PID-Controlled Propeller Arm

RESEARCH & DEVELOPMENT

EIURO85

Authors:

Durand Hugo (22011675)
Denis-Martin Hugo (22210717)

June 22, 2024

Contents

Introduction

Our project involves controlling a twin-propeller arm using a cascade control loop system. In response to an input angle given by the potentiometer, the arm converges towards the angle by adapting the speed of the motors accordingly. The main objective is to achieve precise and stable control of the propeller arm's position through continuous feedback and adjustment. This project will be used for practical sessions with Mr DUCARD for future robotics promotions, providing students with hands-on experience in implementing and fine-tuning control systems.

1 Building on previous work

We are picking up the project where our predecessors left off, building upon the solid foundation they established:

- The hardware part of the robotic arm with already mounted propellers, ESC fixed to the wooden stick pivoting on a stable wooden base.
- An incorporated level shifter sn74ls07n to convert the 3.3V Pulse Width Modulation (PWM) signal from the microcontroller to a 5V signal suitable for the ESCs. This voltage re-mapping ensures reliable control of the motor speeds.
- A cascade control simulation on xcos and a draft code to control the stm32 to perform the embbeded control the arm.

2 STM32 register programming

The code provided by the previous student groups didn't work for us and was a bit confusing. The whole implementation was built in the main file and in a bizarrely unstructured way, so we decided to rebuild the software from scratch and avoiding to use dependencies of any sort which can be altered over time.

We decided to build our program without using the HAL, TM library or the predefined CubeMX functions. The aim was to understand step by step how the microcontroller registers work. There are some very good tutorials on Youtube, on the Controller Tech website [2] and some great code examples on Github [1], based solely on the STM32's registers and memory addresses, to help us in this process.

2.1 Clock Configuration

In order to achieve the fastest possible execution of our code, we chose to configure the clock at the highest speed permitted by the manufacturer.

```
The system Clock is configured as follow :
  *            System Clock source            = PLL (HSE)
  *            SYSCLK(Hz)                     = 100 000 000
  *            HCLK(Hz)                       = 100 000 000
  *            AHB Prescaler                  = 1
  *            APB1 Prescaler                 = 2
  *            APB2 Prescaler                 = 1
  *            HSE Frequency(Hz)              = 8 000 000
  *            PLL_M                          = 4
  *            PLL_N                          = 100
  *            PLL_P                          = 2
```

Figure 1: System Clock Configuration

We have chosen to activate the HSE (High-Stability External) oscillator because it offers better frequency accuracy and stability compared to internal oscillators (HSI), but consume more power due to the external crystal and associated circuitry. But as our application is always plugged and require quite precise timing and synchronization for our control system loop to control the twin propeller arm, it was the best choice.

Configuring the PLL (Phase-Locked Loop) of the MCU allows us to reach a system clock frequency running at 100MHz.

2.2 4 LED TOGGLING

Used as an indicator that the board has been reset and that the MCU has restarted, the 4 LEDs in the development kit light up and go out one after the other.

Configuring them also taught us to confront an initialisation problem. We had developed the two PWMs in parallel to control the motors on the same LED pins and we realised when we merged the two codes that they gave us the following result:

```
GPIOD->MODER |= (1<<26); // For GPIO OUTPUT
GPIOD->MODER |= (2<<26); // AF for PWM modulation
// resulting to GPIOD->MODER | = (3<<26); Analog mode
```

2.3 TIMER DELAY

In order to adjust the timer period to correspond with a time interval of microseconds or milliseconds, it is necessary to know precisely the system clock configuration while configuring the timer for delay.

As, the clock runs at 100MHz, we set the prescaler value to 99 and the arrival time to the MAX value ($0xffffff_{(16)} = 16777215_{(10)}$).

This way the timer have a frequency of 1MHz and therefore increments its counter every microsecond (1 µs):

$$\begin{aligned} Overflow_{Period} &= (ARR + 1) \cdot Timer \cdot Increment \cdot Period \\ &= (16777215 + 1) \cdot 1us \\ &= 16.777216 seconds \end{aligned} \tag{1}$$

This configuration is useful when we need a timer that can measure long durations without resetting, giving us a wide range of delay or timing measurements in microseconds up to 16.777216 seconds.

2.4 USART For debug and QT Communication

For reasons of speed and efficiency, the two uarts are configured in DMA mode with an interrupt so that the display of data on Putty (DEBUG) and the sending of data to QT (real-time display interface) are done without disturbing the micro-controller in the execution of its tasks to keep the cpu time occupied with motor control and angle calculation.

2.5 Motor Initialization

As explained in the LED toggling, we disable the output mode of the pins PD12 and PD13 to configure them as AF. We configure the timer 4 to generate a PWM with a frequency of 50 Hz (writing into the PSC and ARR of TIM4 register)

```
void o_Motor_Initialization_o(void){
  MotorPWM_PIN_Init();
  MotorPWM_TIMER_Init();
  MotorPWM_PWM_Init();
  ESC_Calibration();
}
```

According to the norm explained by ControllerTech [3], we need to set the maximum pulse (2ms). The ESC will sound one beep indicating it has been calibrated for the maximum pulse. Then we send the minimum pulse (1ms) and wait for the ESC to sound the beep again. Once it does that it means the calibration is complete.

2.6 I2C3 Screen Communication

It have been configured an I2C3 Communication between the Arduino and the STM32 to established a tactile screen interface to the system, in order to modify dynamically the filter & cascade control parameters.

Figure 2: Starting screen

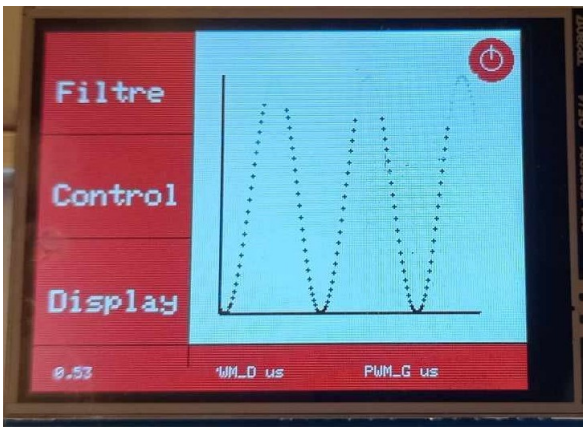

Figure 3: Home screen

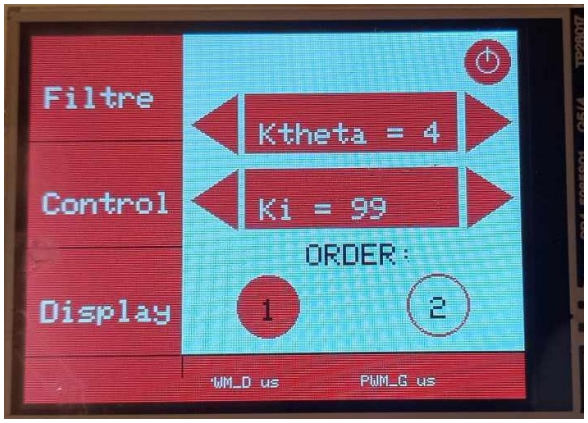

Figure 4: Filter tab

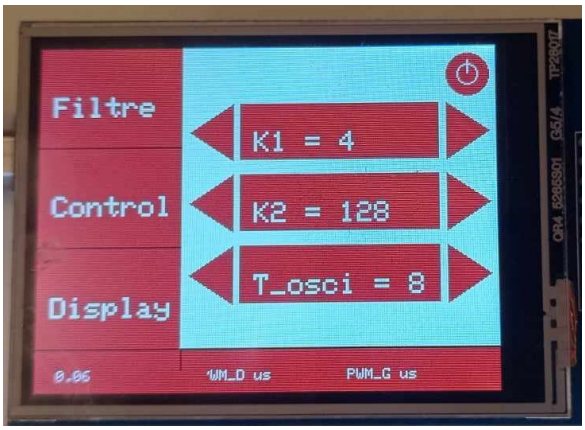

Figure 5: Control tab

The touch screen is composed of two main menus, the starting screen and the home screen. When on the starting screen, the motors are off and will get started when the button is pressed. Then in the home menu, we have a display of the measurement of the angle theta and in the different tabs on the left, we can adjust different parameters as the different gains and the order of the system. There is also a stop button to shut down the motors and come back to the starting menu.

3 Complementary Filtering for θ estimation

To balance the arm, utilizing the gyroscope and accelerometer of the STM32 microcontroller is essential. The gyroscope provides precise angular velocity measurements, allowing the system to detect any rotational movement or tilt of the arm.

The accelerometer, on the other hand, measures the linear acceleration along multiple axes. This helps in detecting any positional changes or deviations from the intended equilibrium. By combining data from both the gyroscope and accelerometer, the STM32 can accurately determine the arm's orientation and movement dynamics. Some documentation about how to use this two sensors are discribed in this two folowing websites [4] [5].

As part of this approach, we implemented an additional filter (low-pass, high-pass) to filter out the high-frequency noise from the accelerometer and the low-frequency noise from the gyrometer seen in the Sensor Fusion course with Mr DUCARD, shown in Figure 6 :

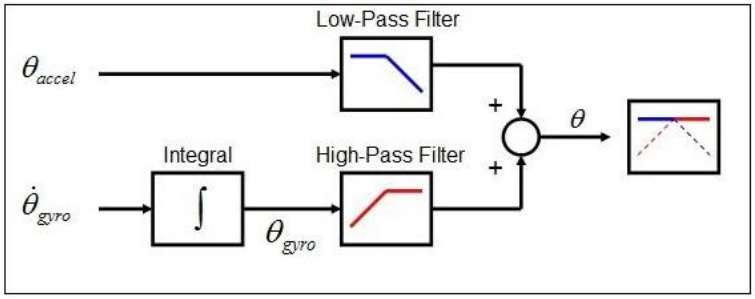

Figure 6: Complementary Filter for Theta Estimation [6]

3.1 3-axis Accelerometer LSM303DLHC

3.1.1 Exctracting theta

The goal is to estimate the angle θ in the navigation frame, extracting the accelerometer data computed in the body frame. First we retrieve the Direction Cosine Matrix (DCM) noted $\mathbf{C}_{\mathbf{n}}^{\mathbf{b}}$ by doing successive rotation yaw, pitch and roll of the system to get its orientation in the navigation frame.

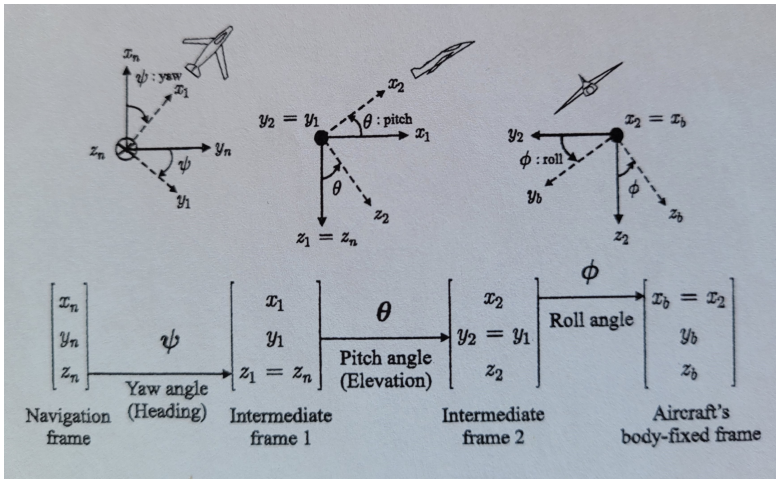

Figure 7: Successive rotation in 3D

$$(C)_n^b = (C)_2^b \cdot (C)_1^2 \cdot (C)_N^1 = \begin{pmatrix} c\theta c\psi & c\theta s\psi & -s\theta \\ s\phi s\theta c\psi - c\phi s\psi & s\phi s\theta s\psi + c\phi c\psi & s\phi c\theta \\ c\phi s\theta c\psi + s\phi s\psi & c\phi s\theta s\psi - s\phi c\psi & c\phi c\theta \end{pmatrix} \tag{2}$$

The embedded sensor LSM303DLHC on the STM32F411 Discovery board can be simplified as a "3 Dimension" sensor with the representation of 3 spring-loaded weights.

The MEMS accelerometers measure the acceleration in three dimensions and allow the user to extract the acceleration components a_x, a_y, and a_z in each axis.

We know the coordinates of $\vec{g}$:

$$\text{In the navigation frame <n>:} \quad (\vec{g})^n = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} = g \cdot \mathbf{l}_3, \quad \text{with} \quad \mathbf{l}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3}$$

$$\text{In the body frame <b>:} \quad (\vec{g})^b = g \cdot (C)_n^b \cdot \mathbf{l}_3, \quad \text{with the Direction Cosine Matrix} \quad (C)_n^b$$

In the most general case, accelerometer measurement in the body frame is expressed as:

$$\begin{aligned} \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}^b &= (C)_n^b \cdot (\ddot{x} - g \cdot \mathbf{l}_3) \quad \text{with } \ddot{x} \text{ the acceleration of the robot} \\ &= -g \cdot (C)_n^b \cdot l_3 \quad \text{assuming the propeller arm acceleration is negligible} \end{aligned} \tag{4}$$

We get as a result θ_acc = $\arcsin(\frac{a_x}{g})$

3.1.2 Noise of the accelerometer

Table 3. Sensor characteristics (continued)

Symbol	Parameter	Test conditions	Min.	Typ.(1)	Max.	Unit
LA_TCSo	Linear acceleration sensitivity change vs. temperature	FS bit set to 00		±0.01		%/°C
LA_TyOff	Linear acceleration typical Zero-g level offset accuracy(3),(4)	FS bit set to 00		±60		mg
LA_TCOff	Linear acceleration Zero-g level change vs. temperature	Max delta from 25 °C		±0.5		mg/°C
LA_An	Acceleration noise density	FS bit set to 00, normal mode(Table 8.), ODR bit set to 1001		220		ug/(√Hz)
M_R	Magnetic resolution			2		mgauss
M_CAS	Magnetic cross-axis sensitivity	Cross field = 0.5 gauss H applied = ±3 gauss		±1		%FS/ gauss
M_EF	Maximum exposed field	No permanent effect on sensor performance			10000	gauss
M_DF	Magnetic disturbance field	Sensitivity starts to degrade. Use S/R pulse to restore sensitivity			20	gauss
Top	Operating temperature range		-40		+85	°C

Figure 8: Extract of the LSM303DLHC datasheet p11/42

The LSM303DLHC datasheet provides at the page 11, the acceleration noise density: $N_{acc} = 220\ ug/\sqrt{Hz}$.
Therefore the standard deviation σ_{acc} :
$\sigma_{acc} = N_{acc} \cdot \sqrt{B_{acc}} = 220.10^-6 \cdot 9.81 \cdot \sqrt{100} = 0.022\ m/s^2$ with B_{acc} = 100 Hz the bandwidth of use of the sensor.

3.2 L3GD20 gyroscope sensor

3.2.1 Extracting turn rate theta dot

The gyroscope compute the angular velocity in each axis of the system's body frame. As we have done for the accelerometer, we compute the relation which links the roll rate (p), pitch rate (q) and yaw rate (r) from the body frame to the Euler angles in the navigation frame.

$$\begin{aligned} \text{The angular velocity } (\vec{\Omega})_n^b : \begin{pmatrix} p \\ q \\ r \end{pmatrix}^b &= \dot{\psi} \cdot \vec{z_n} + \dot{\theta} \cdot \vec{y_1} + \dot{\phi} \cdot \vec{x_2} \\ \iff \quad p \cdot \vec{x_b} + q \cdot \vec{y_b} + r \cdot \vec{z_b} &= \dot{\psi} \cdot \vec{z_n} + \dot{\theta} \cdot \vec{y_2} + \dot{\phi} \cdot \vec{x_b} \end{aligned} \tag{5}$$

Using the rotation matrices computed in the 3.1 chapter, we express the Euler angles in the body frame coordinates and we get the following :

$$\begin{cases} p = & \dot{(\phi)} - \dot{(\psi)} \sin(\theta) \\ q = & \dot{(\psi)} \cos(\theta) \sin(\phi) - \dot{(\theta)} \cos(\phi) \\ r = & \dot{(\psi)} \cos(\theta) \cos(\phi) - \dot{(\theta)} \sin(\phi) \end{cases} \tag{6}$$

If we reformulate the system in matrix form and take the inverse of it. We would we be able to get the Euler angles, a function of the pitch, roll and yaw of the body frame propeller arm. As the gyroscope provides an angular rate, we can extract back the angle information from the gyroscope by multiplying the pitch turn rate with the time difference of the sequence of reading. We consider the initial angle as zero.

$$\theta = \theta_{old} + q * dt \tag{7}$$

3.2.2 Noise of the gyrometer

Table 4. Mechanical characteristics(1)

Symbol	Parameter	Test condition	Min.	Typ.(2)	Max.	Unit
FS	Measurement range	User-selectable		±250		dps
				±500		
				±2000		
So	Sensitivity	FS = 250 dps		8.75		mdps/digit
		FS = 500 dps		17.50		
		FS = 2000 dps		70		
SoDr	Sensitivity change vs. temperature	From -40 °C to +85 °C		±2		%
DVoff	Digital zero-rate level	FS = 250 dps		±10		dps
		FS = 500 dps		±15		
		FS = 2000 dps		±75		
OffDr	Zero-rate level change vs. temperature	FS = 250 dps		±0.03		dps/°C
		FS = 2000 dps		±0.04		dps/°C
NL	Non linearity	Best fit straight line		0.2		% FS
Rn	Rate noise density			0.03		dps/(√Hz)
ODR	Digital output data rate			95/190/ 380/760		Hz
Top	Operating temperature range		-40		+85	°C

Figure 9: Extract of the L3GD20 datasheet p9/44

The L3GD20 datasheet provides at the page 9, the rate noise density of the gyrometer : $R_n = 0.03 dps/\sqrt{Hz}$. The corresponding standard deviation $\sigma_{gyro} = R_n \cdot \sqrt{B_{acc}} = 0.03 \cdot \sqrt{400} = 0.6\ dps = 0.6 \cdot \frac{\pi}{180}$ rad/s taking bandwidth of use of the gyroscope of 400 Hz.

3.3 Complementary Filter Implementation

We fully models the complementary filter on Matlab Simulink, representing faithfully the sensors noise and bias. We decided for now to implements the backward discretization, shown in **??**.

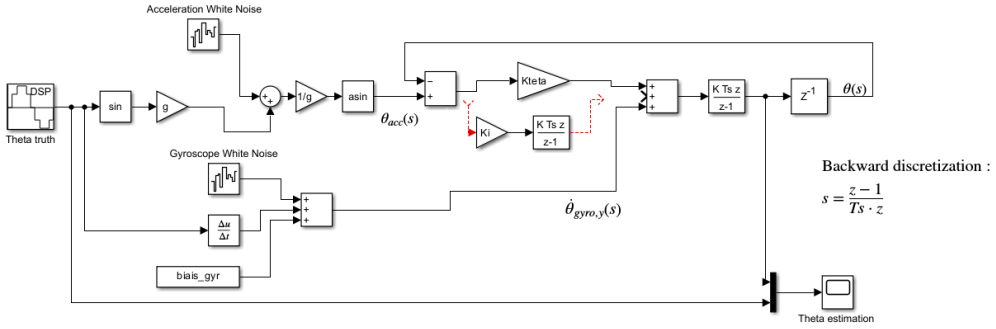

Figure 10: Complementary Filter Simulation on Simulink

Context setting for this simulation :

Acceleration White Noise	: 0.022	Kteta	: 20
Gyrometer White Noise	: 0.01047	Ki	: 99
Gyrometer Bias	: 1		

Plot showing the result of the simulation 1st (Figure 11) & 2nd order (Figure 12)

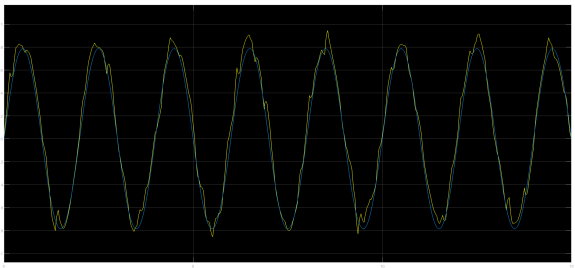

Figure 11: Scope of the curve first order system response to the sinusoidal input: $0.785398 \cdot (\sin \pi \cdot t)$

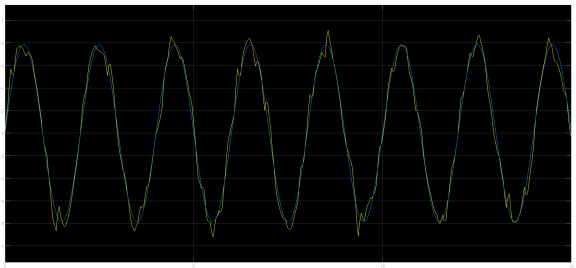

Figure 12: Scope of the curve second order system response to the same sinusoidal input

Basing our calculation on the following draft, we will now implement the complementary filter on the software of the board :

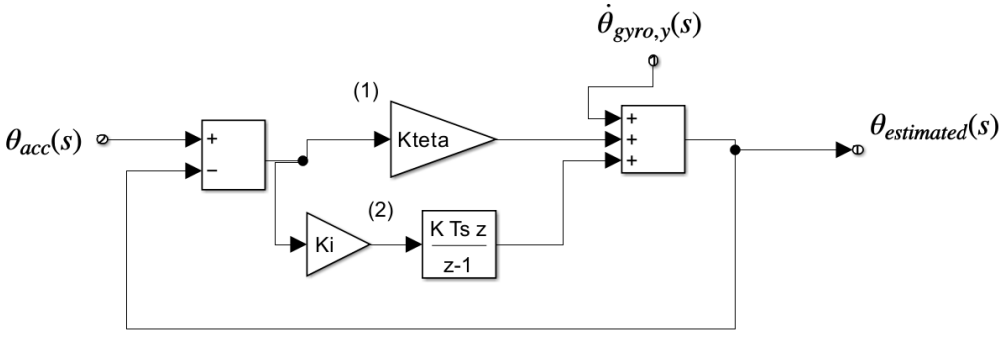

Figure 13: Complementary Filter schematic

3.3.1 Discretization of a First Order System

We first implement the first order complementary filter which involves only the Ko of the schematic, corresponding to the (1) of Figure 13.

We can express $\theta(s) = f(\theta_{gyro,y}(s), \theta_{acc}(s))$:

$$\begin{aligned} \theta(s) &= \frac{1}{s} \cdot (\dot{\theta}_{gyro,y}(s) + Ko \cdot (\theta_{acc}(s) - \theta(s))) \\ (s + Ko) \cdot \theta(s) &= (s \cdot \theta_{gyro,y}(s) + Ko \cdot \theta_{acc}(s)) \\ \theta(s) &= \frac{s}{s + Ko} \cdot \theta_{gyro,y}(s) + \frac{Ko}{s + Ko} \cdot \theta_{acc}(s) \end{aligned} \tag{8}$$

We need to implement the backward discretization ($s = \frac{1-z^{-1}}{T}$) of the equation on the STM32F411E DISCO microcontroller.

$$\begin{aligned} \theta(z) &= \frac{1}{KoT + 1 - z^{-1}} \cdot (T(1 - z^{-1})\theta_{gyro,y}(z) + KoT\theta_{acc}(s)) \\ (KoT + 1 - z^{-1}) \cdot \theta(z) &= T(1 - z^{-1})\theta_{gyro,y}(z) + KoT\theta_{acc}(s) \\ (1 + KoT)\theta[k] &= \theta[k-1] + T \cdot (\theta_{gyro,y}[k] - \theta_{gyro,y}[k-1]) + KoT \cdot \theta_{acc}[k] \end{aligned} \tag{9}$$

It is ready to be implemented in an interrupt loop function in C:

```
1
2    float TiltCompensatedThetaEstimation(void){
3
4    theta_acc = GetThetaAcc_rad();
5    gyro_y = GetThetaGyro_rad();
6
7   theta = pre_theta + T*(gyro_y - pre_gyro_y) + T*Ko*theta_acc;
8   theta = theta / (Ko*T + 1);
9
10  pre_theta = theta;
11  pre_gyro_y = gyro_y;
12  pre_theta_acc = theta_acc;
13
14  return theta;
15 }
16
```

3.3.2 Discretization of a Second Order System

Then we implement the second order complementary filter which involves the Ko with $\frac{1}{Ki \cdot s}$ of the schematic, corresponding to the (2) of Figure 13.

We can express again $\theta(s) = f(\theta_{gyro,y}(s), \theta_{acc}(s))$:

$$\begin{aligned} \theta(s) &= \frac{1}{s} \cdot (\dot{\theta}_{gyro,y}(s) + (Ko + \frac{Ki}{s}) \cdot (\theta_{acc}(s) - \theta(s))) \\ (s + Ko + \frac{Ki}{s}) \cdot \theta(s) &= s\theta_{gyro,y}(s) + (Ko + \frac{Ki}{s}) \cdot \theta_{acc}(s) \end{aligned} \tag{10}$$

Sparing the calculations, using the Euler backward discretization we obtain :

$$\begin{aligned} (\frac{1}{T} + Ko + KiT) \cdot \theta[k] = \theta[k-1] \cdot (\frac{2}{T} + Ko) - \frac{1}{T} \cdot \theta[k-1] \\ + \frac{1}{T} \cdot (\theta_{gyro,y}[k] - 2\theta_{gyro,y}[k-1] + \theta_{gyro,y}[k-2]) \\ + \theta_{acc}[k] \cdot (Ko + KiT) - Ko\theta_{acc}[k-1] \end{aligned} \tag{11}$$

It is ready to be implemented in an interrupt loop function in C:

```
float TiltCompensatedThetaEstimation2(void){

   theta_acc = GetThetaAcc_rad();
   gyro_y = GetThetaGyro_rad();

  theta =   pre_theta* (2/T + Ko) -1/T*pre_pre_theta + 1/T*(gyro_y - 2*pre_gyro_y +
    pre_pre_gyro_y) + (Ko + Ki*T)*theta_acc - Ko*pre_theta_acc;
  theta = theta / (1/T + Ko + Ki*T);

  pre_pre_theta = pre_theta;
  pre_theta = theta;

  pre_pre_gyro_y = pre_gyro_y;
  pre_gyro_y = gyro_y;
  pre_theta_acc = theta_acc;

  return theta;
}

```

3.4 Simulating the brushless motor mathematically

In order to fully models the system, I need to know the motor response to each generated PWM throttle. I've realize a test bench [7] to measure the thrust generated from a 1050 us to 1900 us throttle. To do so, I place one motor on a balance putting the two-blades propeller upside-down pointing toward the grounds (pushing in the opposite direction of a normal drone).

On the figure below, we can see the test performed, from left to right, we have the balance measuring directly the thrust, the power generator displaying the current amperage needed and the throttle corresponding.

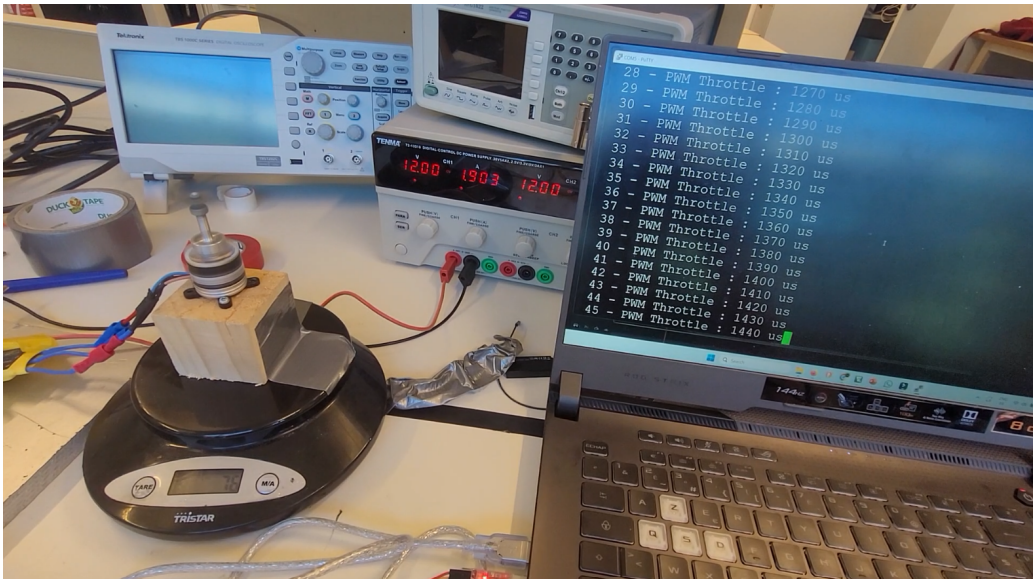

Figure 14: Test bench of the brushless motor

Using a python script, we computed the corresponding linear function representing as accurately as possible the point cloud of the test. Therefore we have :

$$Thrust[g] = 0.17 \cdot PWM_Throttle[us] - 176.21 \tag{12}$$

The drop down at the end of the simulation that we can see on the graph is due to the limitation we have fixed on the power generator (as the power supply used in our system will provided 5A max).

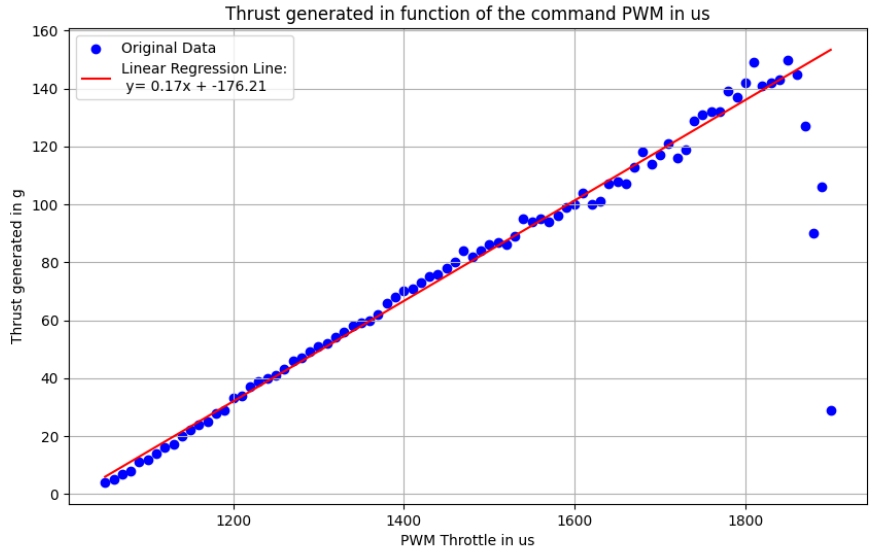

Figure 15: Plot curve of the Thrust Linear Function from the Simulation Test

Then, we obtain finally the generated force in function of the throttle :

$$MotorForce[N] = Thrust[g] \cdot g[N/kg] \tag{13}$$

3.5 System modeling of a One-degree-of-freedom Twin Propeller Arm

Thereafter, by taking over the Lab session with Mr DUCARD, we are remodelling the system with this time two propeller in each end of the arm.

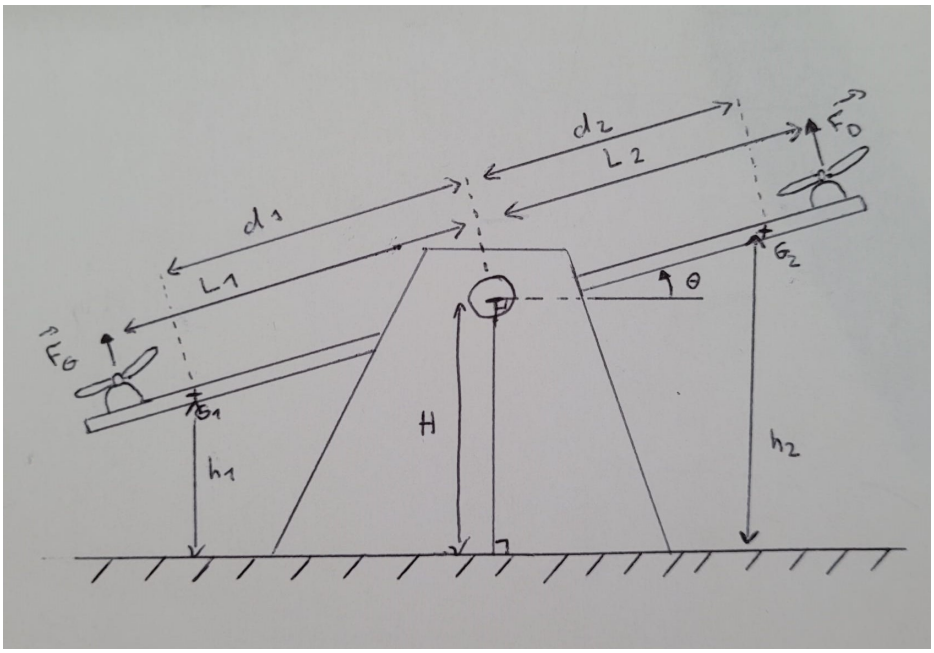

Figure 16: System schematic of the Twin Propeller Arm

Neglecting the friction for a start

$$\Rightarrow \dot{E} = \Sigma(Power) = \Sigma(\vec{F} \cdot \vec{v}) \quad \Rightarrow \text{the sum of non-gravitational forces by their velocity} \tag{14}$$

Method 1:

$$\begin{aligned} E_{tot} &= E_{kinetic} + E_{potential} \\ &= \frac{1}{2} \cdot I_{bar} \cdot \ \dot{\theta}^2 + mgh \text{ (= 0 as the center of gravity does not move)} \end{aligned} \tag{15}$$

So

$$\dot{E_{tot}} = I_{bar} \cdot \dot{\theta} \cdot \ddot{\theta} = \Sigma(\vec{F} \cdot \vec{v}) \quad = (F_D - F_G) \cdot l_1 \cdot \omega \tag{16}$$

$$\ddot{\theta}(t) = \frac{1}{I_{bar}} \cdot (F_D - F_G) \cdot l_1 \cdot \omega \tag{17}$$

Method 2:

$$\begin{aligned} I_{bar} \cdot \ddot{\theta}(t) &= \Sigma(M_i) \\ &= \vec{l_1} \wedge \vec{F_G} + \vec{l_2} \wedge \vec{F_D} + \vec{d_1} \wedge \vec{P_G} + \vec{d_2} \wedge \vec{P_D} \\ &= l_1 \cdot (F_D - F_G) \cdot \vec{z} + (mg\dot{\cos}\theta \cdot d_1 - mg\dot{\cos}\theta \cdot d_2) \cdot \vec{z} \text{ with } \begin{vmatrix} \mathbf{d2} \\ 0 \\ 0 \end{vmatrix} \wedge \begin{vmatrix} -\mathbf{mg}\sin\theta \\ -\mathbf{mg}\cos\theta \\ 0 \end{vmatrix} \text{ and } \begin{vmatrix} -\mathbf{d1} \\ 0 \\ 0 \end{vmatrix} \wedge \begin{vmatrix} -\mathbf{mg}\sin\theta \\ -\mathbf{mg}\cos\theta \\ 0 \end{vmatrix} \\ &= l_1 \cdot (F_D - F_G) \cdot \vec{z} \end{aligned} \tag{18}$$

Creating out of the equation the model block of the system, we get the following Simulink model shown in Figure [17]:

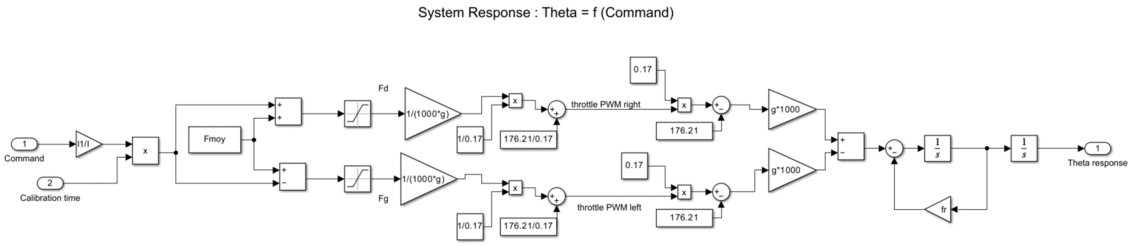

Figure 17: System response model block of the Twin Propeller Arm

We convert on the left to a throttle ready to be implemented on the code and on the right we simulated the force generated in each brushless motor to get the theta response in output.

3.6 Cascade Control of the System

Controlling the robotic arm with a cascade control one loop over the $\delta(\theta_{ref} - \theta estim)$ of the system and another loop on the integration of δ being $\dot{\delta}$ the theta turn rate.

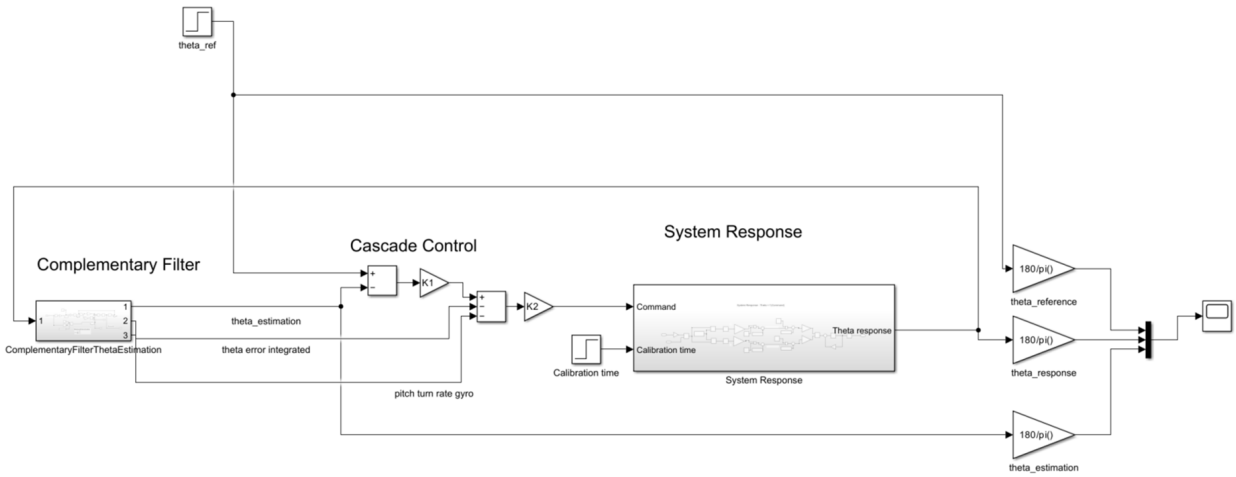

Figure 18: Cascade control of the twin propeller arm

To test the cascade control loop model block, we have performed the following simulation test : We defined the system response to 0 in the calibration phase as we don't turn on the motor during this time lapse. Then after Tcalibration = 5s, we turn on the motor to follow the theta of reference (yellow step input) changing from -45° to 45°. The theta response of the system takes approximately 1 sec to reach the expected extreme value. In red, we have drawn the theta estimation of the complementary filter.

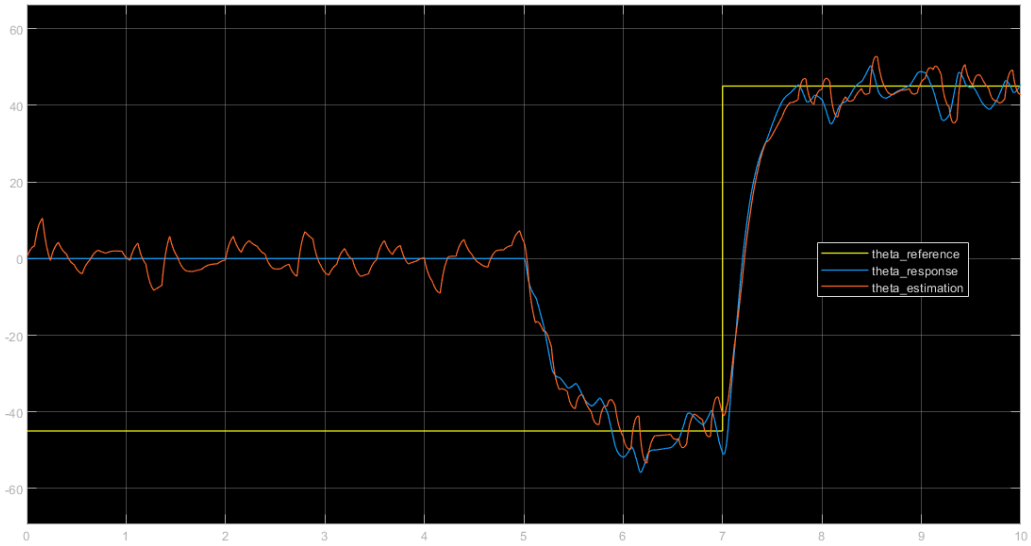

Figure 19: Plot of the final result representing the total system response

3.7 Implementation in a timer interrupt loop

```
    void Control_Loop_ThetaEstimation_CommandThrottle(void){

  //---> Compute the reference theta_ref
  theta_ref = 0;

  PrintConsole(INFO, "\r\nTheta: %.2f deg",theta_data.theta*180/M_PI);

  command = K2 * (K1 * (theta_ref - theta_data.theta) - theta_data.biais - gyro_data.
    pitchTurnRate_radps);

  GetAccData_read_fast();
  GetGyroData_read_fast();
  TiltCompensatedThetaEstimation2();

  /*  Divided by 10 because PWMd = 100 duty cycle = 1000us */
  PWMd =  (((saturate_force_input(Fmoy + command*l1/I)/(1000*G))/0.17 +176.21)/0.17 ) / 10;
  PWMg =  (((saturate_force_input(Fmoy - command*l1/I)/(1000*G))/0.17 +176.21)/0.17 ) / 10;

  MotorPWM_Set(PWMg, PWMd);

  SendValuesToQT(Get_Timestamp(), acc_data.theta_acc*180/M_PI, theta_data.theta*180/M_PI,
                 PWMg, PWMd);

  PrintConsole(INFO, "\r\nPWMg: %lf and PWMd: %lf",PWMg,PWMd);

  theta_data.biais = (acc_data.theta_acc - theta_data.theta)*Ki*T - theta_data.pre_biais;
  theta_data.pre_biais = theta_data.biais;    \

/*
 * Function: GetAccData_read_fast
 * ------------------------------
 * Reads accelerometer data and computes the angular position based on the X-axis acceleration
    .
 * The computed angle is stored in acc_data.theta_acc in radians.
 */

void GetAccData_read_fast(void){

  acc_data.pre_pre_theta_acc = acc_data.pre_theta_acc;
  acc_data.pre_theta_acc = acc_data.theta_acc;


  short accelX = ((ReadFromAccelerometer(0x29) << 8) | ReadFromAccelerometer(0x28));

  // 4,000 MilliGs / 65,535 because FS = +- 2g, convert MilliG to G, and then to m/s
  float accX = accelX * 0.061f * G / 1000.0f;

  // Compute and return the angle in radians
    acc_data.theta_acc = asinf(accX / G);
}

/*
 * Function: GetGyroData_read_fast
 * -------------------------------
 * Reads gyroscope data and updates gyro_y and pitchTurnRate_radps based on the Y-axis
    gyroscope reading.
 * The function also updates previous gyro_y values for filtering purposes.
 */

void GetGyroData_read_fast(void){

  gyro_data.pre_pre_gyro_y = gyro_data.pre_gyro_y;
  gyro_data.pre_gyro_y = gyro_data.gyro_y;

  short Y = GetAxisValue(0x2A, 0x2B);
  gyro_data.gyro_y= gyro_data.pre_gyro_y + Y*T;
  gyro_data.pitchTurnRate_radps = Y;

}

}
```

References

[1] Darwen, Terrence, STM32F4 Register programming examples, https://github.com/tmdarwen/STM32, 2021, Provided code to handle the accelerometer and gyrometer with respectively I2C & SPI.

[2] ControllerTech, STM32F4 Register programming tutorial, https://controllerstech.com/, Provided code and explanation for USART, I2C, GPIO, Clock, TIMER, ... A vast amount of knowledge to begin with.

[3] ControllerTech, Interface BLDC motor with STM32, https://controllerstech.com/how-to-interface-bldc-motor-with-stm32/, Provided code and explanation for how to configure correctly the ESC to drive the BLDC motors.

[4] NCBI, Understanding the Basics of Accelerometer and Gyroscope Sensors, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6111699/.

[5] Atadiat, Towards Understanding IMU Basics of Accelerometer and Gyroscope Sensors, https://atadiat.com/en/e-towards-understanding-imu-basics-of-accelerometer-and-gyroscope-sensors/.

[6] Shah Mihir Rajesh*, Sanman Bhargava, Soumya B, Mr.K.Sivanathan - Mission Planning and Waypoint Navigation of a MicroQuad Copter by Selectable GPS Co-Ordinates - *International Journal of Advanced Research inComputer Science and Software Engineering* Volume 4, Issue 4, April 2014 ISSN: 2277 128X

[7] Carbon Aeronautics, YouTube Video: https://www.youtube.com/watch?v=XjhY2tyhYZ8&list=LL&index=4&t=269s&ab_channel=CarbonAeronautics.

A OVERVIEW OF CONFIGURED PINS, TIMER, PROTOCOLS INTERRUPTION

Resuming all the peripherals configured and pinout used :

```
- The system Clock is configured as follow :
  *            System Clock source            = PLL (HSE)
  *            SYSCLK(Hz)                     = 100 000 000
  *            HCLK(Hz)                       = 100 000 000
  *            AHB Prescaler                  = 1
  *            APB1 Prescaler                 = 2
  *            APB2 Prescaler                 = 1
  *            HSE Frequency(Hz)              = 8 000 000
  *            PLL_M                          = 4
  *            PLL_N                          = 100
  *            PLL_P

- UART2 communication :
  // USART2 TX --> PA2 // Write
  // USART2 RX --> PA3 // Read
 [Baud rate : 115 200 bds]

- USART6 communication :
  // USART6 TX --> PC6 // Write
  // USART6 RX --> PC7 // Read
  [Baud rate : 115 200 bds]

- TIMER 5 : Delay timer
- TIMER 4 : PWM Generation Motor control
- TIMER 3 : Interruption timer for control loop
- TIMER 2 : Interrution button oscillation removal

- The gyrometer L3GD20 pins are connected to the STM32F411 (SPI1):
//    CS  --> PE3   // Chip Select
//    SPC --> PA5   // Clock
//    SDI --> PA7   // Data In
//    SDO --> PA6   // Data Out

- The accelerometer LSM303DLHC pins are connected to the STM32F411 (I2C1):
// SCL  --> PB6
// SDA  --> PB9
[ ACCELEROMETER_READ  0x33 ]
[ ACCELEROMETER_WRITE 0x32 ]

- We have chosen to connect my LCD display to the same I2C1 bus pins :
// SCL  --> PB6
// SDA  --> PB9
[ SLAVE_ADDRESS_LCD 0x4E ]

- I2C3 Slave for Touch screen Master communication
// SCL  --> PA8
// SDA  --> PC9
[ I2C_OWN_ADDRESS 0x12 ]

Power mode input pins:
// GND
// Pull-Up Mode --> PA0

Potentiometer theta reference pins:
// GND 3V
//Read value --> PA4
```

B NOTICE OF HOW TO USE THE QT INTERFACE

Go to the repository directory :

`.\PropellerArm\uvision_project_HDMHD\PythonQTinterface\src`

Open a terminal and run the command : **C:/Python312/python.exe ./main.py**
It should open this interface, plotting two empty graphs :

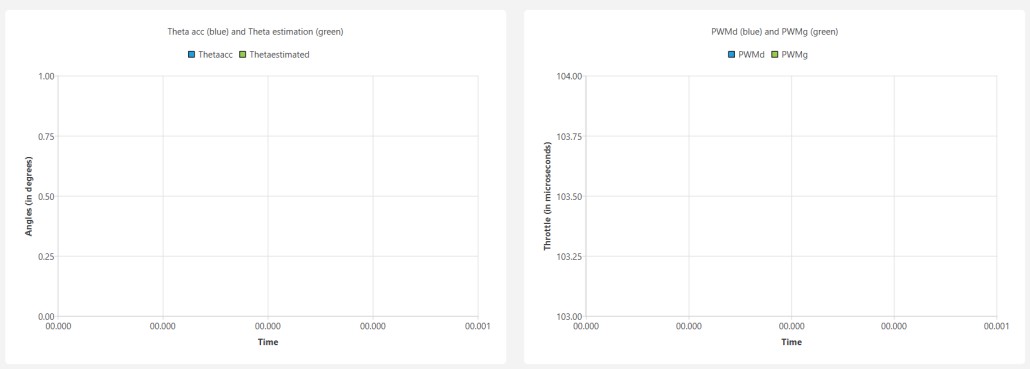

Figure 20: QT Interface prototype to debug the errors

Before check the port COM number to be connect using Device Manager in Windows for example. Then make sure the communication UART6 is not running on the stm32 (pins PC7, PC6) because it could cause to miss a start bit or an error of lecture and breaks the interface.

Then go to the menu Serial and select the right port COM:

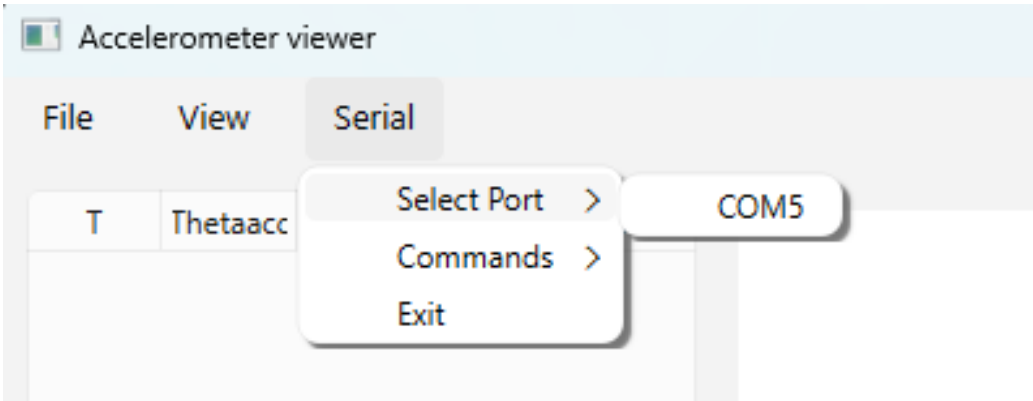

Figure 21: Select the port to establish the communication

Resulting, the plot is starting to show the measured data and command sent to the motor in real time. It is really helpful to debug the simulation :

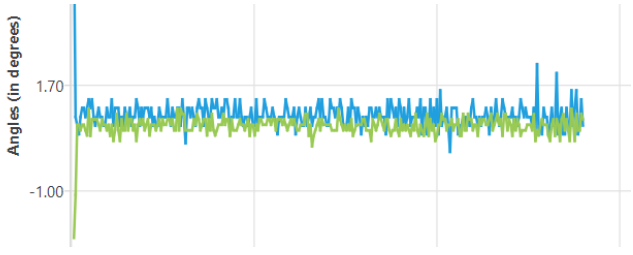

Figure 22: Plot starting showing data in real-time