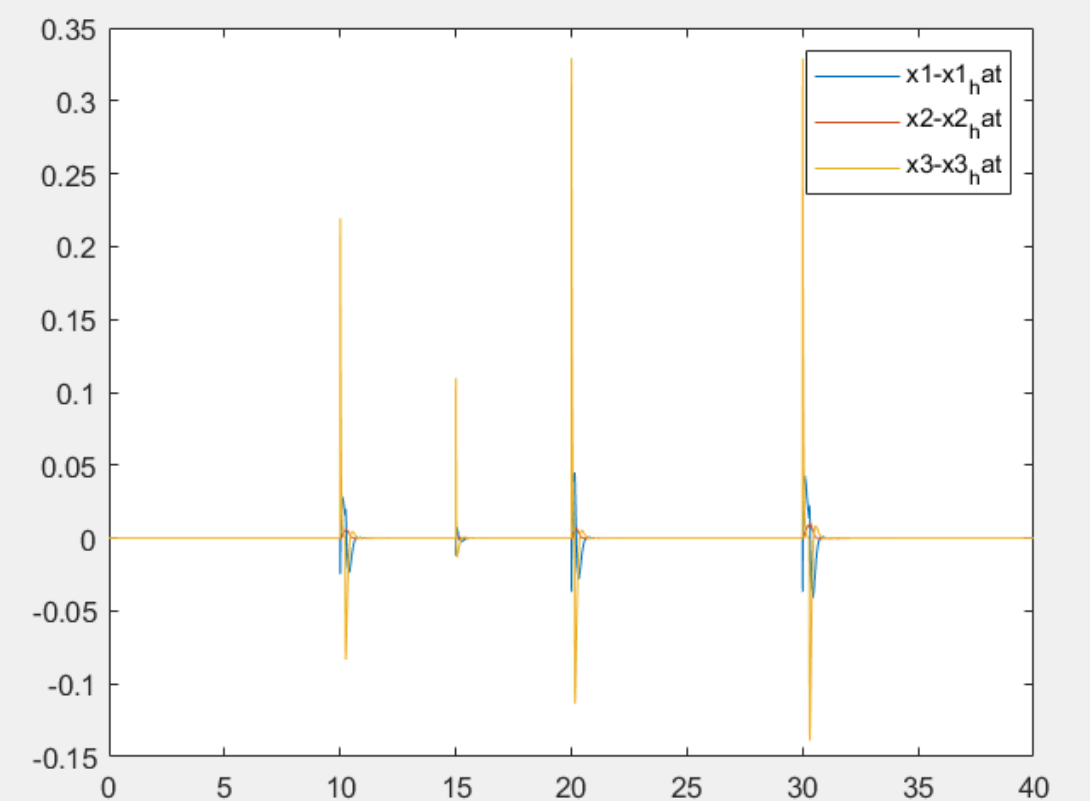


Figure 3.10: Error between estimated state and actual state with disturbance

As a result, even in the presence of disturbance, both the controller and the observer meet the required specifications.

An important note is that, in our design, we decided not to consider a periodic reference, since it seemed legitimate to consider using references greater than $2\pi$ for our system, since the motor is capable of making more revolutions than $2\pi$ .

## 3.11 SIL

Following the MIL phase, it is possible to proceed with the SIL phase, or Software In the Loop.

In this phase, the code generated by MATLAB for the controller and state observer is implemented and tested in a simulation environment.

To do this, a special file is created to compare the output of the simulated system with the output of the system implemented in C code through the SIL/PIL mode that the MATLAB environment itself provides:

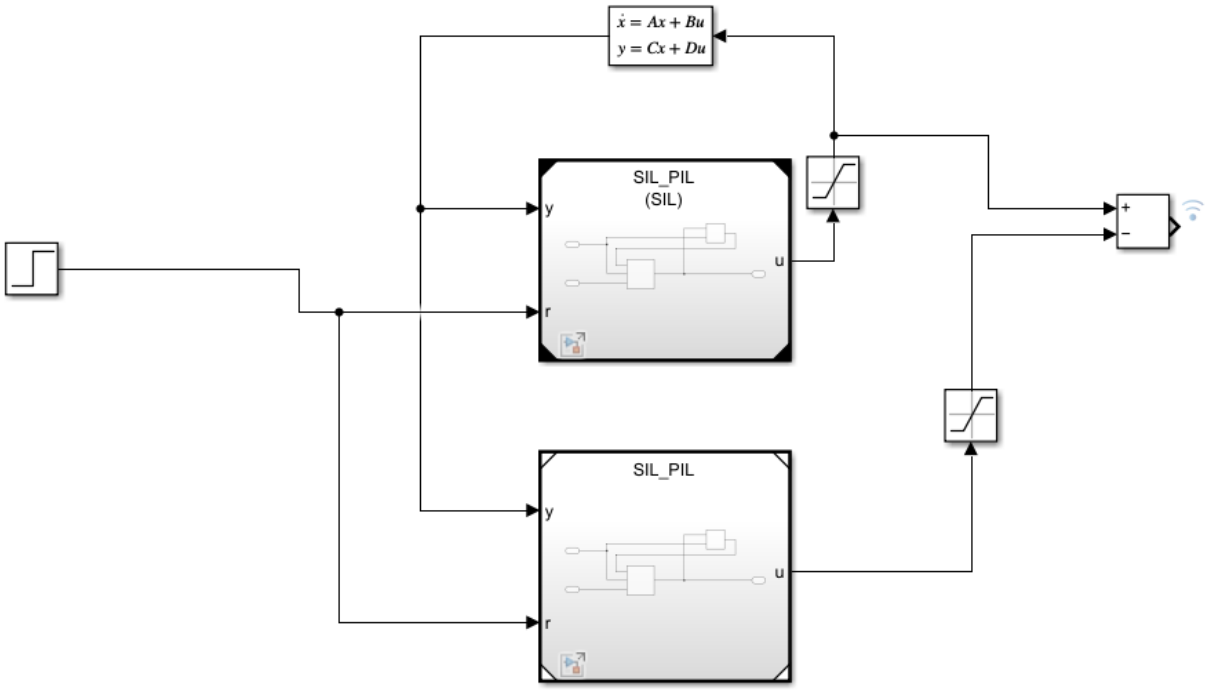


Figure 3.11: Simulink diagram for SIL simulation

In which the behavior of the discretized controller and state observer can be simulated:

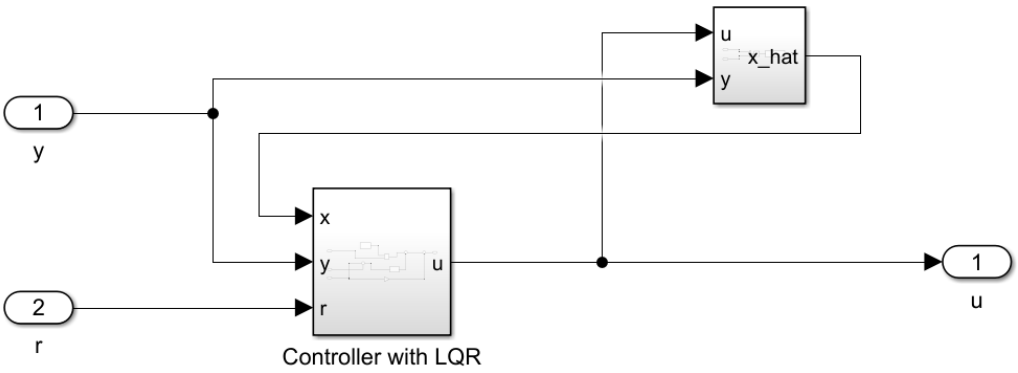


Figure 3.12: Scheme of the controller and observer for SIL simulation

The goal is to show that the output of the simulated system and the output of the system implemented in C code are superimposable, demonstrating that the code generated by MATLAB is correct.
In our case, the reference was set to $\pi$, and since the system's ability to reject disturbances has already been demonstrated, no disturbance was introduced.
By running the simulation, it is possible to visualize the trend of the difference of the output of the simulated system and the output of the system implemented in C code:

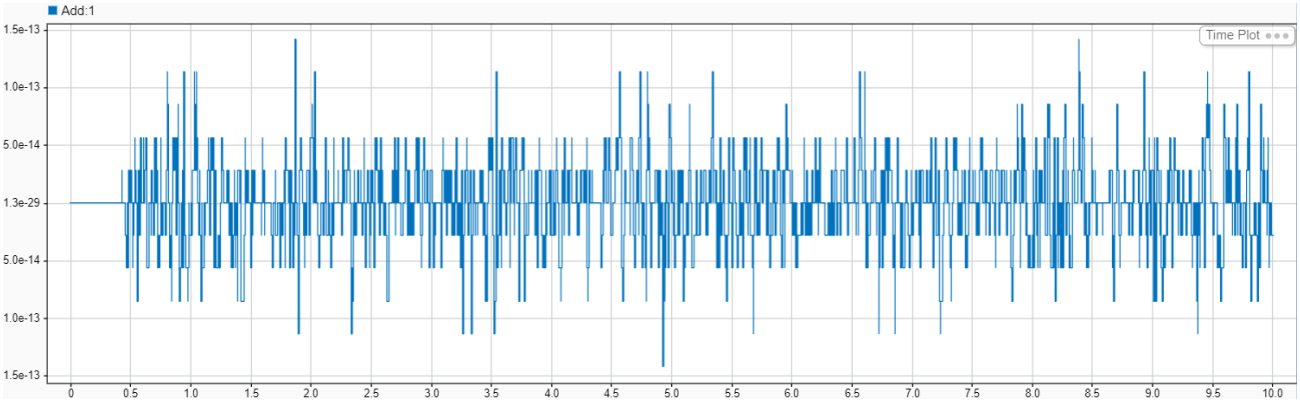


Figure 3.13: Error between simulated system and system implemented in C code on Windows machine

As can be seen, the error is on the order of $10^{-15}/10^{-29}$, demonstrating that the code generated by MATLAB is correct and implementable on the provided microcontroller.

## 3.12 PIL

The next step for the design is to execute the PIL mode, Process in the Loop. Specifically, the same file created for the SIL simulation can be configured to also test in PIL mode by connecting the appropriate hardware, i.e., the microcontroller on which to run the code. Again, the purpose here is to verify that the output of the simulated system and the output of the implemented system are superimposable, but of greater interest is to verify that the sampling time chosen is sufficient to ensure proper execution of the code.

In particular, it is possible to visualize the error trend between the output of the simulated system and the output of the implemented system in C code:

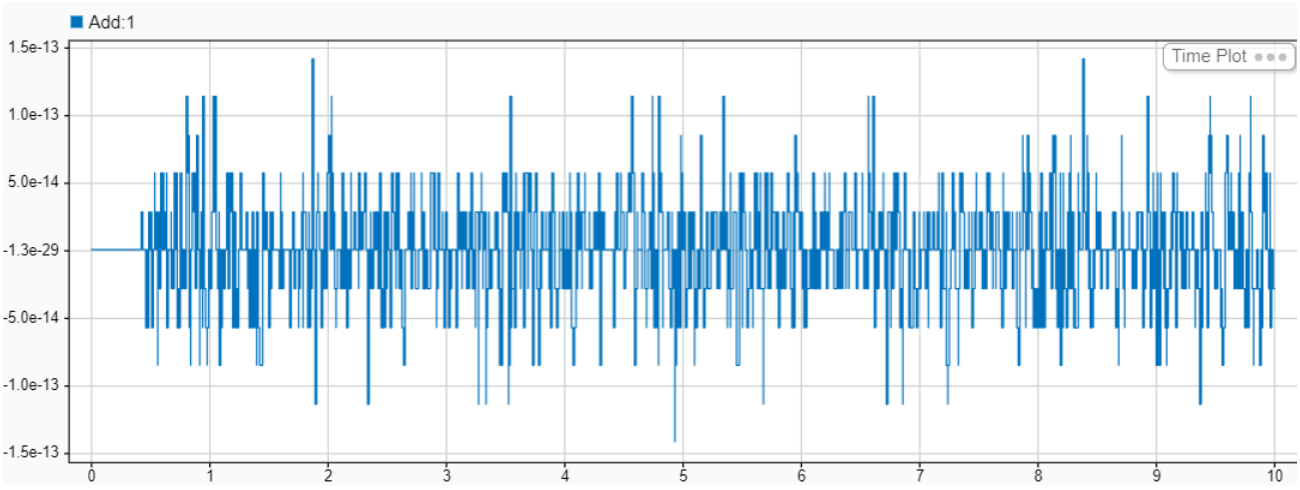


Figure 3.14: Error between simulated system and system implemented in C code on the microcontroller

From the generated results, not only is the difference between the simulation and PIL imperceptible, as in the previous case, but thanks to MATLAB's *CodeProfileAnalyzer* mode, it is possible to verify that

the chosen sampling time is sufficient to ensure proper code execution:

Task Execution Times

| Section Name | Maximum Execution Time | Average Execution Time | Maximum Self Time | Average Self Time | Calls |
|---|---|---|---|---|---|
| SIL_PIL_initialize | 0.0015 | 0.0015 | 0.0015 | 0.0015 | 1 |
| SIL_PIL_Init | 0.0021 | 0.0021 | 0.0021 | 0.0021 | 1 |
| SIL_PIL [0.005 0] | 0.0780 | 0.0775 | 0.0780 | 0.0775 | 2001 |

Figure 3.15: Code Profile Analyzer

As can be seen, the time taken to execute the code is $0.0775ms$ which is equal to about 1.55% of the chosen sampling time, demonstrating that the chosen sampling time is sufficient to ensure the proper execution of the code.

## 3.13 On the hardware

Finally, the last step is to actually use the process within the control loop. The corresponding simulink scheme is as follows:

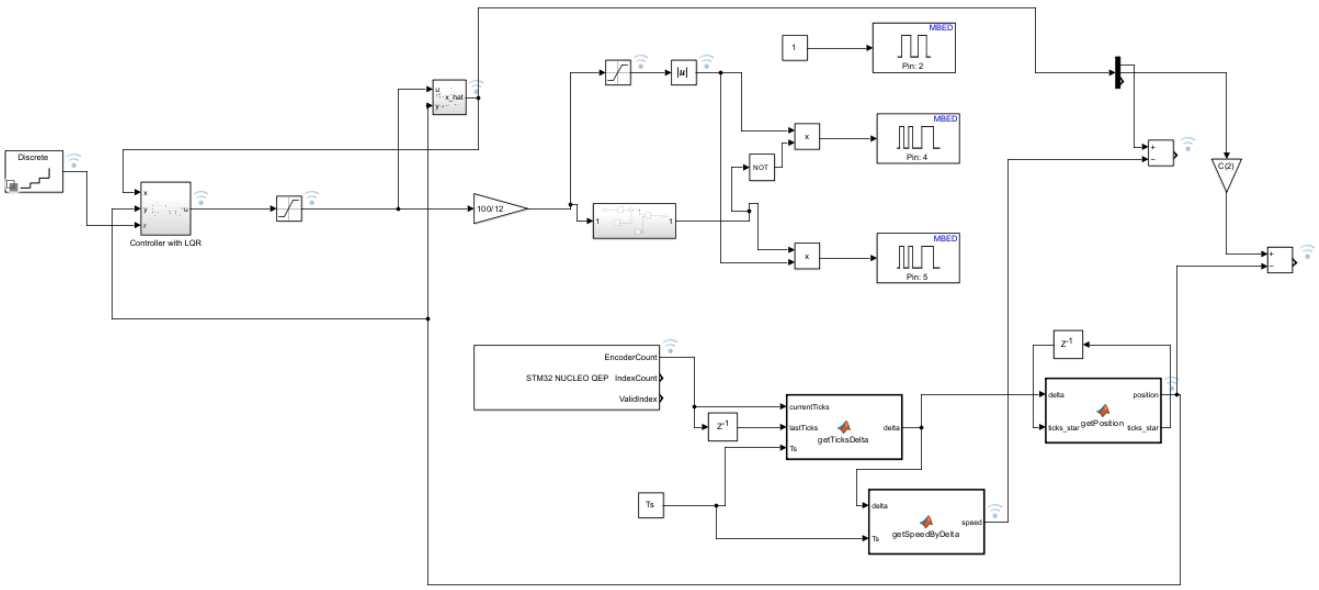


Figure 3.16: Simulink scheme for the real system

It should be noted that the functions provided during the course were exploited for reading the angular position of the motor, with the particular difference that the periodicity of the position was removed as introduced in the 3.10.2 section; in addition, the function for calculating the velocity is not used, given the presence of the observer, but has been retained to make comparisons with the variable estimated by the observer himself. From the schematic, it is possible to visualize the trend of the angular position of the motor as a result of sending a stair reference consisting of values equal to 0 ,$\frac{\pi}{2}$,$\pi$ and $2\pi$:

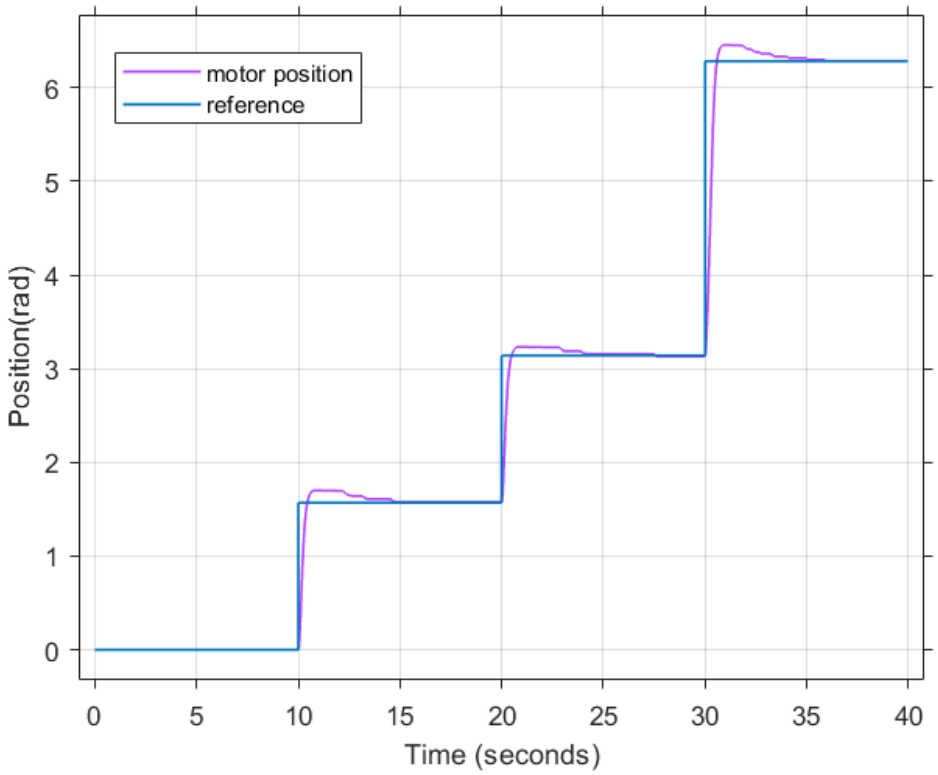


Figure 3.17: Real motor position

As can be seen, the engine follows the reference correctly, presenting a settling time equal to $\approx 3s$. In addition, you can see the control action sent to the motor (correctly scaled to the input pins between $-100$ and 100):

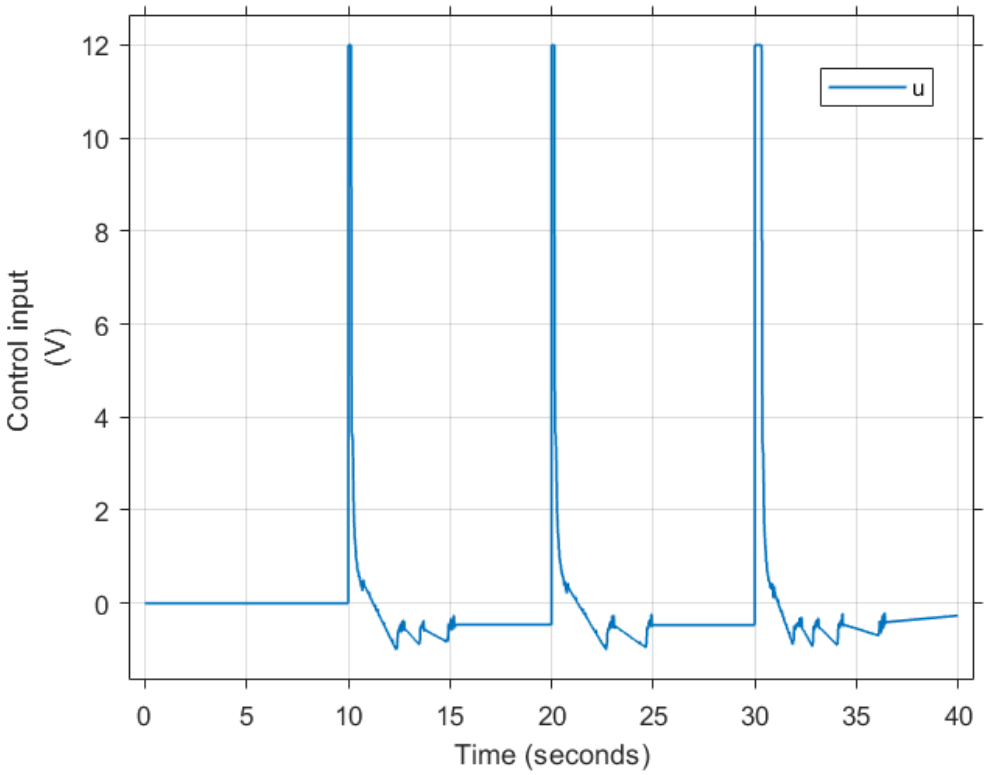


Figure 3.18: Control action sent to the motor

And, in line with what was seen during the previous steps, the error between the position estimated by the observer and the actual position of the motor can be visualized:

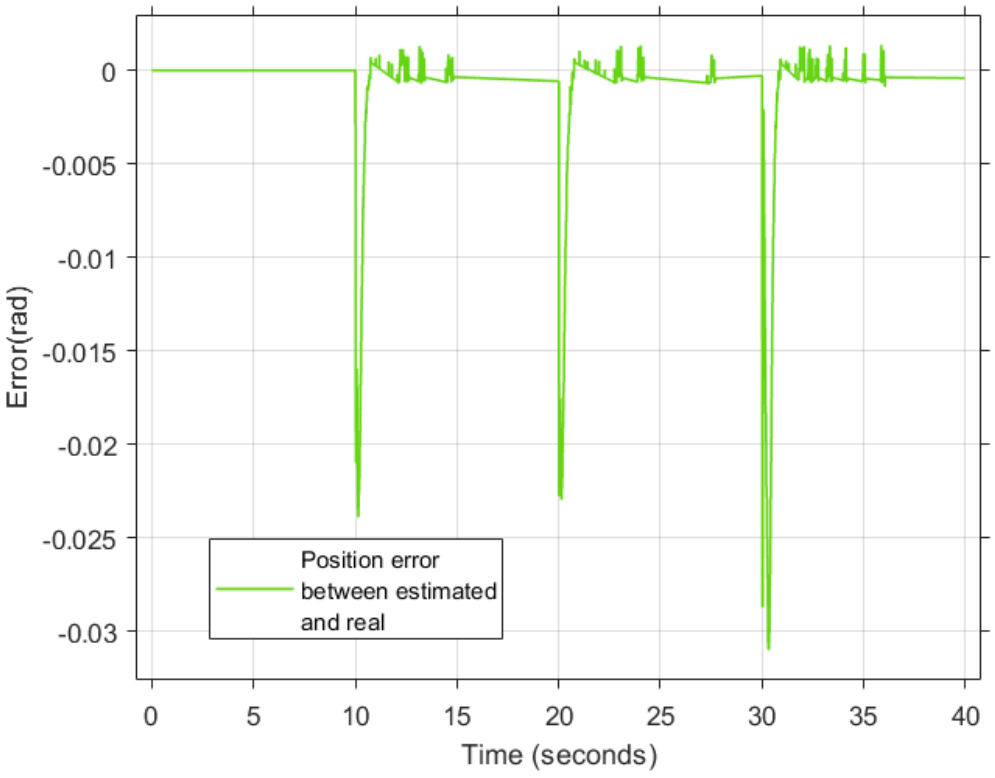


Figure 3.19: Error between estimated position and actual position of the motor

And also regarding the speed estimated by the observer and the actual speed of the motor:

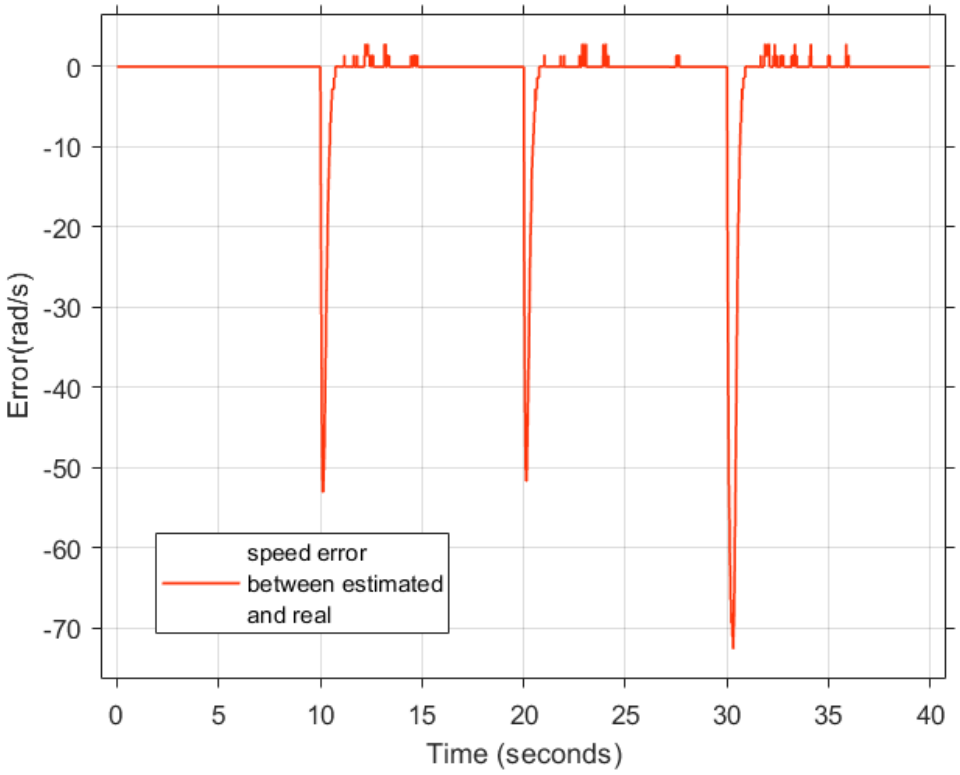


Figure 3.20: Error between estimated speed and actual speed of the motor

It can be said that, in line with what happens during the previous steps, the system meets the required specifications, and the controller and the state observer are able to ensure the correct operation of the system.

To get a clearer view of the difference between what we get in simulation and what we get in reality, you can view the comparison performed in 5.1.

## 3.14 Differences between integral action and anti-windup term

In this section we want to highlight the differences between the system output with the integral action and the system output with the anti-windup action, remembering that the term anti-windup was chosen to be introduced in the design. To perform this analysis, the Simulink scheme for MIL simulation was used, but we modified the disturbance term to be a square-wave signal, so that the differences between the two systems could be visualized as a result of the action of a very high and constant disturbance that would saturate the integral action.

Afterthe effect of the disturbance, which acts at $t = 10$ seconds and ends at $t = 20$ seconds, it is possible to visualize the response pattern of the system with the integral action:

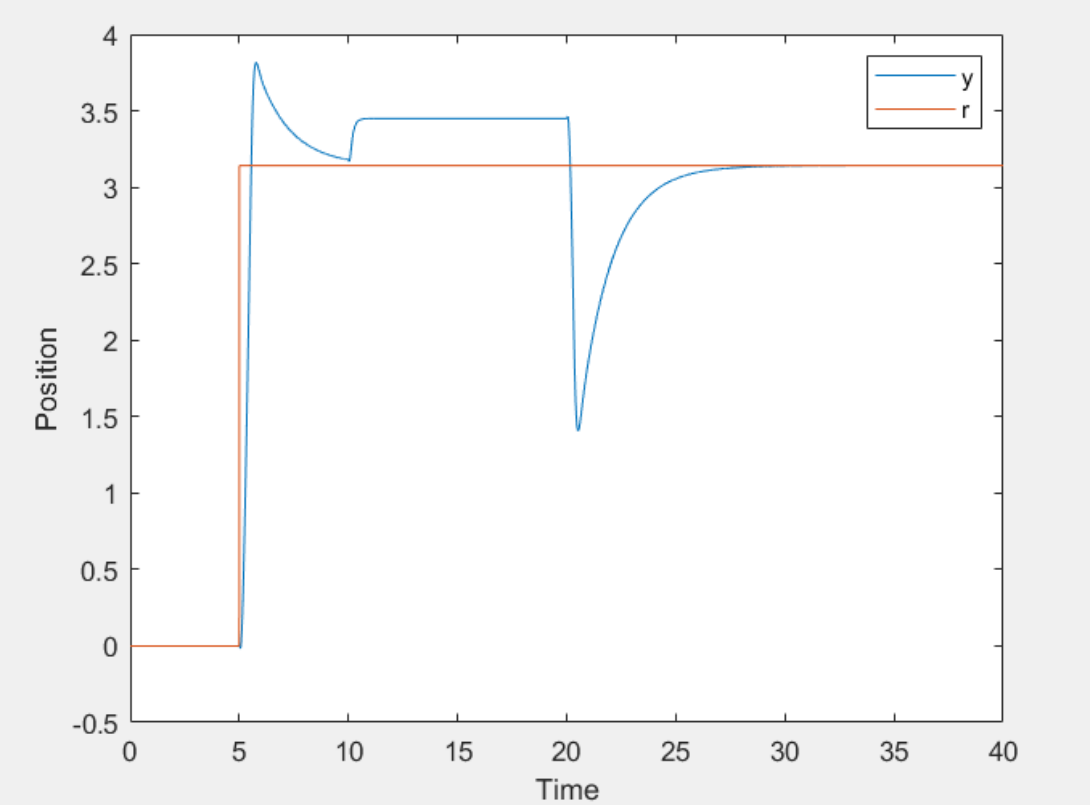


Figure 3.21: System response with integral action

and the trend of the system response with anti-windup action:

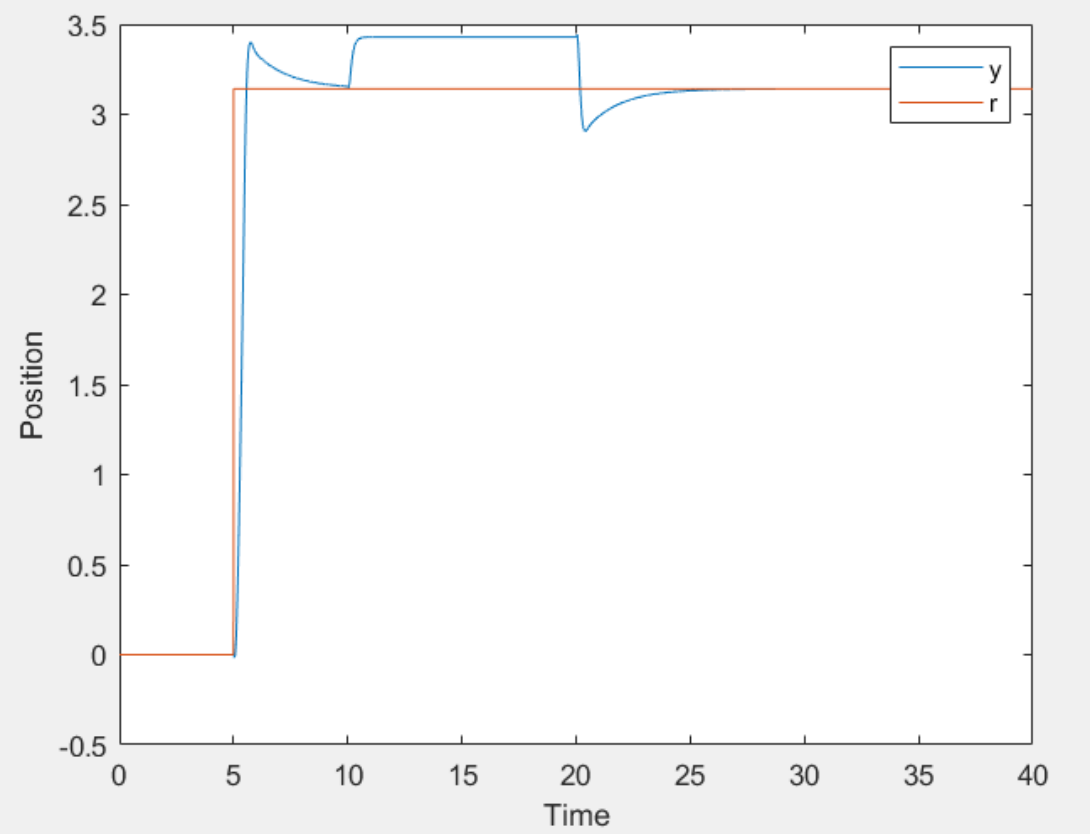


Figure 3.22: System response with anti-windup action

It can be seen that, following the action of the disturbance, the system without the anti-windup term,

loaded with saturation, takes a longer time to return to the reference than the system with the anti-windup term.

## 3.15 Controller Limitations

At the end of the controller design, it can be seen that the system is able to follow the desired reference, even in the presence of disturbances, and that the state observer is able to correctly estimate the system state.

However, it is possible to note that the system has limitations, in particular, the type of controller chosen, namely an LQR controller, is unable to manage the given constraints on the system input.

To do so, it is possible to choose a different type of controller, such as an MPC controller, which can manage them.

This, however, was not the goal of the project, as the LQR controller can guarantee satisfactory performance, both on the real and simulated system.

# CHAPTER 4

# TORQUE CONTROL

The purpose of this chapter is to design a controller that can ensure that the electric motor can deliver the desired torque. To do this, it is necessary to model the electrical and mechanical system of the motor. A torque motor can be modeled according to the following block diagram [1]:

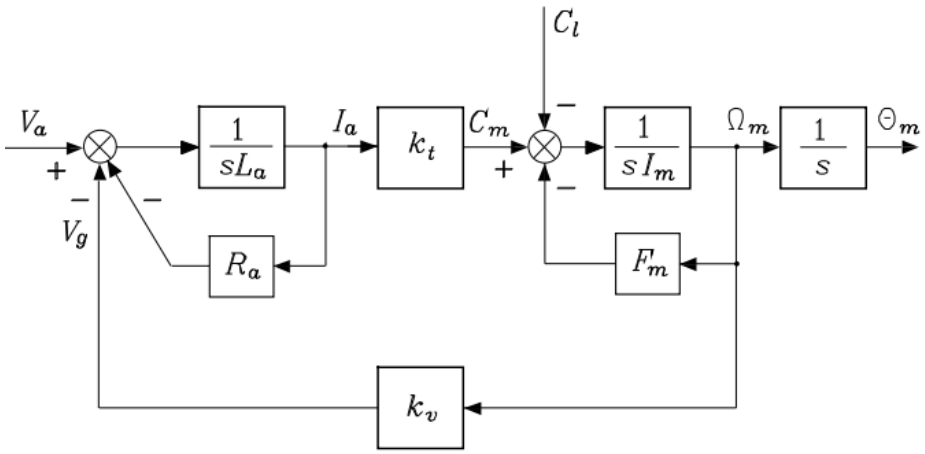


Figure 4.1: Model of a Torque Motor

Where:

- $I_m$ is the inertia of the motor;
- $F_m$ is the coefficient of viscous friction;
- $K_t$ is the torque constant;
- $K_e$ is the electromotive force constant;
- $R$ is the resistance of the motor;
- $L_a$ is the inductance of the motor;

- $V_a$ is the voltage applied to the motor;
- $C_{load}$ is the load applied to the motor.

The mathematical model of the motor is given by the following differential equations described in 2.1

In order to design the controller, it is noted that it is not necessary to know the values of the inertia $I_m$ and the viscous friction coefficient $F_m$ of the motor, since they do not affect the closed-loop behavior of the system of torque evolution. Instead, it is necessary to know the values of the torque constant $K_t$, resistance $R$ and inductance $L_a$ of the motor. So it is necessary to make experimental measurements to derive the values of the system parameters.

## 4.1 Parameter Estimation

### 4.1.1 Estimation of R and k

The parameter $R$ was measured directly with a multimeter by measuring the resistance between the two motor terminals. This measurement gave a value of $R = 2.0\Omega$. For the estimation of the torque constant $K_t$, which is assumed to be equal to the electromotive force constant $K_e$ and will henceforth be called just $k$, it was necessary to make experimental measurements. In particular, the same procedure as described in 2.3 was followed to estimate the torque constant. The results obtained were $K_m = \frac{1}{k} = 0.6876$, so $k = 1.4543 Nm/A$.

## 4.2 Measurement of the current

To obtain the estimate of the inductance $L_a$ of the motor, it is necessary to make experimental measurements using the current sensor provided to us. For this purpose, the interface with the current sensor and the microcontroller was made. As stated on the datasheet, the sensor must be connected in series with the motor circuit so that the current flows in the direction indicated by the sensor. To measure the current, it was necessary to calibrate the sensor because the value read from the analog pin did not match the actual current. The datasheet states that for the ACS712 sensor the output voltage value is about 2.5V for 0A, therefore, it was necessary to measure the output voltage value for a current of 0A so as to attest to the correct offset of the sensor at zero current. For this purpose, many output voltage values for currents of 0A were collected and averaged. The value obtained was $V_{0A} = 2.5530V$. The procedure can be summarized with the graph in 4.2. These steps are documented in the file *Embedded_controller_project/sensor_calibration/sensor_calibration.m*.

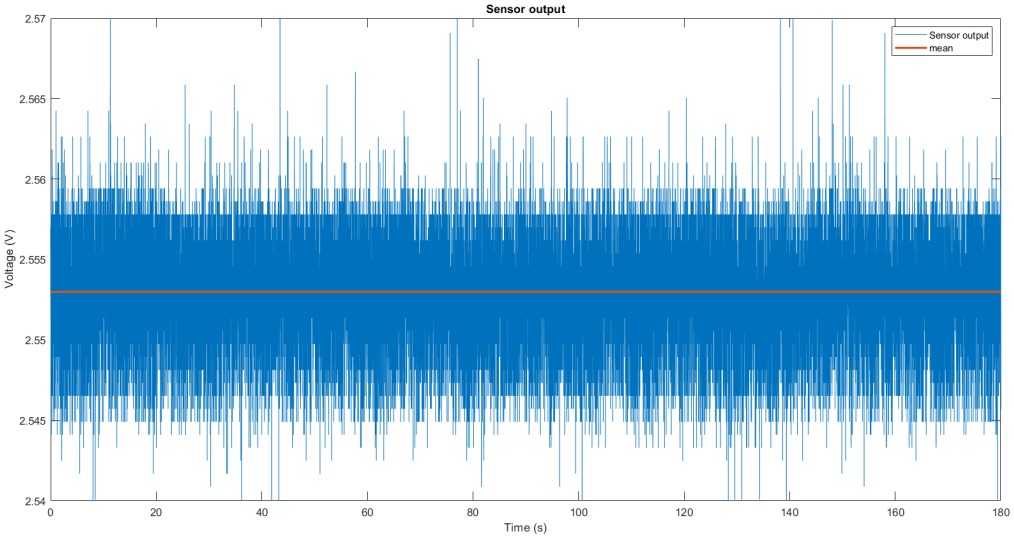


Figure 4.2: Calibration of the current sensor

According to the documentation, the analog pin on the STM32 board reads values between 0 and 1. Where, the maximum value corresponds to the microcontroller's reference voltage, which is $V_{ref} = 3.3V$. So, all read values must be rescaled by multiplying by $V_{ref}$. At this point, to obtain current, the algorithm 4.1 is run, which yields the actual current output from the sensor.

Listing 4.1: Algorithm for current calculation

```
1    current = (analogRead(pin) * Vref) / (2^pin_resolution-1);
2    current = (current - V_0A) / Sensitivity;
```

### 4.2.1 First order lag filter

As can be seen from the graph 4.2, the values are affected by noise, so a filter had to be applied to reduce the noise. The filter used is a first order lag filter, whose objective is to remove, from an input signal, the high-frequency components, the transfer function of the filter is given by:

$$T(f) = \frac{1}{T_f s + 1} \tag{4.1}$$

where $T_f$ is the time constant of the filter, at least equal to $T_f = \frac{T_s}{2}$, with $T_s$ the sampling time, to guarantee the Nyquist-Shannon sampling theorem.
Considering the differential filter equation, we have:

$$T_f \frac{d\hat{y}}{dt} + \hat{y} = y \tag{4.2}$$

where $\hat{y}$ is the output of the filter, while y is the input of the filter as well as our data.
An exact solution of the differential equation is given by:

$$\hat{y}(k+1) = e^{-\frac{T_{fs}}{T_f}} \hat{y}(k) + (1 - e^{-\frac{T_{fs}}{T_f}}) y(k) \tag{4.3}$$

where $T_{fs}$ is the filter sampling time.
Having $\alpha = e^{-\frac{T_{fs}}{T_f}}$, we can write the equation as:

$$\hat{y}(k+1) = \alpha\hat{y}(k) + (1-\alpha)y(k) \tag{4.4}$$

and applying a time shift of one sample, we obtain:

$$\hat{y}(k) = \alpha\hat{y}(k-1) + (1-\alpha)y(k-1) \tag{4.5}$$

This filter was implemented in a Matlab Function, choosing $T_s = 0.005$, $T_f = \frac{T_s}{2}$ so the aliasing effect is reduced by about 90% for white noise, and $T_{fs} = 0.000125$, 20 times lower than $T_f$, so we get $\alpha = 0.9512$, the results can be appreciated in the graph 4.3. Choosing this values for the filter, we can weight more the filtered value than the new value, so we can reduce the noise.

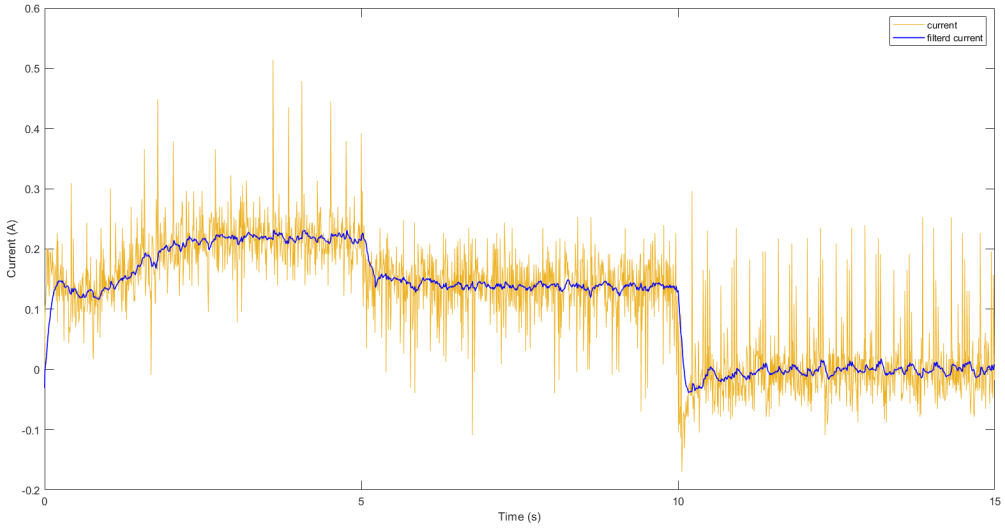


Figure 4.3: Calibration of the current sensor with filter

### 4.2.2 Estimation of $L_a$

To estimate the inductance of the motor, there are several methods, but they require the use of instruments such as oscilloscopes, signal generators or impedance meters. In this case, having none of these tools available, we tried to derive the inductance from the differential equations of the motor.
Specifically, the inductance is given by:

$$L_a = \frac{V_a - R \cdot i - k \cdot \omega}{\frac{di}{dt}} \tag{4.6}$$

Where $V_a$ is the voltage applied to the motor, $R$ is the estimated resistance of the motor, $k$ is the estimated torque constant, $\omega$ is the measured angular velocity of the motor, $i$ is the measured current flowing in the motor, and $\frac{di}{dt}$ is the derivative of the current with respect to time.
At this point, a module was built in Simulink for acquiring the data needed for inductance estimation, as

documented in the file *Embedded_controller_project/inductance/Tuning_L.slx*.
The acquired data were stored and loaded into Matlab, where the Matlab module *Embedded_controller_project/inductance/Inductance.m* tries to calculate the inductance by filtering the data appropriately and then calculating the numerical derivative of the current.
Unfortunately, the results obtained were highly variable and not consistent with the expected values, so it was decided not to use this method to estimate the inductance and to consider the inductance as a known parameter provided by the motor manufacturer, which is approximately $L_a = 2.3mH$.
Having a value of $L_a$ that is very different from the actual value is not a problem for torque control, since a well-designed controller can guarantee the desired performance even in the presence of parametric uncertainties. So, the motor model (in LaPlace's domain) becomes:

$$\frac{\omega(s)}{V_a(s)} = \frac{k}{L_a s + R} = \frac{1.4543}{0.0023s + 2} \tag{4.7}$$

## 4.3 MIL

Again, the V-Model methodology will be adopted for the implementation of the project, which involves a verification phase for each phase of the project, thus ensuring that the system meets the required specifications. The verification phases will use simulation methods such as MIL, SIL and PIL, leading up to the implementation of the system on the provided microcontroller and the testing of the system under real conditions. First, the model of the electric motor was built in Simulink, as documented by the file *Embedded_controller_project/torque/motor_model.slx*. The model was built from the differential equations of the electric motor, as described in 2.1, in order to be able to test the previously estimated parameters. For the design of the controller, an important assumption was made: only the electrical part of the motor was considered for the definition of the model to be controlled, whereas, the mechanical part was considered as a slowly varying external disturbance, comparable to a step. This assumption can be justified by the fact that the mechanical part of the motor is much slower than the electrical part, therefore, it is reasonable to consider it as an external disturbance.
At this point, the high-level requirements that the closed-loop system must meet can be summarized as follow:

- The system must reject stepped disturbances;
- The system must be fast enough to ensure an acceptable dynamic response;
- The system must be stable.

To ensure that the system meets the requirements, a PI controller is proposed, as it is able to ensure good dynamic response and rejection of step disturbances due to the presence of the integrator. The PI controller was designed using the root locus method, as documented in the file *Embedded_controller_project/torque/PI_controller.m*. In addition, an acceptable response speed is on the

order of $0.1s$. Accordingly, the designed controller is:

$$C(s) = K_p + \frac{K_i}{s} = \frac{0.5625 * (s + 80)}{s} \tag{4.8}$$

This controller guarantees that the closed-loop settling time is: $t_s = 0.2s$ and the frequency of the closed loop system is $\omega_3 = 23.4rad/s$, which gives us for a sampling time $T_s = 0.005s$: $\omega_n = \frac{2}{T_s} = 1256.6rad/s >> 6\omega_3$. In figure 4.4, it is possible to appreciate the response of the closed-loop system to the unit step.

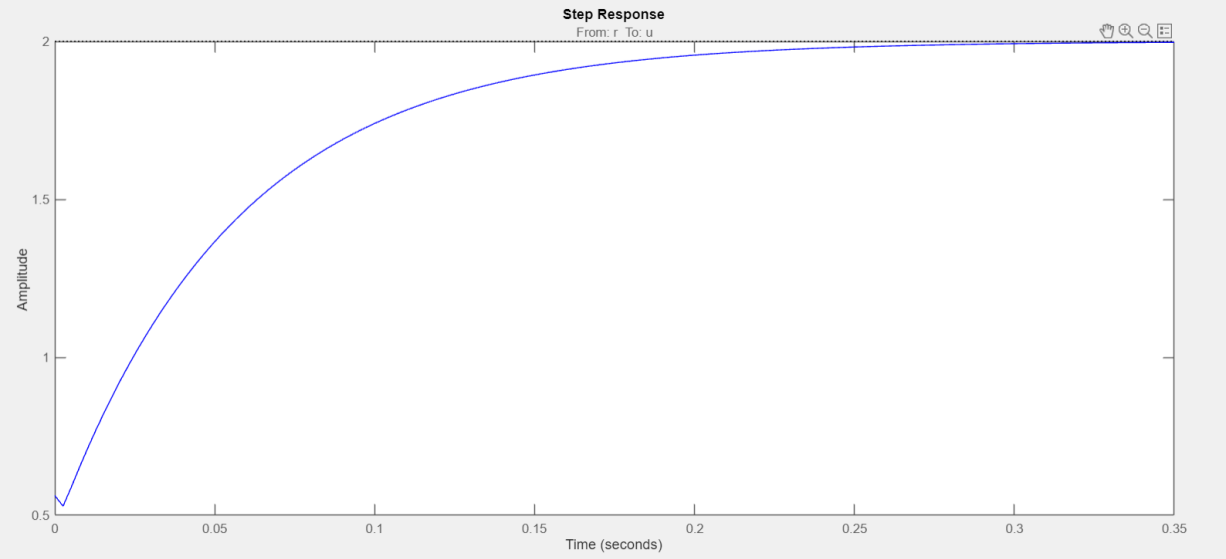


Figure 4.4: Step response of the system

We now proceed to discretize the controller, using Tustin's method. The discretized controller is:

$$C_d(z) = \frac{1.792z - 1.542}{z - 1} \tag{4.9}$$

whereas, the steps described in 3.6 were followed for the implementation of the antiwindup. The simulink schematic of the closed-loop system is documented in the file *Embedded_controller_project/torque/Torque_controller_MIL.slx*, as shown in figure 4.5.

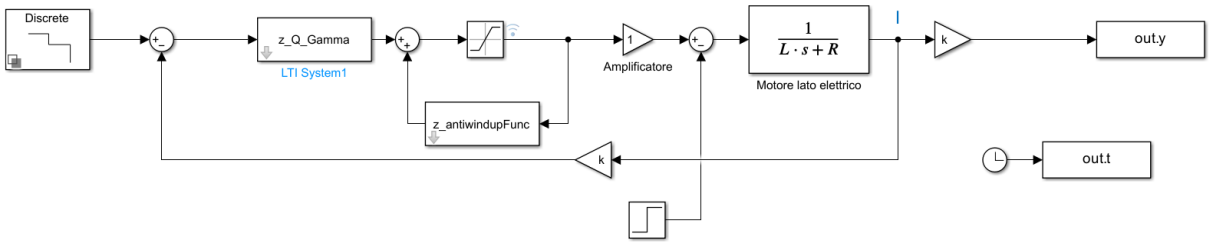


Figure 4.5: Simulink model of the system in closed loop

The closed-loop behavior of the system is shown in figure 4.6, where it can be seen that the system is stable and chases the desired references, while the control effort is shown in figure 4.7.

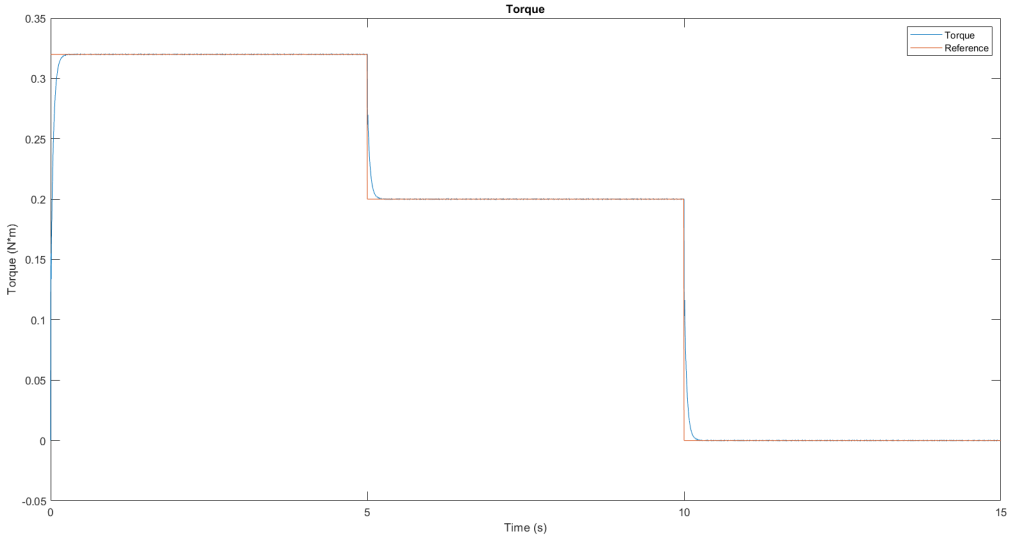


Figure 4.6: Step response of the system in closed loop

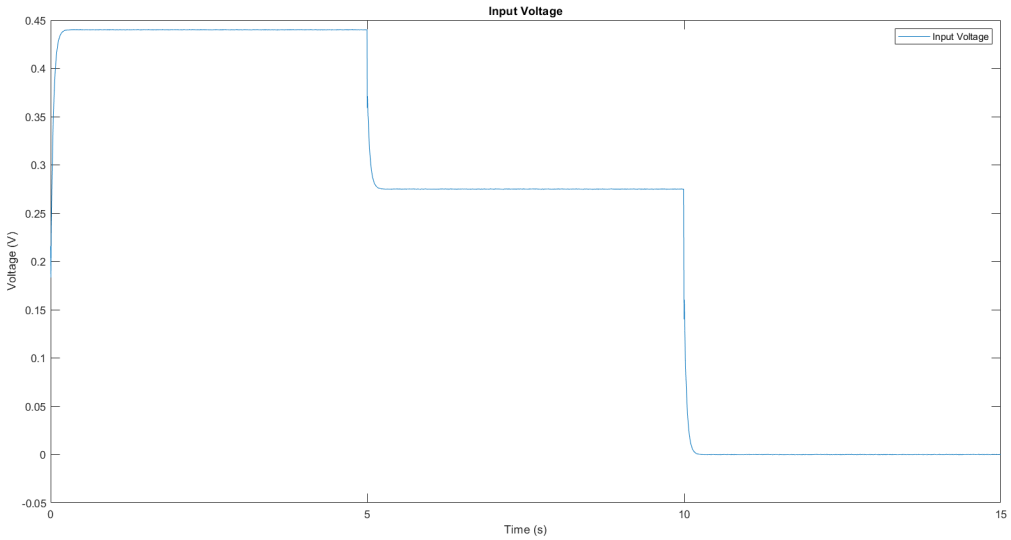


Figure 4.7: Input of the system in closed loop

In addition, it was verified that the system meets the requirements in that it is capable of rejection of stepped disturbances, as shown in the figure 4.8.

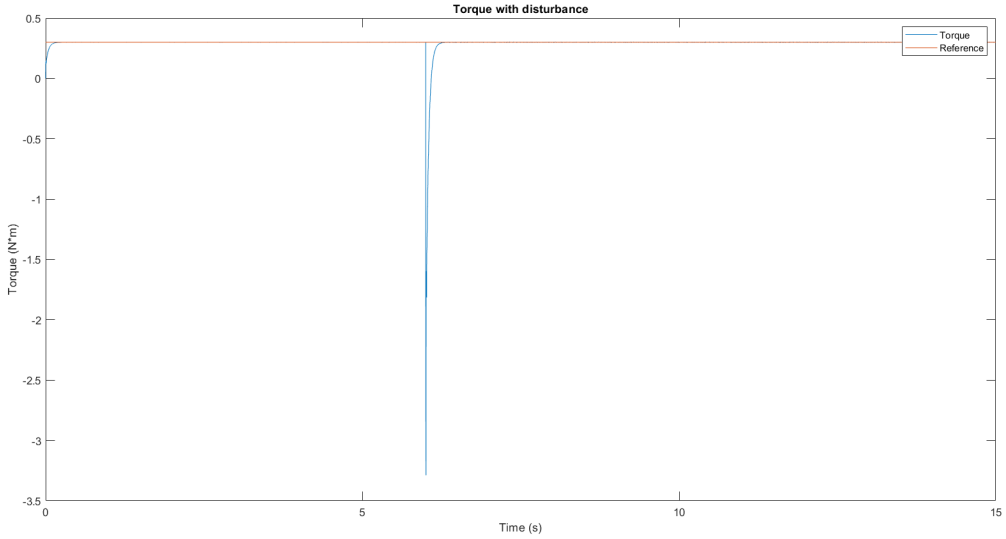


Figure 4.8: Disturbance rejection of the system in closed loop

## 4.4 SIL

Following the MIL phase, it is possible to proceed with the SIL phase.
In this phase, the code generated by MATLAB for the controller and state observer is implemented and tested in a simulation environment.
To do this, a special file is created to compare the output of the simulated system with the output of the system implemented in C code through the SIL/PIL mode that the MATLAB environment itself provides:

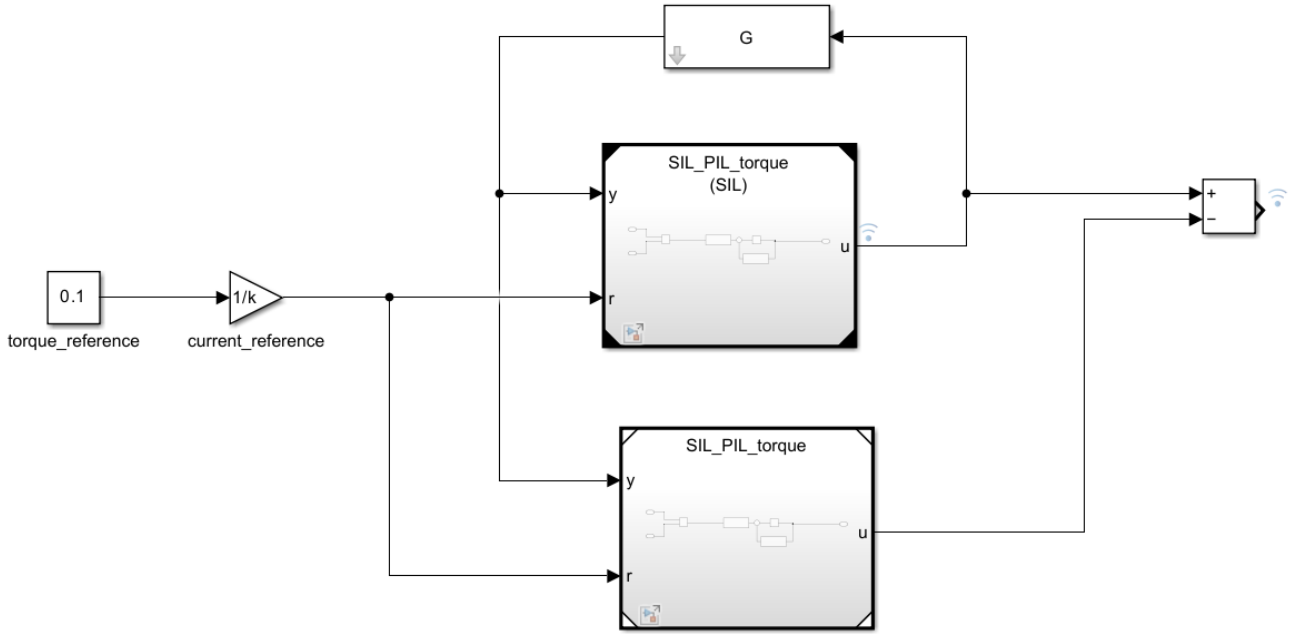


Figure 4.9: Simulink diagram for SIL simulation of the torque control system

In which the behavior of the discretized controller can be simulated:

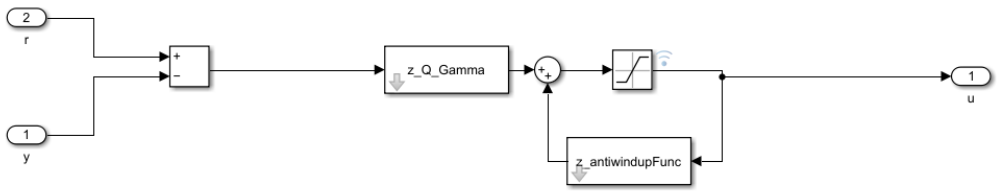


Figure 4.10: Scheme of the controller SIL simulation of the torque control system

The goal is to show that the output of the simulated system and the output of the system implemented in C code are superimposable, demonstrating that the code generated by MATLAB is correct.
In our case, the reference was set to 0.1, and since the system's ability to rejection disturbances has already been demonstrated, no disturbance was introduced.
By running the simulation, it is possible to visualize the trend of the difference of the output of the simulated system and the output of the system implemented in C code:

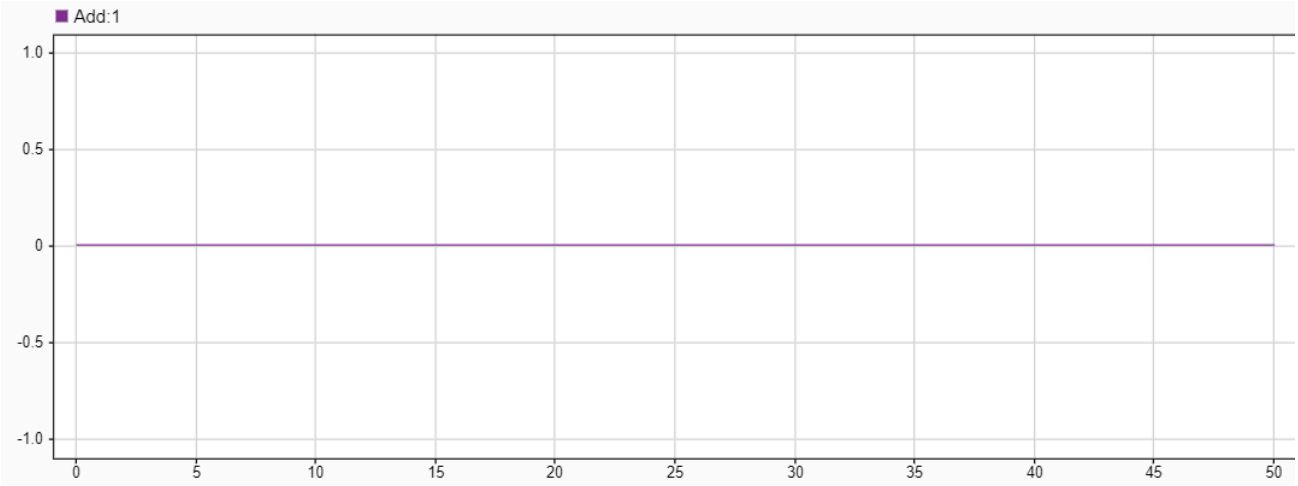


Figure 4.11: Error between simulated system and system implemented in C code on Windows machine of the torque control system

As can be seen, the error is 0, demonstrating that the code generated by MATLAB is correct and implementable on the provided microcontroller.

## 4.5 PIL

As we did for the position control system, we can proceed with the PIL phase.
Again, the purpose here is to verify that the output of the simulated system and the output of the implemented system are superimposable, but of greater interest is to verify that the sampling time chosen is sufficient to ensure proper execution of the code.
In particular, it is possible to visualize the error trend between the output of the simulated system and the output of the implemented system in C code:

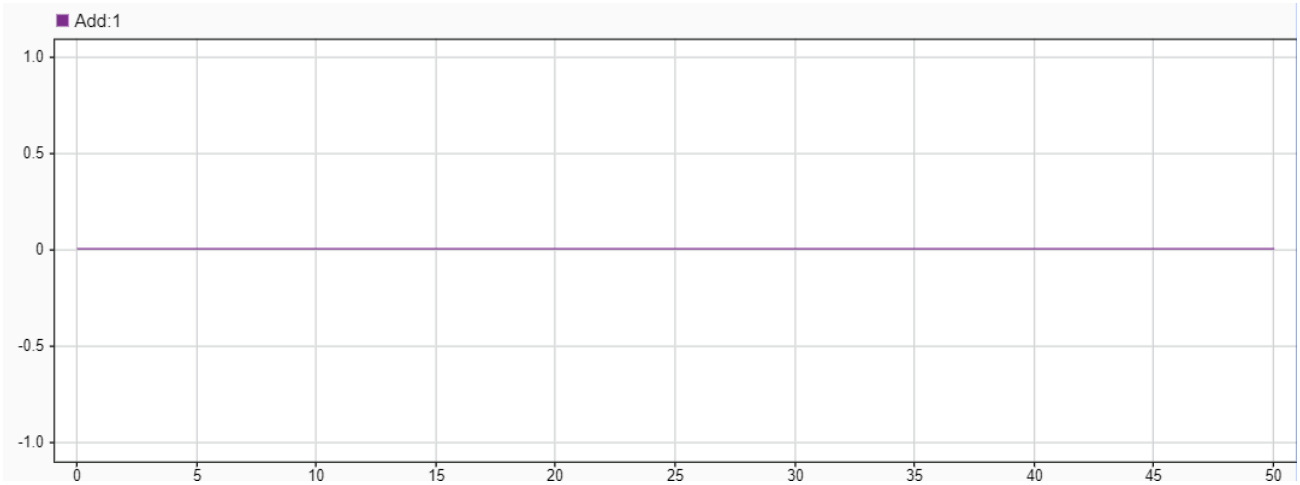


Figure 4.12: Error between simulated system and system implemented in C code on the microcontroller of the torque control system

From the generated results, not only is the difference between the simulation and PIL imperceptible, as in the previous case, but thanks to MATLAB's *CodeProfileAnalyzer* mode, it is possible to verify that the chosen sampling time is sufficient to ensure proper code execution:

| Section Name | Maximum Execution Time | Average Execution Time | Maximum Self Time | Average Self Time | Calls |
|---|---|---|---|---|---|
| SIL_PIL_torque_initialize | 0.0015 | 0.0015 | 0.0015 | 0.0015 | 1 |
| SIL_PIL_torque_Init | 0.0016 | 0.0016 | 0.0016 | 0.0016 | 1 |
| SIL_PIL_torque [0.005 0] | 0.0137 | 0.0134 | 0.0137 | 0.0134 | 10001 |

Figure 4.13: Code Profile Analyzer

As can be seen, the time taken to execute the code is $0.0137ms$ which is equal to about 0.274% of the chosen sampling time, demonstrating that the chosen sampling time is sufficient to ensure the proper execution of the code.

## 4.6 On the hardware

The code generated by MATLAB is implemented on the provided microcontroller and the system is tested in real conditions.
The simulink diagram used is the following:

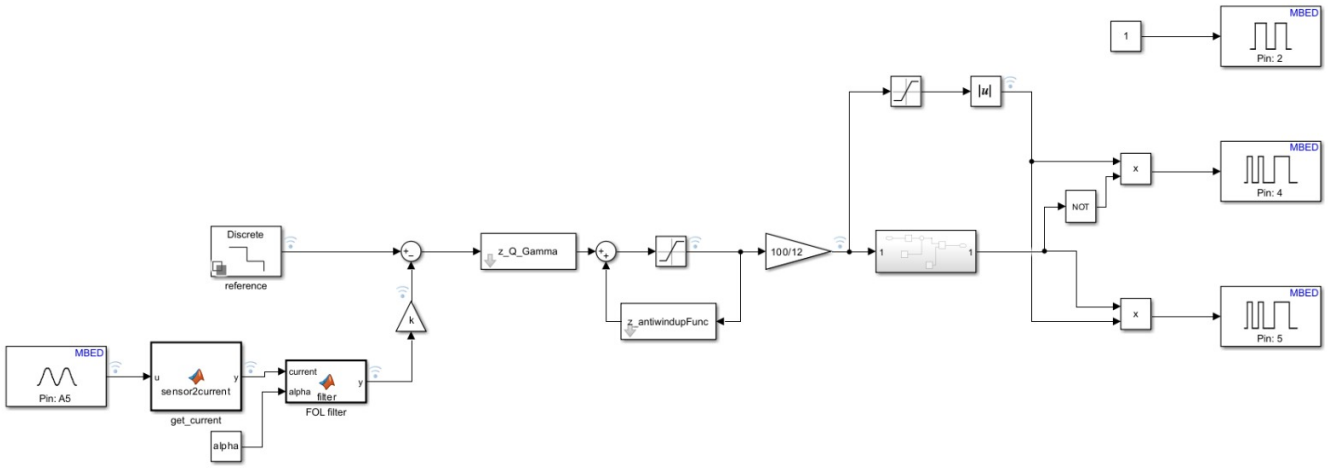


Figure 4.14: Simulink diagram for the torque control system

where we highlight the presence of the *ADC_Read* block to read the voltage provided by the current sensor and the MATLAB function to translate the voltage into current. In addition, as described above, the following MATLAB function is concerned with the first order lag filter.

The response of the system is shown below:

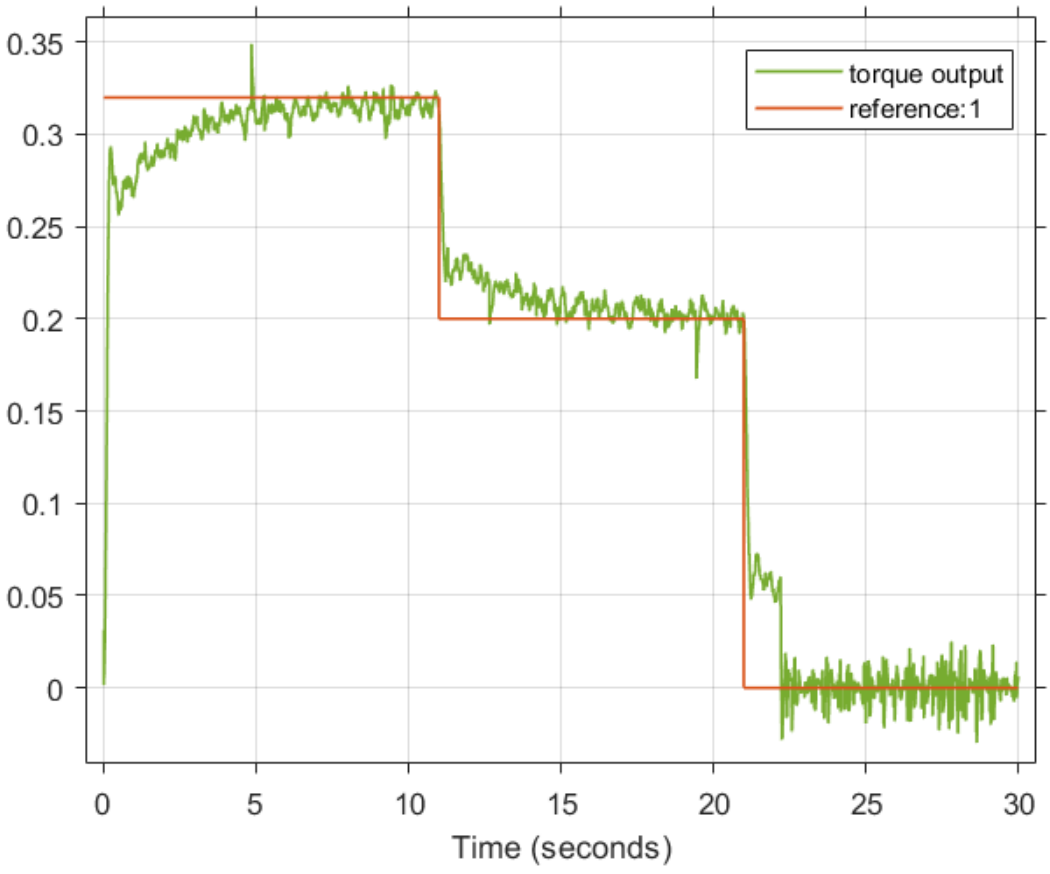


Figure 4.15: Step response in the torque control system

It can be seen that the response is much slower than seen in the simulations since the mechanical part of the motor was neglected which is much slower than the electrical part and using the real motor it is

impossible to neglect it.

It is also noted that in order to chase lower references the motor has to overcome inertia, consequently, if one starts with too low torques, the motor is not able to to overcome inertia and will stand still while guaranting the torque reference. Conversely, however, if high torques references are provided, the motor will be able to overcome inertia and reach the various references, also the low ones.

In addition, the control input related to the previous experiment is shown:

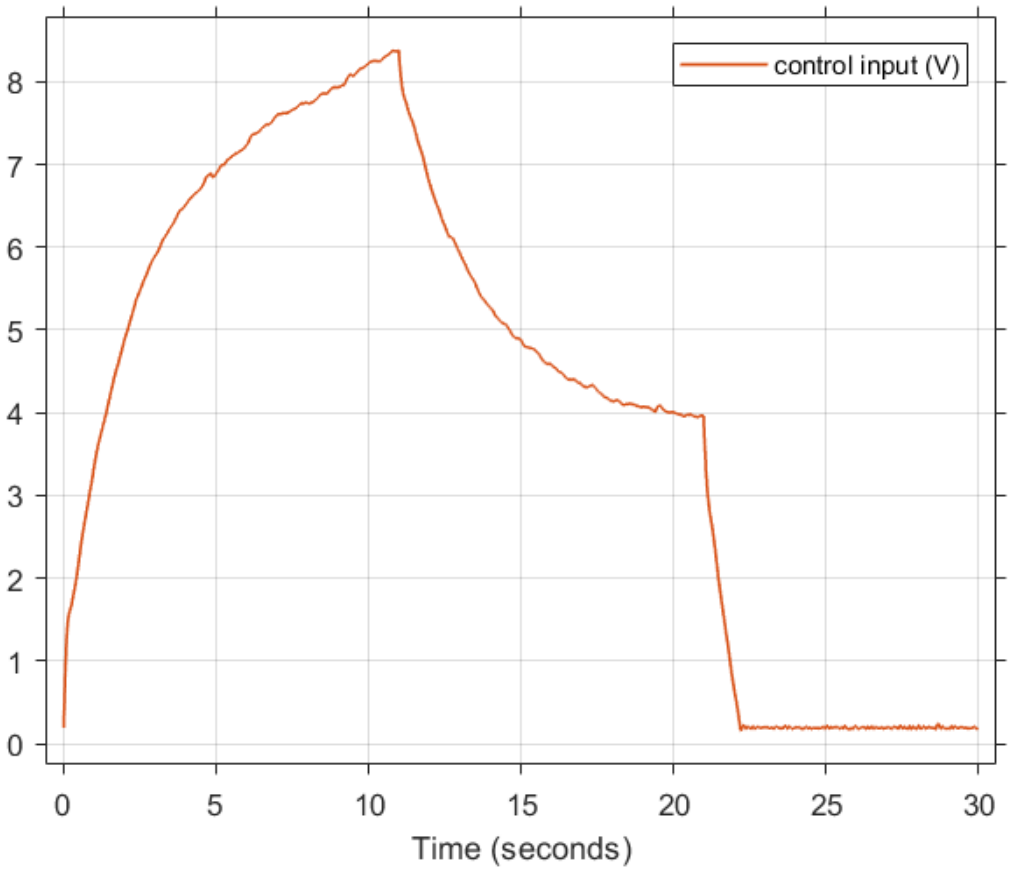


Figure 4.16: Control input in the torque control system

It should also be noted that the anti wind up term, again, has been implemented and is working correctly. As can be seen by giving the input an unattainable torque for the motor:

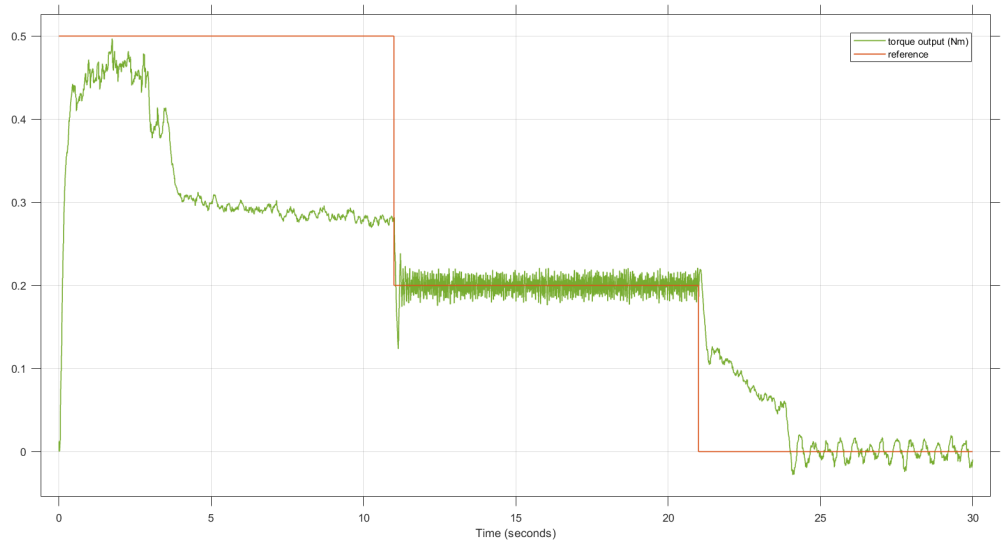


Figure 4.17: Step response in the torque control system with anti wind up

which is accompanied by a saturated control action:

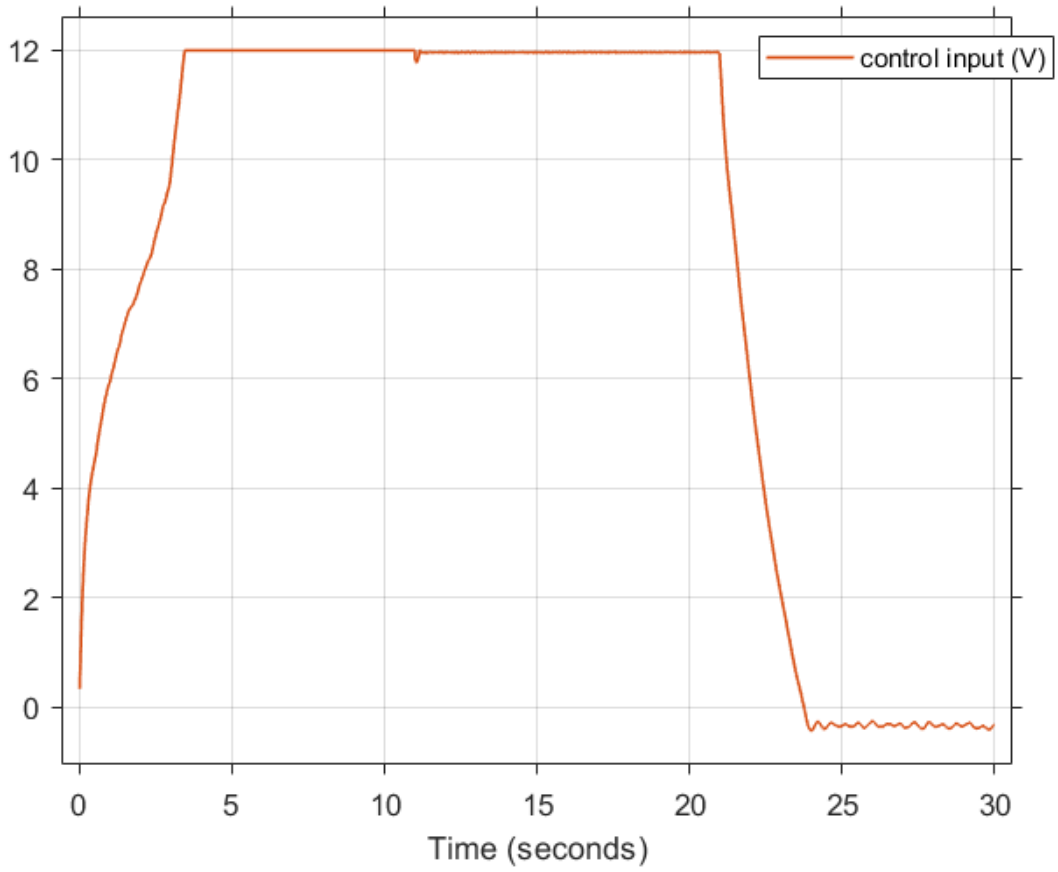


Figure 4.18: Control input in the torque control system with anti wind up

it is shown how it correctly succeeds in immediately discharging the accumulated error and correctly resuming torque tracking.

# CHAPTER 5

# DIRECT CODING

An alternative way to develop the control algorithm based on automatic code generation, as seen in previous chapters, is to directly develop the ad hoc code for controlling the system.

In this chapter, we start with the observer control system developed in 3 and proceed with its implementation in C code, via the STM32CubeIDE development environment, which makes it easy to write code for STM32 microcontrollers, in our case for the STM32F401RE board, as well as to compile and upload it directly to the board via USB.

The same procedure was performed for the development of code for torque control, starting from the control system developed in 4. For the development of the code, clock and interrupt-based control loop synchronization mode was used as the control loop synchronization mode, so as to ensure all the system functionalities we required, such as reading data from sensors, calculating control, and sending data via USART.

In addition, the time for the control loop is less than the threshold of $\frac{T_s}{10}$, with $T_s$ sampling time, consequently we are able to execute it in the mode: "Read, compute, send".

## 5.1 Development of C code for position control

For the development of the C code for position control, we started from the same pin configuration of the STM32F401RE board used for the speed control performed during the course:

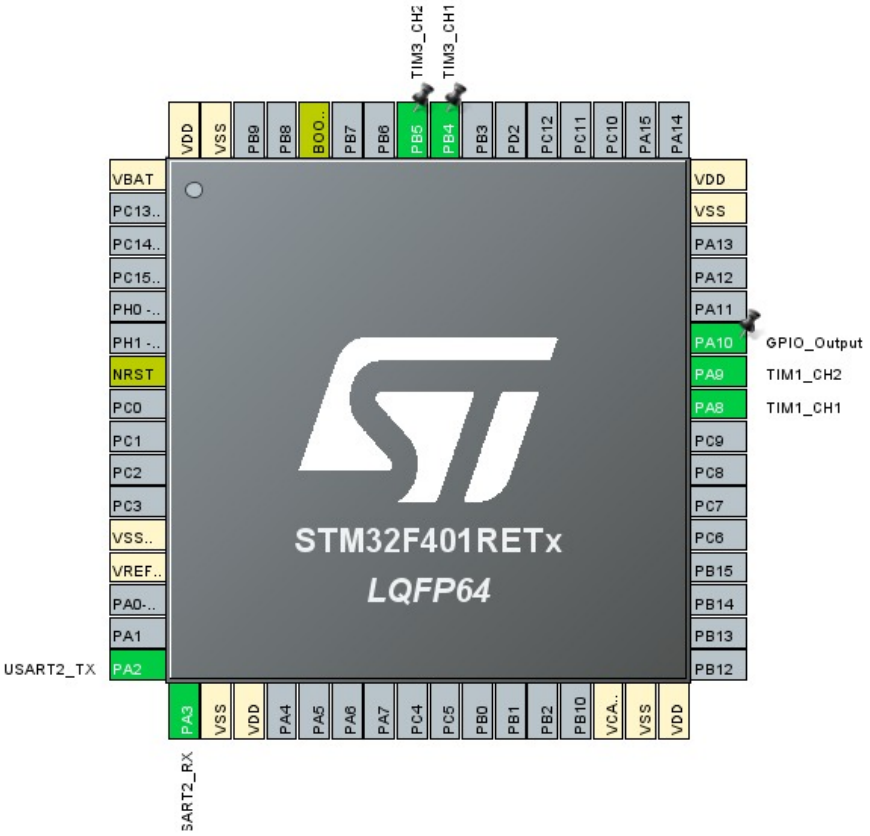


Figure 5.1: Pin configuration of the STM32F401RE board for position control

The same was done for the clock configuration:

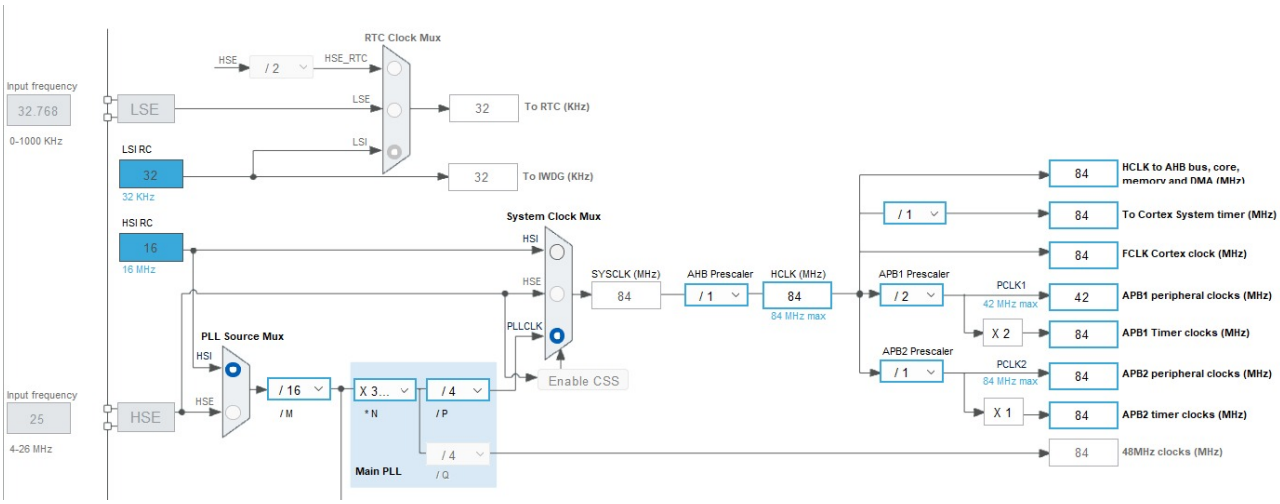


Figure 5.2: Clock configuration of the STM32F401RE board for position control

Following the pin and clock configuration, we proceeded with writing the C code for position control. Initially, code was written for the data record, including the angular position of the motor, the voltage applied to the motor, the reference, and other debugging information useful for control.

Next, omitting the description of the functions in common with speed control, we proceeded to write the code for calculating the position derived from the encoder data.

Furthermore, the data structures needed to store the dynamic matrices of the observer system and the vector

of controller gains.

Next, the ”main” function was revisited, in which an initial delay was inserted to allow the motor, which without any control code, move at full speed, to stop. In addition, in the section where data is sent via USART, the records previously described.

In writing the code for the control loop, the function *get_x_hat* was used to calculate the estimated state of the system from the dynamic matrices previously initialized.

Furthermore, since we have never implemented a dynamic system in the form of a C function, we proceeded with the analysis of the correctness of the code by implementing the same code in MATLAB verifying that the results were consistent.

In particular, starting with the MATLAB function:

```
function [x_hat,x_new]= fcn(u, y, x_now)
A = [0.931605363824459  -11.346656884140103    0.039099543321700;
    0.002741925544432    0.119498209981127    5.550203919352659e-05;
   -0.007094455468349   -2.896588572120294    0.837920445653216];

B = [-0.004047022543127    1.941673905307397;
   -5.744773077210588e-06    0.150674103104834;
    0.036776145051735    0.495672910737163];

C = [0.965802681912229  -5.673328442070051    0.019549771660850;
    0.001370962772216    0.559749104990563    2.775101959676330e-05;
   -0.003547227734175  -1.448294286060147    0.918960222826608];

D = [-0.002023511271564    0.970836952653698;
   -2.872386538605294e-06   0.075337051552417;
    0.018388072525868   0.247836455368581];

x1 = x_now(1);
x2 = x_now(2);
x3 = x_now(3);

x1_stimato = C(1) * x1 + C(4) * x2 + C(7) * x3 + D(1) * u + D(4) * y;
x2_stimato = C(2) * x1 + C(5) * x2 + C(8) * x3 + D(2) * u + D(5) * y;
x3_stimato = C(3) * x1 + C(6) * x2 + C(9) * x3 + D(3) * u + D(6) * y;

x1_new = A(1) * x1 + A(4) * x2 + A(7) * x3 + B(1) * u + B(4) * y;
x2_new = A(2) * x1 + A(5) * x2 + A(8) * x3 + B(2) * u + B(5) * y;
x3_new = A(3) * x1 + A(6) * x2 + A(9) * x3 + B(3) * u + B(6) * y;


x_new = [x1_new ;x2_new ;x3_new];
x_hat = [x1_stimato ;x2_stimato ;x3_stimato];
```

Figure 5.3: Observer in MATLAB in the form of a function

you can simulate the system and verify that the results are consistent with those obtained with the discrete observer implemented by MATLAB itself:

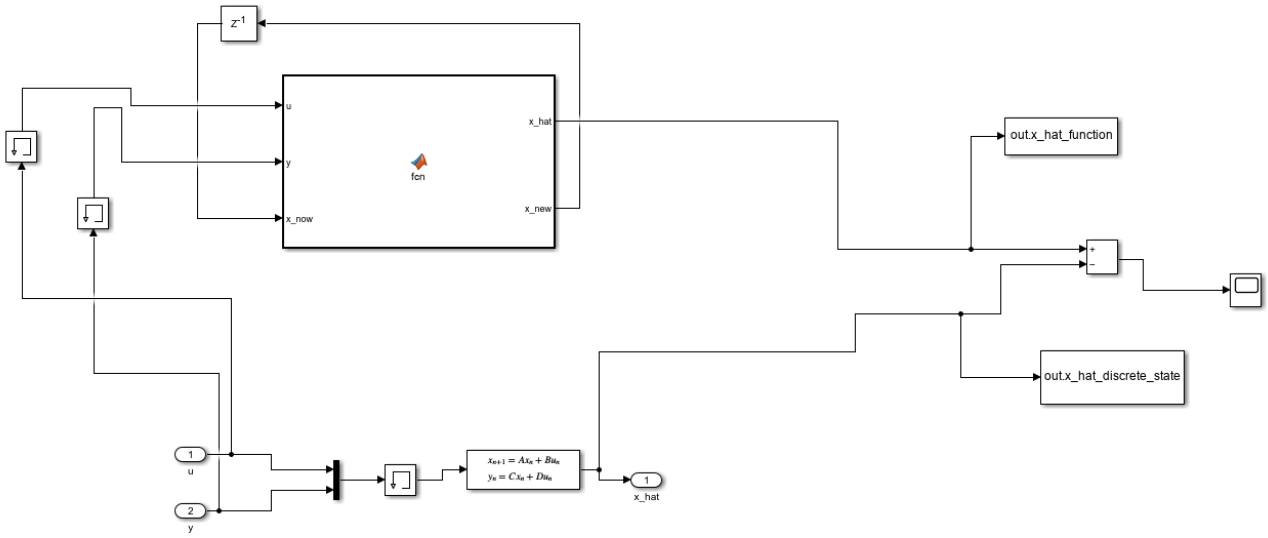


Figure 5.4: Comparison between observer in MATLAB and observer in function form

Following the simulation with a stair reference with values equal to $0, \frac{\pi}{2}, pi$ and $2\pi$, we proceeded with the verification of the correct operation of the implemented function:

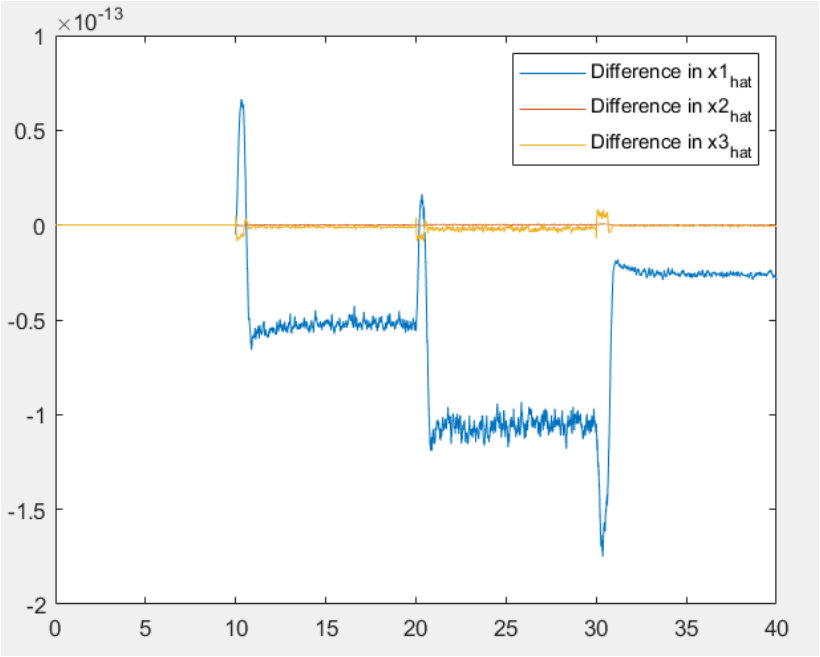


Figure 5.5: Comparison between state estimated by the observer in MATLAB and by the observer in function form

The difference between the two observers is minimal, and therefore it can be said that the implemented C code is correct.

Following the verification of the correct operation of the observer, we proceeded with the writing of the code for the controller, specifically for the calculation of the input control from the estimated state, reference, and current error.

In the control loop, the terms of anti wind up, saturation of the input and the change of reference, since if done in the ”main” function, it does not meet the required timelines.

Finally, at the end of each control cycle, code was written to save the data in the circular buffer to allow visualization and monitoring of the data via the MATLAB script *readFromCOM*.

By making the code build and loading it on the STM32F401RE board, it was possible to verify the correct operation of the control in position, by exchanging data and plotting them on MATLAB.

In particular, this is the response of the system to a stair reference:

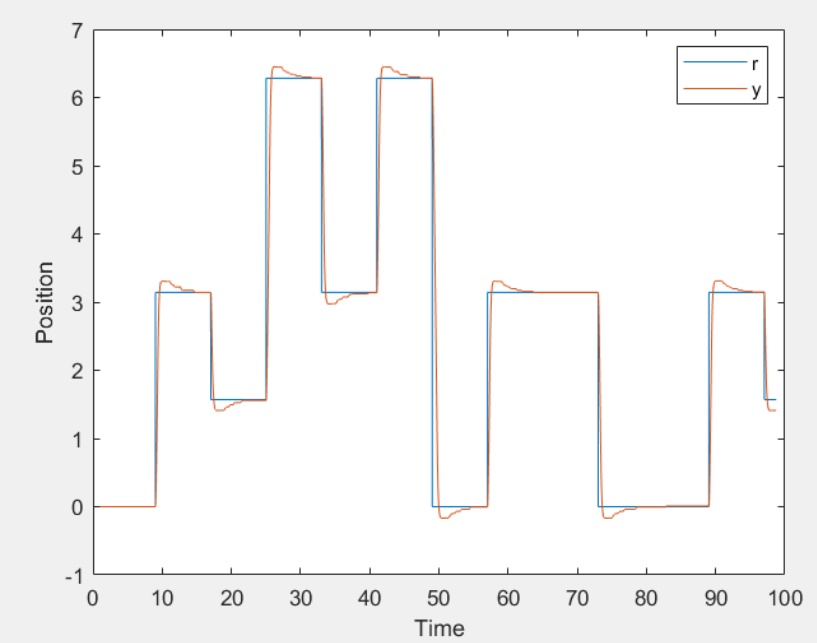


Figure 5.6: Response of the system to a scaled reference

Finally, we proceeded to analyze the differences between the responses to the system at the same reference in the three case scenarios: simulated, automatically generated code, and directly written code:

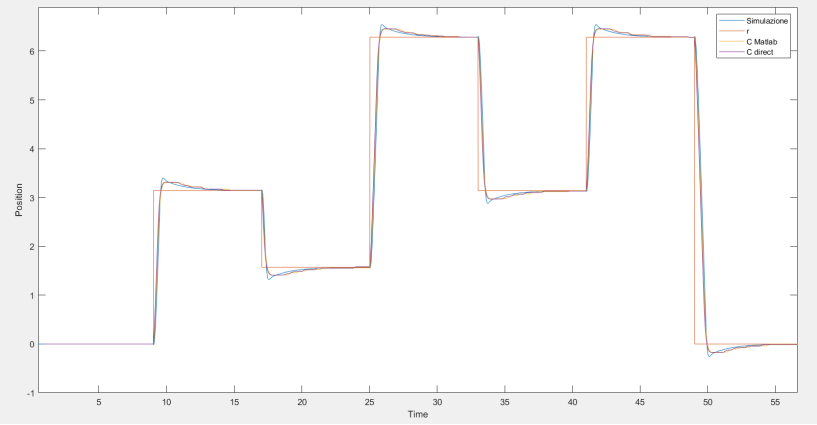


Figure 5.7: Comparison of simulated position control, automatically generated code, and directly written code

And in particular, zooming in on one of the references can be observed:

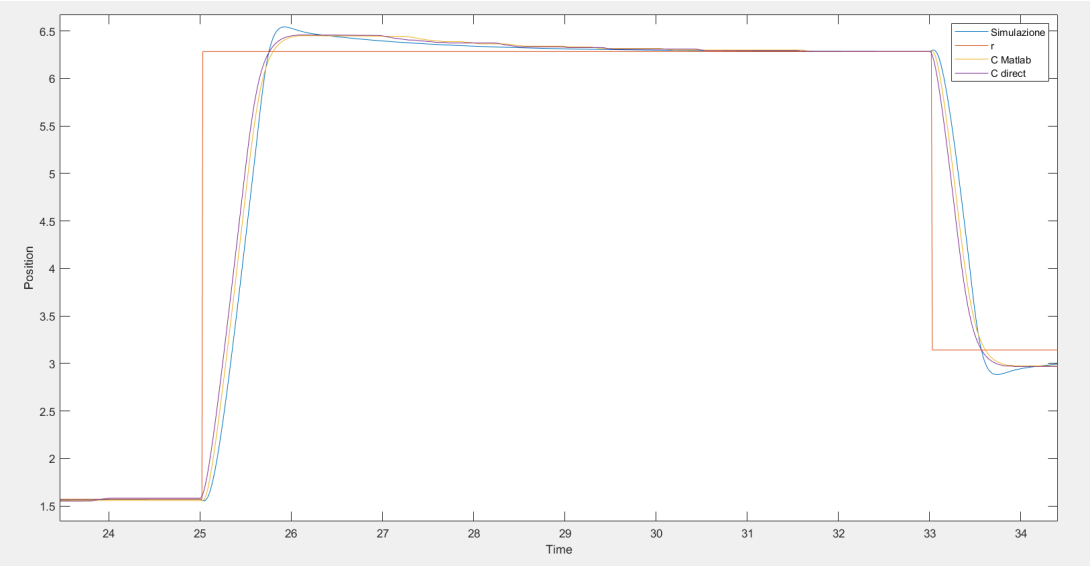


Figure 5.8: Comparison between simulated positional control, automatically generated code, and directly written code (Zoom in)

It can be seen that the differences between the three systems are minimal, and considering that the system model was identified in an approximate way, one can verify its validity.

## 5.2 C code development for torque control

Starting from the same ioc diagram as in-position control, with the addition of ADC readout to allow reading from the current sensor:

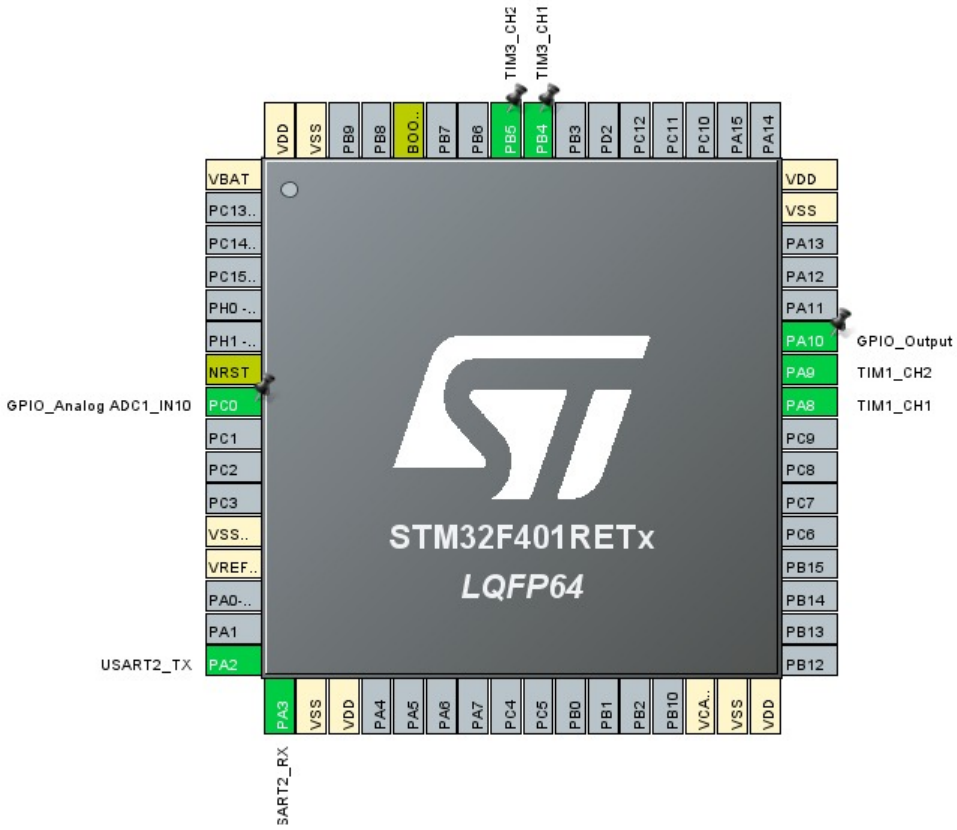


Figure 5.9: Ioc diagram for torque control

the C code for torque control was drafted. In particular, unlike the previous code, it was no longer necessary to use auxiliary data structures, since the change is only in the control law which involves the reading of the current sensor value by ADC, with subsequent rescaling of the value to obtain the actual current; filtering of the same, via first order lag filter, as already described in **??**; calculation of the error on the torque and application of the anti wind up term for the correct calculation of the control input, appropriately saturated. Also in this case the control cycle ends with saving the data in the dedicated record data structure and sending the data, from the main, via USART port. Starting with a stair reference with values ranging from 0.0 and 0.4 we are able, thanks to the *readFromCOM* MATLAB function, to visualize the trend of the motor torque, demonstrating its effectiveness:

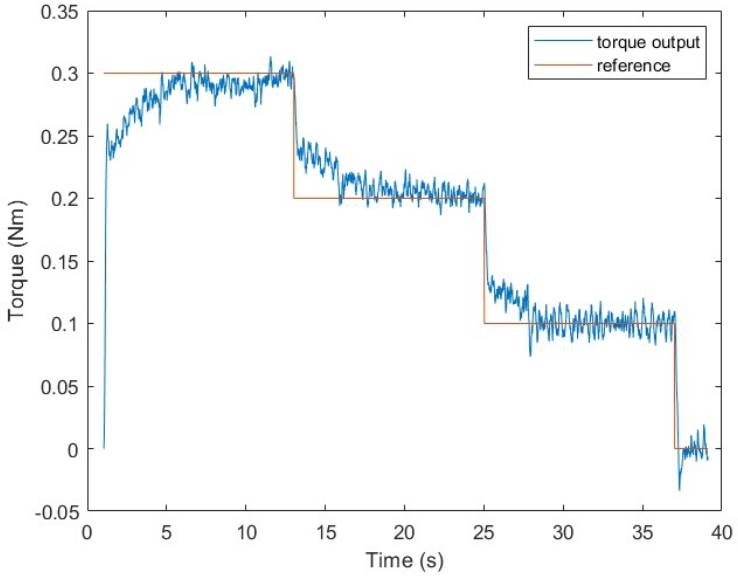


Figure 5.10: Response of the system to a scaled reference

Finally, we proceeded to analyze the differences between the responses to the system at the same reference in the two case scenarios: automatically generated code and directly written code:

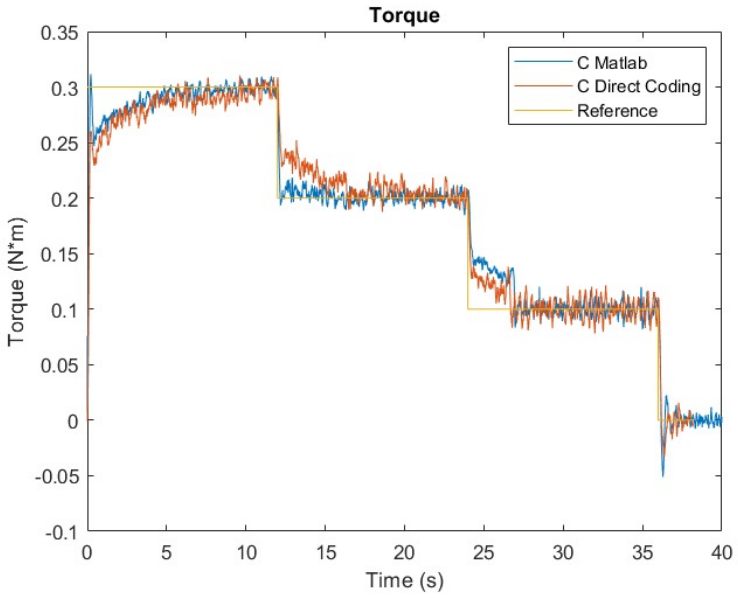


Figure 5.11: Comparison of automatically generated code and directly written code

It can be seen that the differences between the two systems are minimal, and considering that there is a bit of aleatory due to the sensor and also to the wires that connect it to the board so the difference is acceptable.

In the simulated case, the mechanical dynamic was supposed to be a disturbance and was not taked into account, so the difference would be due to this, and so we've decided to not include it in the comparison.

# CONCLUSIONS

At the end of the project, it can be said that all the set goals have been achieved. Two different types of controllers were designed and implemented, in position and in pair, both with the mode of automatic code generation and in direct coding mode. In addition, we would like to conclude that a speed control was not implemented in the project, as it seemed more interesting to implement the other two types of control, aware of the fact that for the implementation of speed control, the implementation would be very similar to that of position control. In addition, the choice was made to design only one controller per control type, consequently it was not necessary to introduce a switch for selecting the controller that enjoyed bumpless transfer. Furthermore, using the stair generator for the reference, it was not useful to introduce state chart diagrams for the change of reference.

Further improvements could be obtained by a better estimation of the DC motor parameters, which we were not able to obtain with the given tools and due to the lack of information in the datasheet.

# REFERENCES

[1] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2010. ISBN: 1849966346.

# LIST OF FIGURES