Lab 3:

**3.1 Description**

The goal of this laboratory is to implement a controller in order to drive and control DC motors of the Turtle-Bot. To reach that goal, we have to design a PI controller, fed by angular speed of the wheel, measured in round per minute [rpm], as the reference signal, in order to control the angular speed of the wheel, which is rotating under the action of a DC motor.

**3.1.1 DC Motor**

A DC motor is a class of rotary electrical motors that converts direct current (DC) electrical energy into mechanical energy. There are three common parts in all types of DC motors:

* a *stator*, which can be implemented as a *permanent magnet*;
* a *rotor* *winding* or a *coil*, which can be simply a *wire*;
* a *commutator* is connected to the rotor winding and its purpose is to force the direction of the current flowing into the rotor to be always the same;

However, the DC motor which is used in the Turtle-Bot uses Brushed implementation. *brushes* are used to supply current to the commutator and each brush is connected to one side of the power supplier.

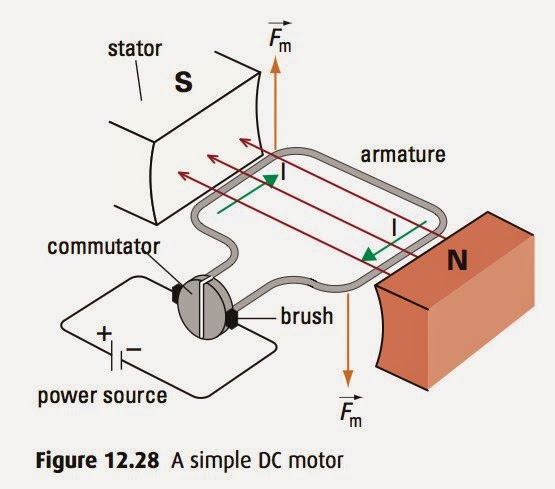
[](http://sph3u2014.blogspot.com/2015/01/dc-motors-and-induction.html)

Figure 1: DC Motor simplified scheme

The stator generates a magnetic field and the rotor winding, which is positioned within the field generated by the stator, when a current flow through it, is subjected to a magnetic force perpendicular to the direction of the stator field. The commutator has a ring-like shape and present gaps, so that each part of the commutator is connected to power source of opposite polarity, ensuring the current flows in the proper direction.

Inverting the power source polarity, invert the direction of the current, allowing the motor to rotate in both directions.

**3.1.2 DRV8871**

Since we want to use a *PWM* as a control signal for the motor, we use a chip that can convert a PWM input into a proper output, both in terms of *voltage* and *current*, to drive the motor. For such purpose, the chosen chip is a *DRV8871*, which is a brushed DC motor driver. The motor can be controlled bidirectionally by implementing an H-bridge.

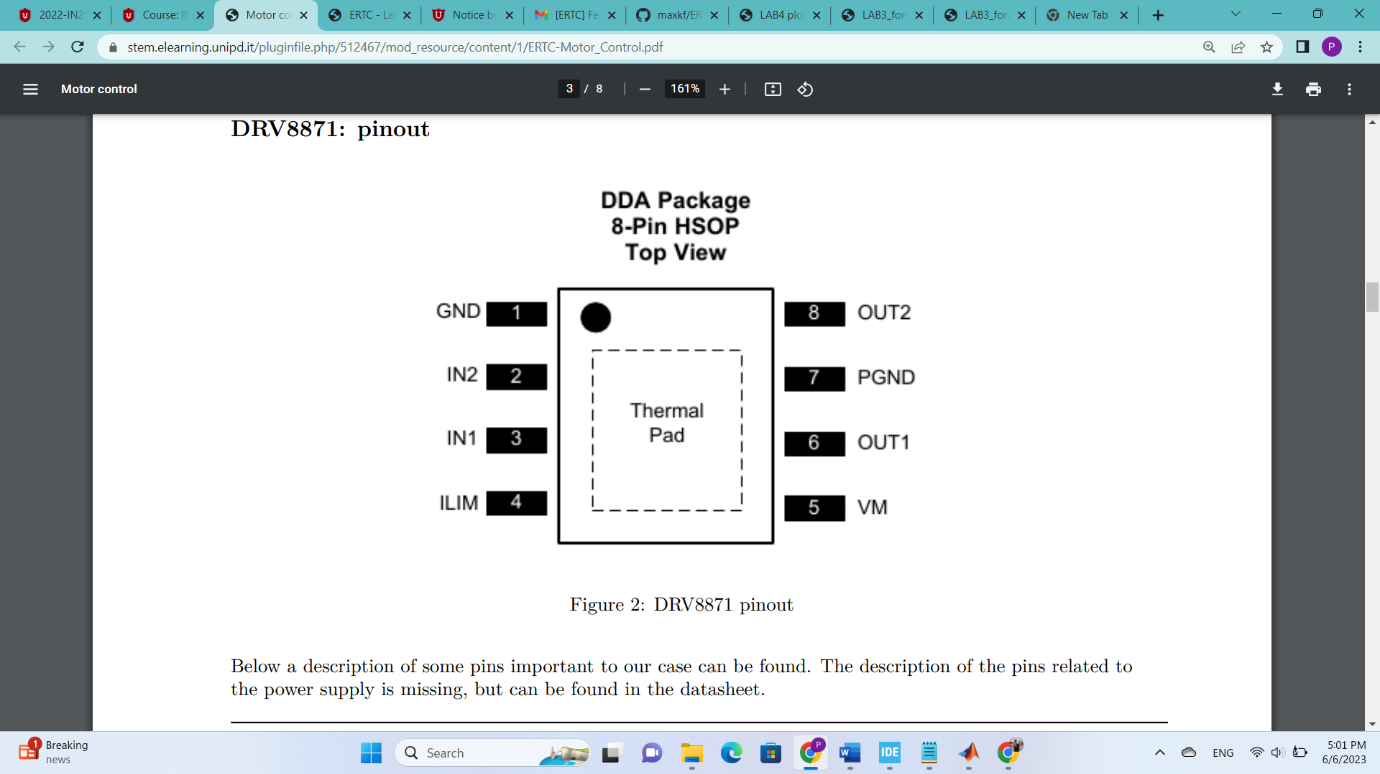


Figure 2: DRV8871 pinout

Below a description of some pins important to our case can be found.

|  |  |  |  |
| --- | --- | --- | --- |
| Pin name | Pin number | Pin type | Description |
| IN1 | 3 | Input | Input of the chips that allow to control the output of the h-bridge. These pins receive the PWM signal from the microcontroller |
| IN2 | 2 | Input |  |
| OUT1 | 6 | Output | H-bridge output that goes to the motor |
| OUT2 | 8 | Output |  |

The table below describes how the input signals can be used to control the output.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mode name | IN1 | IN2 | OUT1 | OUT2 |
| Coast | 0 | 0 | Hi-z | Hi-z |
| Reverse | 0 | 1 | L | H |
| Forward | 1 | 0 | H | L |
| Brake | 1 | 1 | L | L |

*Coast* mode allows the motor to coast to a stop, which means that the motor is not driven and will move by inertia. On the other hand, *Brake* mode will stop the motor faster. When used to drive the motor with PWM control signal with a duty cycle < 100% and, for example, in *forward* mode, it is possible to alternate between forward and brake mode or between forward and coast mode.

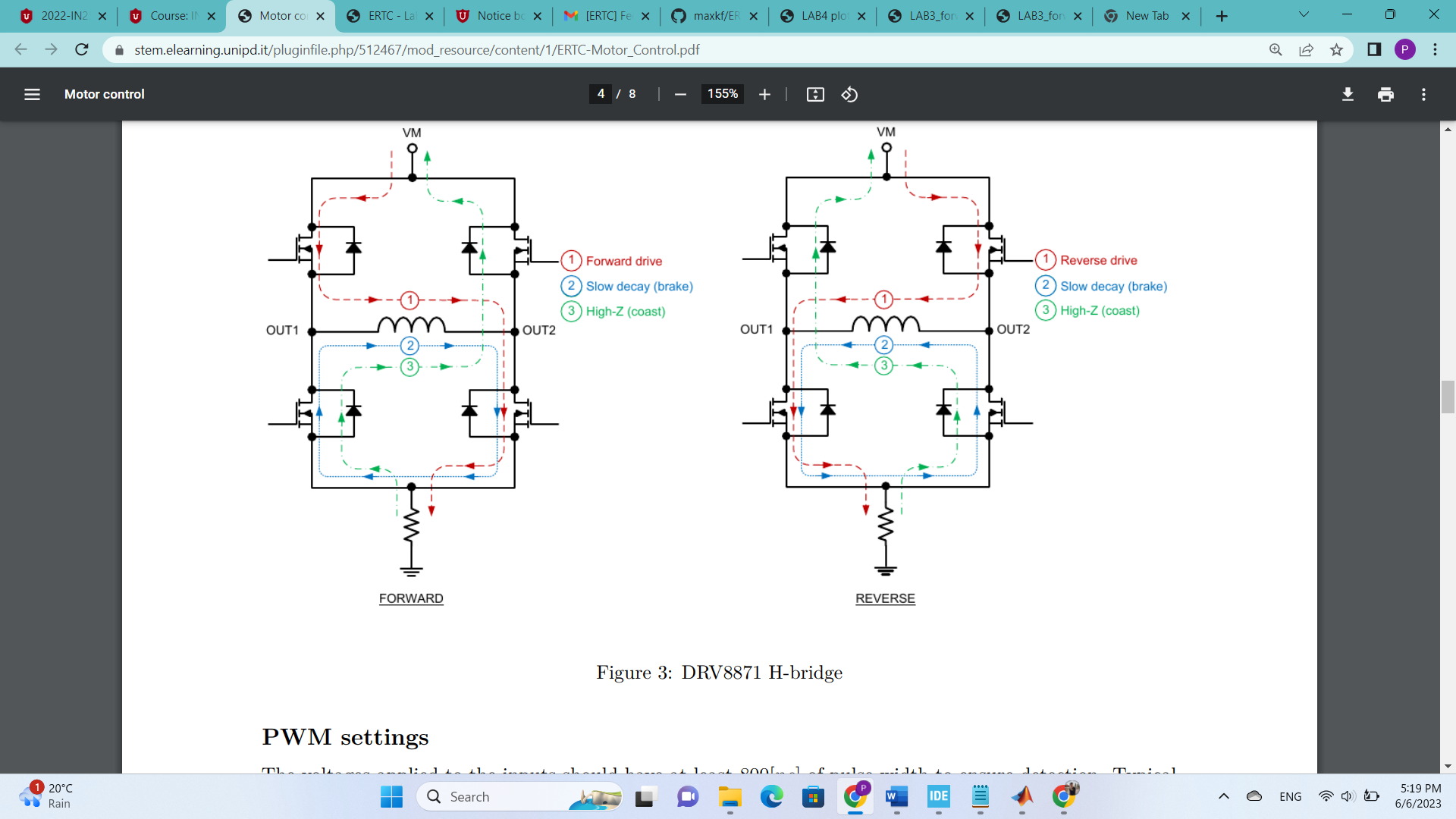


Figure 3: DRV8871 H-bridge

**3.1.3 PWM setting**

The voltages applied to the inputs should have at least 800[ns] of pulse width to ensure detection. If the PWM frequency is 200[kHz], the usable duty cycle range is 16% to 84%.

**3.1.4 Position Encoder**

The position encoder is a sensor that transform a position information into an electrical signal. A *quadrature* encoder employs two outputs *A* and *B* which are called quadrature outputs, as they are *90 degrees* out of phase; the direction of the motor depends on which phase’s signal lead over the other;

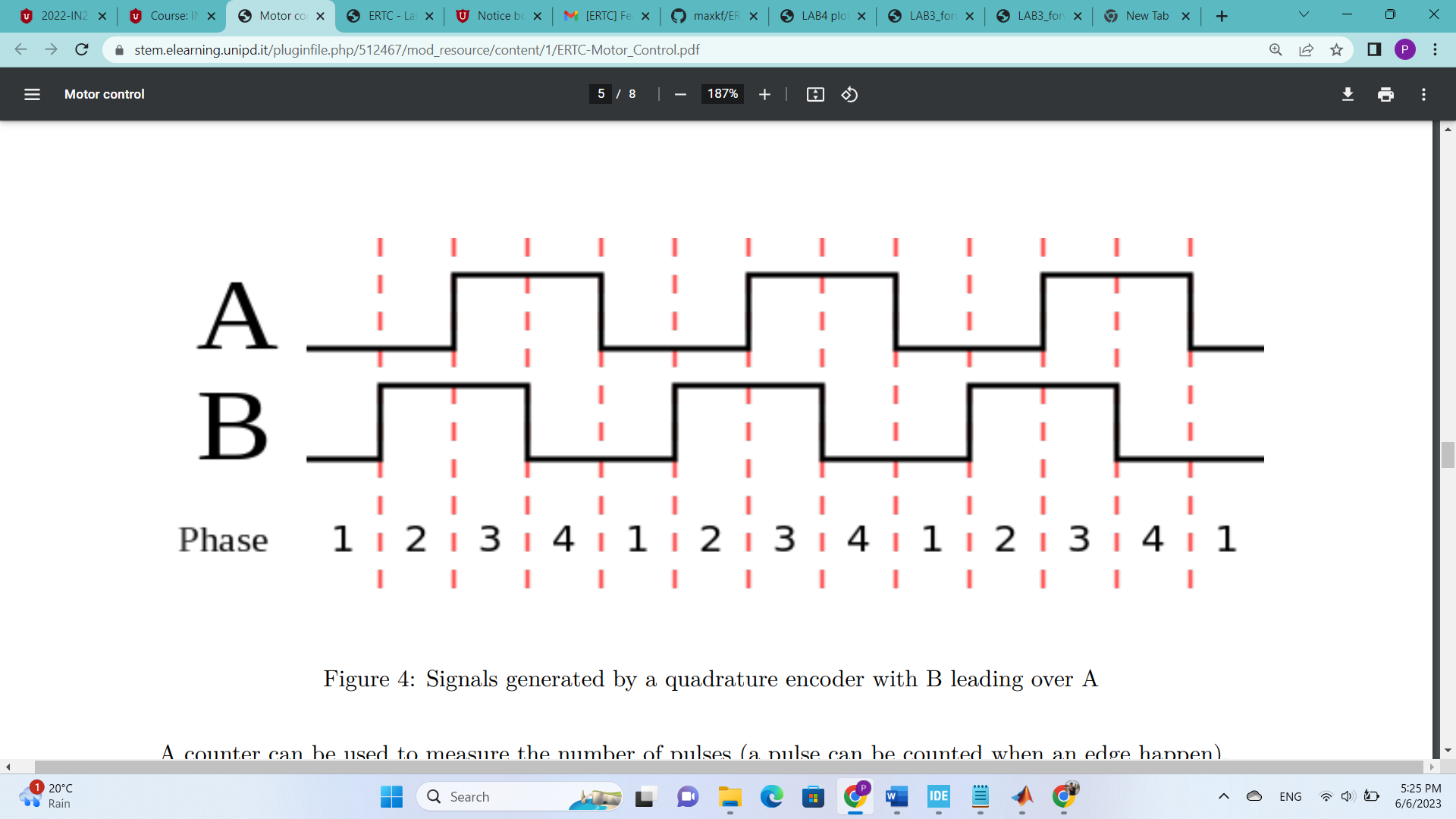


Figure 4: Signals generated by a quadrature encoder with B leading over A

A counter can be used to measure the number of pulses (a pulse can be counted when an edge happens).

**3.1.5 PID Controller**

PID (Proportional Integral Derivative) controllers use a control loop feedback mechanism to control process variables and are the most accurate and stable controller.

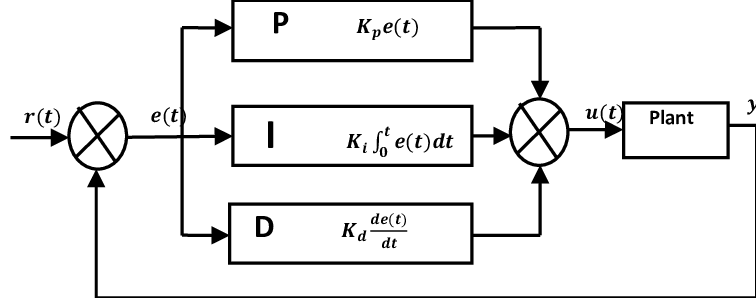
[](https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.researchgate.net%2Ffigure%2FThe-Model-of-a-PID-Controller-Configuration-From-Figure-3-the-mathematical-expression_fig3_344378374&psig=AOvVaw326qZwgY5yoeCXLIo8yvy7&ust=1686151834616000&source=images&cd=vfe&ved=0CBEQjRxqFwoTCLiY_J37rv8CFQAAAAAdAAAAABAT)

Figure 5: PID Controller diagram

Using the derivative control mode is a bad idea when the process variable has a lot of noise on it. 'Noise' is small, random, rapid changes in the process variable, and consequently rapid changes in the error. Because the derivative mode extrapolates the current slope of the error, it is highly affected by noise. Therefore, we only need to design and implement PI controller for this lab.

**3.1.5 Anti-Windup feature**

Integral windup refers to the situation in a PID controller where a large change in setpoint occurs (say a positive change) and the integral term accumulates a significant error during the rise (windup), thus overshooting and continuing to increase as this accumulated error is unwound (offset by errors in the other direction). The specific problem is the excess overshooting.

Integral windup can come from derivative action or a large reference. In our case, since we are not using the derivative control mode in our PID, only the reference may cause the overshooting.

In order to solve the problem, we need an Anti-windup architecture, basically a saturation over the control output of the integrator, that keeps the error caused by the integral term of the controller in a desired range and prevents accumulation of the error and consequently the overshooting. The figure below illustrates a common Anti-windup architecture.

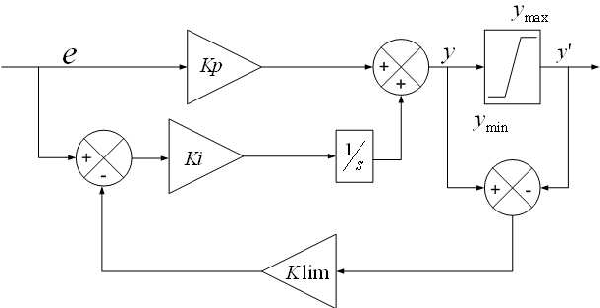
[](https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.semanticscholar.org%2Fpaper%2FAnti-windup-Schemes-for-Proportional-Integral-and-Ghoshal-John%2F9774270fa07e4be5dbe77dc7bf2a285167b82b68%2Ffigure%2F4&psig=AOvVaw1ggQ7BcApXXRdJ2R-6ZISv&ust=1686154937334000&source=images&cd=vfe&ved=0CA4QjRxqFwoTCJCZyuaGr_8CFQAAAAAdAAAAABAD)

Figure 6: Anti-windup architecture

**3.2 Implementation details**

**3.2.1 Hardware setup and peripheral list**

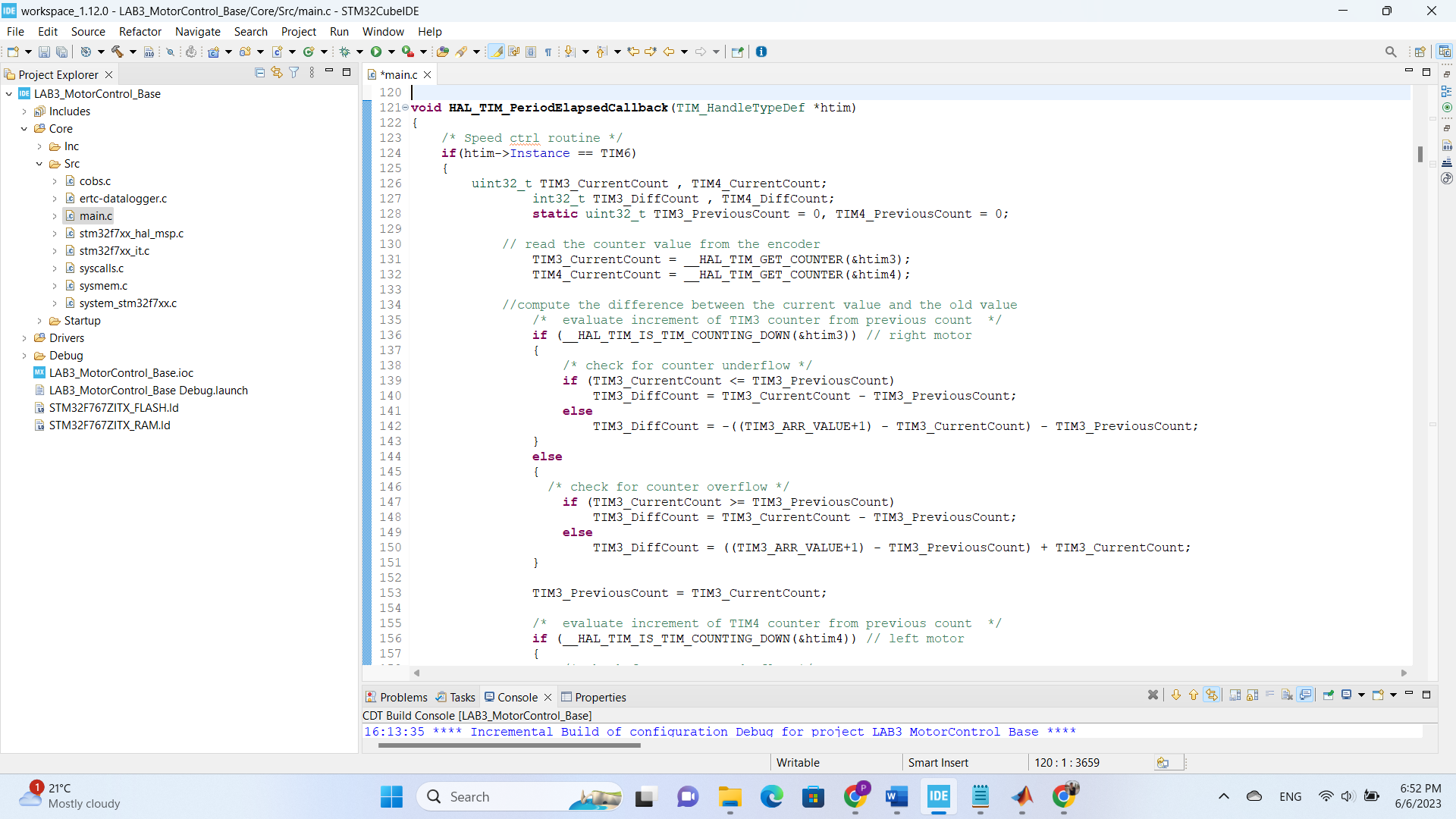
* TIM3 (16-bit, general-purpose timer), channel 1 and channel 2: Encoder for motor 1
* TIM4 (16-bit, general-purpose timer), channel 1 and channel 2: Encoder for motor 2
* TIM6 (16-bit, basic timer): Provide the clock for the controller
* TIM8 (16-bit, advanced timer), channel 1 and channel 2: PWM for motor 1
* TIM8 (16-bit, advanced timer), channel 3 and channel 4: PWM for motor 2

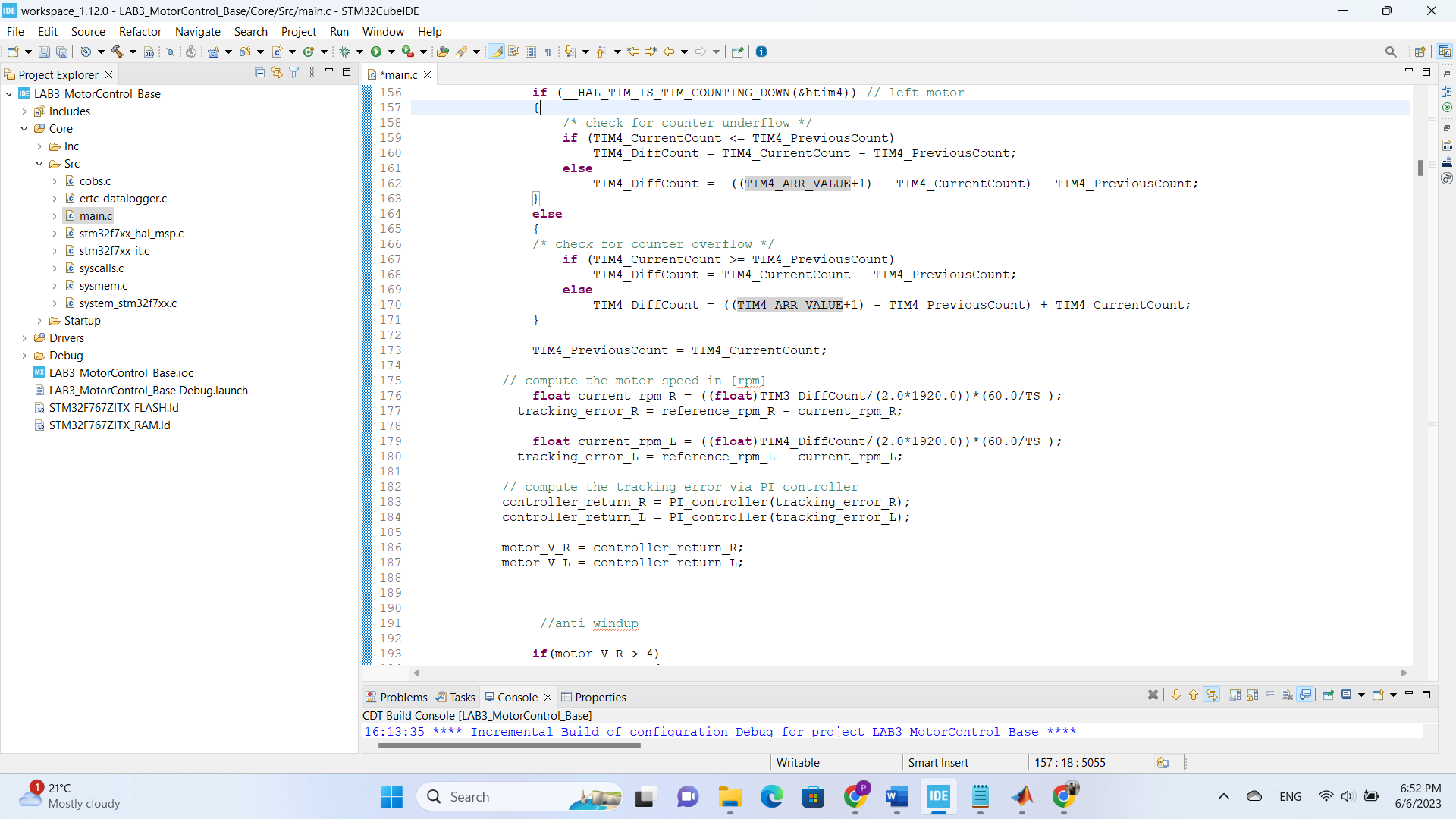
**3.2.2 Motor and Encoder**

The module encompasses a gearbox with a 120:1 ratio, which means that for every round performed by the wheel, the motor rotates 120 times.

The encoder provides 16 pulses per round. This means that the number of pulses per round from the wheel-side is 1920, assuming to use encoder mode 2X; this number is doubled when considering encoder mode 4X. Notice that this mode halves the quantization error, when compared with the 2X mode.

The selected timer needs to be configured in encoder mode. Also, the counter period which can be setup to be equal to the number of pulses counted in one round (in our case 3840 since we are using 4X encoder mode) has to be taken into account.





Listing 1 – encoder code in order to have the current speed of each motor

**3.2.3 PWM**

Configuration for the timer generating the PWM signal (TIM8):

* Clock source: APB1-Timer\_clocks at 96[MHz];
* Prescaler (PSC): 959 ⇒ = 96 ∗ 106/96 ∗ 10 = 105 [Hz]
* Counter period (Auto Reload Register): 399 ⇒ = 4 ∗ 102 ∗ 10−5 = 4 ∗ 10−3 [s]

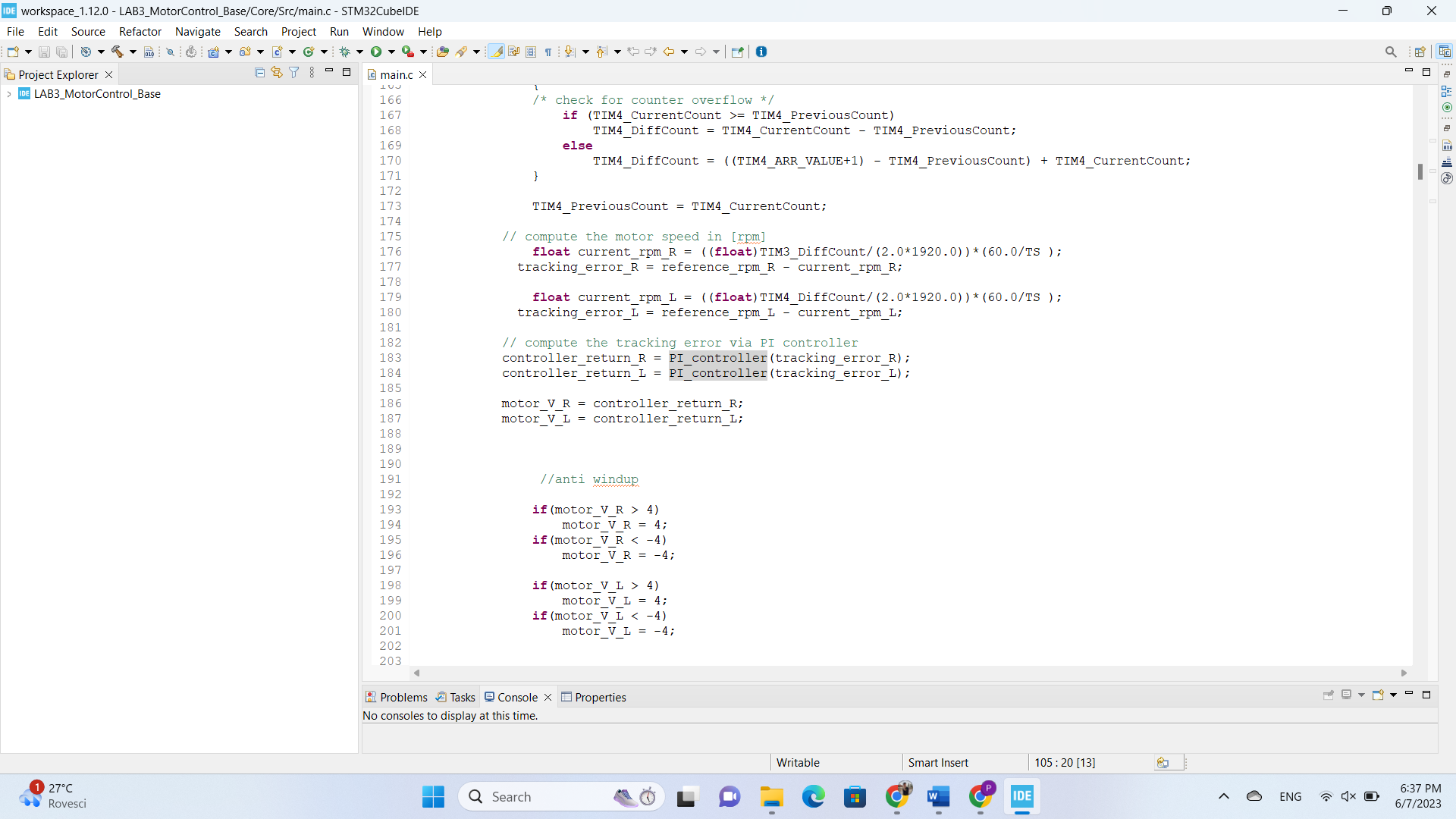
**3.3 Exercises**

**3.3.1 Exercise 1**

In order to control the DC motors, we use a PI Controller.

By using the encoders (TIM3, TIM4), implemented in the way that has been explained in section 3.2.2, the number of pulses for each wheel is available. Considering that the counter period is the number of pulses per round from each wheel-side (in our case since we are using the 4x mode, it is 3840), we can evaluate the speed of the motors in round per minutes as follow:

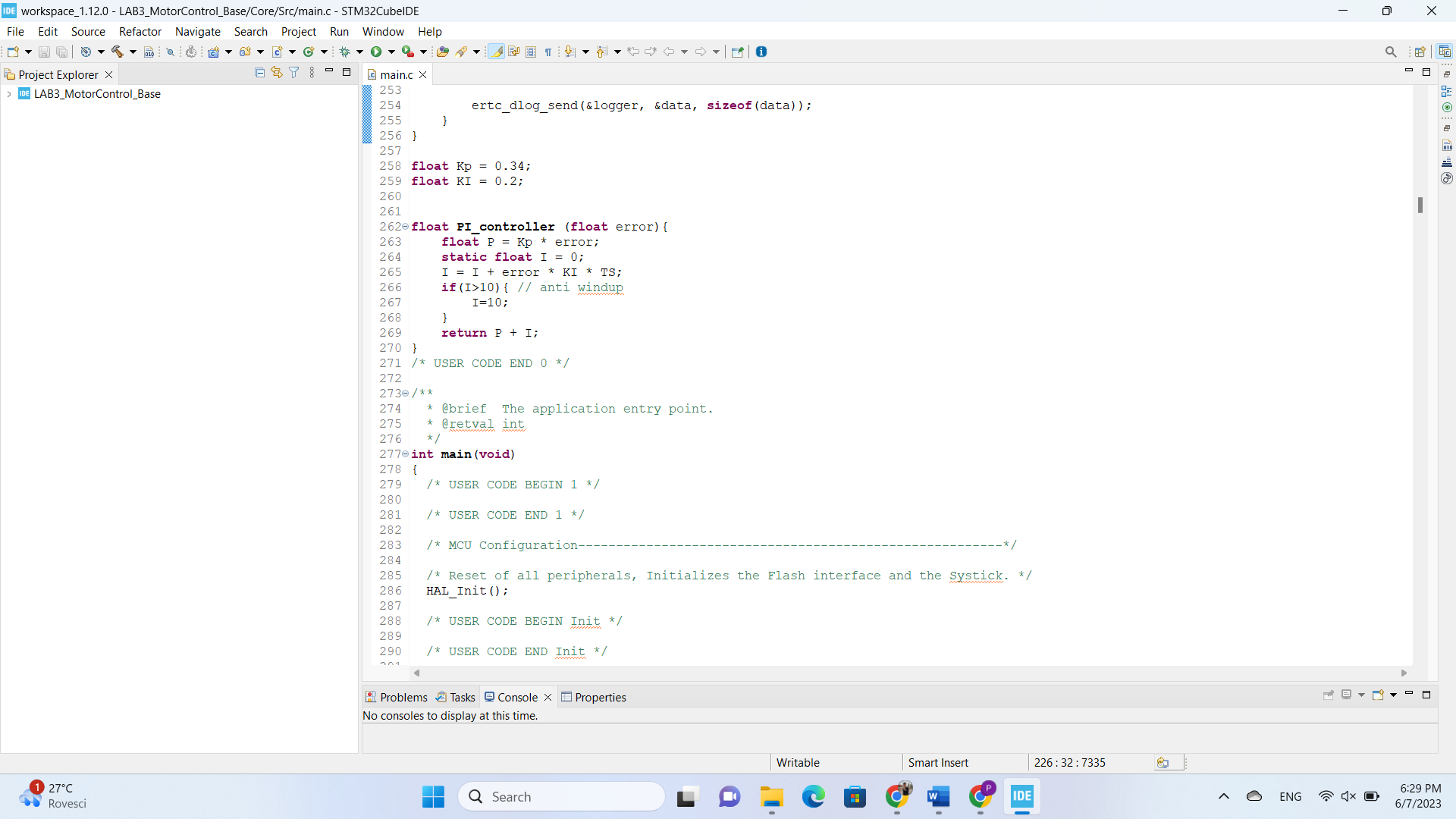
Which [s] is the sampling time. We also define the reference for the speed of each motor in rpm, which are hardcoded. Therefore, we can calculate the tracking error which will be considered as the input of the function PI\_controller. The output of the function is the control signal (in rpm) used to control the DC motors.



Listing 2- calculating the control signal for DC motors in rpm

We establish the closed loop according to the following logic: The input of the function PI\_controller is the tracking error. The error signal is multiplied in parallel by two parameters, the first is proportional gain Kp producing the proportional term, and the second is the product of integral gain Ki and the sampling period TS, producing the integral term.

To avoid overshooting in the response, caused by accumulation of the integrator error, we set up an Anti-windup scheme which is a saturation over . The threshold of the error which is 10 has been selected by trial and error. The optimal values of the Kp and Ki are found by trial and error.



Listing 3 – PI Controller

After summing up the proportional and integral term and producing the output of the function (control signal in rpm), it will be fed to the DC motors through TIM8 as PWM signals. Notice that TIM6 is acting as a clock in the system. All the steps needed to accomplish the goal, should be executed from inside the HAL\_TIM\_PeriodElapsedCallback function, which is a callback function called when a timer’s period is elapsed.

Since we are assuming a linear relationship between the duty cycle and the voltage applied to the motor, which we supposed to be in the range of [0, ]. , depending on the power supply source, is equal to ≈ 8[V].

**#define** VBATT 8.0

**#define** V2DUTY ((**float**)(TIM8\_ARR\_VALUE+1)/VBATT)

**#define** DUTY2V ((**float**)VBATT/(TIM8\_ARR\_VALUE+1))

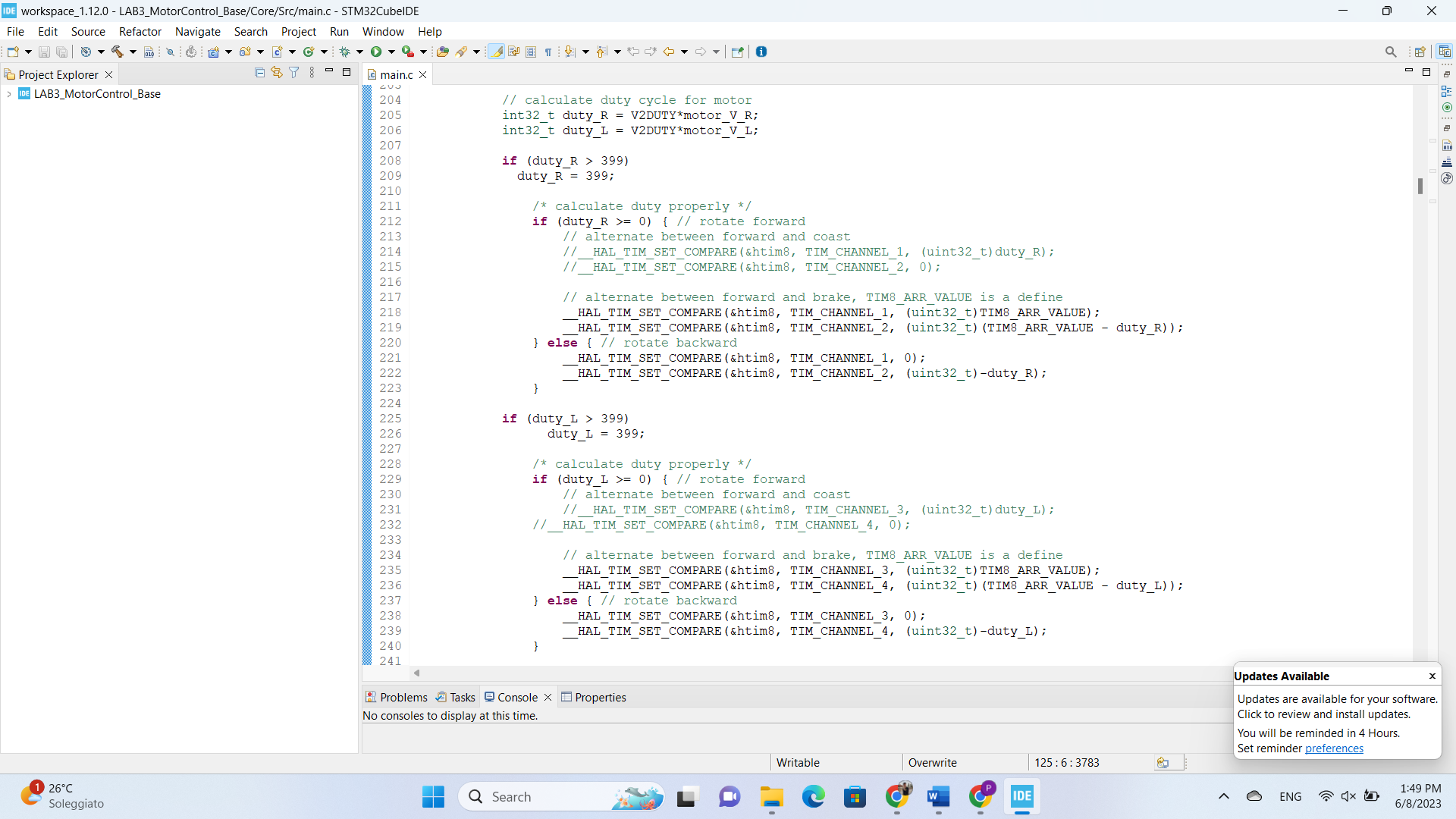
// calculate duty cycle for motor

int32\_t duty\_R = V2DUTY\*motor\_V\_R;

int32\_t duty\_L = V2DUTY\*motor\_V\_L;

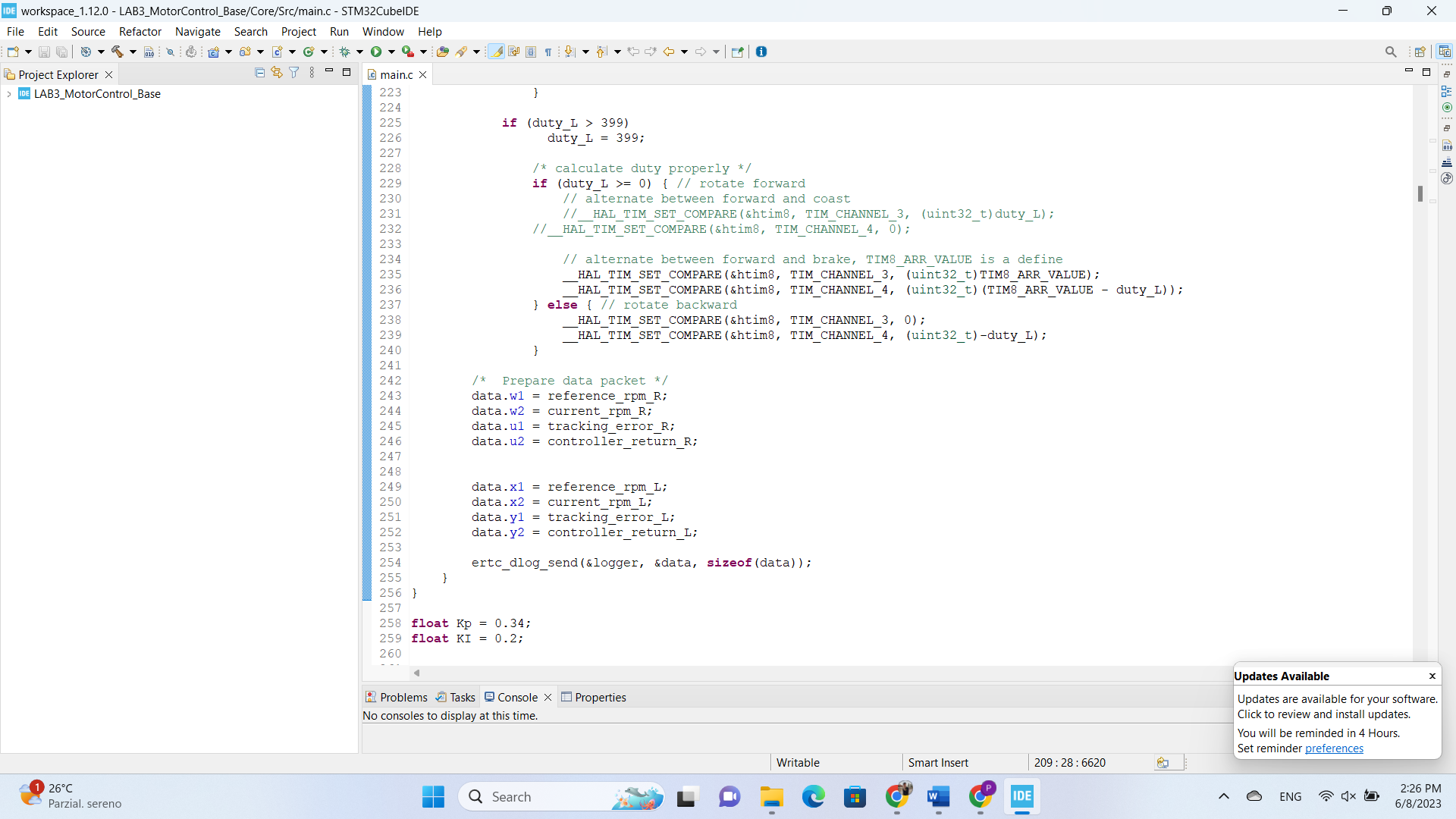
Listing4- calculating duty cycle

According to the Counter Period of TIM6, generating the PWM signal, a saturation of 399 is set over each duty cycle to prevent over driving the motors.



Listing 5- calculating duty cycle and command the motors

Since we are implementing the control using the Forward/Break technique in Exercise 1, we set up the command to the motors in the way that is shown in Listing 5, and in the end, by using the data logger, we save and plot all relevant information. The set up is as follow:

  
Listing 6- setting up data logger

The results are plotted as shown in below:

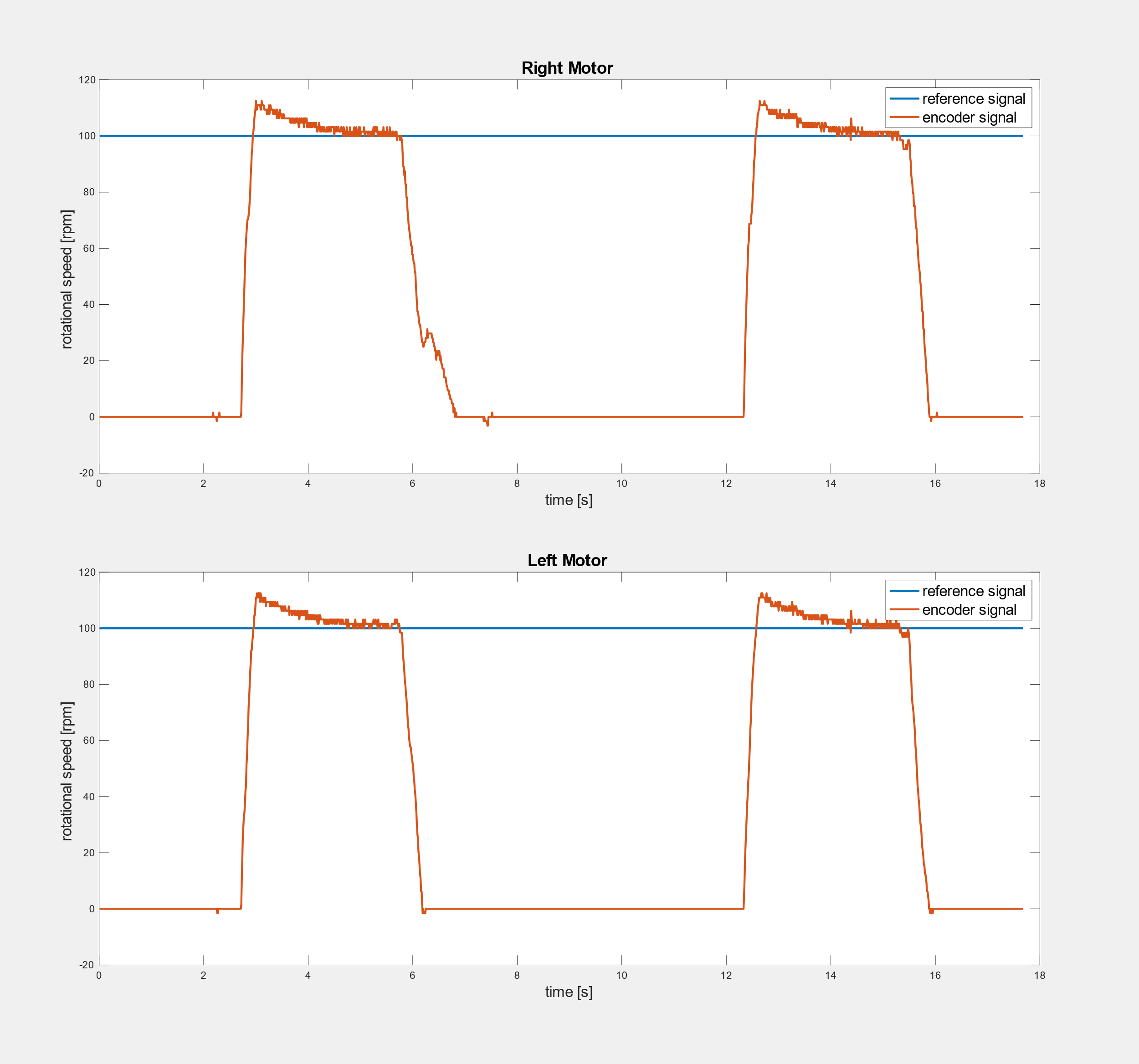


Figure 6: The results of the control using the Forward/Brake technique

To conclude, the results show that after turning on the motors, it takes a few milliseconds for motors to reach desired speed (acceptable rise time); even after being turned off for a while, the over-shoot does not go higher than a certain and reasonable level of speed (effect of the Anti-windup scheme). Lastly, the step response is settled at the reference value in few seconds. Notice that the noise in the response is because of the nature of the system and the digital controller.

**3.3.2 Exercise 2**

In order to implement the controller with the Forward/Coast technique, we only need to change the alternative piece of code, showed in listing 5. The result is illustrated in below:

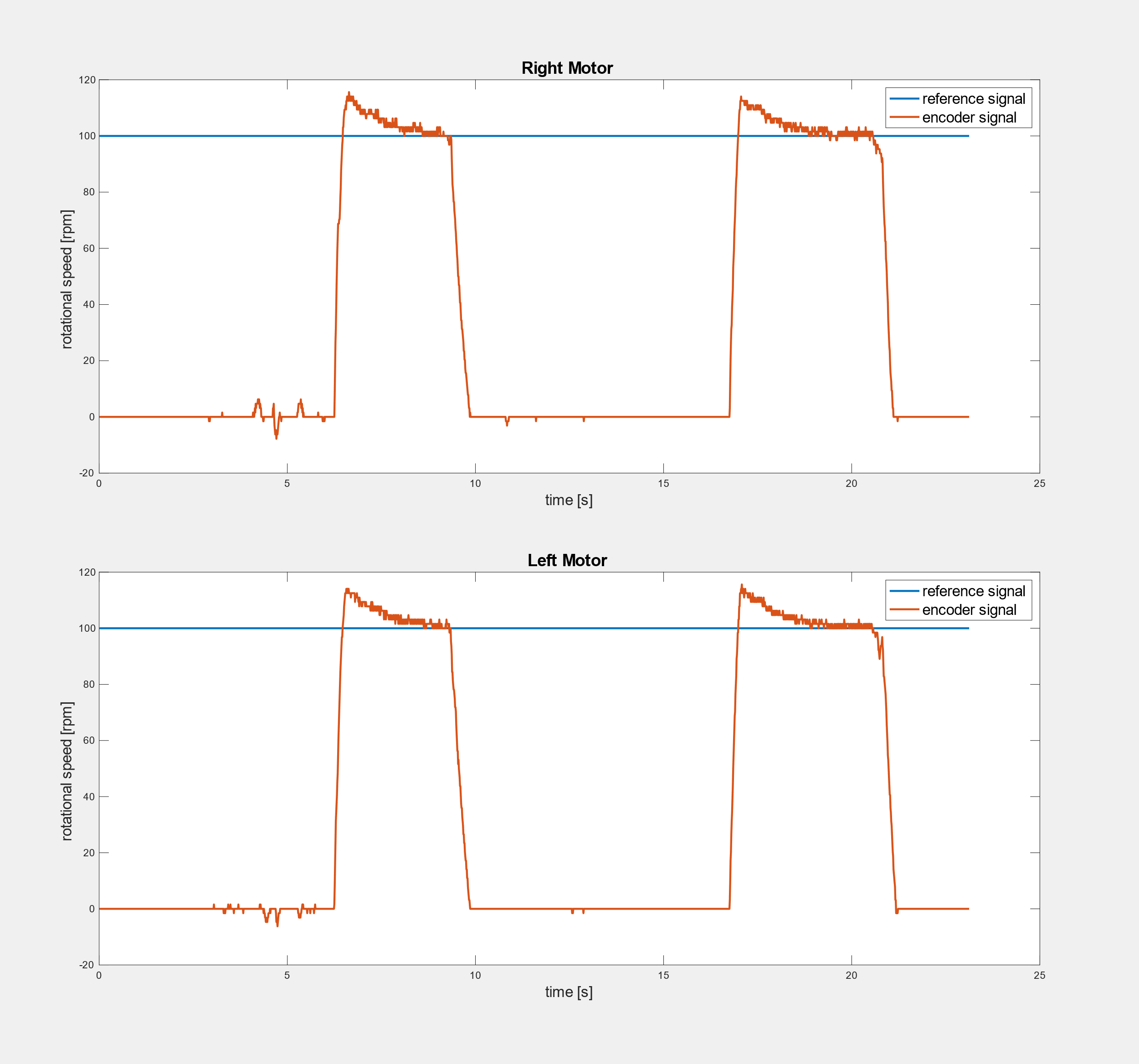
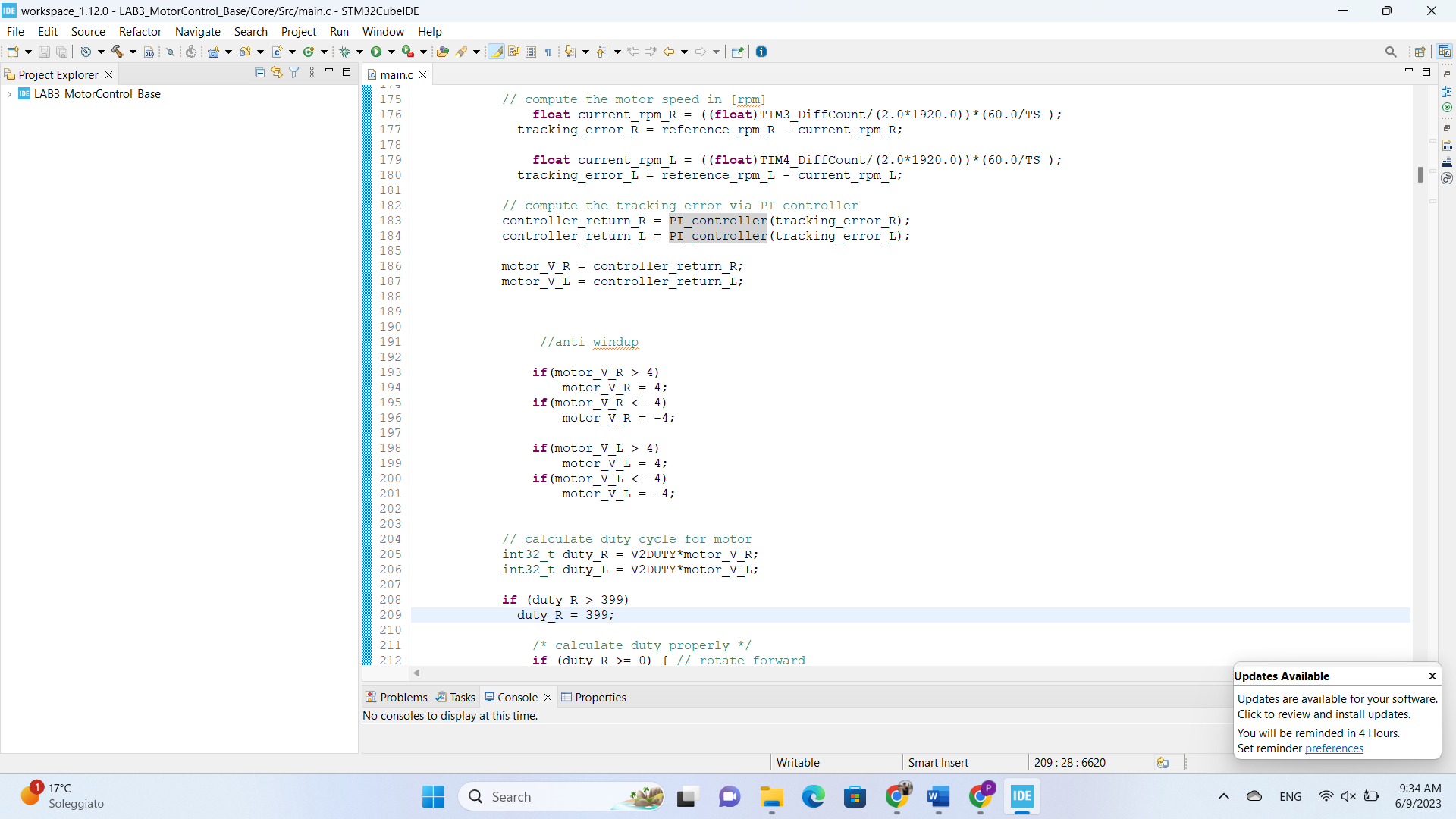


Figure 7: The results of the control using the Forward/Coast technique

Coast mode allows the motor to coast to a stop, which means that the motor is not driven and will move by inertia. On the other hand, brake mode will stop the motor faster. Therefore, the step response of motor implemented by Forward/Coast technique has more noises (even when the motor is set to be off) which can be interpreted as the inertia.

**3.3.1 Exercise Bonus**

The Anti-windup is implemented in the way that has been explained in section 3.3.1, particularly in Listing 3. As we mentioned before the threshold for the saturation has been chosen by trial and error. However, the first Anti-windup method we used was to set directly a saturation over the voltages fed to the motors. After saturating the voltage directly by changing the threshold to different values (for example, 4 in Listing 7), the overshoot problem was still unsolved. Hence, we decided to change the Anti-windup method to saturating only the integral term of the control signal (which is the motor voltage), shown in Listing 3.



Listing 7 – wrong Anti-windup method

The step response of the motors using both Forward/Break and Forward/Coast techniques, when the Anti-windup implementation is not used are shown in Figure 8 and Figure 9. To conclude, the step response of the system (no matter implemented by which technique), without implementing an Anti-windup feature, has a large overshoot due to the accumulation of the error caused by integral term. Adding an Anti-windup feature enhances the overall performance and reduced the overshoot in the system, as expected (Figure 6 and Figure 7).

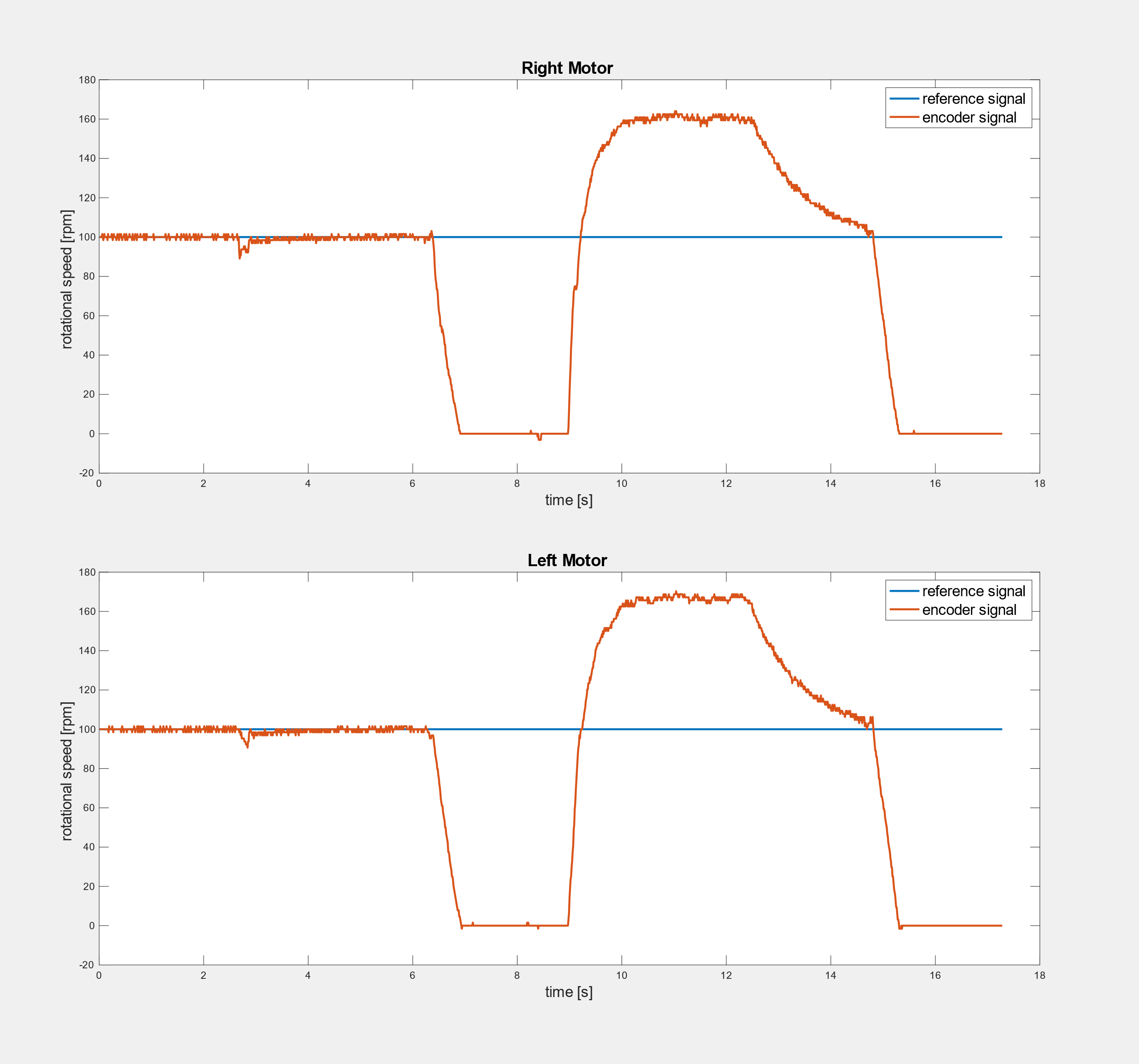


Figure 8: The results of the control using the Forward/Break technique without Anti-windup

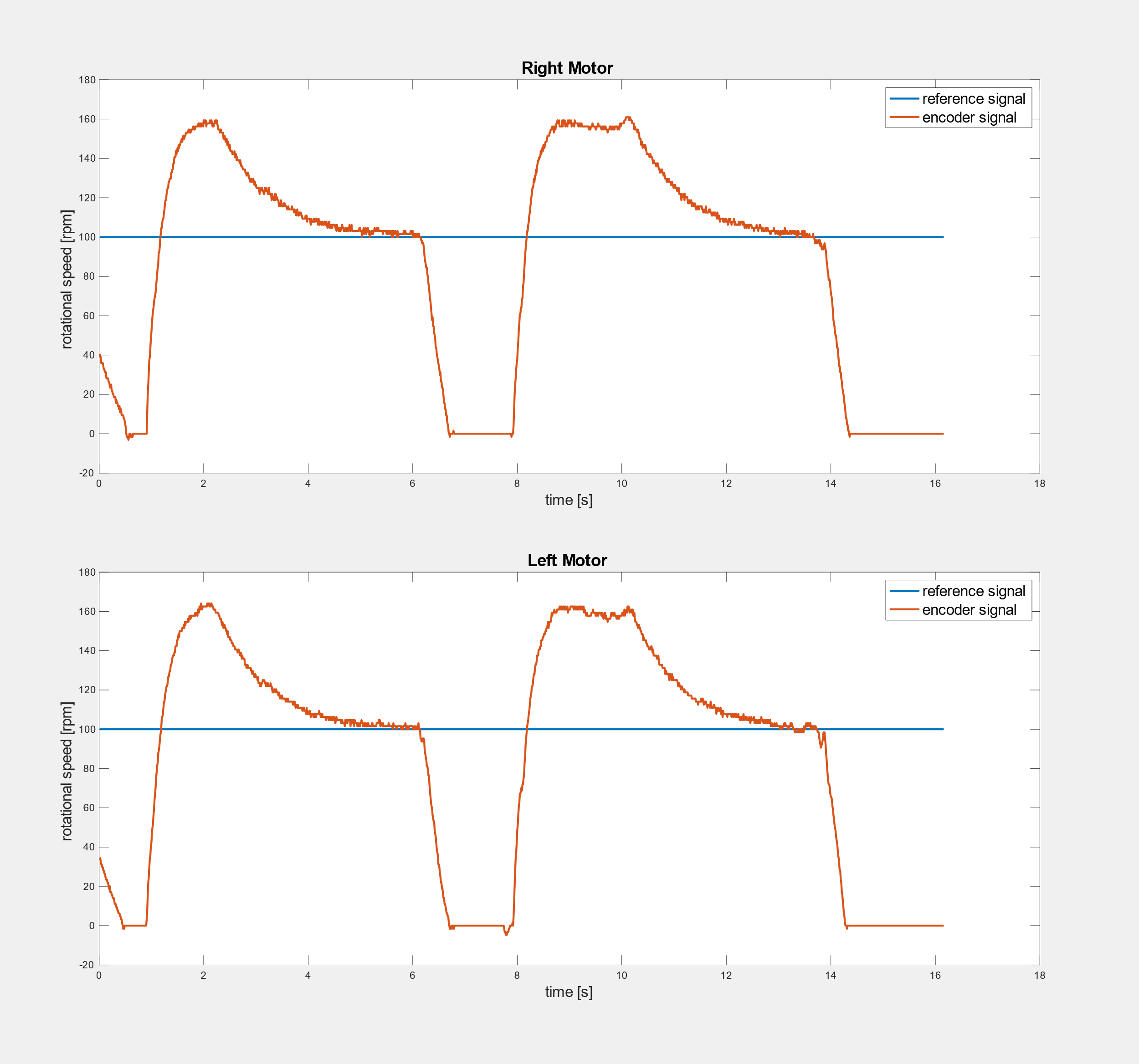


Figure 9: The results of the control using the Forward/Coast technique without Anti-windup