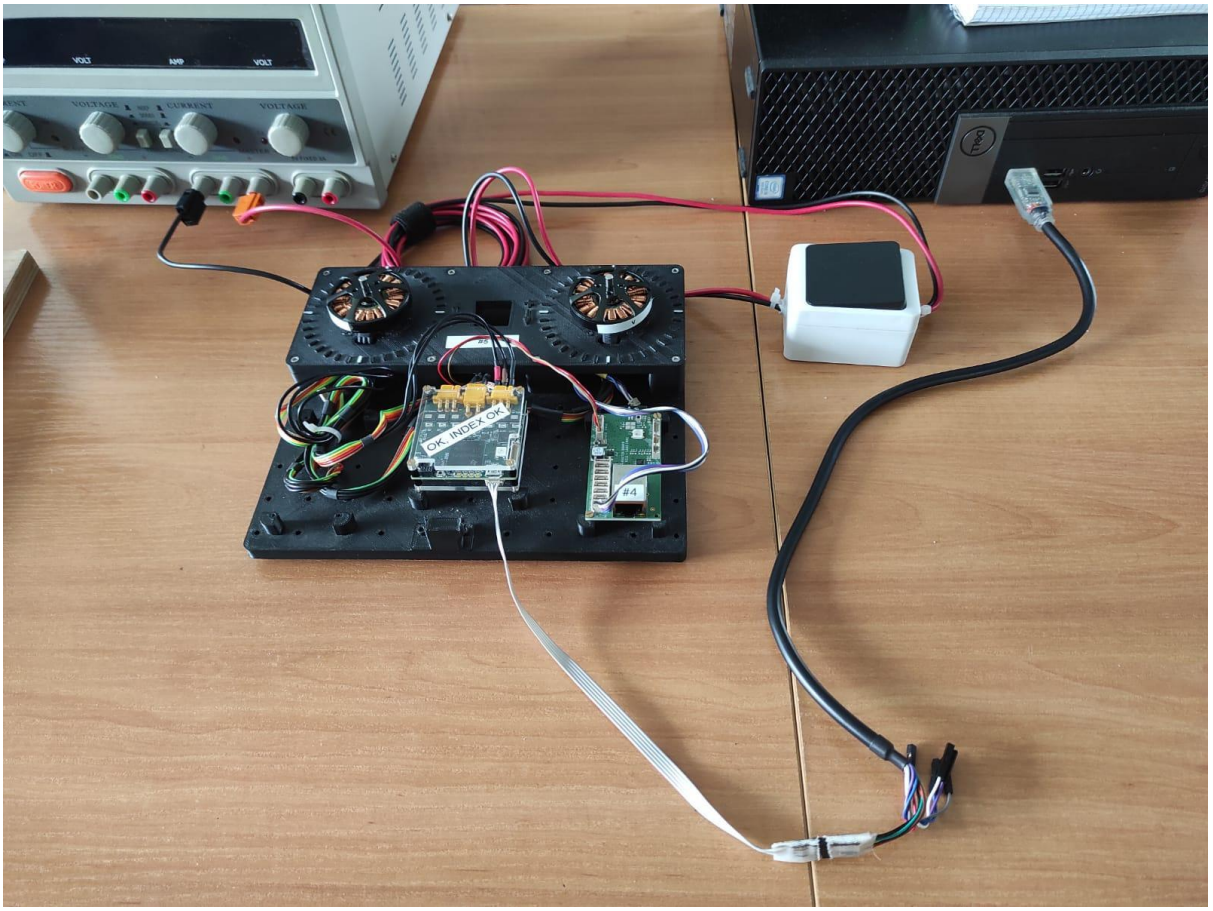
Cogging torque compensation

Introduction

The phenomenon known as cogging refers to the unwanted torque caused by the interaction between the permanent magnets of the rotor and the stator slots of a permanent magnet machine. At lower frequencies, this phenomenon can trigger oscillations in the speed control, impairing the stability and precision of the control in position.

The purpose of this report is to address and mitigate the cogging present in the motor of the Solo quadruped robot. To achieve this compensation, we implemented a position-based algorithm, as detailed in the article entitled "Cogging Torque Ripple Minimization via Position-Based Characterization".

Setup



Our test bench consists of a motor, MN4004-KV300, the Omodri driver controller and a C232HM-DDHSL-0 cable (used to communicate the Omodri with the computer). All the codes for the anti-cogging are at the following link.

Algorithm

The main purpose of the algorithm is to determine the current that keeps the motor at rest for each angular position. To achieve this, we use proportional-integral (PI) control based on the motor's position. When the motor position coincides with the desired position and the speed is zero, we record both the current and the current position of the motor. Subsequently, we set the desired position, which varies from 0 to 2π before making a complete turn in the opposite direction, oscillating between 2π and 0. The graphical representation of this data is shown in the following chart.

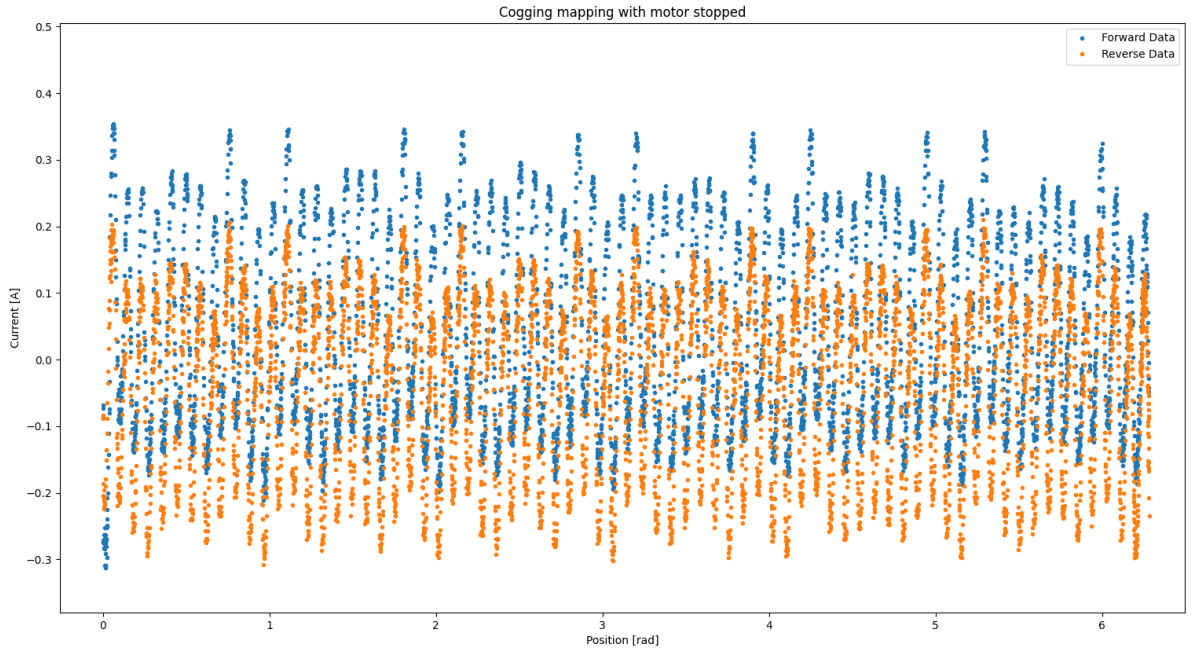


Figure 2: Position and currents of the cogging torque mapping at standstill

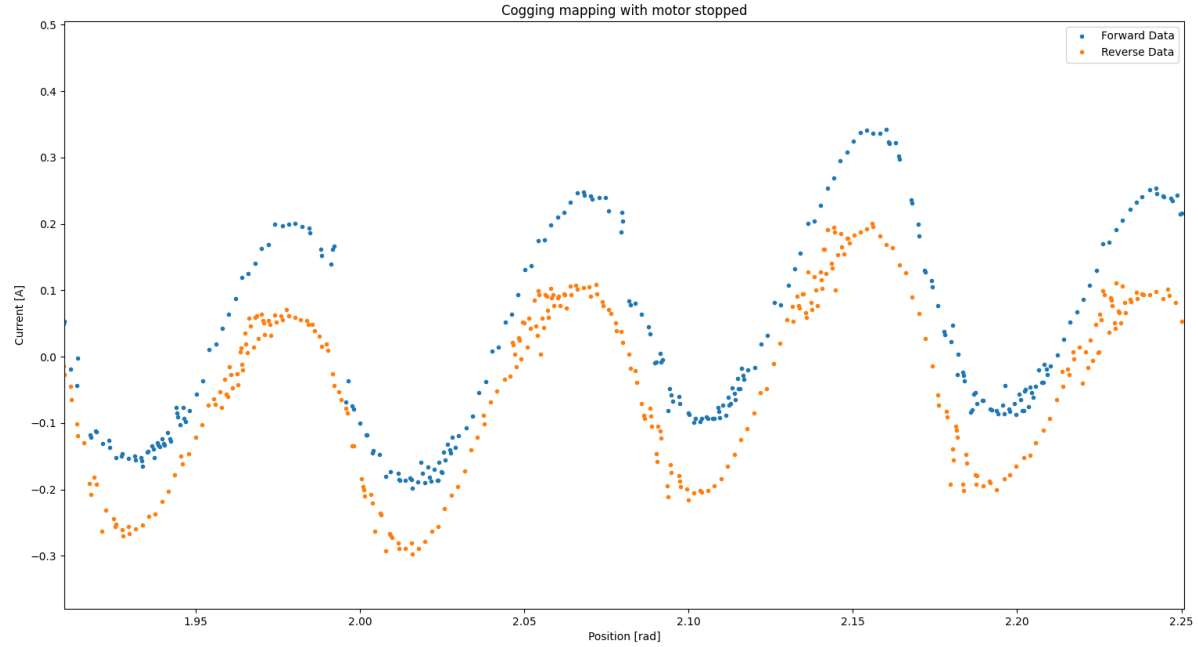


Figure 3: Figure 2 Magnified

We can see that the currents acquired during the first "forward" lap show a positive deviation from the currents of the second lap. This deviation is attributed to friction in the motor, known as stiction. When seeking stability in the commanded position, the required current can be expressed as the sum of $I_{sticion} + I_{cogging}$. It is important to note that the friction force is associated with the direction of the motor's movement, while the cogging torque depends exclusively on the position.

In this context, we can calculate the cogging current for a given position, represented by $I_{cogging\theta} = (I_{forward\theta} + I_{reverse\theta})/2$. If you want to calculate $I_{stiction}$ this is equivalent to half the difference between the two currents, i.e,$I_{stiction} = (I_{forward\theta} - I_{reverse\theta})/2$. It should be noted that, in our application, $I_{sticion}$ was not used.

Implementing position control

We implemented position control with the following equations:

$$error = \theta_{desired} - \theta_{actual}$$
$$ierr += error$$
$$I = kp * error + ki * ierr$$

Initially, the control was developed in a Python code with a communication frequency of 1 kHz with Omodri. However, due to the need for extreme precision and the rapid occurrence of small variations, the 1 kHz frequency was not enough to stabilize the motor in the desired position. Therefore, the control was subsequently implemented directly on the Omodri, which operates at a frequency of 40 kHz.

The controller originally took a long time to position the motor exactly on the reference. To mitigate this problem, the control condition has been relaxed, now requiring the error in position to be less than 0.002 radians and the speed to be equal to 0 to achieve stability. In addition, the encoder data is not obtained at the full resolution available. Although the encoder is divided into 20,000 intervals with a resolution of $2\pi 20000 \approx 0.0003200002\pi \approx 0.0003$ radians, the reference position is incremented by 0.002 radians with each piece of data obtained.

As for the controller constants, Kp and Ki were set to 30 and 20, respectively, to ensure that the motor reacts effectively to small position errors. However, starting operation with such a high value for Kp can make the controller unstable. It is therefore recommended to start with Kp = 3, allow the motor to get closer to the reference and gradually increase Kp to 30. The value of Kd was kept at 0.006, which is a standard value in other controllers in the project.

It's important to mention that, to make it easier to adjust the constants, we used Pygame in the computer code, mapping the above constants. In the specific case of mapping the cogging chain, this process took 26 minutes.

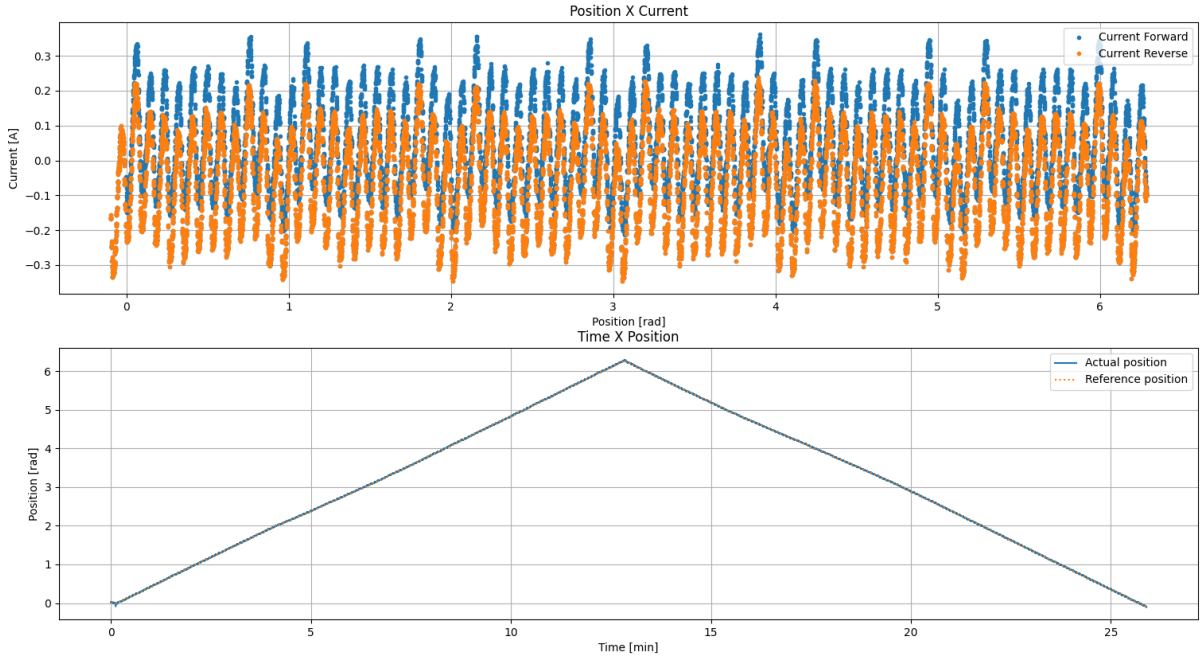


Figure 4: Motor data during the experiment. Note that the final time was 26 minutes.

Generating the anti-cogging chain table

To calculate the current that will compensate for the effects of the cogging torque, $I_{cogging}$, it is necessary to average the forward and reverse currents. In addition, we want this current to be evenly spread out in space, since we will be modeling the cogging current with Fourier sequences. The algorithm for generating the desired cogging current behaves as follows:

1. Divide the space into intervals of 0.002 radians, we'll call each interval θ_i.
2. Calculate the average forward current in this interval, I_{fi}
3. Calculate the average reverse current in this interval, I_{ri}
4. Calculate $I_{cogging}$ by computing the average of the two currents above, $I_{cogging} = (I_{fi} + I_{ri}) / 2$

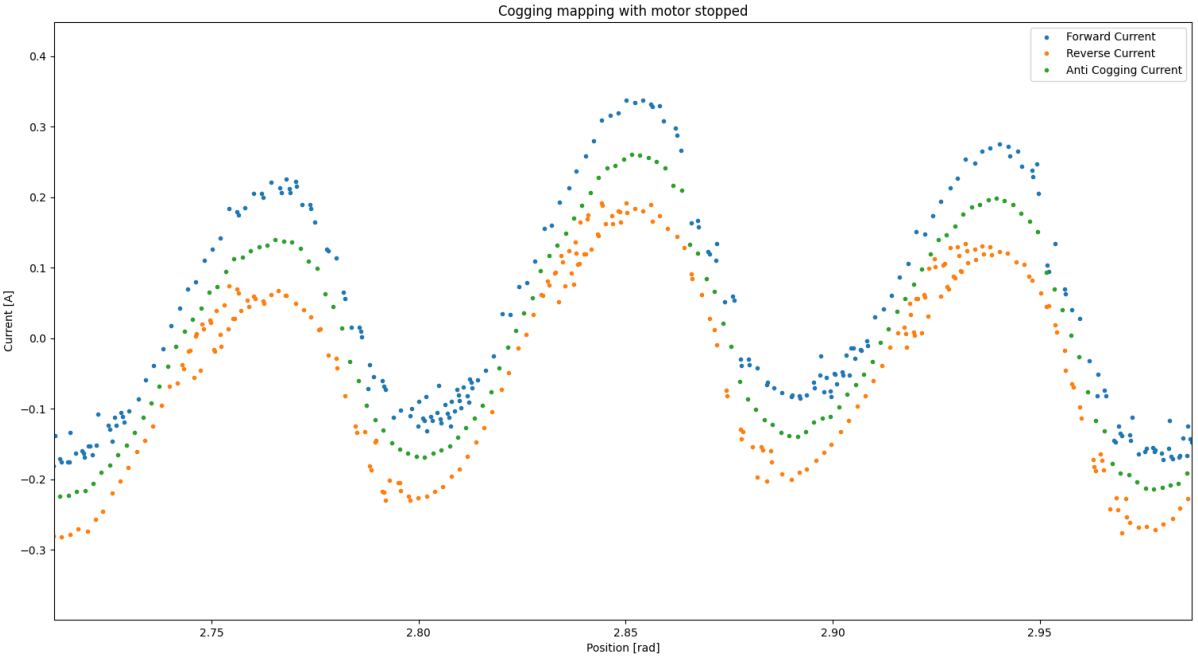


Figure 5: Forward, reverse and anticogging currents

The graph above shows I_f e I_r e $I_{anticogging}$ calculated, respectively in blue, orange and green. The next step was to model the cogging current with fourier functions and increase the number of points. I used 7200 points and obtained the data in the graph below.

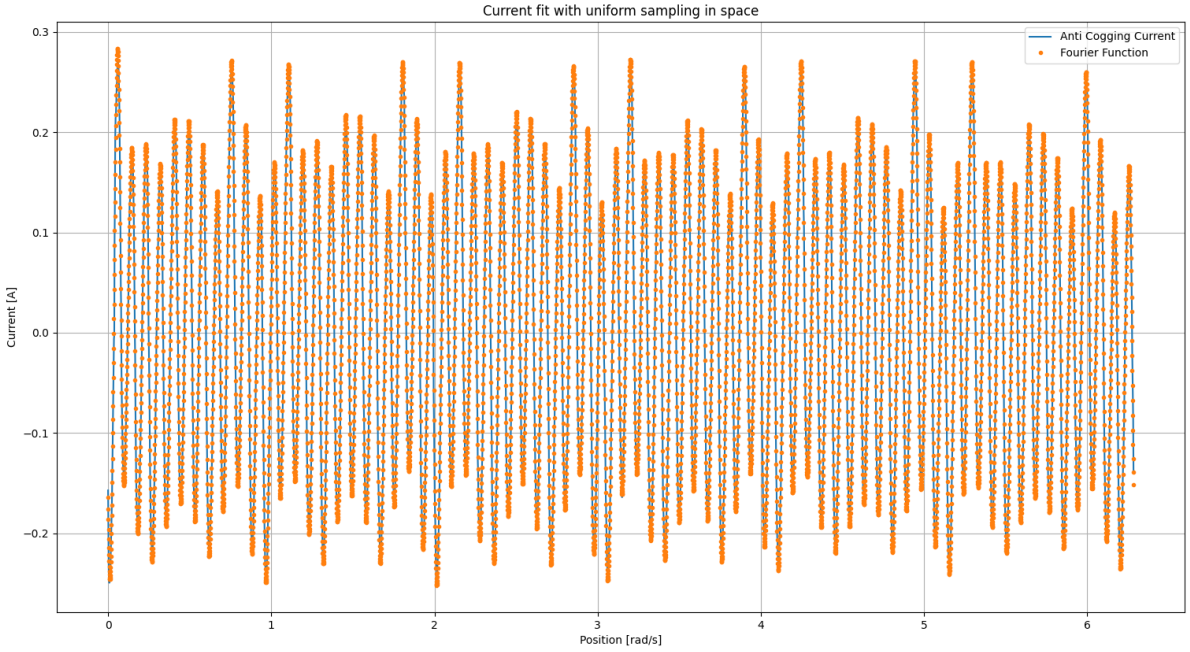


Figure 6: Fourier fit in the anticoggin current

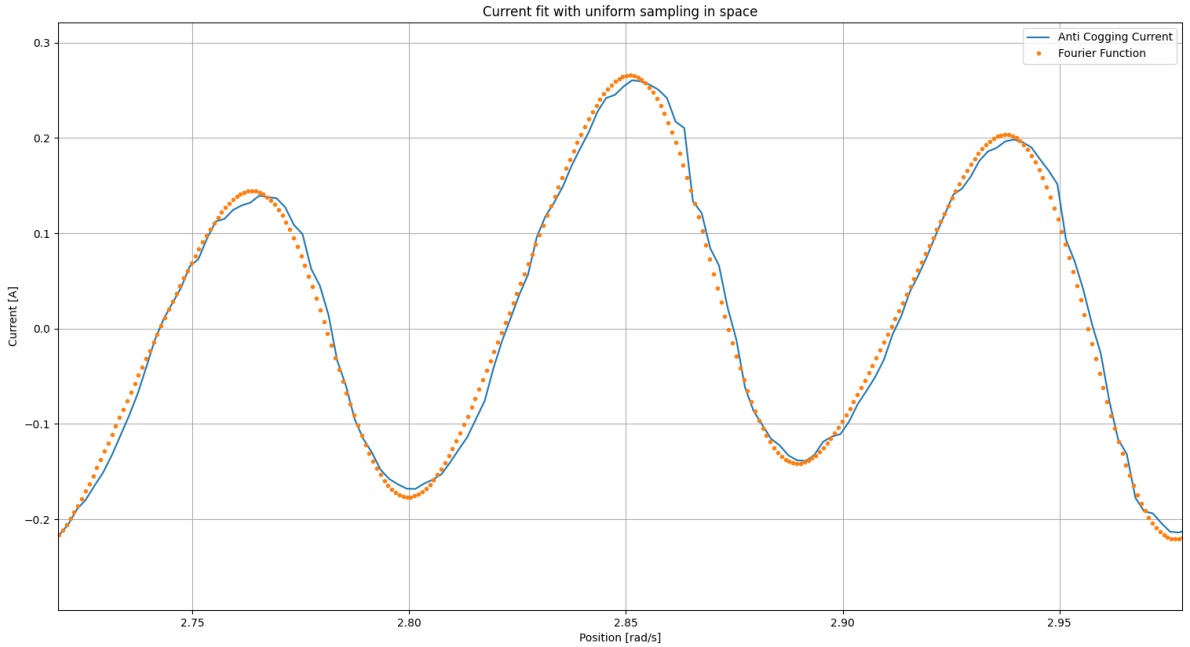


Figure 7: Figure 6 Magnified

Applying the anti-cogging chain

At this stage, $I_{anticogging}$ is represented by 7200 floats between -03 and 0.3, and this is a problem because it takes up a lot of space. To solve this, I multiplied the values by 2^{16} and used a vector of 16-bit signed integers to store the values. This strategy halved the space needed compared to floating point storage.

The resolution of a 16-bit signed integer to represent an amplitude interval of 0.6 is equivalent to $0.6/2^{16} = 0,000009155$. As the resolution of the current is in the order of milliamps, we will not suffer any loss in resolution when using this representation by integers. It's worth noting that the smallest packet size supported by our microcontroller, the F28388DZWTS, is precisely 16 bits.

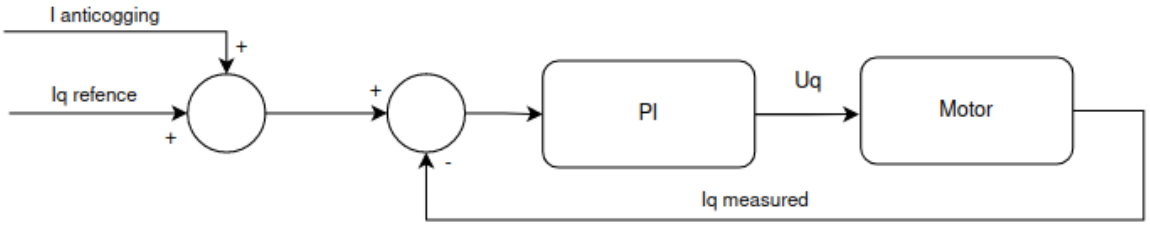


Figure 8: Diagram of PI control of Uq with anticogging current

As for where the cogging current is applied, it must be added to the desired reference current value just before it is applied to the PI control that will generate Uq, as shown in the diagram above.

Results

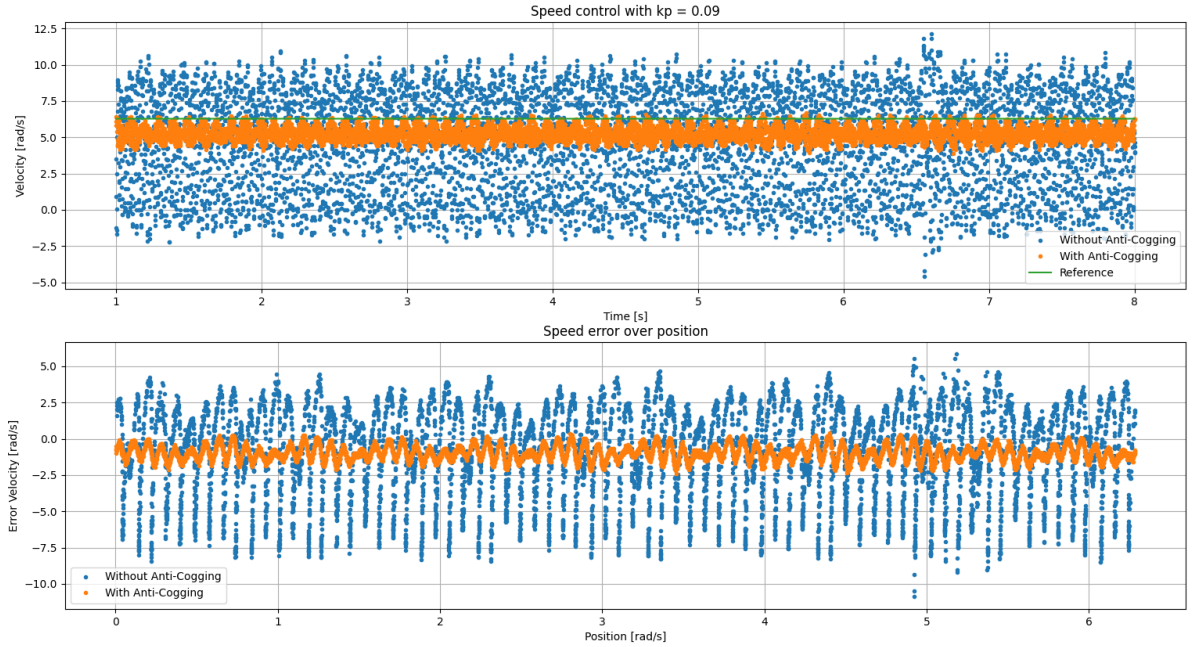


Figure 4: Comparison of the effects of anti-cogging on speed control

To evaluate the effectiveness of the anti-cogging system, we carried out a test with a proportional speed control, $I = (V_{reference} - V_{actual}) * kp$. The upper graph shows the reference speed, 2πin green, the motor speed over time without anti-cogging (in blue), and the motor speed with anti-cogging (in orange). The bottom graph shows the speed errors for each test.

It is clear from the graph that the oscillations are significantly reduced with the implementation of anti-cogging. Calculating the squared error from the second graph shows a reduction from 14 to 1.3 (rad/s)².

In addition, another notable effect, although not possible to visualize in this report, is the tactile experience when manually turning the motor. Normally, we notice the oscillations caused by cogging. However, by activating anti-cogging, we observed a clear reduction in these effects during manual rotation of the motor.

Instructions for mapping or testing anticogging

Accessing the cogging repository we have the following folder structure

- codes
  - Mapping
    - get_data_stopped.py
    - generate_tableau.py
    - cogging_tableau.txt
  - Testing
    - test_anticogging.py
- open_motor_drive_initiate_master

To carry out the mapping:

1. Access the code from the folder "open_motor_drive_initiate_master" m
2. Change the MAPPING_COGGING_ENABLE variable to 1 in the foc.h file
3. Pass the code to uOmodri
4. Open the file get_data_stopped.py in the Mapping folder
5. Launch this python code
6. Turn the motor by hand until the encoder is indexed
7. Increase the constant Kp to a value of 30
8. Wait for the code to finish running and create the "stopped_data/stopped_data.txt" file. It takes about 30 minutes
9. Launch the code generate_tableau.py
10. Copy the vertor declaration from the file "cogging_tableau.txt" and insert it at the end of the FOC.H file.

To carry out the anticogging test:

1. Access the code from the folder "open_motor_drive_initiate_master" m
2. Change the MAPPING_COGGING_ENABLE variable to 0 in the foc.h file
3. Pass the code to uOmodri
4. Open the test_anticogging.py file in the Testing folder
5. Launch this python code
6. Turn the motor by hand until the encoder is indexed
7. At this point, you should feel that the anticogging has been applied and the engine turns smoothly