LAAS internship report - Heitor

Heitor Abreu de Andrade

LAAS internship report - Heitor

ABREU DE ANRADE Heitor

With guidance from

FLAYOLS Thomas

SAUREL Guilhem

MANHES Jérôme

Summary

Introduction to temperature estimation

This report describes the development process for estimating the temperature of a three-phase motor. Our approach involves using two different models: a thermal model, which is based on the relationship between the energy consumed and the current supplied to the motor, and a resistive model, which deduces resistance and temperature based on speed and current measurements. Finally, we apply a Kalman filter to combine the estimation of the two models. The specific motor in focus is the MN4004-25 (KV: 300), however, it is worth noting that the algorithm developed can be replicated on other motors that are equipped with encoders and current sensors.

Thermal Model

Introduction

The thermal model is used to calculate the rate of temperature change based on the electric current and the temperature difference in relation to the environment. The mathematical calculations for deriving the thermal model, the method used to determine the constants and the results obtained will be presented below. The thermal model is based on the method in Ben Katz's blog.

Mathematical Model

ΔT: Temperature difference between motor and environment
Q_{in}: Thermal energy received by the motor
Q_{out}: Thermal energy released by the motor
C_{th}: Thermal capacity of the motor
R_{th}: Thermal resistance of the motor
i: Current fed into the motor
R: Motor resistance

Using a 1st order thermal model:

$$\Delta T = \frac{Q_{in} - Q_{out}}{C_{th}}$$

$$I.\ \ \Delta T' = \frac{Q'_{in} - Q'_{out}}{C_{th}}$$

$$Q_{in} = i^2 * R * t$$

$$II.\ \ Q'_{in} = i^{2} * R$$

$$III.\ \ Q'_{out} = \frac{\Delta T}{R_{th}}$$

$$II,\ III \rightarrow\ I$$

$$\Delta T' = \frac{i^{2} R}{C_{th}} - \frac{\Delta T}{R_{th} * C_{th}}$$

$$\Delta T' = K_1 * i^2 + K_2 * \Delta T$$

Also,

$$\Delta T' = (T_{motor} - T_{ambient})' = T'_{motor}$$

$$T'_{motor} = K_1 * i^2 + K_2 * \Delta T$$

Running the experiment

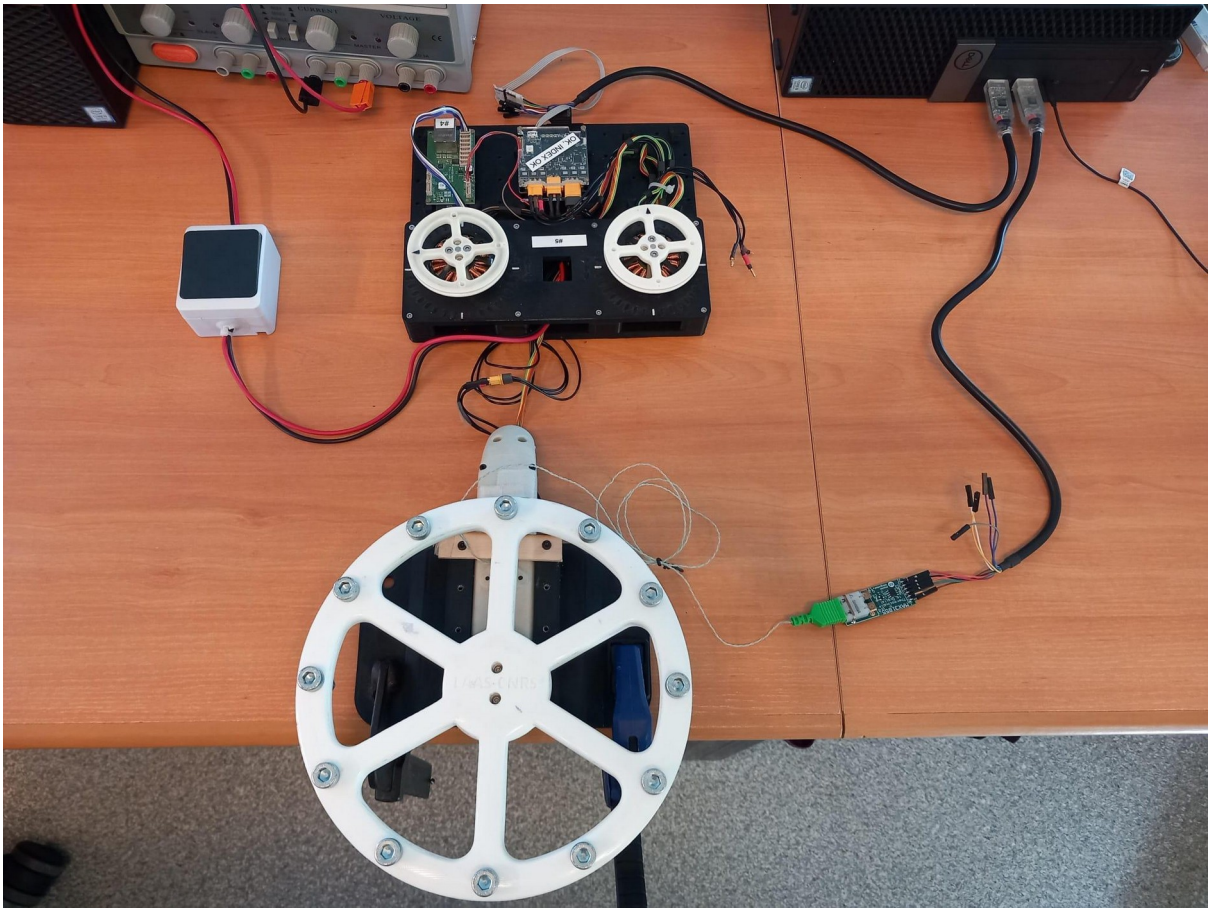

Fig. 1: Setup with motor, Omodri and connectors C232HM

mo
tem dri
to t

d
we'
position, id, iq, ud and uq of our motor.

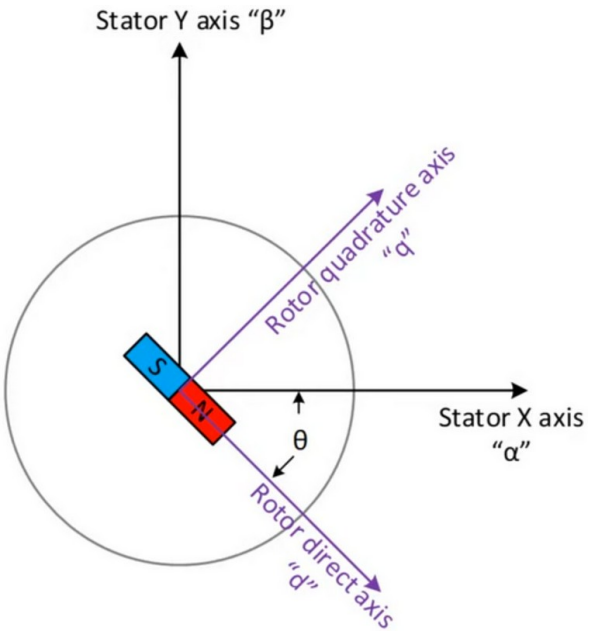

Fig. 3: "q" and "d" rotor axis

The brushless motor has two axes, the quadrature rotor, "q", and the direct rotor, "d". The current id mentioned above means the projection of the current passing through the motor on the "d" axis. Note that as id is aligned with the rotor axis, it does not generate torque and we can use it to heat the motor without it moving.

For the experiment in question, we calculated the error in the motor's position, and set iq to be proportional to the error. In this way, we can control the current by turning the motor by hand. Note: When the code starts, it assumes the position of the motor at that moment to be 0.

```
error = ud.position1 - 0 kp=5/28
I = (error*kp)
```

In this setup, we collect temperature data (from the sensor) and motor current data (from the Omodri) over time.

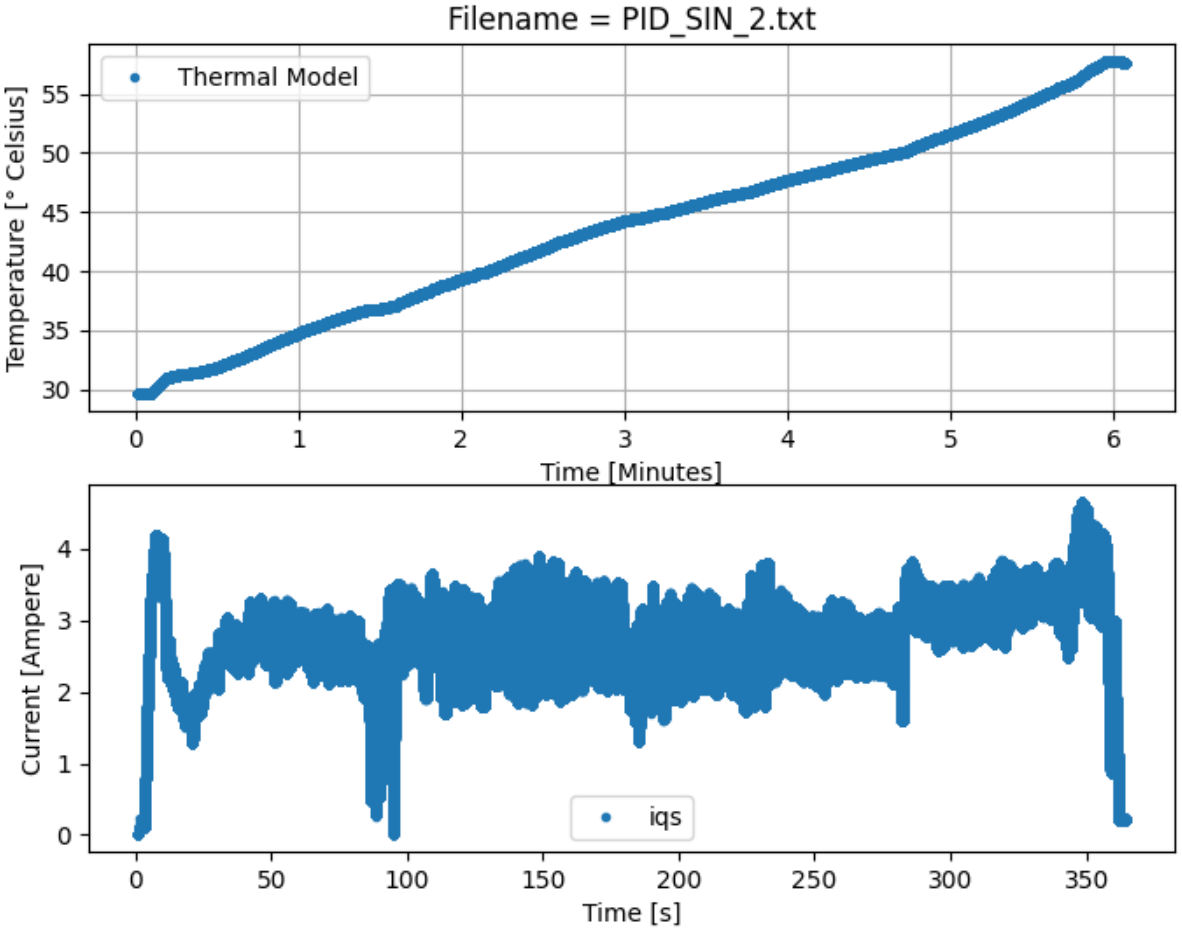

Fig. 4: Motor temperature and current data

Calculating model variables

$$\Delta T' = K_1 * i^2 + K_2 * \Delta T$$

We created a code that computes the variables $\Delta T'$, i^2 *and* ΔT every 5 seconds. Basically, at each time interval, we calculate i^2 as the average of the currents passing through the motor. $\Delta T'$ and the motor temperature at the end of the interval minus the temperature at the beginning of the interval, divided by 5 seconds. Finally ΔT is the motor temperature at the end of the interval minus the ambient temperature.

```
"""
   Compute the arrays variables needed to the thermal model ΔT' =K1*i**2+K2*ΔT
   ΔT'     = deltas_temperature_derivative
   i**2    = is_squared_mean
```

```
   ΔT      = deltas_temperature
"""

# Initialize variables
is_squared_mean, deltas_temperature,deltas_temperature_derivative = [], [], []
time, is_squared, count = 0, 0, 0

# Initialize variables with temperature initial of the motor
T_AMB = temps_measured[0]
motor_temperature = T_AMB

# Time interval to compute and store the data
dt=5 # seconds

for i in (range(len(times))):
   # Add current squared to do the mean after 5s
   is_squared += ids[i]**2 + iqs[i]**2
   count += 1

   if(times[i]-time>=dt):

      # Compute and Store i**2ΔT and ΔT'
      is_squared_mean.append(is_squared/count)
      deltas_temperature.append(temps_measured[i] - T_AMB)
      deltas_temperature_derivative.append((temps_measured[i] - motor_temperature)/dt)

      # Reinitialize variables
      is_squared = 0
      count = 0
      time=times[i]
      motor_temperature = temps_measured[i]
```

Finding constants K1 and K2

Our model is linear so we can do least squares to find the constants. To do this, we use the statsmodels library.

```
import pandas as pd
import statsmodels.api as sm

# Creating data frame with data
x = pd.DataFrame({'is_squared_mean': is_squared_mean, 'deltas_temperature':
```

```
deltas_temperature})
y = deltas_temperature_derivative

# Fitting the model
model = sm.OLS(y, x).fit()

# Getting constants
K1 = model.params["is_squared_mean"]
K2 = model.params["deltas_temperature"]
print(f'ΔT' = {K1}*i**2 + {K2}ΔT")
```

Calculating the temperature with the thermal model

The temperatures are calculated at each iteration of the two-step code:

1. Calculate the temperature derivative: $T'_k = K_1 * i_k^2 + K_2 * \Delta T_k$
2. Integrate the derivative over time: $T_{k+1} = T_k + dt * T'_k$

```
   # Getting data from the experiments
   times, ids, iqs, vds, vqs, velocities, temps_measured, positions =
get_data(filename)

   # Initializing the estimation with the temperature of the motor measured
by the sensor
   temps_thermal=[temps_measured[0]]

   # Computing temperatures with the thermal model
   for i in range(len(times) - 1):

       current = ids[i]*ids[i]+iqs[i]*iqs[i]                  # i**2
       delta_t = temps_thermal[-1] - T_AMBIENT                # ΔT
       temperature_derivative = (current * K1) + (delta_t * K2)# ΔT'
       DT = times[i+1] - times[i]                             # dt

       temp_thermal = temps_thermal[-1] + ((temperature_derivative) * DT) #
T[k+1] =T[k] + dt*T'[k]
       temps_thermal.append(temp_thermal)
```

Results

The model was trained with the data shown in Figure 4, and below we present the results of the tests in which it was trained, as well as the results of other tests carried out with the engine. A crucial point to note about the thermal model is that it is unable to

estimate the initial temperature on its own. Therefore, in the tests carried out, the first temperature of the thermal model is obtained from the temperature sensor. Later on, we'll discuss how we deal with this initial estimate in the thermal model. In summary, the thermal model follows the measured temperature, especially during temperature increases, with a maximum difference of 10 degrees during cooling.

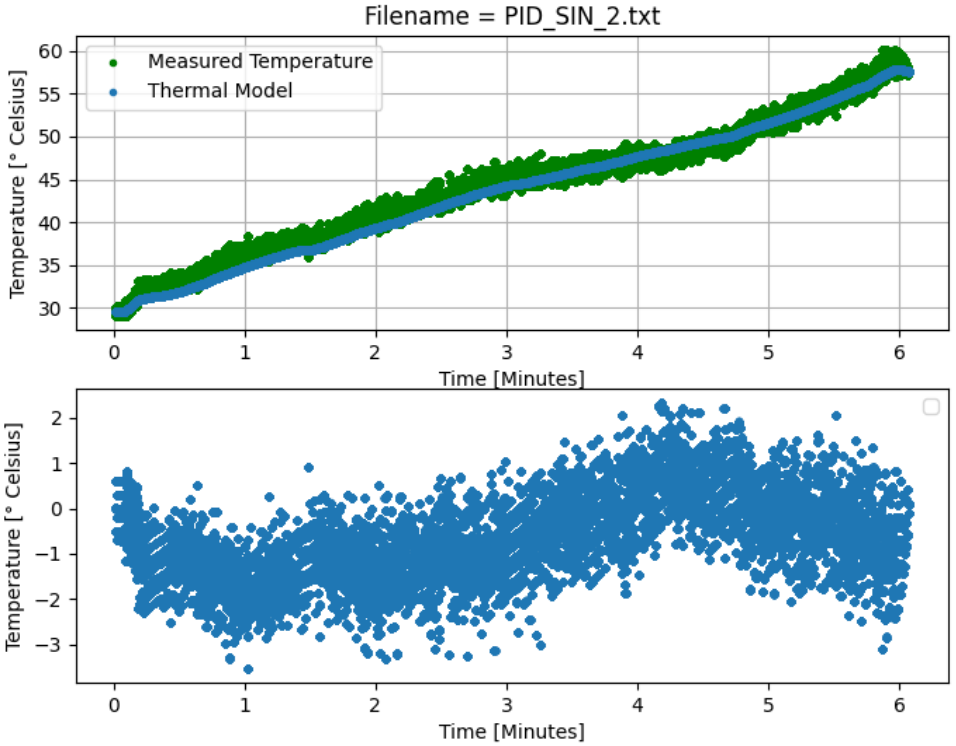

Fig. 5: Estimation of the thermal model and its error (1)

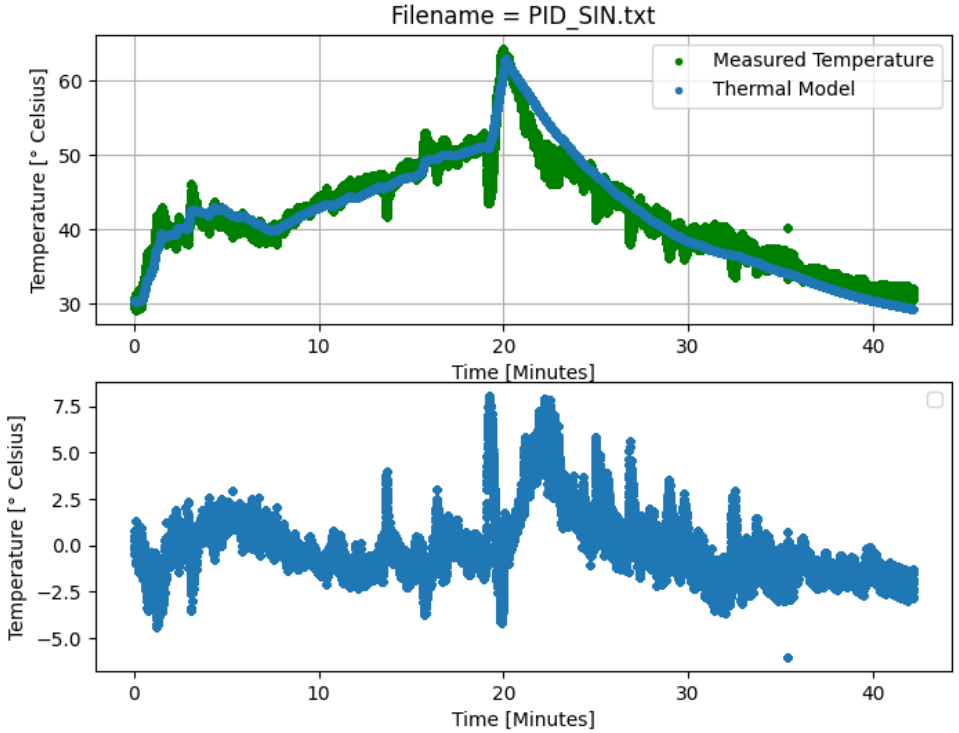

Fig. 6: Estimation of the thermal model and its error (2)

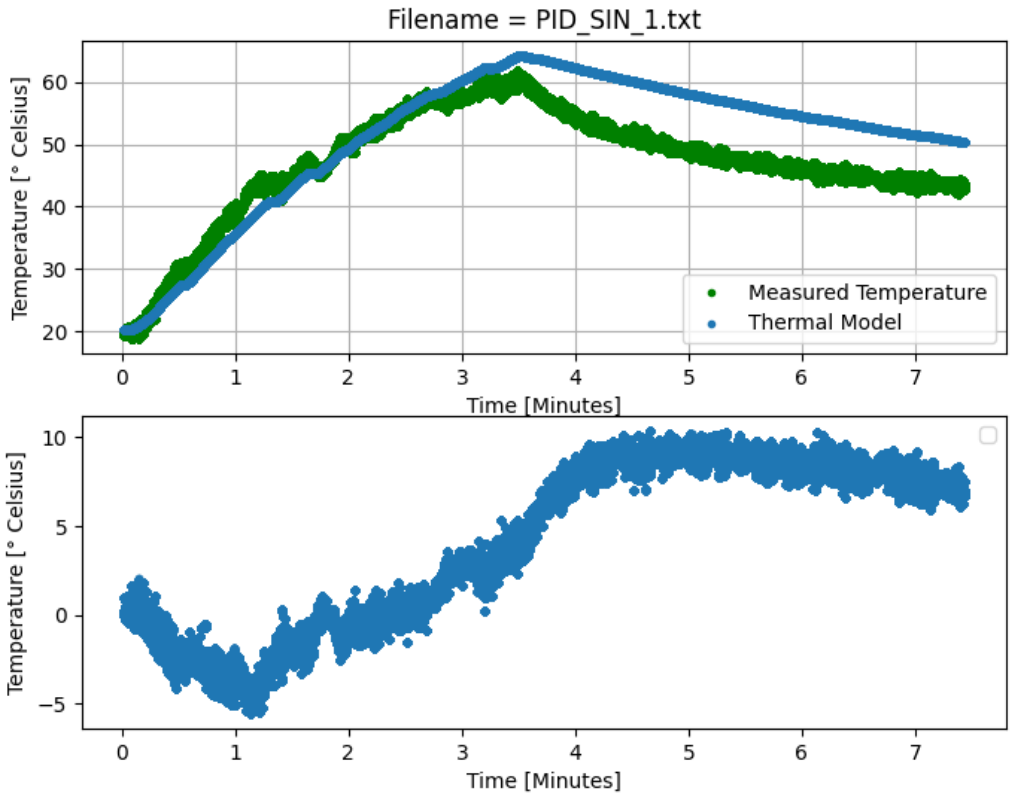

Fig. 7: Estimation of the thermal model and its error (3)

Resistive model

Introduction

The resistive model starts from the premise that variations in temperature are proportional to variations in resistance. We can therefore obtain an indirect measure of temperature by measuring resistance.

Non-linearity

Ben Katz warns in his blog about the non-linearity between motor voltage and current. This non-linearity is caused by the dead time in the motor driver's commutation, and it generates a limitation in the resistance calculation.

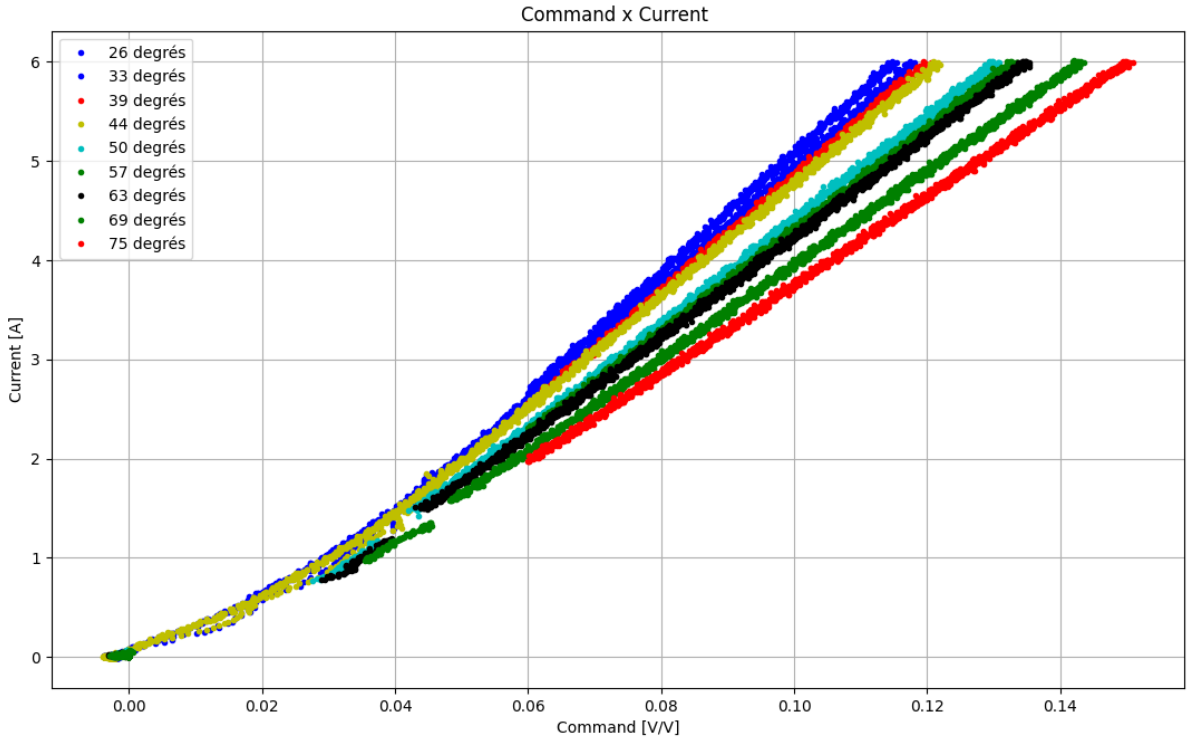

Fig. 8: Command and current ratio at different temperatures

The graph above shows the relationship between the command/PWM applied to the motor and the current generated. We observed this relationship at different temperatures, and each temperature results in a different curve on the graph. This graph shows two important things. Firstly, we can differentiate between temperatures based on voltage and current, as the curves have different slopes. However, this is only safe from about 2 amperes; below this value, the curves start to overlap. Secondly, we note that the resistance does not strictly follow the relationship $R = U/i$. This is evidenced by the fact that, when we project the red line further to the right, it does not intersect the origin. Therefore, for a more accurate approximation, we will use the expression $R = (U + B)/i$

Mathematical model with the engine stopped

Considering that temperature is an affine function of resistance, we can derive our first resistive model.

$$R = \frac{U-B}{i}$$

$$T = aR + b$$

$$T = \frac{aU}{i} - \frac{aB}{i} + b$$

$$T = k_1 \frac{u}{i} + k_2 \frac{1}{i} + k_3$$

Finding constants

Once again, we will use the least squares method to determine the model's constants. The data used to train the model is voltage, current and temperature, shown in the graph in Figure 8.

```
# Computate resistance(u/i) and inv_current(1/i)
resistance = [tension[i]/current[i] for i in range(len(i))]
inv_current = [1/current[i] for i in range(len(i))]

x = pd.DataFrame({'Resistance': resistance, 'inv_current': inv_current})
x = sm.add_constant(x)
y = temperatures

# Fit the Model
model = sm.OLS(y, x).fit()

# Get Constants
k1 = model.params["Resistance"]
k2 = model.params["inv_current"]
k3 = model.params["const"]
```

With this, we obtained the following mathematical model

$$T = 490\frac{u}{i} + - 125\frac{1}{i} - 97.5$$

Results

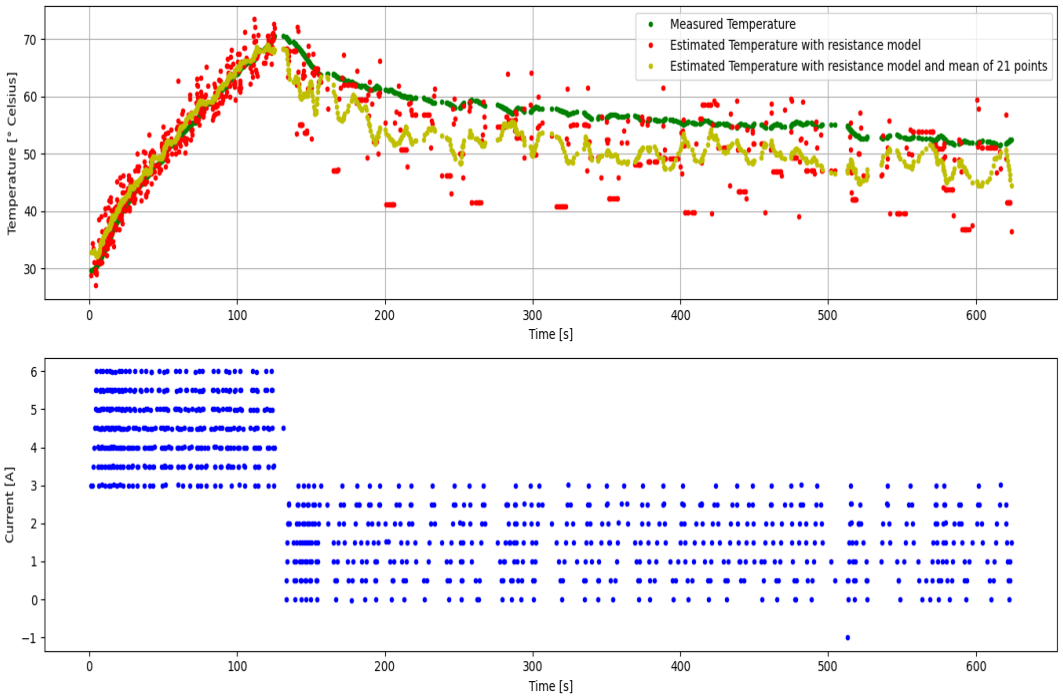

Fig. 9: Estimation of the resistive model and current used in the test

In the upper graph, we show the engine temperature captured by a thermocouple in green, the temperature estimate calculated by the resistive model in red and an average of the estimates in yellow.

The lower graph shows the current applied to the motor. Each time we measure the temperature, we vary the current by 0.5 Amps up or down. During the motor's heating process, the current remains between 3 and 6 A until the motor reaches 70 degrees Celsius. On the other hand, during the cooling of the motor, we send currents between 0 and 3 A.

We noticed that the resistive model tracks the temperature measurement well when the currents are above 3 A. While the results are more inaccurate with currents below 3 A. In the worst case, the average of the model has an error of 11 degrees.

Mathematical model with the engine running

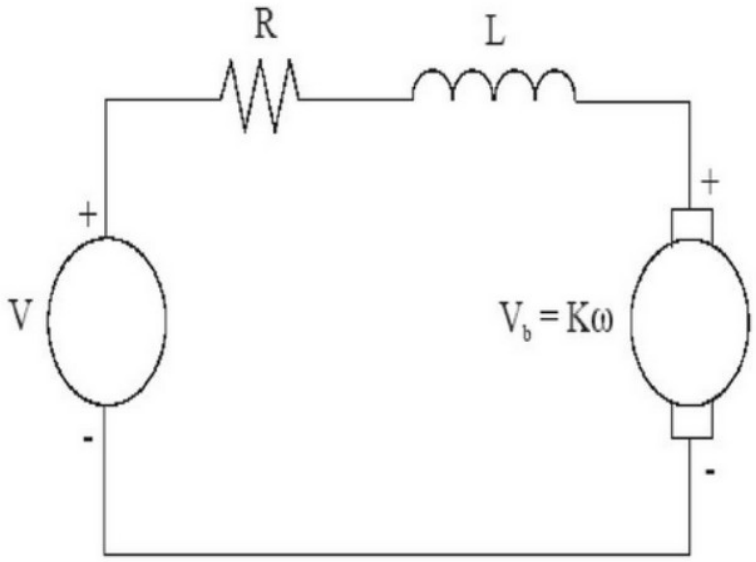

Fig. 10: Motor wiring diagram

To model the engine running we assume the following:

1. The voltage on the inductance $L\frac{di}{dt}$ is 0, because the temperature bandwidth is very low compared to the dynamics of the current control $\frac{di}{dt}$
2. The applied voltage, V, has the effects of dead time, so we can express V as $V = U_{commanded} + B$

Thus:

$$U - B = Ri + Kw$$

$$R = \frac{U - B - Kw}{i}$$
$$T = aR + b$$
$$T = \frac{aU}{i} - \frac{aB}{i} - \frac{aKw}{i} + b$$
$$T = k_1 \frac{u}{i} + k_2 \frac{1}{i} + k_3 \frac{w}{i} + k_4$$

This is the original modeling of the rotating motor, however, the least squares method did not converge on an adequate result with this model. Next, we will present the modifications that allow the least squares method to find the desired constants.

Kr calculation

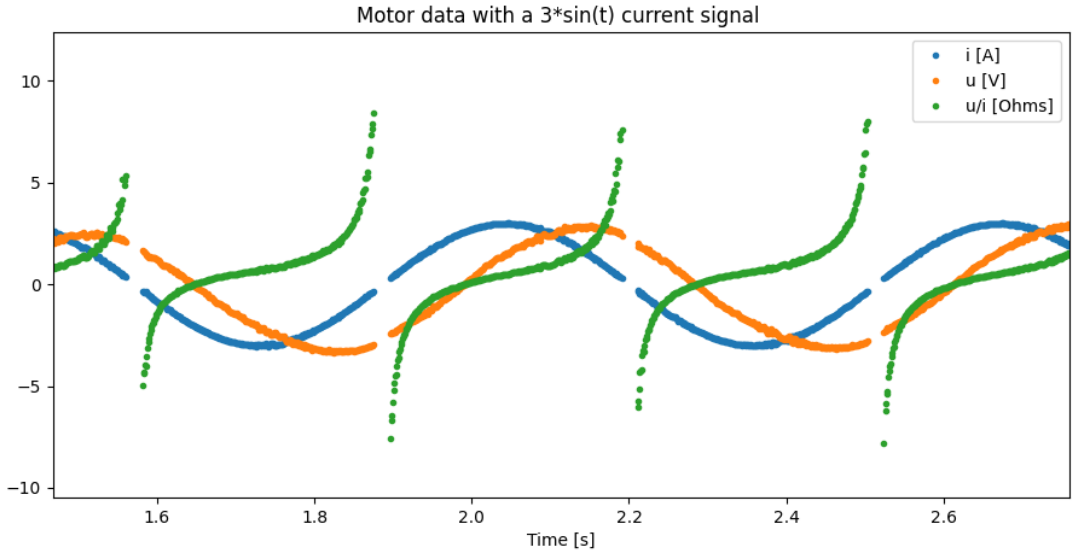

Fig. 11: Motor current, voltage and "resistance"

The graph above shows the current in blue, the voltage command sent to the motor in orange, and the motor's "resistance at standstill" (u/i) in green. Naturally, there is a lag between the voltage applied to the motor and the current, caused by the counter-electromotive force. On the other hand, we know that the voltage across the resistor must be in phase with the current. With this premise, we will try to find one of the model's constants.

Ignoring the non-linearity of the voltage, U, we can find the following equation:

$$U_r = Ri = U - K_r w$$
$$i = (U - K_r w)R$$
$$i = (U - K_1 w)K_2$$

With this equation, we'll perform a curve fit to find out Kr and k2 and use the equation $R = (U - K_r w)/i$ to calculate the resistance.

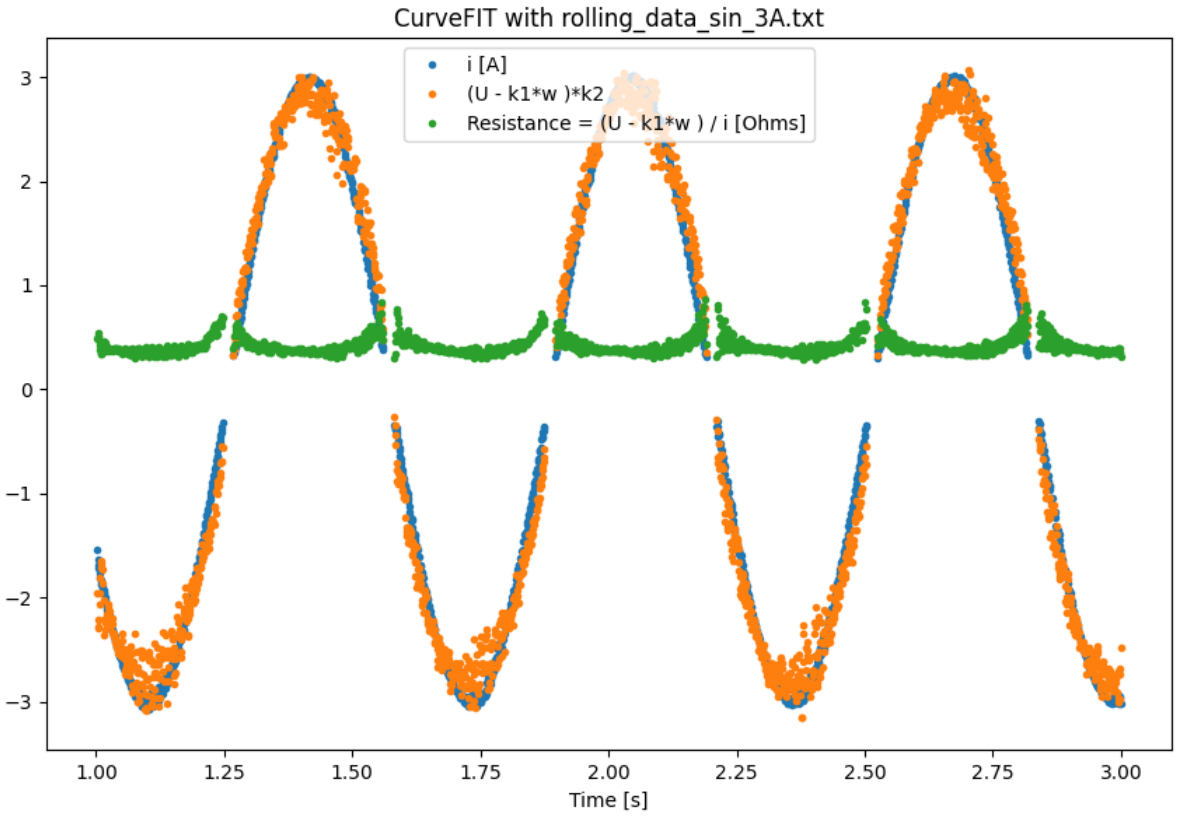

*Fig. 12: Fitting the equation (U-k1*w)*k2 to the current*

In the graph above, we have the current in blue, the result of the curve fit in orange and the calculation of the resistance with the equation mentioned above.

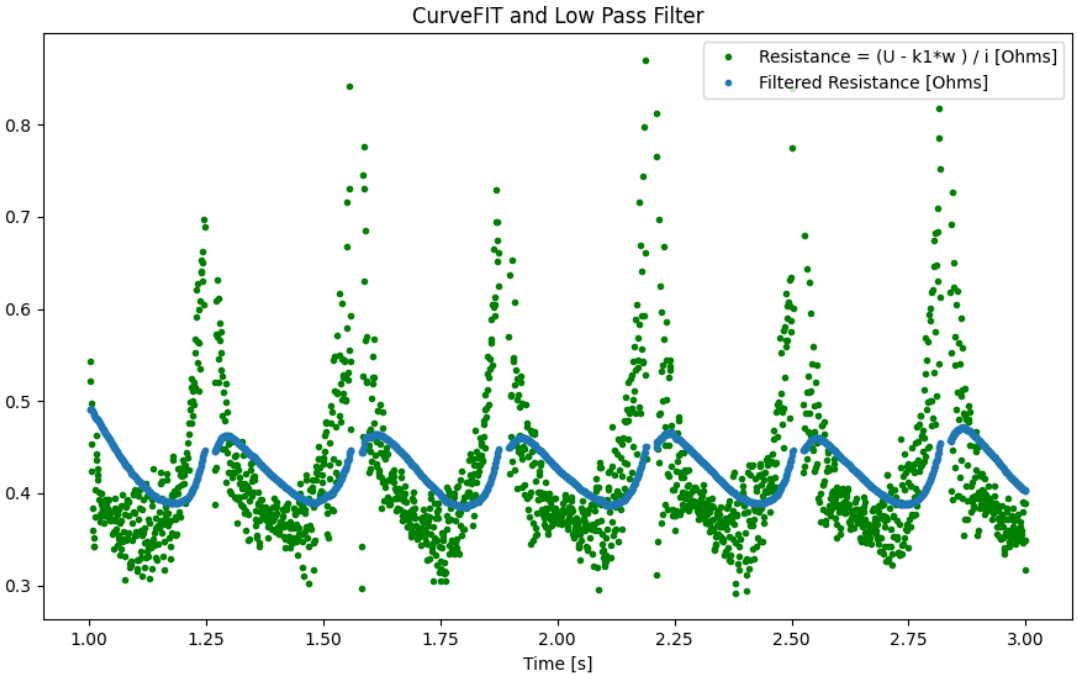

Fig. 13: Approximate graph at the estimated resistance

The graph above shows the details of the resistance calculated in the previous step. We can see that the calculated resistance, in green, still has a lot of variation, so we applied a low-pass filter. The low-pass filter is calculated as follows:

filtered[i+1] = 0.01 * measured[i] + 0.99 * filtered[i]

Where "filtered" are the new filtered values and "measured" are the raw resistance values.

Considering the non-linearity of the voltage

After finding Kr, we had to consider the non-linearity of the voltage in our model, so we used voltage = U - B.

$$T = a\frac{(U-B)-\frac{Kw}{r}}{i} + b$$

$$T = a\frac{U-\frac{Kw}{r}}{i} - a\frac{B}{i} + b$$

$$T = k_1\frac{U-\frac{Kw}{r}}{i} - k_2\frac{1}{i} + k_3$$

Thus, a new term emerged, 1/i. This term initially faced the same noise problem as the resistance, however, we can't simply apply a low-pass filter because, as it is an AC signal, the filter would have a point centered on 0. To resolve this issue, we start working with the absolute value of the current and then apply the low-pass filter.

With the model we have so far, the temperature depends only on our resistance and the inverse of the current. However, this model presented a problem which can be seen in the image below.

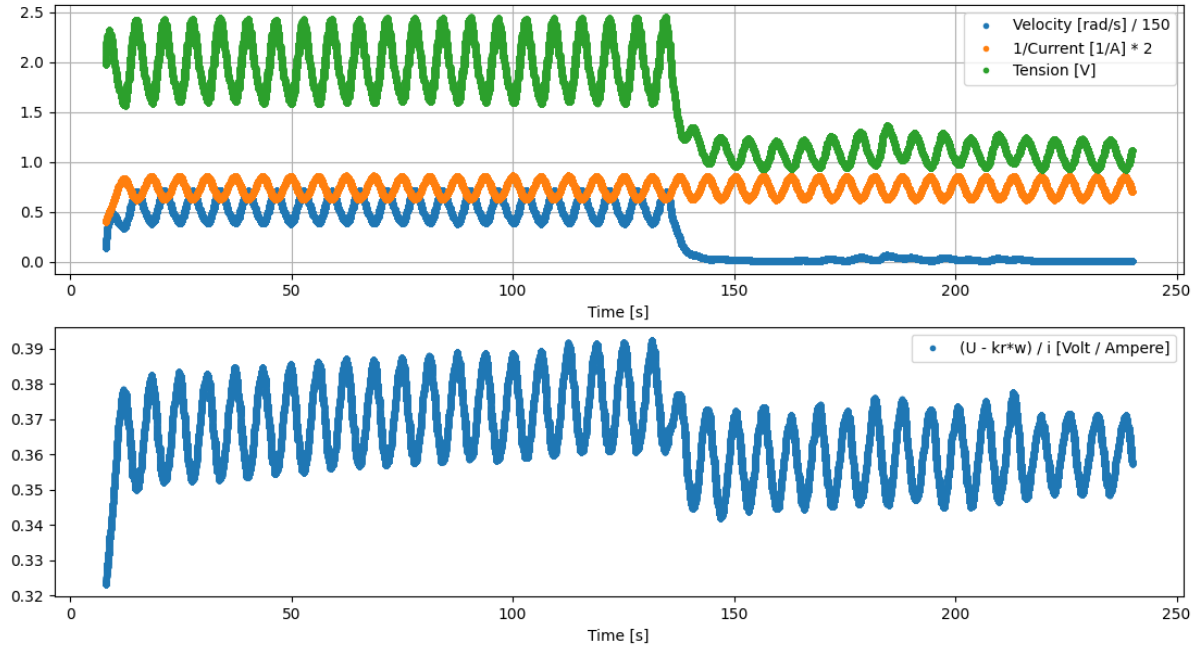

Fig. 14: Motor parameters after filtering

In the first graph we can see the speed in blue, the inverse of the current in orange and the commanded voltage in green. The lower graph shows our resistance. We noticed that at around the second 140, there was a considerable drop in speed, which led to a drop in the voltage applied and consequently a drop in the resistance as well.

$$T = k_1 \frac{U - K_r w}{i} - k_2 \frac{1}{i} + k_3$$

According to our model, this drop in speed ended up resulting in a sudden drop in temperature, which didn't happen in the real engine. To solve this, we added speed as a separate term in our model, so that it is sensitive to these variations. Speed is also treated in the same way as the inverse of current, i.e. it is taken as its modulus and passed through a low-pass filter.

So our final model is of the form

$$T = k_1 \frac{U - K_r w}{i} - k_2 \frac{1}{i} + k_3 w + k_4$$

Results

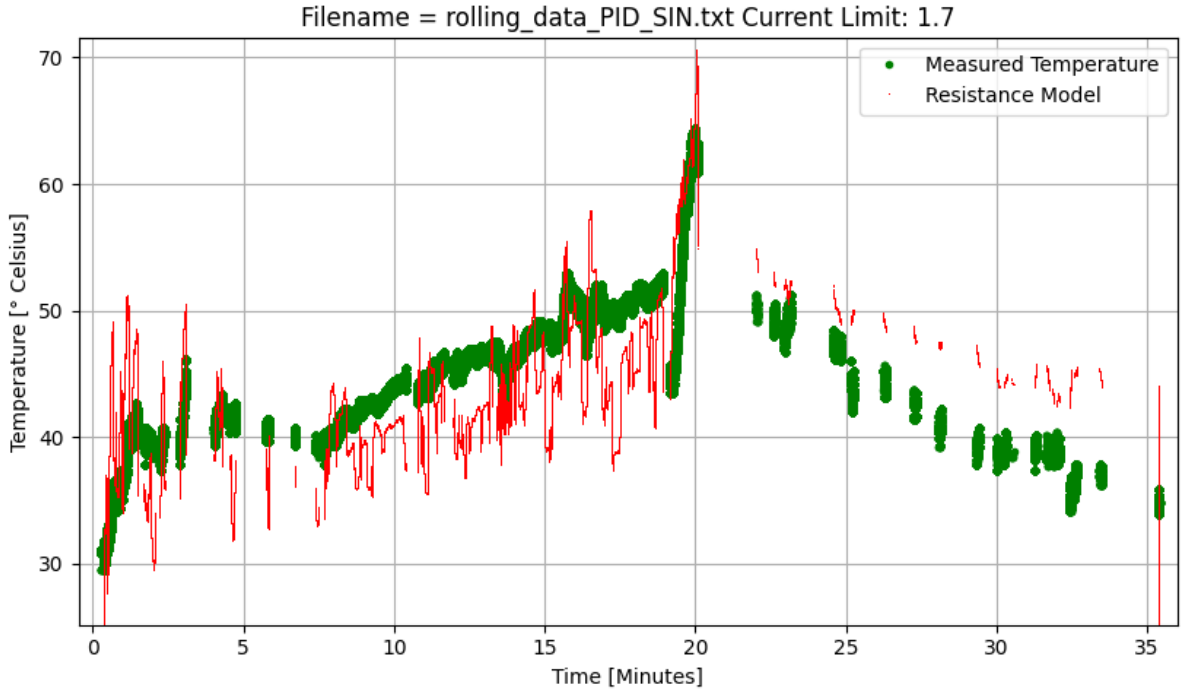

Fig. 15: Estimation of the resistive model (1)

Above, we see the temperature measured by the sensor in green, and the temperature estimated by the resistive model in red. When the current is less than 1.7 Amperes, the resistive model is unable to correctly estimate the temperature because of the non-linear relationship between voltage and current. Thanks to this, we have these empty sections in the graph. We can see that the resistive model follows the temperature, but it has a very high variation, reaching errors of more than 10 degrees.

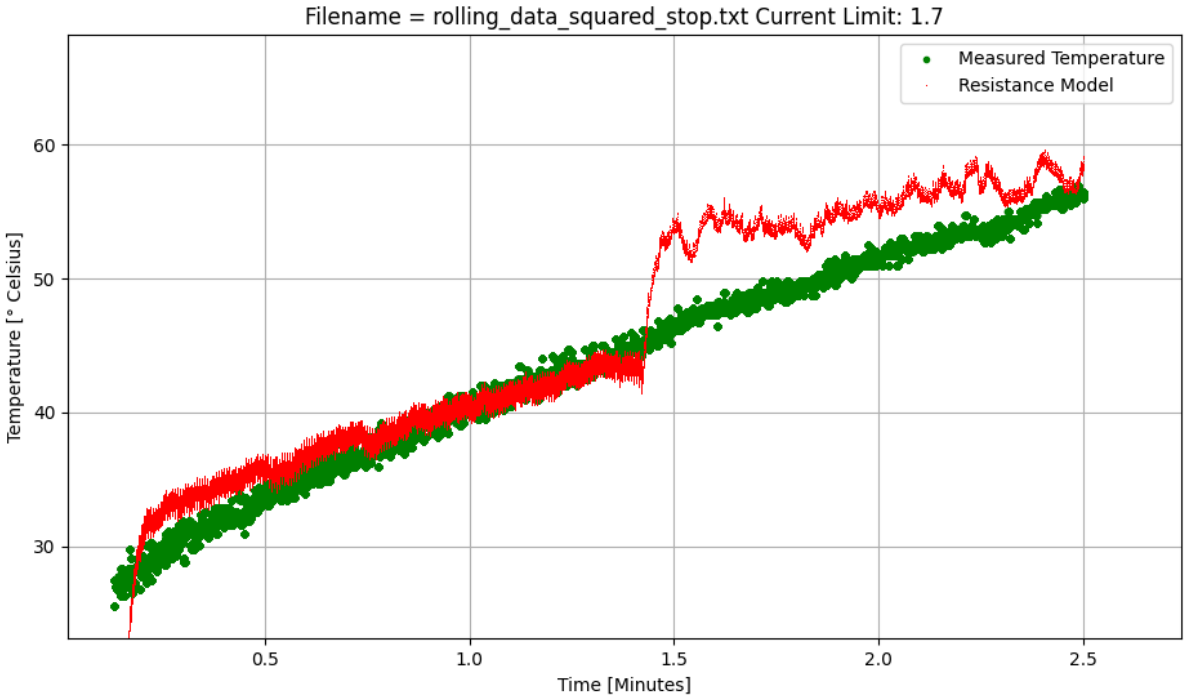

Fig. 16: Estimation of the resistive model (2)

In the test graph above, we used high currents to turn the motor, [5, -5] Amps. And we forced the motor to stop close to 90 seconds which caused a jump in the temperature estimate. This jump shows that our model is still disturbed by the change in speed, but manages to maintain an error of less than 10 degrees if the current is high.

Kalman filter

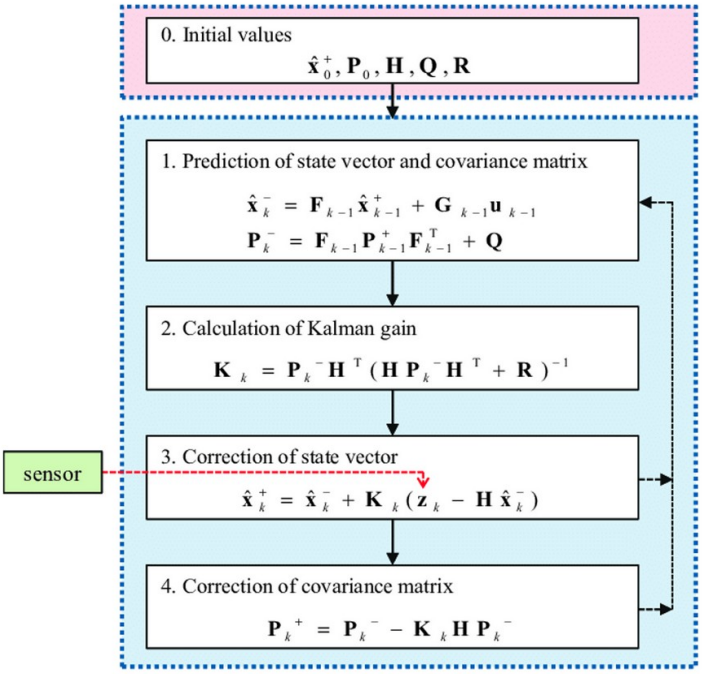

Fig. 17: Diagram of how the Kalman Filter works

The Kalman filter operates in two distinct steps: the prediction of the state and the subsequent updating of the state based on an observed measurement. In this document, my main focus will be to explain the application of the Kalman filter. However, if you would like a more in-depth understanding of how it works, I recommend visiting the website available at this link.

To apply the Kalman filter, we define the following variables:

x	State variable	2x1	Output
A	State transition matrix	2x2	System Model
P	State covariance matrix	2x2	Output
z	Measurement	1x1	Input
R	Measurement	1x1	Input

	covariance		

We define our state as the temperature and its derivative x = [T , T']. The dynamics of the state are:

$$T_{k+1} = T_k + dt * T'_k$$
$$T'_k = K_1 * i^2 + K_2 * (T_k - T_{amb})$$

The first equation is a discrete integration of the temperature, while the second is taken from the thermal model.

$$x_{k+1} = Ax$$
$$A = [\,[1,\ dt],\ [0,\ 1]]$$

The first line of A is derived from the dynamics equations of x. On the other hand, the temperature derivative does not depend solely on x. Therefore, just for the sake of defining A, we assume that the temperature derivative is constant, and we define the second line of A as [0, 1].

The state covariance matrix was defined as $P = [\,[10, 0],\ [0,\ .01]]$ The initial estimate is that the engine is at room temperature and this is not always the case. To indicate this lack of confidence in the initial value, it is necessary to put a high value on the variance of the temperature, the first element of the P matrix, which is why we chose 10. The last term in the P matrix is the variance of the temperature derivative and we put 0.01, as this is the average value of the variance of the temperature derivatives observed in the tests.

As the resistive model does not work at low currents, below 1.7 A, z is the temperature estimation of the thermal model. When the current is above this value, z is defined by the resistive model.

To define R, we calculated the temperature error of the resistive model in relation to the measured temperature and took the variance of this error. This left R with a value of 20.

Results

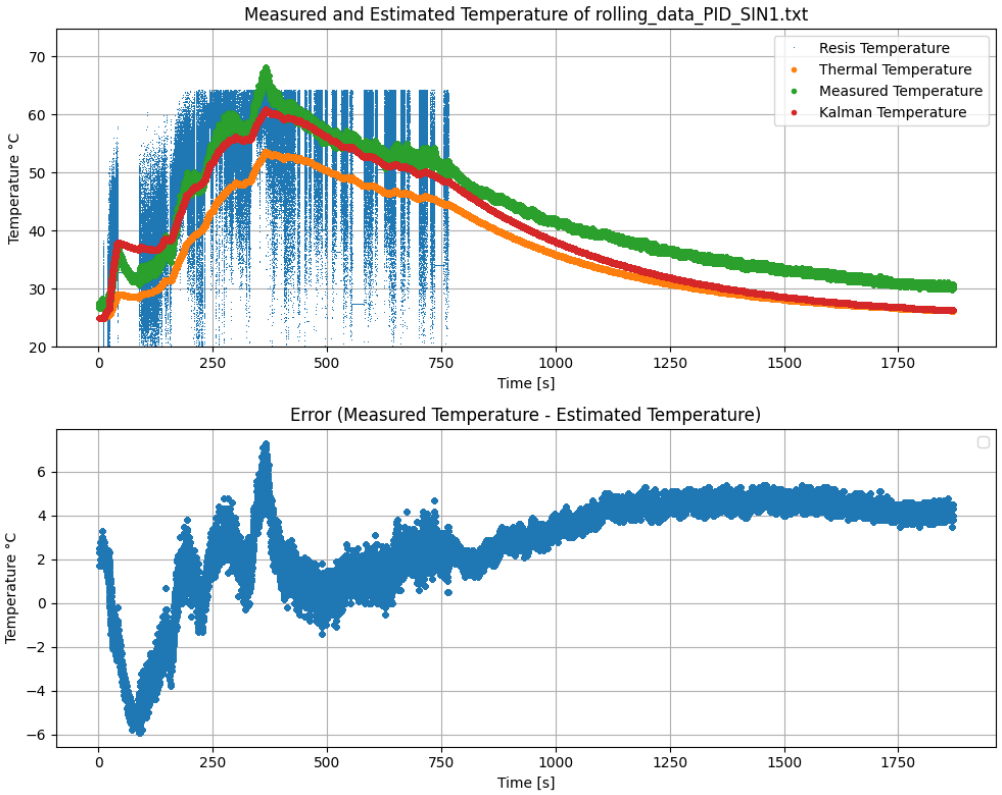

Fig. 18: Estimation of the Kalman Filter and its error (1)

The upper graph shows the measured temperature from the thermal model and the resistive model, and the temperature calculated by the Kalman filter. The lower graph shows the error of the temperature estimated by the Kalman filter and the measured temperature. We can see that the filter performs satisfactorily, showing an error of less than 7 degrees throughout the experiment and being able to follow the temperature both when the engine is heating up and when it is cooling down.

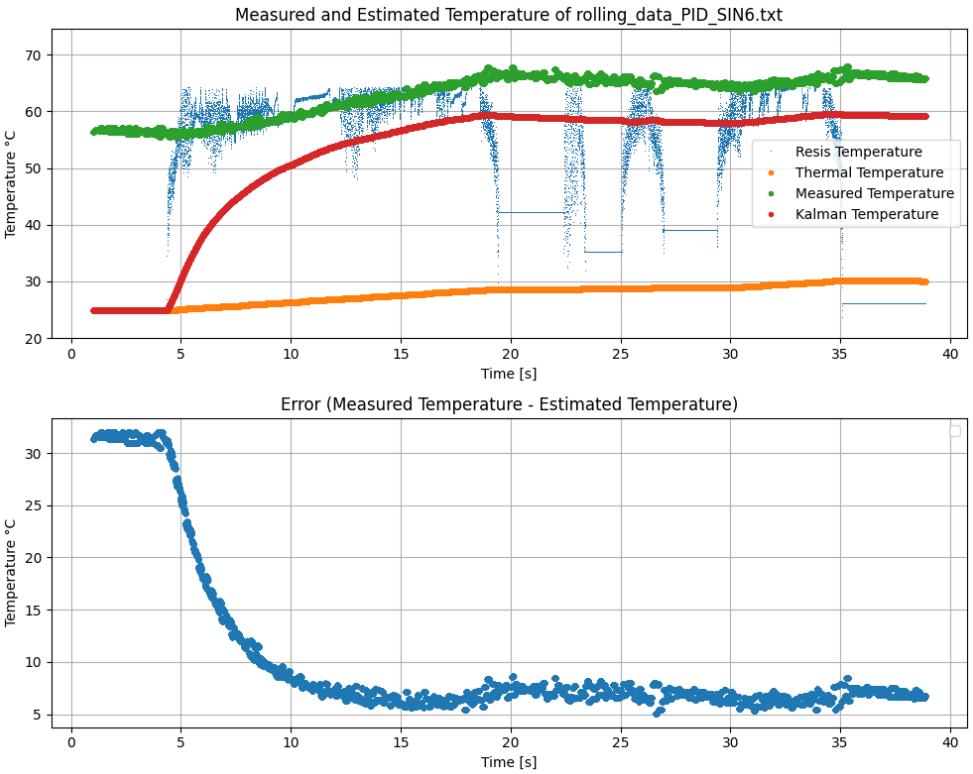

Fig. 19: Estimation of the Kalman Filter and its error (1)

In this other experiment, we show the behavior of the system when the estimation is started with the engine hot. The thermal model, orange curve, only works with the temperature derivative and a fixed initial estimate. So it doesn't notice that the engine temperature is too high. On the other hand, when we supply the measurement from the resistance model to the Kalman filter, it is able to perceive the temperature divergence and achieve an error of less than 10 degrees Celsius in around 5 seconds.

Conclusion

In this work, we developed two models for estimating temperature that had their limitations, but we managed to combine the two with the Kalman filter to obtain an accurate and fast estimate. Finally, since this method uses only two common sensors in motors, a current sensor and an encoder, it can be applied at no additional cost.

Introduction to cogging

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

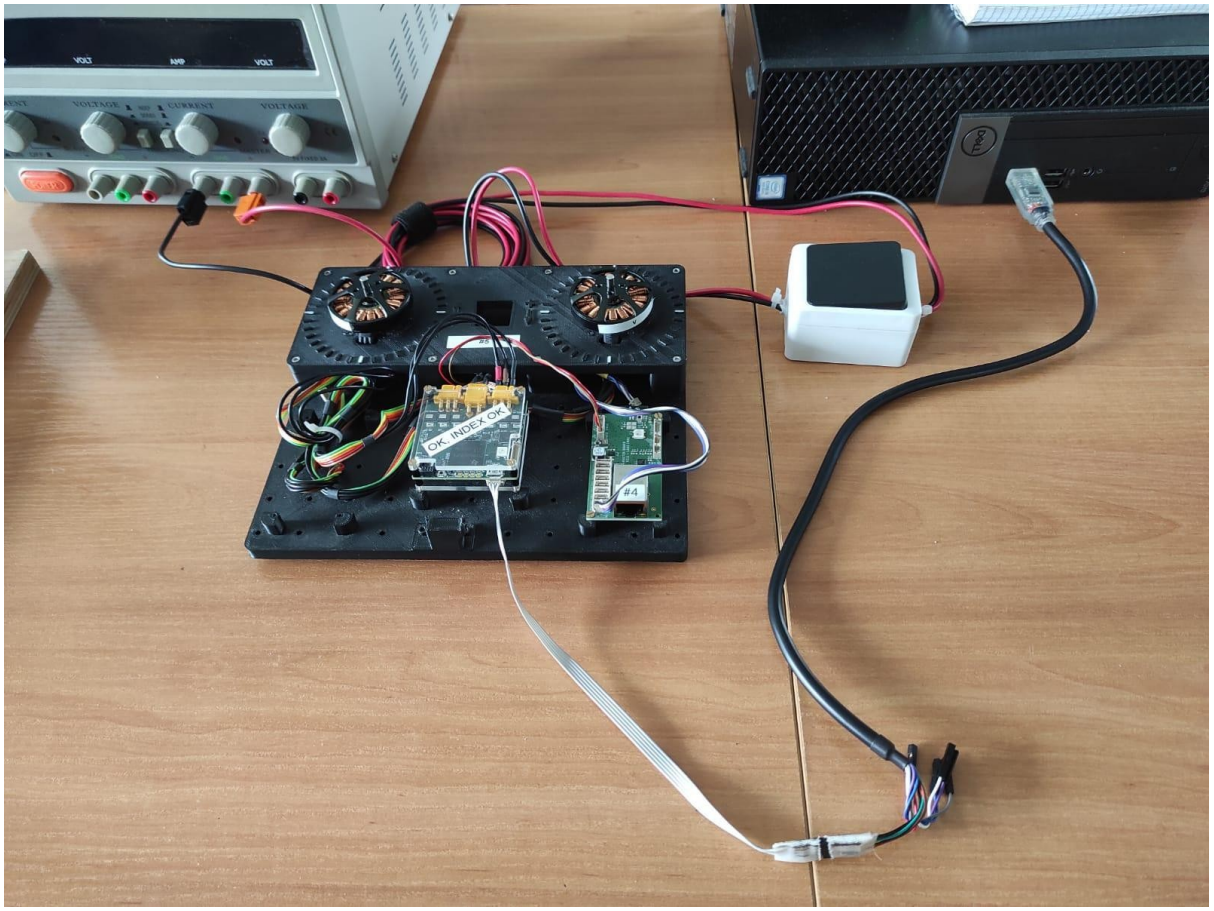

Fig. 20: Setup with motor, Omodri and connectors C232HM

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).

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.

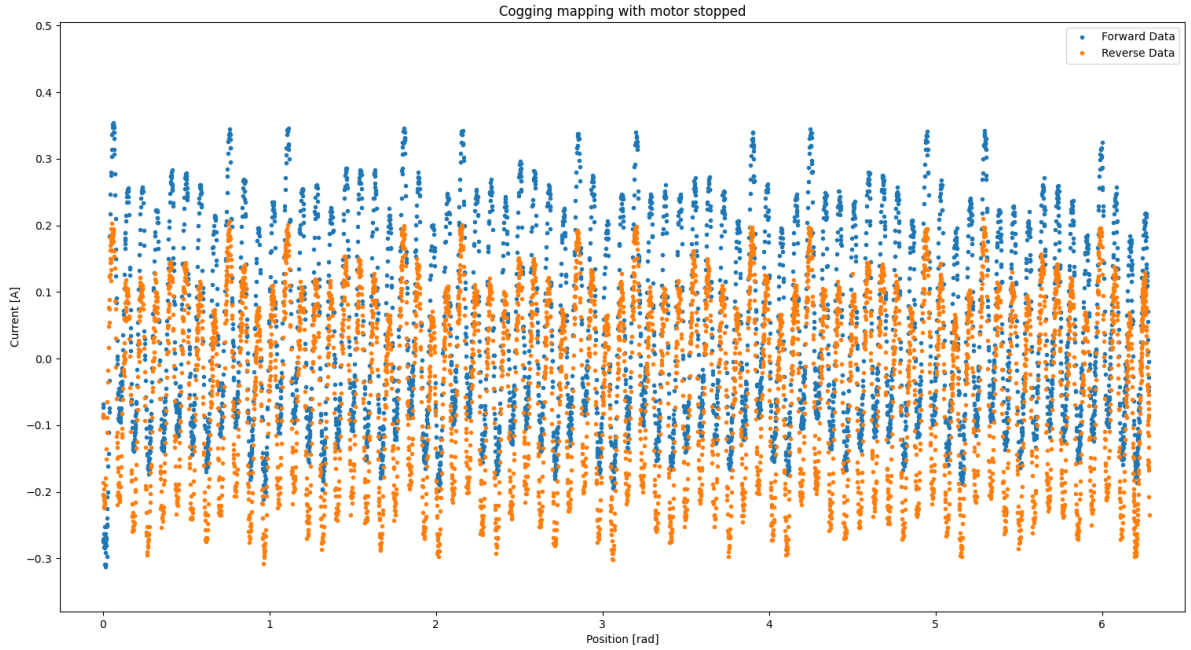

Figure 21: Position and currents of cogging torque mapping at standstill

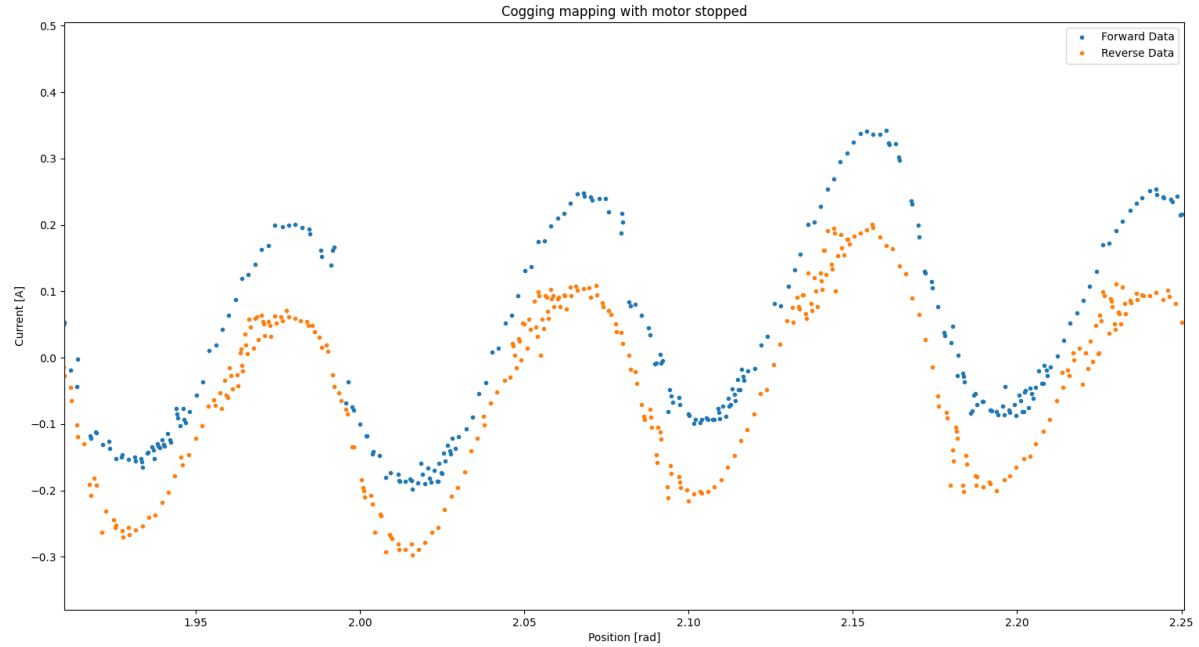

Figure 22: Figure 2 Magnified

We can see that the currents acquired during the first "forward" turn show a positive deviation from the currents of the second turn. 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 position error 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 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.

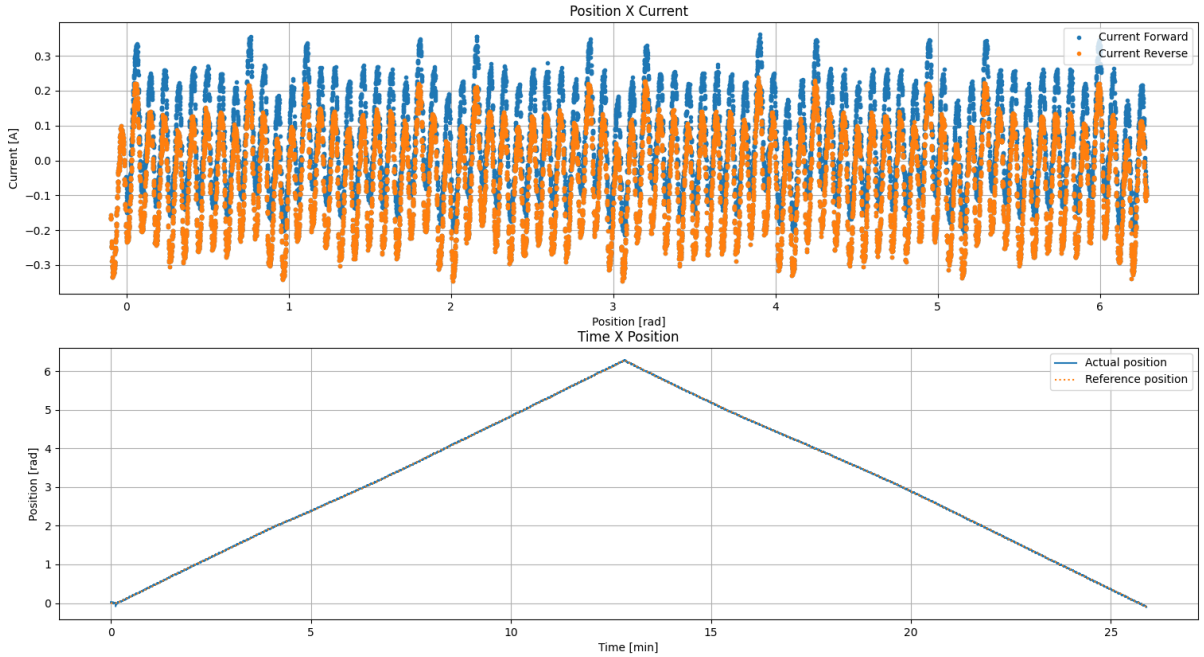

Figure 23: Motor data during the experiment. Note that the final time was 26 minutes.

Generating the anti-cogging current vector

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,

   $I_{cogging} = (I_{fi} + I_{ri}) / 2$

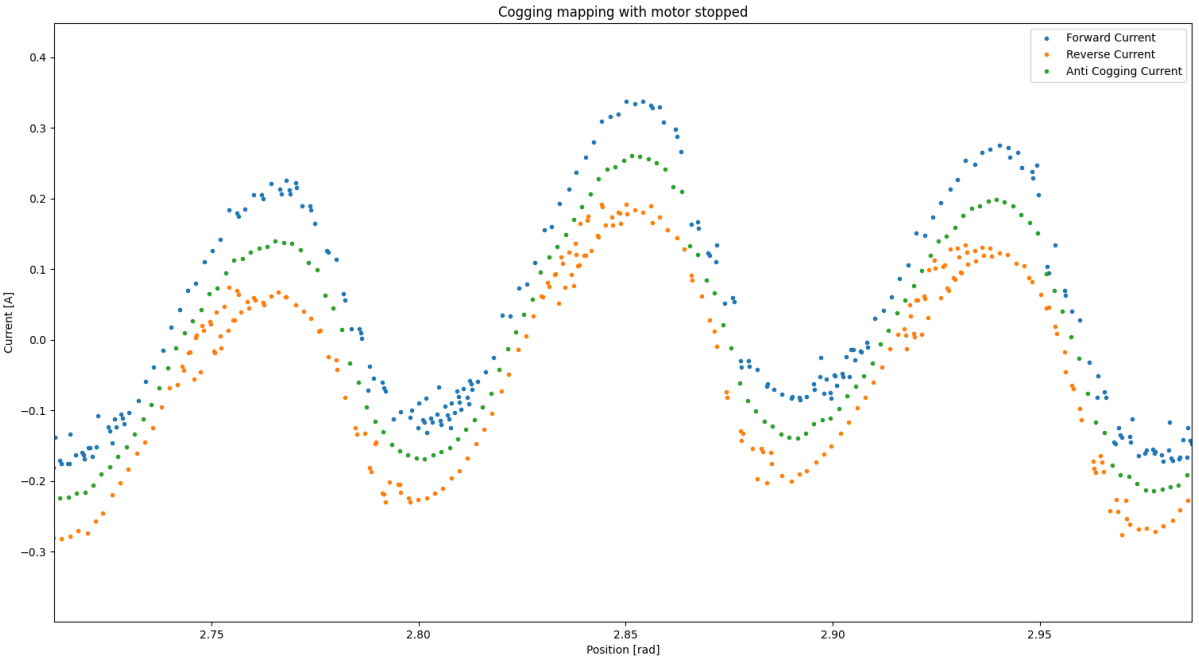

Figure 24: Forward, reverse and anticogging currents

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.

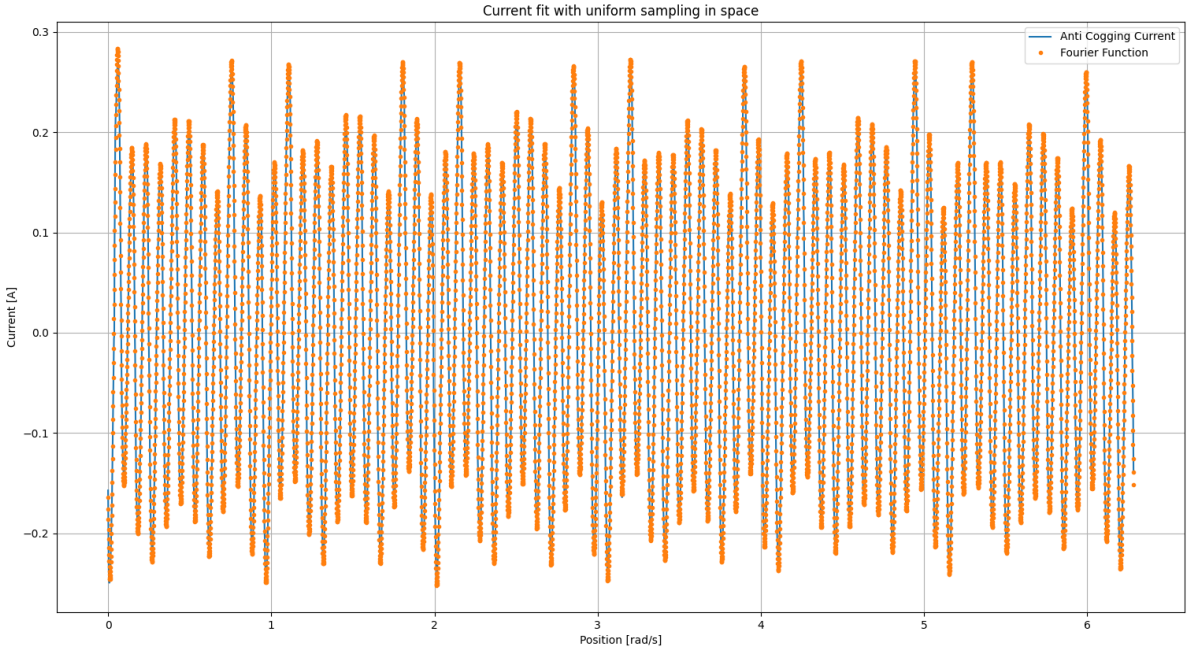

Figure 25: Fourier fit in the anticoggin current

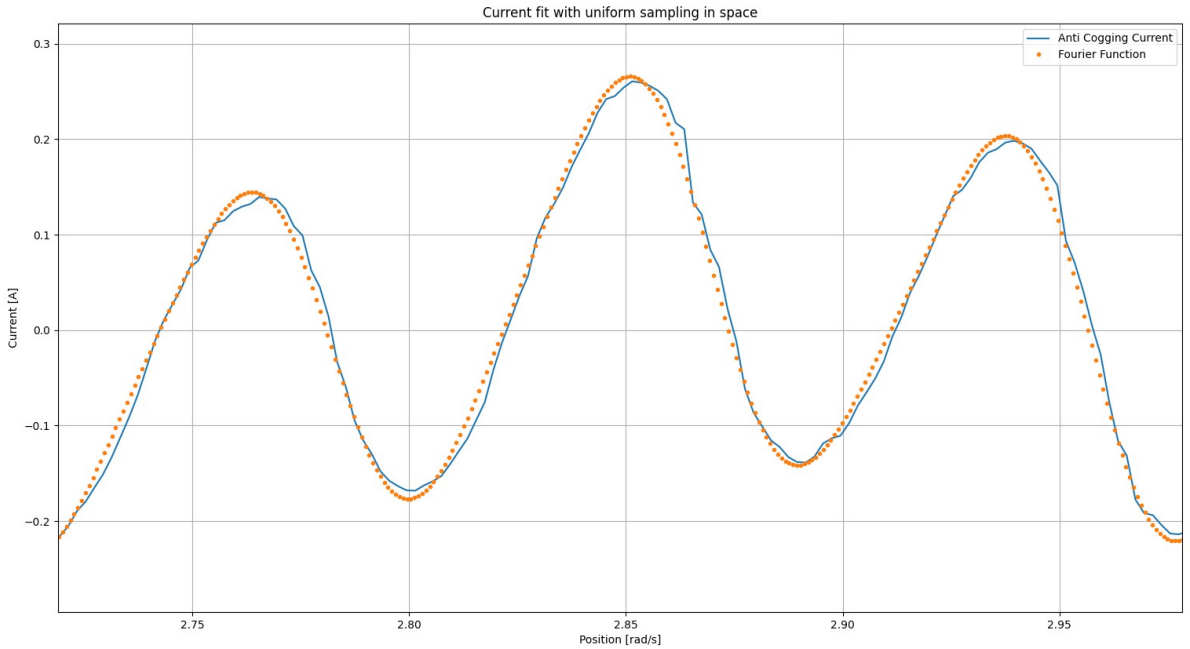

Figure 26: Figure 6 Magnified

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 milliampere range, we won't suffer any loss in resolution when using this integer representation. It's worth noting that the smallest packet size supported by our microcontroller, the F28388DZWTS, is precisely 16 bits.

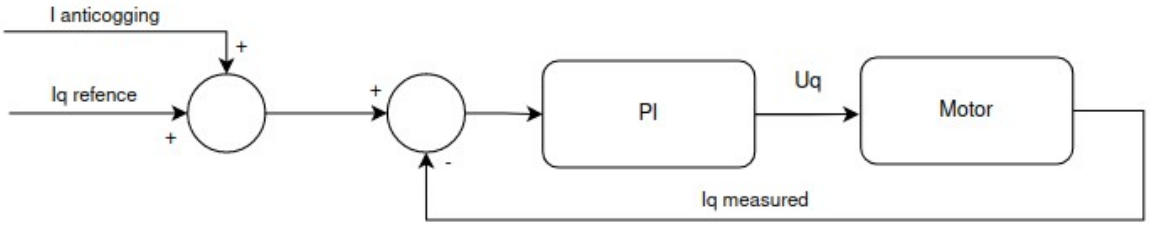

Figure 27: Diagram of PI control of Uq with anticogging current

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

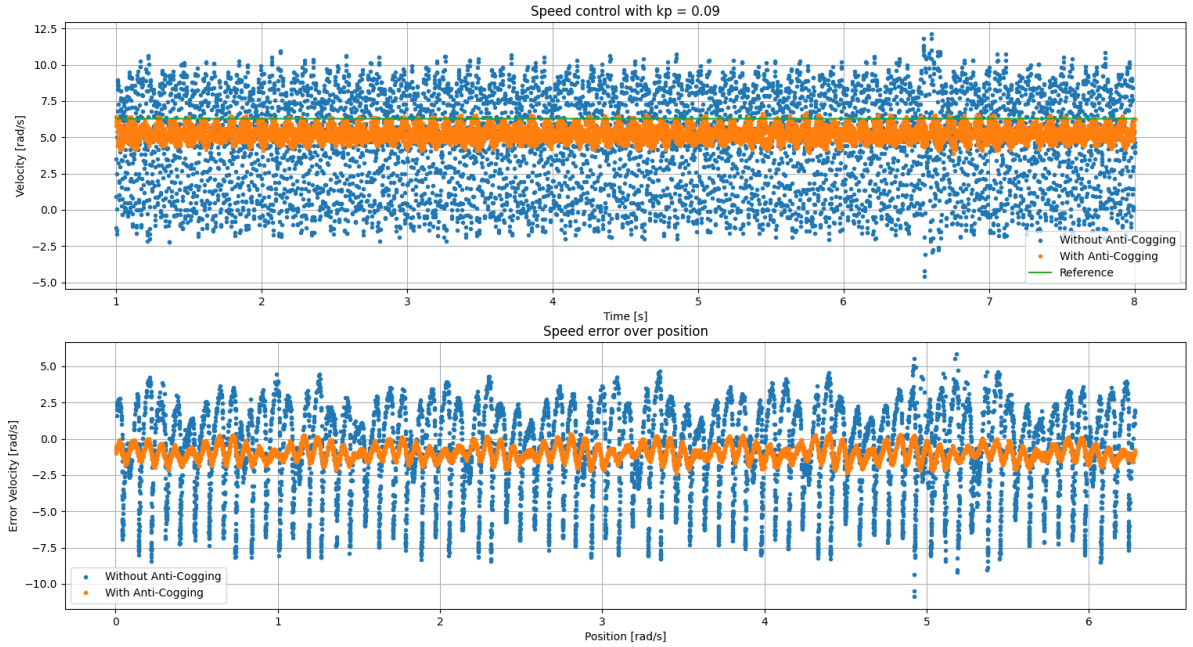

Figure 28: Comparison of the effects of anti-cogging on speed control

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, 2π in 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

Estimating the Resistance and Inductance of an Induction Motor

Introduction

The Omodri motor controller code incorporates a proportional-integral (PI) controller to modulate the voltage applied to the motor, based on a desired reference current. The constants of this controller, represented by Kp and Ki, are directly proportional to the motor's resistance (R) and inductance (L). Initially, the resistance and inductance values are defined statically in the program. However, the main aim of this task is to automate the calculation of the R and L parameters, providing a more efficient approach in motor replacement situations.

Setup

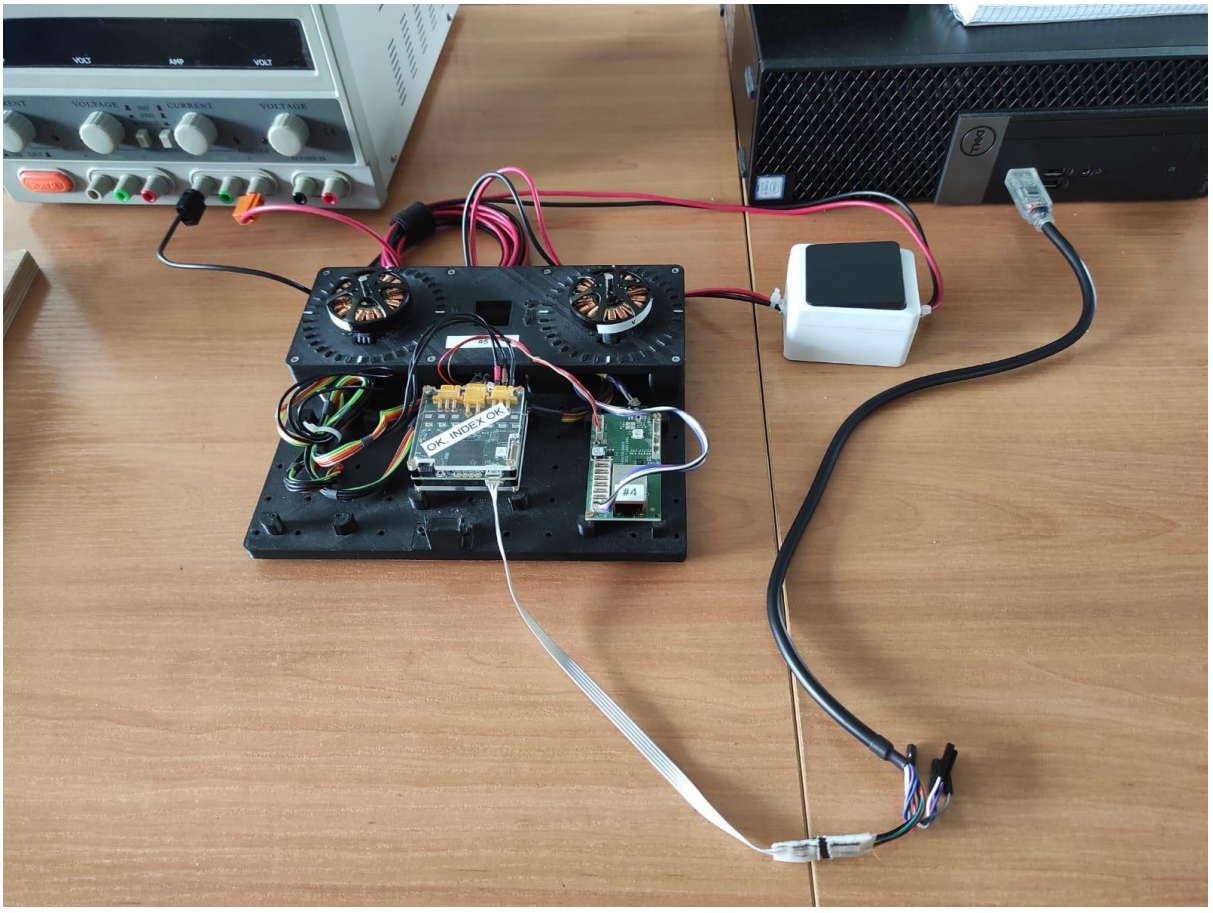

Fig. 29: Setup with motor, Omodri and connectors C232HM

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). The codes used can be found at the following link.

Resistance Estimation

Equivalent circuit

In order to calculate the resistance and inductance, we will apply a constant duty cycle to phases b and c, working in a similar way to a ground, while varying the voltage of phase a. We can visualize our motor as equivalent to the following circuit.

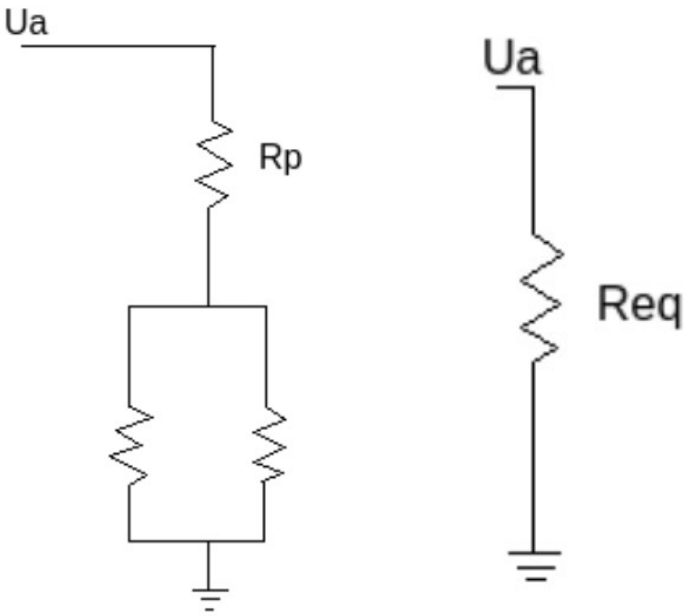

Fig. 30: Equivalent circuit and resistance when setting dtc_b = dtc_c = 0.5

Rp is defined as the equivalent resistance of the phase and Req as the equivalent resistance of the motor, $R_p * \frac{3}{2} = R_{eq}$. To calculate the resistance we will take the voltage and current at two different points (duty cycle, current) and then determine the ratio between the voltage variation and the current variation.

$$R = \frac{\Delta V}{\Delta I}$$

Non-linearities in the controller

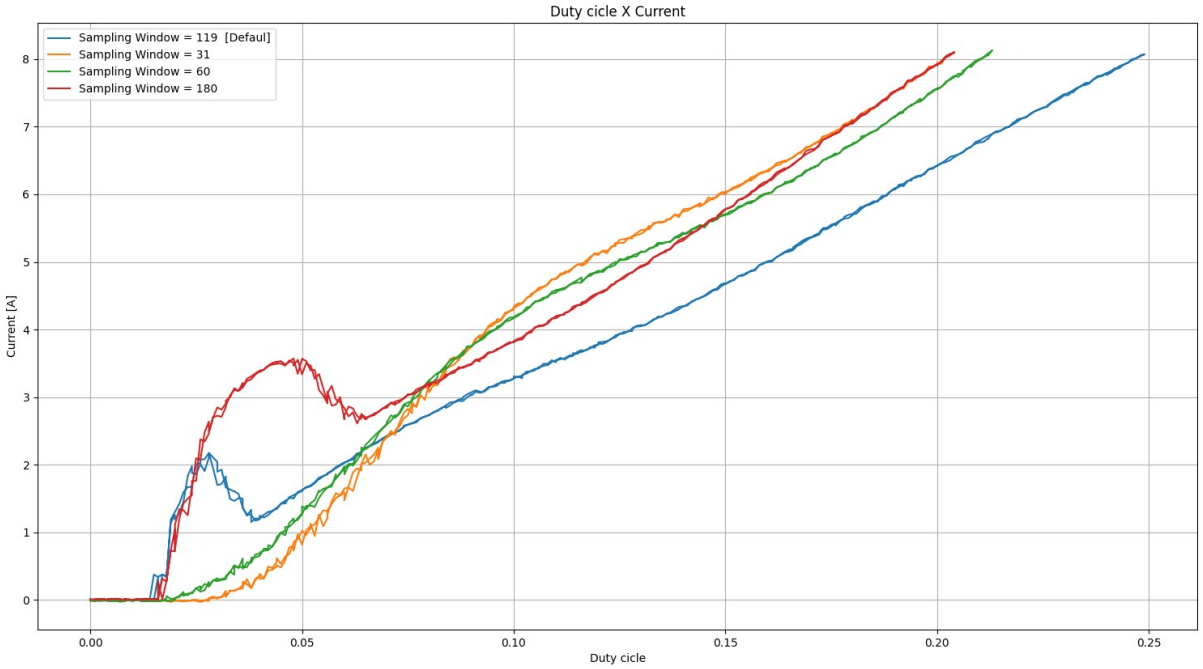

Fig. 31: Duty cycle and current ratio for different SAMPLING_WNIDOW values

The graph above illustrates the non-linearity present in our motor driver at low currents due to SAMPLING_WINDOW. This parameter indicates the number of system clocks the ADC capacitor will be connected to the signal to be converted. The exact formula for the acquisition time is (ADC_SAMPLING_WINDOW + 1) * PERIOD_SYSTEM, and our system works at 200 MHz, so a period of 5 ns.

The omodri configuration has a sampling window of 119, i.e. 600 ns of acquisition time. For these reasons, and in order to avoid non-linearities at low currents, we set the omodri's duty cycle operating range between 0.1 and 0.9. With regard to calculating the resistance, we initially set our 3 phases to 0.5, and then increased the duty cycle of phase a only.

Algorithm for calculating resistance

The resistance calculation algorithm consists of the following steps:

1. **Starting the Phases:** Start the motor's three phases with a duty cycle of 0.5.
2. **Gradual Duty Cycle Increase:** Gradually increase the duty cycle until the current exceeds a defined value, called $I_{limInferior}$. The duty cycle at which this occurs is called $dtc_{inferior}$.
3. **Duty Cycle Maintenance:** Keep the duty cycle $dtc_{inferior}$ applied to the motor and set $I_{inferior}$ to the average of 20 measured currents.
4. **Return to Duty Cycle Increase:** Again, increase the duty cycle from $dtc_{inferior}$ until the current exceeds another value,$I_{limSuperior}$, allowing $dtc_{superior}$ to be set.

5. **Maintaining the Upper Duty Cycle:** Keep the duty cycle $dtc_{superior}$ applied to the motor and set $I_{superior}$ to the average of 20 measured currents.
6. **Resistance calculation:** Calculate the resistance as $R = \frac{(dtc_{superior} - dtc_{inferior}) * V_{Supply}}{I_{superior} - I_{inferior}}$

Results

The calculated line-to-line resistance was 480 mΩ. Assuming that the resistance reported on the [MN4004-KV300 motor website] is also the line-to-line resistance, we have a reported resistance of 0.452 mΩ, thus concluding a measurement with an error of 0.03mΩ, i.e. less than 7%.

Inductance estimation

Applied signal

The impedance of an inductance is expressed by $Z = j\omega L$ indicating that it varies proportionally to the frequency of the signal applied to the circuit. According to Ohm's Law (U = Z * I), where U is the voltage amplitude and I is the current, we can infer that for a constant voltage amplitude and increasing frequencies, the impedance will increase, resulting in a corresponding decrease in the current amplitude.

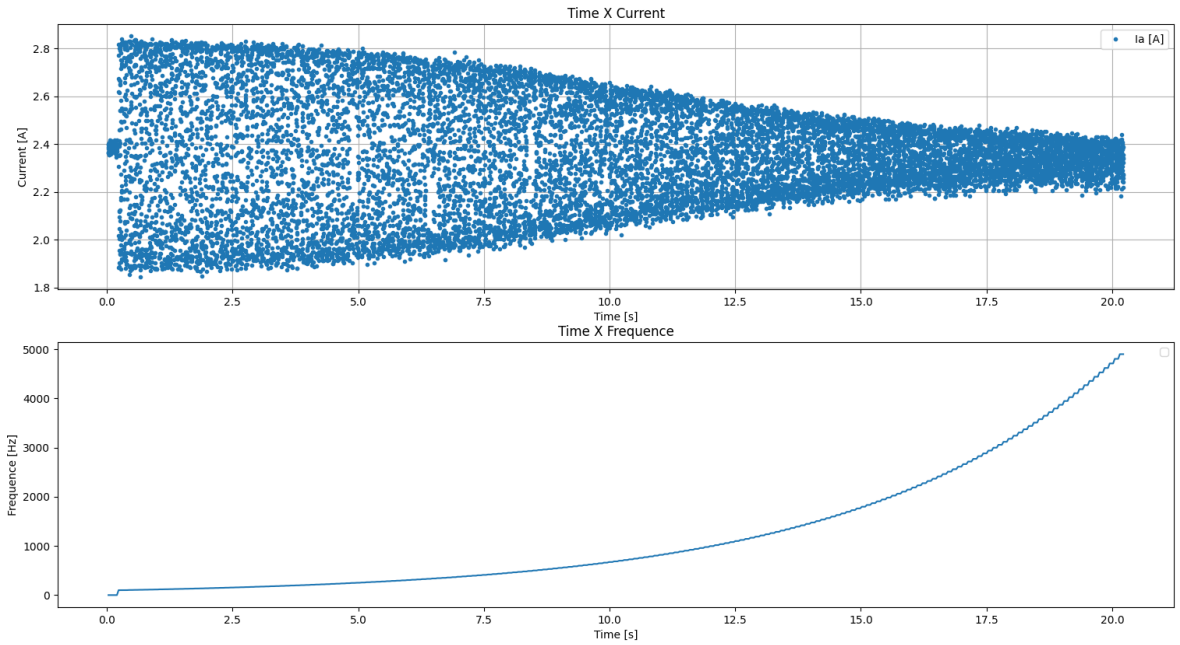

Fig. 32: Current amplitude and frequency during inductance calculation

For the data in the image above, a sinusoidal voltage was applied with constant amplitude but increasing frequencies. In this way, the upper graph shows the mentioned behavior of increasing impedance and consequent reduction in current amplitude.

Calculating the new impedance

We have seen that applying a frequency ω can cause a gain K in the initial current, I_0. To calculate the new impedance of the circuit with the applied frequency, we can follow the equations:

$$I_1 = I_0 * K$$

$$Z_0 = \frac{U_0}{I_0}$$

$$Z_1 = \frac{U_0}{I_1}$$

$$Z_1 = \frac{U_0}{I_0 * K}$$

$$Z_1 = Z_0 * \frac{1}{K}$$

We can see that with a gain of K in the current, we can calculate the new impedance of the circuit by multiplying the original impedance by 1/K.

Calculating inductance

It is possible to calculate the inductance from the alternating current impedance (Zac = Z with a frequency ω applied) from the following equations:

$$Z_{ac}^{\ 2} = resistance^2 + reactance^2$$

$$reactance = \sqrt{Z_{ac}^{\ 2} - resistance^2}$$

$$L = reactance \ / \ (f_{ressonance} * 2 * \pi)$$

The algorithm for calculating impedance is as follows:

1. **Applying a Sinusoidal Voltage:** Initially, we apply a sinusoidal **voltage** with a low frequency and define I_0 as the generated amplitude.
2. **Progressive Frequency Increase:** We progressively increase the frequency until the gain in current, represented by $K = \frac{I\omega}{I_0}$, reaches or exceeds the value of $\frac{1}{\sqrt{2}}$.
3. **Calculating the New Impedance:** We calculate the new impedance as the inverse of the gain in current, i.e., $Z_{ac} = \frac{1}{K}$.
4. **Calculating reactance and inductance:** Finally, we calculate the reactance and inductance of the circuit using the equations mentioned above.

Results

The motor's RL parameters are used to calculate the Kp and Ki of the uOmodri current controller. Thus, observing the frequency and time response of this controller can give us an idea of the usefulness of parameter estimation.

First, we set the passband to 1kHz, and then we calculate the controller constants with the equations below. You can find demonstrations of this formula at the following link:

$$Kp = Ls * 2 * \pi * f_{-3db}$$
$$Ki = Rs * 2 * \pi * f_{-3db}$$

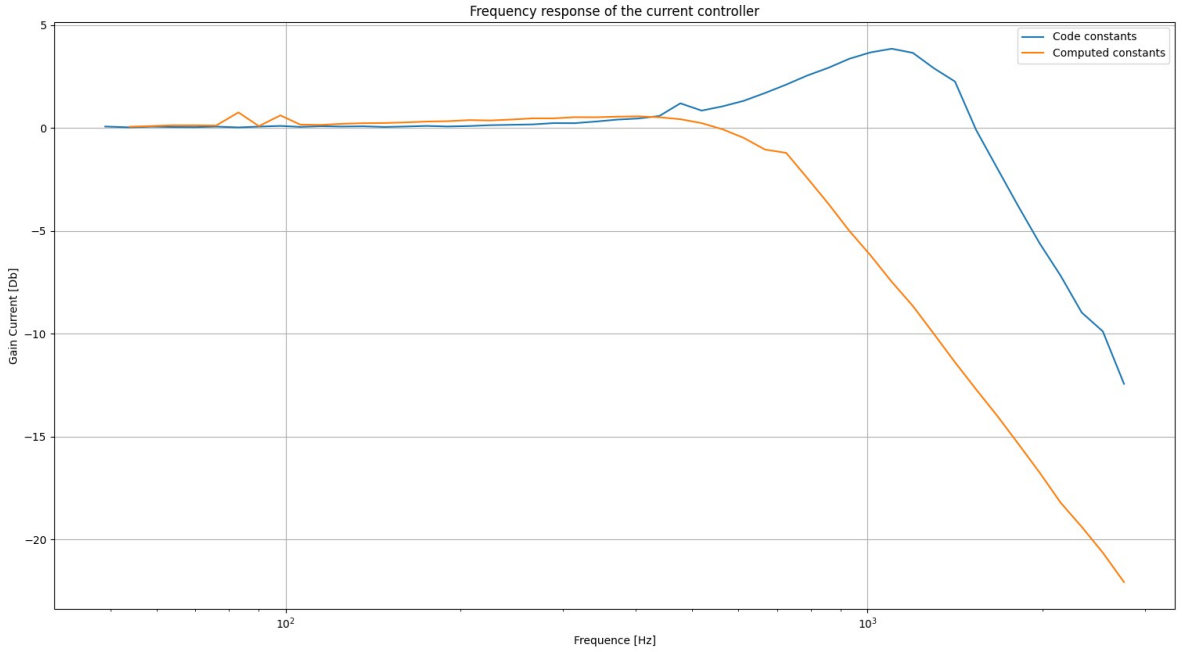

Fig. 33: Frequency response of the current controller

The graph above illustrates the frequency response of the current controller. The PI calculated from the old constants is represented by the blue curve and has a gain of -3db around 1090 Hz. Meanwhile, the controller with the new calculated constants achieves a gain of -3db at 800 Hz.

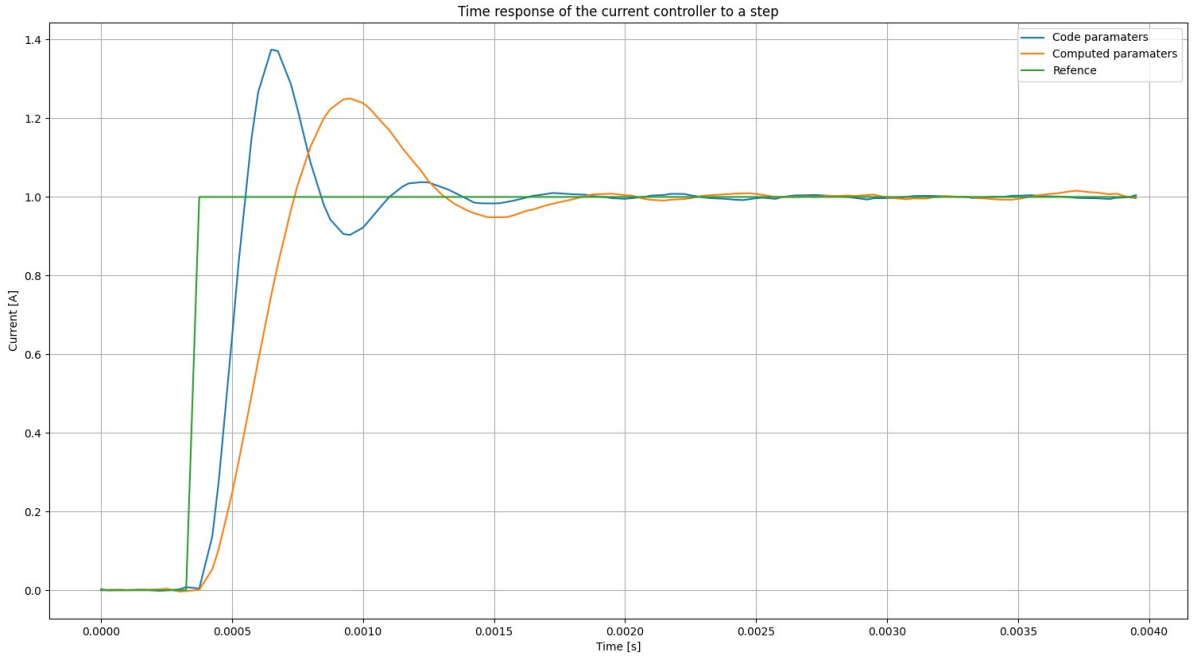

Fig. 34: Time response of the current controller

The graph above illustrates the response of the current controller over time. The current reference is represented by the green curve, the controller calculated from the old code constants by the blue line, and the controller derived from the computed constants by

the orange line. We can see that we have a smaller overshoot with the new constants, going from around 38% to 25%. However, the rise time has increased significantly compared to the old controller, being approximately 0.25 ms.

Next steps

A desirable next step is to study whether we should take into account the influence of preserving the current amplitude in the Clark transformation. To be more precise, when converting the current from the phase frame ia, ib and ic, to the rotor angle reference, id and iq, we choose to preserve the amplitude of the magnitudes, multiplying the clarke transformation by 2/3. You can find more details about this preservation at this link. So, as the current controller acts in the d and q frame, and the calculated resistance and inductance are in the a, b and c frame, we need to study the need to perform the reference transformation before calculating the current controller parameters.

Annexes

Temperature estimation code

```
/**
* @brief           Initialize all necessary variables.
* @param[inout]    *obs Pointer on the Temperature structure associated
to the motor.
*/
void TEMP_initialize(obs* ob){
   ob->temperature = T_AMBIENT;

   // RESISTANCE MODEL
   ob->resis_K1 = 636;
   ob->resis_K2 = -161;
   ob->resis_K3 = -0.224;
   ob->resis_K4 = -125;
   ob->resis_kv = 0.01821;
   ob->TEMPERATURE_CURRENT_THRESHOLD = 1.7f;
   ob->alpha_flt = 0.01;

   ob->velocity_flt     = 0;
   ob->inv_current_flt  = 1/ob->TEMPERATURE_CURRENT_THRESHOLD;
   ob->resis_flt        = 0.36;
```

```
   // THERMAL MODEL
   ob->thermal_K1 = 0.010382;
   ob->thermal_K2 = -0.001252;
   ob->thermal_temperature = 25;

   // KALMAN FILTER
   ob->X[0] = T_AMBIENT;
   ob->X[1] = 0;
}

/**
* @brief Update temperature estimation.
* @param[in] *p_foc Pointer on the FOC motor control structure.
        [inout] *obs Pointer on the Temperature structure associated
with the motor.
*/
void TEMP_update(foc_t* p_foc, obs* ob)
{

   // Initialize variables for themal and resistance models
   static bool started = false;
   if (started == false){
       TEMP_initialize(ob);
       started = true;
   }

   // Compute derivative temperature
   double current = (p_foc->id*p_foc->id) + (p_foc->iq*p_foc->iq);
   float dif_temperature = ob->temperature - T_AMBIENT;
   float dif_thermal_temperature = ob->thermal_temperature - T_AMBIENT;

   double temp_derivative = (current * ob->thermal_K1) +
(diff_temperature * ob->thermal_K2);
   double temp_derivative_thermal = (current * ob->thermal_K1) +
(dif_temperature_thermal * ob->thermal_K2);

   ob->thermal_temperature = ob->thermal_temperature +
temp_derivative_thermal*LOOP_PERIOD;


   // Compute temperature measured by Thermal or Resistance Model
   double temperature_measured;
```

```
   if (fabs(p_foc->iq) < ob->TEMPERATURE_CURRENT_THRESHOLD)
       {temperature_measured = ob->temperature +
temp_derivative*LOOP_PERIOD;}
   else
   {
       // Get module of current, velocity and resistance
       float iq, velocity, resis;
       iq = fabs(p_foc->iq);
       velocity = fabs(p_foc->motor_enc.speed.speedMech[0]);
       resis = fabs((p_foc->uq -
ob->resis_kv*p_foc->motor_enc.speed.speedMech[0])/p_foc->iq);

       // Compute low pass filter
       ob->velocity_flt = ob->alpha_flt * velocity + (1 -
ob->alpha_flt) * ob->velocity_flt;
       ob->resis_flt = ob->alpha_flt * resis + (1 - ob->alpha_flt) *
ob->resis_flt;
       ob->inv_current_flt = ob->alpha_flt / iq + (1 - ob->alpha_flt) *
ob->inv_current_flt;

       // Compute temperature by Resistance Model
       float tr = ob->resis_K1*ob->resis_flt +
ob->resis_K2*ob->inv_current_flt + ob->resis_K3*ob->velocity_flt +
ob->resis_K4;
       if (tr > 200){tr = 200;}
       if (tr < 0){tr = 0;}

       ob->resis_temperature = tr;
       temperature_measured = ob->resis_temperature;
   }

   // Compute Kalman Filter
   TEMP_kalman_filter(temperature_measured, temp_derivative, ob->X);
   ob->température = ob->X[0];


   // Indicates if the maximum temperature has been reached
   if(ob->temperature > TEMP_MAX)
       {ob->otw flag = 1;}
   else
       {ob->otw_flag = 0;}
```

```
    return;

}
```

Kalman Filter Code

```
/**
* @brief Apply Kalman filter on temperature estimation
* @param[in] temperature is the measured estimated by thermal or
resistance model,
                temperature_derivate is T' estimated by thermal model,
                x[2] is an array composed by the temperature estimated
by the kalman filter and its derivative x=[T, T']
*/
inline void TEMP_kalman_filter(double temperature, double
temperature_derivate, double x[2]) {
    // Kalman filter parameters
    static float R = 20;
    static double P[2][2] = {{10, 0}, {0, 0.01}};
    static float A[2][2] = {{1, LOOP_PERIOD}, {0, 1}};
    static float AT[2][2] = {{1, 0}, {LOOP_PERIOD, 1}};

    // Predict State Forward
    double x_p[2] = {x[0] + x[1] * LOOP_PERIOD, temperature_derivate};

    // Predict Covariance Forward
    double P_p_aux[2][2] = {{0, 0}, {0, 0}};

    // P_p_aux = A P
    int i, j, k;
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 2; j++) {
            for (k = 0; k < 2; k++) {
                P_p_aux[i][j] += A[i][k] * P[k][j];
            }
        }
    }

    double P_p[2][2] = {{0, 0}, {0, 0}};
    // P_p = A P AT
    for (i = 0; i < 2; i++) {
```

```
        for (j = 0; j < 2; j++) {
            for (k = 0; k < 2; k++) {
                P_p[i][j] += P_p_aux[i][k] * AT[k][j];
            }
        }
    }

    // Compute Kalman Gain
    double S = P_p[0][0] + R;
    double K = P_p[0][0] * (1 / S);

    // Estimate State
    double residual = temperature - x_p[0];
    x[0] = x_p[0] + K * residual;
    x[1] = x_p[1];

    // Estimate Covariance
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 2; j++) {
            P[i][j] = P_p[i][j] - K * P_p[i][j];
        }
    }

    return;
}
```