POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

Embedded Control

Prof. Ing. Luca De Cicco

Project Work:

Independent joints control of a Pan-Tilt system

Students:
Carnevale Angela
Delzotto Rossella

ANNO ACCADEMICO 2023-2024

Abstract

This report presents the design and implementation of an independent joint control system for a Pan-Tilt system that can orient itself towards a given point in space using a Nucleo Board F446RE and an MPU9250 IMU sensor placed on the end effector of the system.

The goal is to move the camera to a given target point in the 3D space by computing the required pan and tilt movement angles through inverse kinematics. Data from the accelerometer and gyroscope of the IMU sensor are processed to calculate position and velocity the two motors have to reach during the entire motion.

The control approach is based on an independent joints control with velocity and position feedback. A cascaded control structure, made of a P-type position controller and a PI-type velocity controller, drives the motors to minimize the error between the current and the desired positions, keeping the rotation speed limited. The control strategy involves a sequential approach, where the Pan axis is adjusted first, followed by the Tilt axis.

The controllers are tuned to achieve stable and accurate responses. Experimental results confirm the good performance of the proposed control scheme, demonstrating precise tracking and smooth motion execution.

Contents

1 Introduction

The aim of this project is to control a pan-tilt system consisting of two rotary joints, by using a Nucleo Board F446RE by STMicroelectronics. The system is equipped with an MPU9250 IMU sensor to provide real-time position and velocity feedback. The control architecture consists of a two-stage approach: a P-type position controller and a PI-type velocity controller for each motor.

The goal is to move the system to a predefined 3D target point (in the xyz space) by computing the necessary pan and tilt angles via inverse kinematics. The movement is executed sequentially, first adjusting the pan angle and then the tilt angle.

The main challenges addressed in this work include having a stable and reliable IMU reading, ensuring smooth transitions between movements, reducing overshoot and steady-state errors.

2 Pan-Tilt System (Gimbal)

A pan-tilt system (or Gimbal system) is a two d.o.f. robot that allows a mounted camera, sensor, or other equipment to rotate along two distinct axes: horizontal (pan) and vertical (tilt). It's composed of two DC motors positioned in quadrature. This allows the device to capture a wide range of angles and views without the need to physically move the base or the entire setup.

Pan-tilt systems are widely used in surveillance applications (to cover a wide area) and in photography or videography (to stabilize images).

Figure 1: Pan-Tilt system.

2.1 Robot Kinematics

2.1.1 Direct Kinematics

We compute the direct kinematics function as a transformation matrix (Eq. 1), with Denavit-Hartenberg (DH) convention. DH parameters are represented in table 1, with $l_1 = 106mm$ and $l_2 = 15.9mm$.

Link	a_i	α_i	d_i	θ_i
Link 1	0	$-\pi/2$	l_1	θ_1
Link 2	l_2	0	0	θ_2

Table 1: DH Parameters

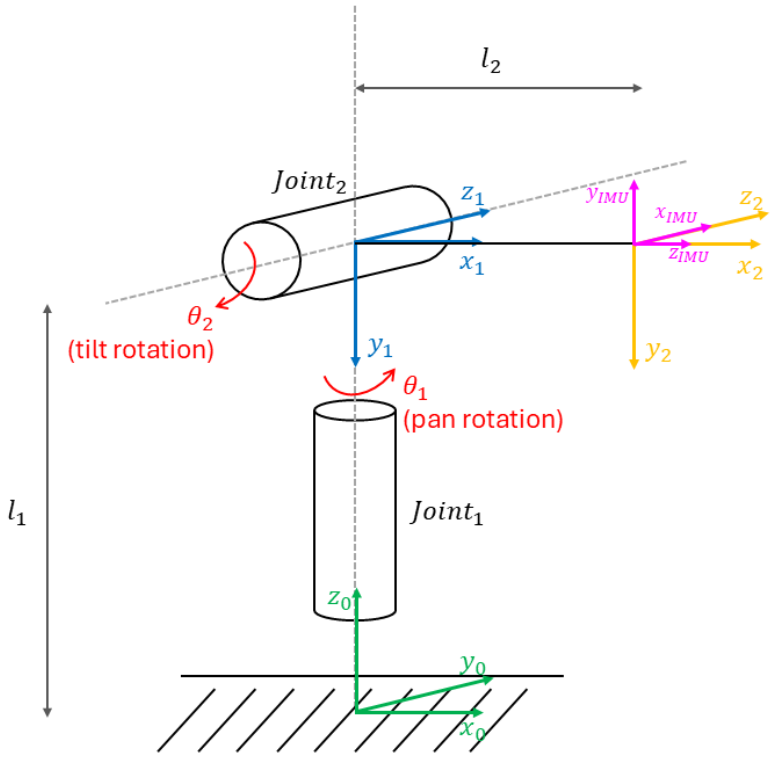


Figure 2: Pan Tilt system.

$$T = \begin{bmatrix} c_1c_2 & -c_1c_2 & -s_1 & l_2c_1c_2 \\ s_1c_2 & -s_1s_2 & c_1 & l_2s_1c_2 \\ -s_2 & -c_2 & 0 & l_1 - l_2s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

where $c_i = \cos\theta_i$ and $s_i = \sin\theta_i$.

We can also compute the rotation matrix between Joint1 and the base reference frame (Eq. 2), the rotation matrix between Joint2 and the zero reference system (Eq. 3) and the position vector of the end effector of the manipulator (Eq. 4).

$$R_{01} = \begin{bmatrix} c_1 & 0 & -s_1 \\ s_1 & 0 & c_1 \\ 0 & -1 & 0 \end{bmatrix} \tag{2}$$

$$R_{02} = \begin{bmatrix} c_1c_2 & -c_1c_2 & -s_1 \\ s_1c_2 & -s_1s_2 & c_1 \\ -s_2 & -c_2 & 0 \end{bmatrix} \tag{3}$$

$$p_2 = \begin{bmatrix} l_2c_1c_2 \\ l_2s_1c_2 \\ l_1 - l_2s_2 \end{bmatrix} \tag{4}$$

2.1.2 Inverse Kinematics

The inverse kinematics problem consists in the determination of the joint variables corresponding to a given end effector pose. [1]

In this project we consider a P point in the xyz space, where the camera (the end effector of the manipulator) must point to. We can compute the pan angle (Θ_1) by projecting the point P onto the $x_0 - y_0$ plane, obtaining P'. (Fig. 3)

$$\Theta_1 = \arctan \frac{y}{x} \tag{5}$$

In Fig. 4 we represent a section of the system. Looking at that representation we can see that the tilt angle can be computed as follows:

$$\Theta_2 = \arctan \frac{z - l_1}{\sqrt{x^2 + y^2}} \tag{6}$$

We have to take into account the singularities of the manipulator and its reachable workspace, due to its structure. In fact, both pan and tilt angles have a limited range of movement: ideally it is $[0, 180]\, deg$ for both angles, but due to physical limitation, the tilt angle falls within the range $[0, 127]\, deg$. Moreover, there are multiple solutions if $x = y = 0$.

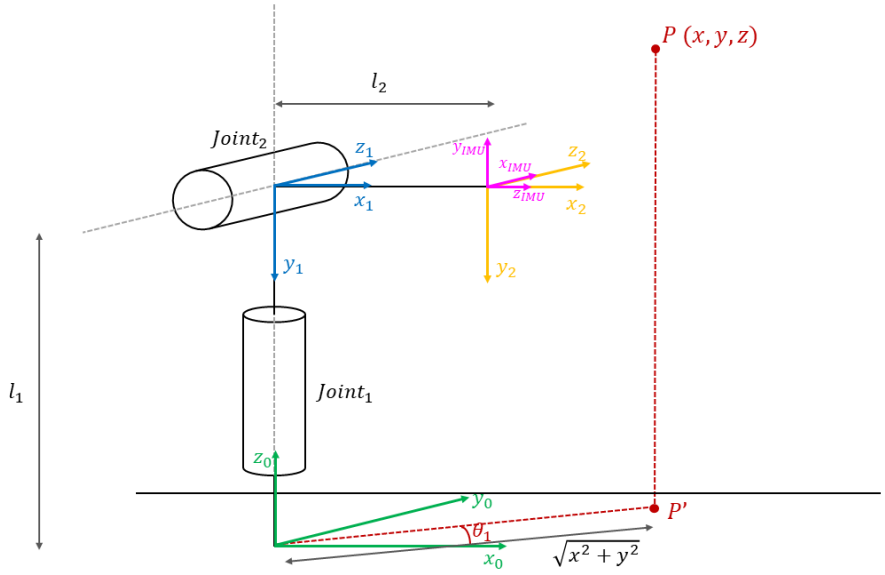


Figure 3: Calculation of pan angle with inverse kinematics.

3 Hardware

3.1 DC Motors

The system includes two Maxon 144240 twin motors (Fig. 5). Each motor is made up of a stator, a fixed component that performs the function of an inductor, since it has a winding crossed by supplied direct current, in this case, by a 12V power supply.

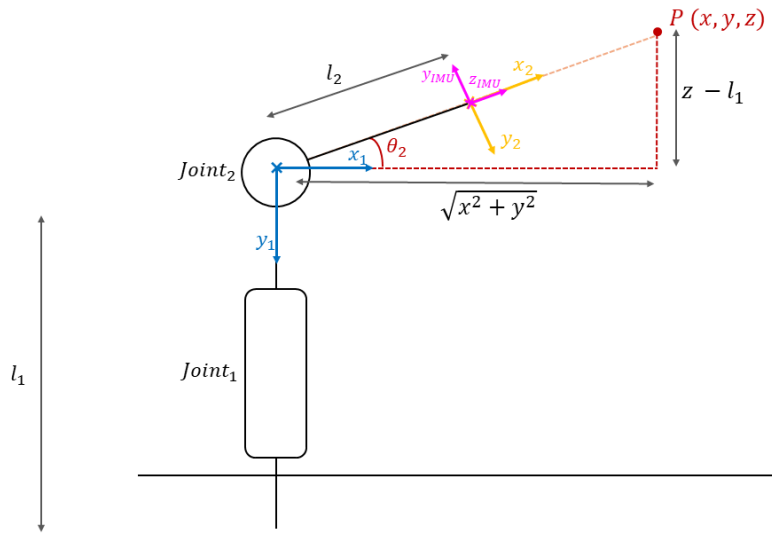


Figure 4: Calculation of tilt angle with inverse kinematics.

The first motor, the one that regulates left and right movement (the *pan* movement), contains an incremental encoder. The second motor, the one that regulates the down and up movement (the *tilt* motor), has a malfunctioning incremental encoder, therefore it is not possible to use the data from this sensor.

Figure 5: DC motor Maxon 144240.

3.2 Motor Driver

The L298N driver (Fig. 6) is used to control the DC motors. The L298N is a dual H-bridge motor driver integrated circuit (IC) manufactured by STMicroelectronics, and it is designed to drive two motors independently or one stepper motor with bidirectional control. [2]

The L298N operates on a supply voltage up to $46V$, making it versatile for low to moderate-voltage motors. It can deliver a continuous output current up to $2A$ per channel, with peak currents reaching $3A$ per channel for a short period of time. Moreover, it has built-in thermal protection to prevent overheating, which is essential for reliable operation.

The IC contains two H-bridge circuits, each capable of independently driving a single DC motor. By controlling the current direction through each H-bridge, the motor can spin in both forward and reverse directions. The L298N supports Pulse Width Modulation (PWM) for speed control, allowing variable speed adjustments based on PWM input signals. Each motor channel has an Enable pin that can be used to turn the motor on or off. Input pins (IN1, IN2 for one motor, IN3, IN4 for the other) control the direction of rotation by setting their state high or low.

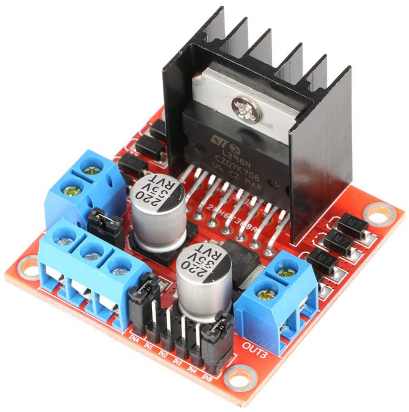


Figure 6: Motor Driver

3.3 IMU Sensor

The MPU-9250 (Fig. 7) is a compact, 9-axis motion-tracking device that integrates a 3-axis gyroscope, a 3-axis accelerometer, and a 3-axis magnetometer. [3] It belongs to the class MPU-92xx, sharing the same registers, except for the magnetometer, of the class MPU-65xx. [4] It is developed by InvenSense and typically communicates with a microcontroller through I2C or SPI protocols, which are common in embedded systems for data transfer.

Key features are:

- 3-Axis Gyroscope: Measures angular velocity (i.e., rotational speed) along the X, Y, and Z axes, which is useful for detecting changes in orientation or rotational movements;
- 3-Axis Accelerometer: Detects linear acceleration across the three axes, useful for tracking movement, determining orientation (like tilt or pitch), and detecting free fall;
- 3-Axis Magnetometer: Measures magnetic field strength and direction, acting as a compass. This sensor is crucial for determining absolute heading relative to the Earth's magnetic field, aiding in accurate orientation data.

The diagrams in Fig. 8a and Fig. 8b show the orientation of the axes of sensitivity and the polarity of rotation.

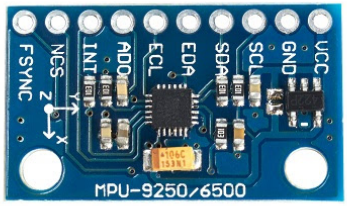
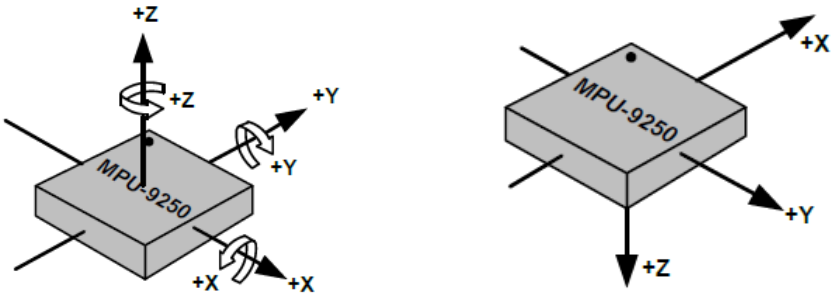


Figure 7: MPU-9250



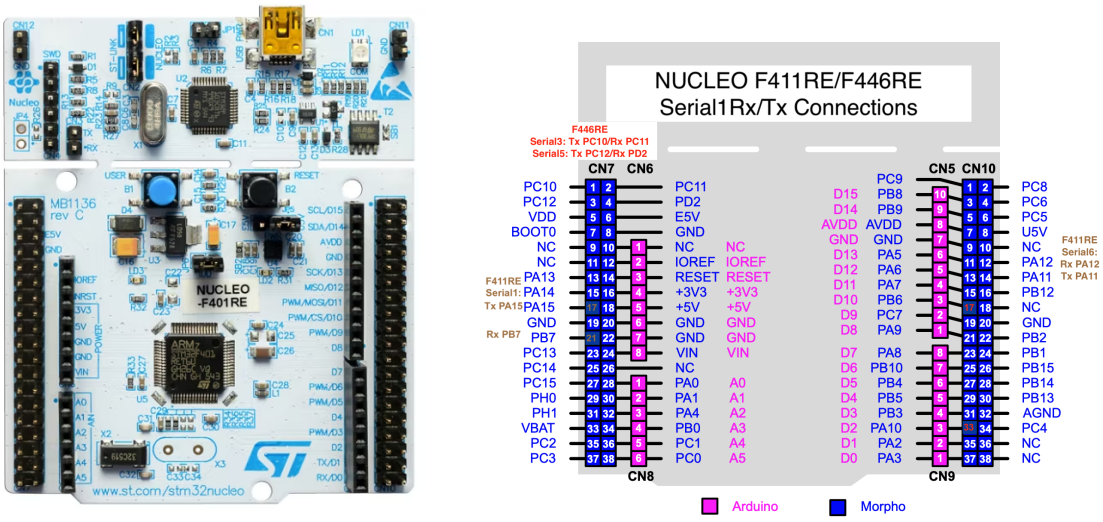
(a) Orientation of axes of sensitivity and polarity of rotation for accelerometer and gyroscope.

(b) Orientation of axes of sensitivity for magnetometer.

Figure 8: MPU-9250 axes orientation.

3.4 Board STM32F446RE

Nucleo-F446RE (Fig. 9a) is a STMicroelectronics development board based on the STM32F446RE microcontroller, part of the STM32 family. This board features a 32-bit ARM Cortex-M4 CPU, running at 180 MHz with 512 KB of Flash memory and 128 KB of RAM. It provides a range of peripherals, including ADC, DAC, timers, and communication interfaces (UART, SPI, I2C, CAN), making it suitable for complex embedded applications. [5] The Pinout of the borad is represented in Fig. 9b.



(a) Board [6]

(b) Pinout

Figure 9: Nucleo STM32F446RE

4 Configuration

4.1 STM32Cube IDE settings

4.1.1 Pin Settings

- PA0 (En_Mot1): Enable Motor1. It is set as TIM2_CH1, which controls the Motor1 speed.
- PA1 (En_Mot2): Enable Motor2. It is set as TIM2_CH2, which controls the Motor2 speed.
- PB4 ($Mot1\_Dir1$): Direction1 of Motor1. It's set as GPIO Output and controls the first direction of the Motor1. When its value is high, the motor rotates in the clockwise direction.
- PB10 ($Mot1\_Dir2$): Direction2 of Motor1. It's set as GPIO Output and controls the second direction of the Motor1. When it has a high value, the motor rotates in the counterclockwise direction.
- PB5 ($Mot2\_Dir1$): Direction1 of Motor2. It's set as GPIO Output and controls the first direction of Motor2. When its value is high, the motor rotates in the clockwise direction.
- PA10 ($Mot2\_Dir2$): Direction2 of Motor2. It's set as GPIO Output and controls the second direction of Motor2. When it has a high value, the motor rotates in the counterclockwise direction.
- PB8 (SCL_IMU): Serial Clock Line (SCL). It is set as $I2C1\_SCL$ for the I2C serial communication protocol between the IMU sensor and the board.
- PB9 (SDA_IMU): Serial Data Line (SDA). It's set as $I2C1\_SDA$ for the I2C serial communication protocol between the IMU sensor and the board.
- PA8 (OUT_VDD): it's set as GPIO Output, with pulldown resistor. It's used to turn on the switch circuit for the IMU sensor.

4.1.2 Connectivity Settings

We set the I2C1 Speed Mode as Standard Mode, i.e. $f_{i2c1} = 100kHz$.

4.1.3 Timers Settings

TIM2 is used for PWM Generation to control motors speed. We set Prescaler (PSC) to 0 and Autoreload value (ARR) to 4499, with $f_{APB1} = 90MHz$, to have a $f_{PWM} = 20kHz$ as follows:

$$f_{PWM} = \frac{f_{APB1}}{(ARR+1)(PSC+1)} \tag{7}$$

Consequentially, we set the duty cycle value by changing the CCR register, as follows:

$$DUTY\_CYCLE = \frac{CCR}{ARR+1} \implies CCR = DUTY\_CYCLE \cdot (ARR+1) \tag{8}$$

TIM6 is used to sample the IMU data and actuate the control law. We chose $f_{TIM6} = 200Hz$, corresponding to $dt = 5ms$. We set $PSC = 9$ and $ARR = 44999$ accordingly to the following equation:

$$f_{TIM6} = \frac{f_{APB1}}{(ARR+1)(PSC+1)} \tag{9}$$

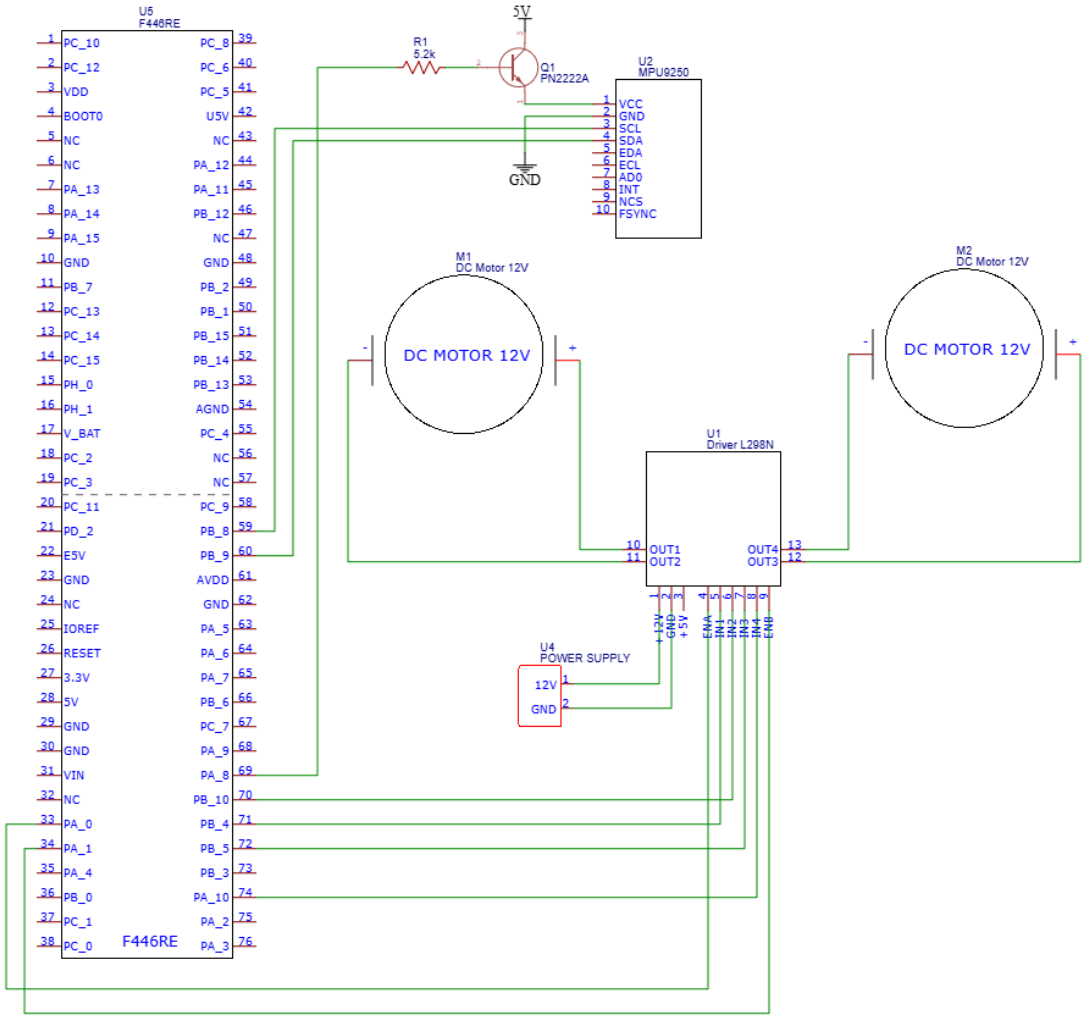


Figure 10: Schematic of the system.

4.2 Schematic

We implement a switch to turn on the IMU sensor when needed and avoid syncronization problems. It is composed of a $5.1k\Omega$ resistor and a bjt transistor (PN2222A [7]). The complete schematic is represented in Fig.10.

5 Reading Angles

In this project, we use data from the IMU sensor. First of all, we conduct a comparison between the angle readings obtained from the IMU sensor (Section 5.1) and those from the encoders (Section 5.2), which are known to provide a more reliable position measurement. The comparison is described in Section 5.3.

5.1 Reading Angles with IMU Sensor

According to the IMU datasheet [3] and register map [8] we write an initialization function `MPU9265_Init` where we enable the gyroscope, with a full scale range of $\pm 500 dps$ and the accelerometer, with a full scale range of $\pm 2g$, to deal with moderate speeds. We also set the Sample Read Register to read data at $400 Hz$.

In order to minimize the drift error we implement the calibration function for the 3-axis gyroscope (`MPU9265_CalibrateGyro`) by reading 1000 raw gyroscope data with the motor stopped in the same position and averaging the readings for each xyz axis. The computed averages represent the gyroscope offsets that compensate the sensor bias. In later calculation, they can be subtracted from raw read data to improve measurement accuracy.

We mounted the sensor in such a way that the IMU axes and the ones of the manipulator's end effector matches as in Fig.2: the IMU's x-axis is aligned with the rotation axis of the Motor2, the IMU's z-axis is parallel to the y-axis of the Motor2 and the IMU's y-axis is aligned with the y-axis of the Motor2 but points in the opposite direction. In Fig. 11 there is a photo of the mounted system in the starting position.

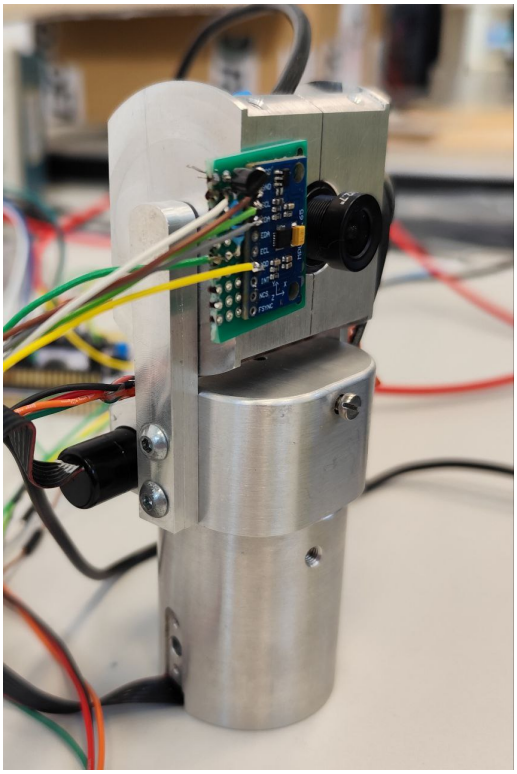


Figure 11: Pan-Tilt system with IMU sensor in the starting position.

We can see that the tilt angle (Θ_2) always corresponds to the rotation around the x-axis (roll angle) of the IMU sensor. Hence, we decide to combine accelerometer and gyroscope data to estimate the tilt angle accurately. We can estimate Θ_2 with the accelerometer data as:

$$\Theta_{2_{acc}} = \arctan \frac{a_y}{\sqrt{a_x^2 + a_z^2}} \cdot \frac{180}{\pi} - offset_{\Theta_2} \tag{10}$$

where a_x, a_y and a_z are the gravitational acceleration components along the xyz axis of the IMU sensor. We set $offset_{\Theta_2} = 87.5\, deg$, so that tilt angle is zero in the starting position. This

computation gives a stable tilt estimation based on gravity but is sensitive to noise and external accelerations. In order to correct these data we use a complementary filter to fuse accelerometer and gyroscope data. It is a weighted combination of both accelerometer and gyroscope data, so that we have a more reliable tilt estimation. Therefore, we can compute the tilt angle as:

$$\Theta_2 = roll_{IMU} = \alpha \cdot (\Theta_{2_{prev}} - \omega_x \cdot dt) + (1 - \alpha) \cdot \Theta_{2_{acc}} \tag{11}$$

where α is the gyroscope weight (we set $\alpha = 0.95$), $\Theta_{2_{prev}}$ is the previous value of tilt angle (in deg), ω_x is the value read by gyroscope (in dps), dt in the sampling time (in sec), $\Theta_{2_{acc}}$ is the tilt angle based on accelerometer data (in deg, eq. 10).

The pan angle (Θ_1) corresponds to the rotation around the y-axis (pitch angle) of the IMU sensor only when the manipulator is in the pose shown in Fig. 11. For this reason we set that as the starting position. We can compute the pan angle with the data of the 3-axis gyroscope of the IMU sensor as:

$$\Theta_1 = \int \omega_y \cdot dt \sim \sum_{i=1}^{N} \omega_{y,i} \cdot dt \tag{12}$$

where ω_y is the angular velocity along the y-axis of the IMU sensor, dt is the sampling time and N is the number of samples. Here we don't need a complementary filter because y-axis of the IMU sensor always corresponds to the gravity acceleration, thus the accelerometer does not give any useful information.

We implement all the functions related to the IMU sensor in Code 2.

5.2 Reading Angles with Encoders

We compute the pan angle for Motor1 using the built-in incremental encoder; for Motor2 we use the AS5660 magnetic encoder [9] to read tilt angle. We use Arduino Nano [10] to read data with the encoders (Code 1).

We compute the pan angle (Θ_1) as follows:

$$\Theta_1 = \frac{360}{2 \cdot PPR} \cdot N_{imp} \tag{13}$$

where $PPR = 550$ is the Pulse Per Revolution and it's calculated with a logical analyzer since we have no datasheet for the incremental encoder and N_{imp} is the number of pulses detected at a given moment.

In order to read tilt angle (Θ_2), we use the library [11], to deal with the magnetic encoder data. It returns raw values in the range $[0, 4095]$ and the angle is computed as follows:

$$\Theta_2 = \frac{RawValue}{TickPerRevolution} \cdot 360 \tag{14}$$

where $TickPerRevolution = 4096$ and it's the resolution of the encoder and 360 is the total amount of degrees in a complete revolution.

5.3 Compare IMU Data and Encoders Data

We compare the pan angle (Θ_1) and tilt angle (Θ_2) computed with the IMU data with the ones respectively read by the incremental encoder and the magnetic encoder.

In Fig. 12 there is the comparison for the pan angle; in Fig. 13 there is the comparison for the tilt angle.

The maximum error between the pan angle data is $\approx 2\,deg$; the maximum error between the tilt angle data is $\approx 6\,deg$. As a result, we can conclude that the IMU sensor estimates both angles quite well, compared to the better encoder measurements.

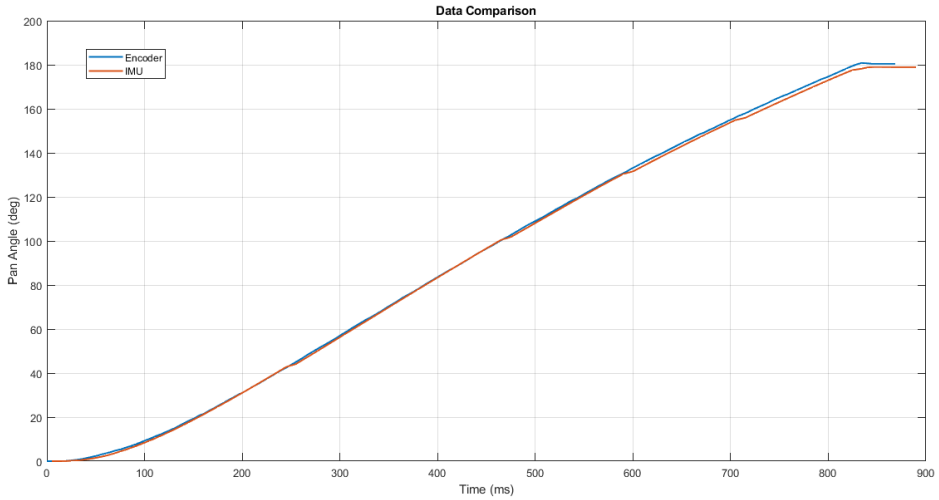


Figure 12: Data comparison for pan angle.

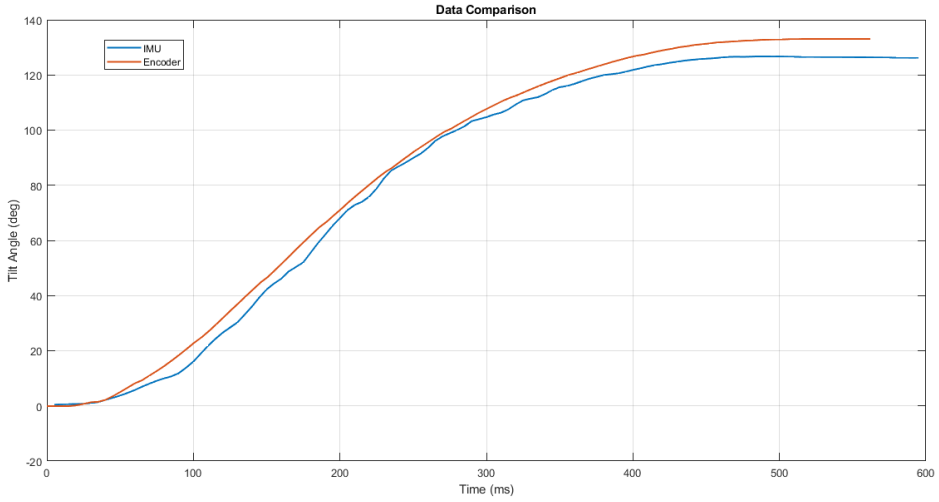


Figure 13: Data comparison for tilt angle.

6 Independent joints control

Independent joint control is a fundamental strategy in robotics where each joint of a manipulator is controlled individually, without accounting for the dynamic coupling between joints. This approach simplifies the control problem by treating the manipulator as a set of decoupled single-input single-output (SISO) systems. [1]

A general feedback-based control loop can be implemented with a proportional–integral–derivative controller (PID controller). The control signal of a PID controller is:

$$u(t) = K_P e(t) + K_I \int e(t)\,dt + K_D \frac{de(t)}{dt} \tag{15}$$

where K_P is the proportional gain, K_I is the integral gain, K_D is the derivative gain and $e(t)$ is the error of the system at time t.

A typical approach is to implement a velocity feedback and a position feedback, with a cascaded control structure: there is an inner velocity control loop and an outer position control loop. In Fig. 14 we represent the structure of the control loops: $C_P(s)$ represents the position controller and $C_V(s)$ represents the velocity controller. The velocity loop is regulated by a Proportional-Integral (PI) controller, while the position loop is managed by a Proportional (P) controller. Both controllers equation are described in Eq.16.

$$\begin{aligned} C_P(s) &= K_{PP} \\ C_V(s) &= K_{PV} + \frac{K_{IV}}{s} \end{aligned} \tag{16}$$

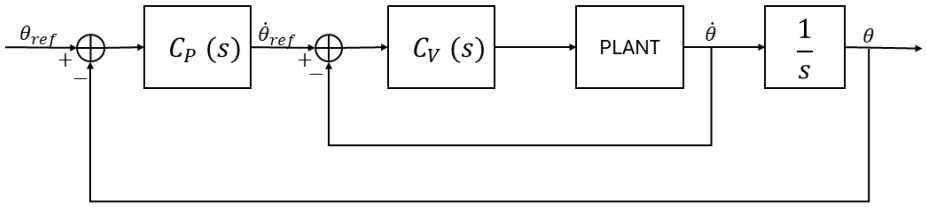


Figure 14: Control loop of a single joint.

6.1 Motors Controllers

We tune the motors' controllers with a trial-and-error method in order to achieve a balance between fast response and stability. We do that by tuning the velocity (PI) controller first, and then tuning the position (P) controller.

At first we set $K_{IV} = 0$ and we gradually increase K_{PV} starting from 0, until we reach a fast response without excessive oscillations. Then we progressively increase K_{IV}, in order to reduce steady-state error.

Later we tune K_{PP} by gradually increasing its value until the motor follows the target position accurately without excessive overshoot.

At last we test the system with different target angles and points. Some of them are described in Section 6.1.1 and Section 6.1.2.

6.1.1 Motor1 Controller

The tuned Motor1's controller has the following parameters:

$$\begin{aligned} K_{PP} &= 18 \\ K_{PV} &= 5 \\ K_{IV} &= 4 \end{aligned} \tag{17}$$

We also set limits, expressed as CCR values, for the output of the velocity controller: $outputMin = 0$ and $outputMax = 675$, which corresponds to 15% of the maximum speed ($CCR = 4500$). This constraint prevents excessive velocity.

In Fig. 15, 16 and 17 we represent the step response of the Motor1 to, respectively, $Pan_{Target} = 40\,deg$, $Pan_{Target} = 90\,deg$ and $Pan_{Target} = 120\,deg$. We can see that the system always reaches the target angle with an error $\approx 1\,deg$ and a settling time of about five seconds, without overshoot.

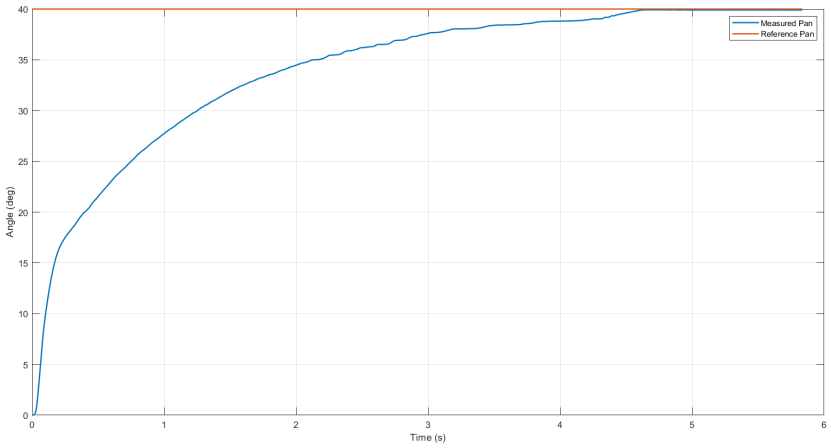


Figure 15: Step response of Motor1 to $Pan_{Target} = 40deg$.

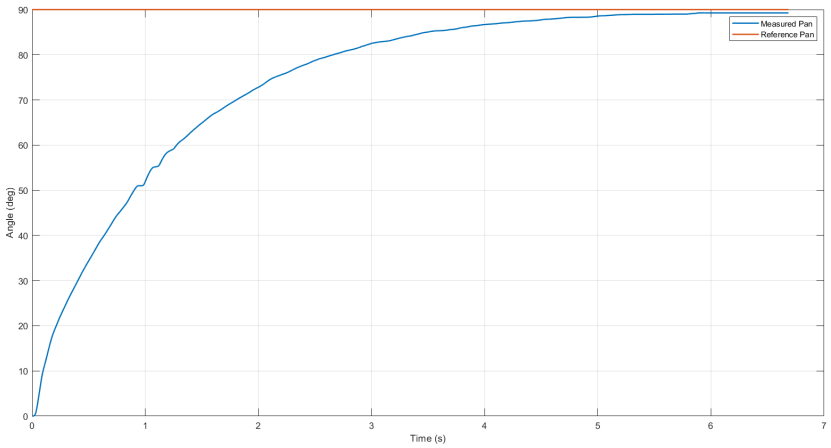


Figure 16: Step response of Motor1 to $Pan_{Target} = 90deg$.

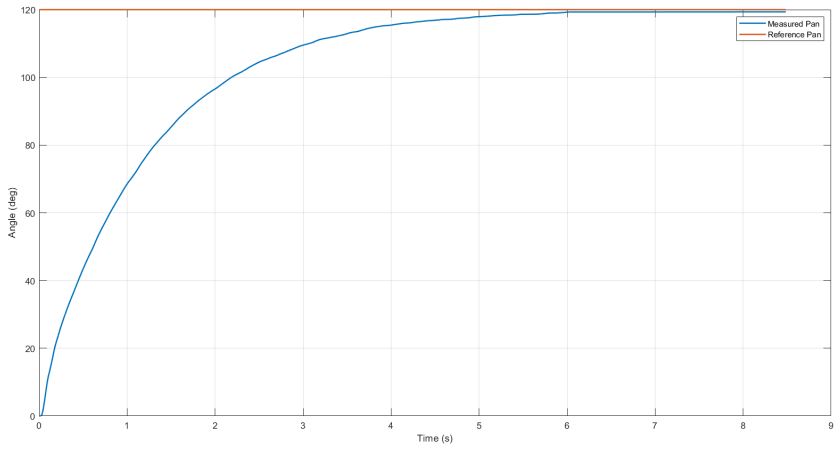


Figure 17: Step response of Motor1 to $Pan_{Target} = 120\,deg$.

6.1.2 Motor2 Controller

The tuned Motor2's controller has the following parameters:

$$\begin{aligned} K_{PP} &= 15 \\ K_{PV} &= 7.5 \\ K_{IV} &= 3.5 \end{aligned} \tag{18}$$

We also set limits for the velocity controller output as CCR values: $outputMin = 0$ and $outputMax = 540$, which corresponds to 12% of the maximum speed ($CCR = 4500$). That allows us to not have an excessive velocity. Compared to those of the first motor, the different values for the controller parameters and the different output limits are related to a different hardware construction of the two arms of the manipulator.

In Fig. 18, 19 and 20 we represent the step response of the Motor2 respectively to $Tilt_{Traget} = 40\,deg$, $Tilt_{Traget} = 90\,deg$ and $Tilt_{Traget} = 120\,deg$. In Fig. 18 and 19 we can see that the system settles to the target with an error of $\approx 2\,deg$ and a overshoot $\approx 5\%$; in Fig. 20 we can see that the system reaches the target without overshoot and with an error of $\approx 1\,deg$. The settling time is always below the second.

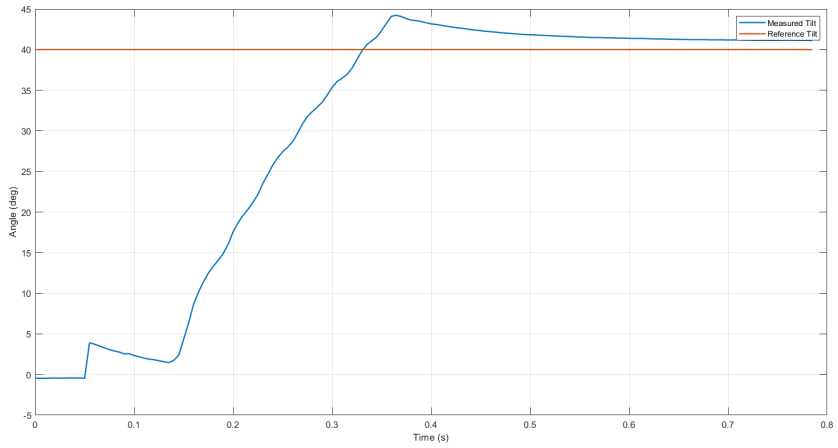


Figure 18: Step response of Motor2 to $Tilt_{Target} = 40deg$.

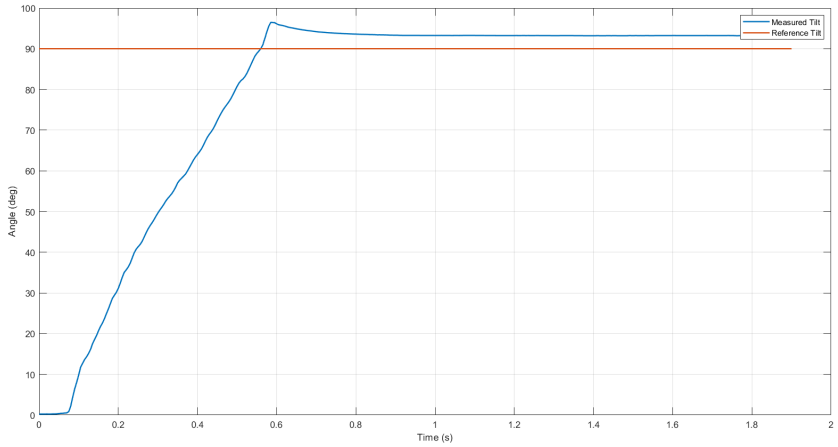


Figure 19: Step response of Motor2 to $Tilt_{Target} = 90deg$.

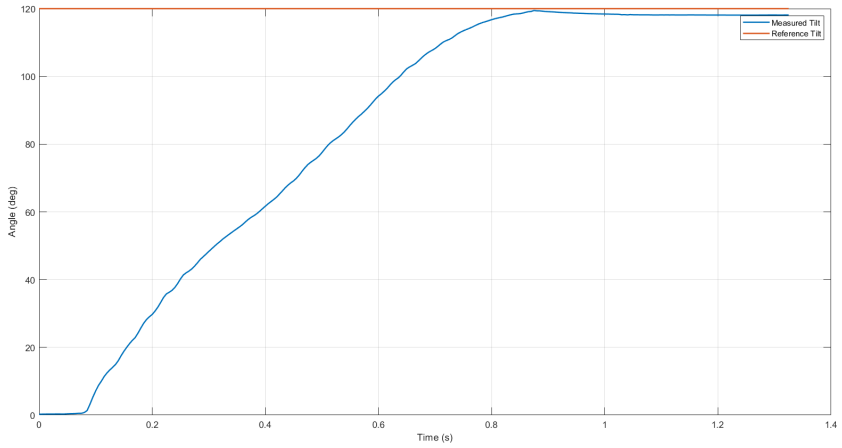


Figure 20: Step response of Motor2 to $Tilt_{Target} = 120\,deg$.

7 Software Implementation

The main code of the project (Code 5) involves the initialization of all the used peripherals, the initialization of all controllers and the control loop.

7.1 Initializing Functions

7.1.1 Initialize IMU sensor

Before using the IMU sensor, we verify its availability by checking its status. This is done through the `CheckStatus` function (Code 2, lines 19-28), which reads the IMU sensor's ID and status to ensure it is accessible.

Once confirmed, we initialize the sensor using a dedicated initialization function (Code 2, lines 32-72). This function activates the IMU sensor and configures all the necessary registers, as detailed in Section 5.1. It is executed once before entering the main while loop.

7.1.2 Initialize Motors

We implement an initialization function for each motor, in Code 3. That function is used once, before entering the main while loop. It allows us to move the motors in the starting position (Fig. 11) with a defined velocity, established by the given CCR value.

7.1.3 Compute Inverse Kinematics

We implement the inverse kinematics described in Section 2.1.2 in Code 4, lines 51-91. This function allows us to compute the desired pan and tilt angles given the point we want to reach with the camera. It is used once, at the beginning of the main code.

7.1.4 Initialize Controllers

We implement an initialization function for the controllers in Code 4, lines 93-105. Here we can set the proportional constant, the integrative constant, and the derivative constant of each controller and the limitation of the output, expressed as a CCR value, if needed.

7.2 Control Loop

In order to have a sequential control of the pan and tilt joints, we use a state machine defined with three states:

1. *PAN_ CONTROL*: in this state we execute the pan movement and control it with feedback. Once the joint reaches the target angle, the state machine switches to the next state.
2. *TILT_ CONTROL*: in this state we execute the tilt movement and control it with feedback. Once the joint reaches the target angle, the state machine transitions to the next state.
3. *SEQUENCE_ DONE*: this state indicates that the desired positions for both joints have been achieved.

The starting state is *PAN_ CONTROL* and it's defined in the main code 5 (line 61) with all the needed variables.

We implement the control loop in the Timer6 callback (Code 5, lines 208-227), so that the actuation of the control occurs each $dt = 5ms$. The control loop follows these steps:

- Read pan and tilt angles with IMU data;
- Control motors based on the actual state of the state machine.

8 Results

As an example, we set the target point as $targetPoint = [-35, -20, 160]\, mm$. The computed angles are $\Theta_1 = 29.74\, deg$ and $\Theta_2 = 126.74\, deg$.

We represent the step response of the two motors in Fig. 21 and Fig. 22. We can see that both motors reach the target angles quite well, with $e \approx 1\, deg$, but with different settling times: Motor1 is slower than Motor2. Motor1 has no overshoot and Motor2 has a little overshoot ($\approx 1\%$).

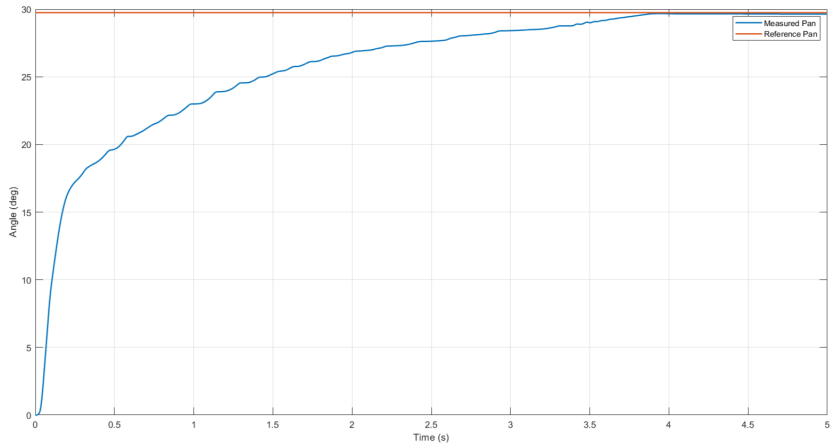


Figure 21: Step response of Motor1 to $Pan_{Target} = 29.74\, deg$.

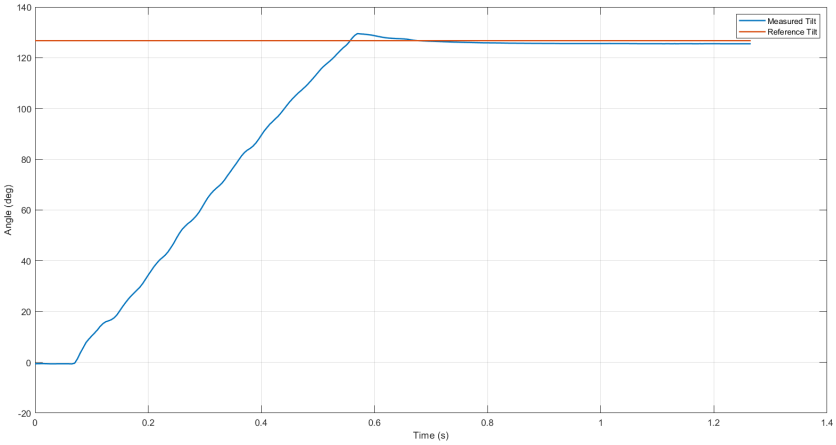


Figure 22: Step response of Motor2 to $Tilt_{Target} = 126.74\,deg$.

9 Conclusions

The developed embedded system successfully implements a pan-tilt control mechanism based on an IMU sensor mounted on the end effector. The system demonstrates accurate positioning at a desired target point by sequentially actuating the motors: first adjusting the pan angle and then the tilt angle. The control systems for the two joints provide a stable and precise response. The controllers for the two motors work independently, each of them with their parameters carefully tuned, due to the different arm construction and structure.

However, a notable limitation is that the system must always return to its initial position before moving to a new target. This restriction happens because the pan angle calculation is only accurate when the IMU's Y-axis is aligned with the Z-axis of the first motor. As a result, the system cannot move continuously between multiple target points in a sequence.

To overcome this limitation, future work should focus on computing and incorporating a rotation matrix evaluated between the IMU sensor and the first motor. This improvement would enable real-time compensation for the orientation differences, allowing for continuous and accurate pan angle calculations regardless of the tilt motor's position. That solution would significantly expand the system's flexibility and usability in dynamic applications.

References

[1] Bruno Siciliano et al. *Robotica–Modellistica, Pianificazione e Controllo*. McGraw-Hill libri Italia, 2008.

[2] STMicroelectronics. *L298: Dual Full-Bridge Driver*. 2023. URL: `https://www.st.com/resource/en/datasheet/l298.pdf`.

[3] InvenSense. *MPU-9250 Product Specification (Rev. 1.1)*. InvenSense Inc. June 2016. URL: `https://invensense.tdk.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf`.

[4] InvenSense, TDK. *MPU-9250 Product Specification*. Version 1.3. 2016. URL: `https://invensense.tdk.com/wp-content/uploads/2016/10/AN-IVS-0001EVB-00-v1-3.pdf`.

[5] STMicroelectronics. *STM32F446xC/E Datasheet: Arm® Cortex®-M4 32-bit MCU+FPU, 225 DMIPS, up to 512 KB Flash/128+4 KB RAM, USB OTG HS/FS, seventeen TIMs, three ADCs and twenty communication interfaces*. Document DS10693 Rev 10. Jan. 2021. URL: `https://www.st.com/resource/en/datasheet/stm32f446re.pdf`.

[6] STMicroelectronics. *NUCLEO-F446RE Development Board*. 2024. URL: https://www.st.com/en/evaluation-tools/nucleo-f446re.html.

[7] Philips. *PN2222A Datasheet*. 2000. URL: https://www.alldatasheet.com/datasheet-pdf/pdf/18722/PHILIPS/PN2222A.html.

[8] TDK InvenSense. *MPU-9250 Register Map and Description Revision 1.6*. Feb. 2015. URL: https://invensense.tdk.com/wp-content/uploads/2015/02/RM-MPU-9250A-00-v1.6.pdf.

[9] ams AG. *AS5600: 12-Bit Programmable Contactless Potentiometer*. ams AG. June 2018. URL: https://files.seeedstudio.com/wiki/Grove-12-bit-Magnetic-Rotary-Position-Sensor-AS5600/res/Magnetic%20Rotary%20Position%20Sensor%20AS5600%20Datasheet.pdf.

[10] Arduino S.r.l. *Arduino® Nano Product Reference Manual*. SKU: A000005. Arduino S.r.l. Jan. 2025. URL: https://docs.arduino.cc/resources/datasheets/A000005-datasheet.pdf.

[11] Rob Tillaart. *AS5600 Arduino Library*. https://github.com/RobTillaart/AS5600. 2024.

A Codes

Code 1: Pan and Tilt angles readings using Arduino Nano and encoders.

```
#include <SPI.h>
#include <Wire.h>
#include <AS5600.h>

const int encoderPinA = 2;  // Encoder pin A -> Arduino pin 2
const int encoderPinB = 3;  // Encoder pin B -> Arduino pin 3
AS5600 as5600; // A4->SDA A5->SCL (Magnetic Encoder)

// Optical Encoder parameter
const int ticksPerRevolution = 2 * 550;
const float degreesPerTick = 360.0 / (2 * ticksPerRevolution);
// Magnetic Encoder parameter
const int encoderTicksPerRevMagnetic = 4096;
const float degreesPerTickMagnetic = 360.0 / encoderTicksPerRevMagnetic;
float offset = 110;

float currentTheta1 = 0.0;
float currentTheta2 = 0.0;

unsigned long lastTime = 0;
const unsigned long interval = 5; //dt


void setup() {

  Serial.begin(115200);
  Wire.begin();

  as5600.begin(4);
  as5600.setDirection(AS5600_CLOCK_WISE);

  pinMode(encoderPinA, INPUT_PULLUP);
  pinMode(encoderPinB, INPUT_PULLUP);

```

```
  attachInterrupt(digitalPinToInterrupt(encoderPinA), readTheta1, CHANGE);

}

void loop() {

  // Sampling every 5 ms
  unsigned long currentTime = millis();
  if (currentTime - lastTime >= interval) {
    lastTime = currentTime;

    readTheta2();

    Serial.print("Tempo (ms):,");
    Serial.print(lastTime);
    Serial.print(",Theta1 (deg):,");
    Serial.print(currentTheta1);
    Serial.print(",Theta2 (deg):,");
    Serial.println(currentTheta2);
  }

}

// Read function for Pan angle
void readTheta1(){

  int stateA = digitalRead(encoderPinA);
  int stateB = digitalRead(encoderPinB);

  if (stateA == stateB) {
    currentTheta1 += degreesPerTick;
  } else {
    currentTheta1 -= degreesPerTick;
  }
}

// Read function for Tilt angle
void readTheta2(){

  currentTheta2 = as5600.readAngle();
  currentTheta2 = (currentTheta2 * degreesPerTickMagnetic+offset);

 while (currentTheta2 < 0) {
        currentTheta2 = currentTheta2 + 360.0;
    }
    while (currentTheta2> 360) {
        currentTheta2 = currentTheta2 - 360.0;
    }
  currentTheta2 = -(currentTheta2-245.0);
}
```

Code 2: Functions related to IMU sensor.

```
/*
 * mpu9265_fun.c
 *
 * Contains functions to read data from IMU MPU92/65 and compute angles
```

```
 *
 *  Created on: 23 giu 2023
 *      Author: rosso
 */
#include "i2c.h"
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <math.h>
#include <mpu9265.h>
#include <mpu9265_fun.h>

float OffsetGyro[3] = {-1.96690583f, 0.816289783f, -1.07923603f}; // offset for drift error of gyroscope

void MPU9265_CheckStatus(uint8_t *id, HAL_StatusTypeDef *status){
     // reading ID device
    *status = HAL_I2C_Mem_Read(&hi2c1,MPU9265_I2C_ADDR<<1,MPU9265_WHO_AM_I,1,id,1,1000);

    if (*status != HAL_OK){
        HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET); // turn on the led
    } else if(*status == HAL_OK){
        HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET); // turn off the led
    }
}

void MPU9265_Init(void)
{
      // Turn on the IMU
      HAL_GPIO_WritePin(OUT_VDD_GPIO_Port, OUT_VDD_Pin, GPIO_PIN_SET);
      HAL_Delay(100);

      // Reset IMU
      uint8_t reset = 0x00;
      HAL_I2C_Mem_Write(&hi2c1, MPU9265_I2C_ADDR << 1, MPU9265_PWR_MGMT_1, 1, &reset, 1, 100);
      HAL_Delay(100);

      // Exit the sleep mode and select internal clock
      uint8_t pwr_mgmt_1 = 0x00; // CLKSEL = 000, SLEEP = 0
      HAL_I2C_Mem_Write(&hi2c1, MPU9265_I2C_ADDR << 1, MPU9265_PWR_MGMT_1, 1, &pwr_mgmt_1, 1, 100);
      HAL_Delay(100);

      // Enable Gyroscope and Accelerometer
      uint8_t pwr_mgmt_2 = 0x00; // Enable all sensors
      HAL_I2C_Mem_Write(&hi2c1, MPU9265_I2C_ADDR << 1, MPU9265_PWR_MGMT_2, 1, &pwr_mgmt_2, 1, 100);
      HAL_Delay(100);

      // Set Full Scale Range of Gyroscope (500dps)
      uint8_t gyro_config = 0x08; // FS_SEL = 01
      HAL_I2C_Mem_Write(&hi2c1, MPU9265_I2C_ADDR << 1, MPU9265_GYRO_CONFIG,
```

```
            1, &gyro_config, 1, 100);
54      HAL_Delay(100);
55
56      // Set Full Scale Range of Accelerometer (2g)
57      uint8_t accel_config = 0x00; // AFS_SEL = 00
58      HAL_I2C_Mem_Write(&hi2c1, MPU9265_I2C_ADDR << 1, MPU9265_ACCEL_CONFIG
            , 1, &accel_config, 1, 100);
59      HAL_Delay(100);
60
61      // Set LPF for Gyroscope and Accelerometer (44 Hz)
62      uint8_t config = 0x03; // DLPF_CFG = 011
63      HAL_I2C_Mem_Write(&hi2c1, MPU9265_I2C_ADDR << 1, MPU9265_CONFIG, 1, &
            config, 1, 100);
64      HAL_Delay(100);
65
66      // Set Sample Read Register (to read data at 200Hz)
67      uint8_t smplrt_div = 0x04; //0x02 (400Hz)
68      HAL_I2C_Mem_Write(&hi2c1, MPU9265_I2C_ADDR << 1, MPU9265_SMPLRT_DIV,
            1, &smplrt_div, 1, 100);
69      HAL_Delay(100);
70  }
71
72  void MPU9265_CalibrateGyro(float *gyroOffset) {
73      int numSamples = 1000;  // Number of samples to read
74      float temp[3] = {0.0f, 0.0f, 0.0f};
75
76      for (int i = 0; i < numSamples; ++i) {
77          float gyroData[3];
78          MPU9265_ReadGyro(gyroData); // read raw data
79          temp[0] += gyroData[0];
80          temp[1] += gyroData[1];
81          temp[2] += gyroData[2];
82
83          HAL_Delay(5); // wait 5 ms between samples
84      }
85
86      // mean of the read values and compute offset
87      gyroOffset[0] = temp[0] / numSamples;
88      gyroOffset[1] = temp[1] / numSamples;
89      gyroOffset[2] = temp[2] / numSamples;
90  }
91
92  void MPU9265_ReadAccel(float* accelData)
93  {
94    uint8_t rawData[6];
95    int16_t accelRaw[3];
96
97    HAL_I2C_Mem_Read(&hi2c1, MPU9265_I2C_ADDR<<1, MPU9265_ACCEL_XOUT_H,
          I2C_MEMADD_SIZE_8BIT, rawData, 6, HAL_MAX_DELAY);
98
99    accelRaw[0] = (int16_t)((rawData[0] << 8) | rawData[1]);
100   accelRaw[1] = (int16_t)((rawData[2] << 8) | rawData[3]);
101   accelRaw[2] = (int16_t)((rawData[4] << 8) | rawData[5]);
102
103   // Conversion from raw to acceleration values
104   accelData[0] = (float)accelRaw[0] / 16384.0f; // Sensitivity scale factor
```

```
    :  2g  range
  accelData[1] = (float)accelRaw[1] / 16384.0f;
  accelData[2] = (float)accelRaw[2] / 16384.0f;
}

void MPU9265_ReadGyro(float* gyroData)
{
  uint8_t rawData[6];
  int16_t gyroRaw[3];

  HAL_I2C_Mem_Read(&hi2c1, MPU9265_I2C_ADDR<<1, MPU9265_GYRO_XOUT_H,
      I2C_MEMADD_SIZE_8BIT, rawData, 6, HAL_MAX_DELAY);

  gyroRaw[0] = (int16_t)((rawData[0] << 8) | rawData[1]);
  gyroRaw[1] = (int16_t)((rawData[2] << 8) | rawData[3]);
  gyroRaw[2] = (int16_t)((rawData[4] << 8) | rawData[5]);

  // Conversion from raw to acceleration values
  gyroData[0] = (float)gyroRaw[0] / 65.5f; // Sensitivity scale factor:
       500dps  range
  gyroData[1] = (float)gyroRaw[1] / 65.5f;
  gyroData[2] = (float)gyroRaw[2] / 65.5f;
}

void MPU9265_ReadAccelGyro(float* accelData, float* gyroData)
{
    uint8_t rawData[14];
    int16_t accelRaw[3], gyroRaw[3];

    HAL_I2C_Mem_Read(&hi2c1, MPU9265_I2C_ADDR<<1, MPU9265_ACCEL_XOUT_H,
        I2C_MEMADD_SIZE_8BIT, rawData, 14, HAL_MAX_DELAY);

    accelRaw[0] = (int16_t)((rawData[0] << 8) | rawData[1]);
    accelRaw[1] = (int16_t)((rawData[2] << 8) | rawData[3]);
    accelRaw[2] = (int16_t)((rawData[4] << 8) | rawData[5]);

    gyroRaw[0] = (int16_t)((rawData[8] << 8) | rawData[9]);
    gyroRaw[1] = (int16_t)((rawData[10] << 8) | rawData[11]);
    gyroRaw[2] = (int16_t)((rawData[12] << 8) | rawData[13]);

    // Accel Data
    accelData[0] = (float)accelRaw[0] / 16384.0f; // Sensitivity scale
        factor:  2g  range
    accelData[1] = (float)accelRaw[1] / 16384.0f;
    accelData[2] = (float)accelRaw[2] / 16384.0f;

    // Gyro Data
    gyroData[0] = (float)gyroRaw[0] / 65.5f - OffsetGyro[0]; // Sensitivity
         scale factor:  500dps  range
    gyroData[1] = (float)gyroRaw[1] / 65.5f - OffsetGyro[1];
    gyroData[2] = (float)gyroRaw[2] / 65.5f - OffsetGyro[2];
}

/* Function to compute roll and pitch based on accelerometer data */
void MPU9265_ComputeRollPitch(float *accelData, float *roll, float *pitch){
      uint8_t rawData[6];
```

```
155         int16_t accelRaw[3];
156 
157         HAL_I2C_Mem_Read(&hi2c1, MPU9265_I2C_ADDR<<1, MPU9265_ACCEL_XOUT_H,
                I2C_MEMADD_SIZE_8BIT, rawData, 6, HAL_MAX_DELAY);
158 
159         accelRaw[0] = (int16_t)((rawData[0] << 8) | rawData[1]);
160         accelRaw[1] = (int16_t)((rawData[2] << 8) | rawData[3]);
161         accelRaw[2] = (int16_t)((rawData[4] << 8) | rawData[5]);
162 
163         // Conversion from raw to acceleration values
164         accelData[0] = (float)accelRaw[0] / 16384.0f; // Sensitivity scale
                factor:  2g  range
165         accelData[1] = (float)accelRaw[1] / 16384.0f;
166         accelData[2] = (float)accelRaw[2] / 16384.0f;
167 
168     *roll = atan2(accelData[1], sqrt(pow(accelData[0], 2) + pow(accelData
            [2], 2))) * 180.0f / M_PI; // roll in degrees
169     *pitch = atan2(-accelData[0], sqrt(pow(accelData[1], 2) + pow(accelData
            [2], 2))) * 180.0f / M_PI; // pitch in degrees
170 }
171 
172 
173 /* Compute Roll, Pitch and Yaw with gyroscope */
174 void MPU9265_ComputeRollPitchYaw(float *gyroData, float dt, float *roll,
        float *pitch, float *yaw){
175     uint8_t rawData[6];
176     int16_t gyroRaw[3];
177 
178     HAL_I2C_Mem_Read(&hi2c1, MPU9265_I2C_ADDR<<1, MPU9265_GYRO_XOUT_H,
            I2C_MEMADD_SIZE_8BIT, rawData, 6, HAL_MAX_DELAY);
179 
180     gyroRaw[0] = (int16_t)((rawData[0] << 8) | rawData[1]);
181     gyroRaw[1] = (int16_t)((rawData[2] << 8) | rawData[3]);
182     gyroRaw[2] = (int16_t)((rawData[4] << 8) | rawData[5]);
183 
184     // Conversion from raw to acceleration values
185     gyroData[0] = (float)gyroRaw[0] / 65.5f; // Sensitivity scale factor:
              500dps  range
186     gyroData[1] = (float)gyroRaw[1] / 65.5f;
187     gyroData[2] = (float)gyroRaw[2] / 65.5f;
188 
189     // Updates roll, pitch and yaw
190     *roll += gyroData[0]  * dt;
191     *pitch +=  gyroData[1] * dt;
192     *yaw += gyroData[2] * dt;
193 
194     // Keep roll angle in range [-180, 180]
195     if (*roll > 180.0f) {
196         *roll -= 360.0f;
197     } else if (*roll < -180.0f) {
198         *roll += 360.0f;
199     }
200     // Keep pitch angle in range [-180, 180]
201     if (*pitch > 180.0f) {
202         *pitch -= 360.0f;
203     } else if (*pitch < -180.0f) {
```

```
        *pitch += 360.0f;
    }
    // Keep yaw angle in range [-180, 180]
    if (*yaw > 180.0f) {
        *yaw -= 360.0f;
    } else if (*yaw < -180.0f) {
        *yaw += 360.0f;
    }
}
```

Code 3: Motors' initializing functions.

```
/*
 * motors.c
 *
 * Contains functions to initialize motors and set direction and speed
 *
 *  Created on: Nov 11, 2024
 *      Author: rosso
 */
#include "tim.h"
#include "gpio.h"
#include "motors.h"

void Motor1_Init(float ccr_value){
    HAL_TIM_PWM_Start(&htim2,TIM_CHANNEL_1);

    TIM2 -> CCR1 = ccr_value;

    HAL_GPIO_WritePin(Mot1_Dir1_GPIO_Port, Mot1_Dir1_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(Mot1_Dir2_GPIO_Port, Mot1_Dir2_Pin, GPIO_PIN_RESET);
    HAL_Delay(2500);

    TIM2 -> CCR1 = 0; // stop motor
    HAL_Delay(1000);

}
void Motor2_Init(float ccr_value){
    HAL_TIM_PWM_Start(&htim2,TIM_CHANNEL_2);

    TIM2 -> CCR2 = ccr_value;

    HAL_GPIO_WritePin(Mot2_Dir1_GPIO_Port, Mot2_Dir1_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(Mot2_Dir2_GPIO_Port, Mot2_Dir2_Pin, GPIO_PIN_SET);
    HAL_Delay(2500);

    TIM2 -> CCR2 = 0; // stop motor
    HAL_Delay(1000);
}
```

Code 4: Functions related to the pan-tilt system and its control.

```
/*
 * control_system.c
 *
 * PAN-TILT SYSTEM
 *
```

```
 * Contains:
 *      Function to read pan and tilt angles, based on IMU data
 *      Inverse kinematics function of the pan-tilt system
 *      Control functions for the two motors
 *      Functions to send data via SWD
 *
 *
 *  Created on: Nov 27, 2024
 *      Author: rosso
 */
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <math.h>
#include "mpu9265.h"
#include "mpu9265_fun.h"
#include "control_system.h"
#include "motors.h"

/* Compute Pan and Tilt angles in degrees starting from IMU data */
void ReadPanTilt(float *PanAngle, float *TiltAngle, float *PanVel, float *TiltVel){
    float dt = 0.005; // dt TIM6 (f = 200 Hz)
    /* Read IMU Data */
    float accelData[3];
    float gyroData[3];
    static float tilt_prev = 0.0f;
    static float pan_prev = 0.0f;

    MPU9265_ReadAccelGyro(accelData, gyroData);

    /* Compute Pan Angle (Theta1) == Pitch IMU (deg) */
    *PanAngle = pan_prev + gyroData[1] * dt;
    pan_prev = *PanAngle;
    /* Compute Tilt Angle (Theta2) == Roll IMU (deg) */
    float tilt_acc = -(atan2(accelData[1], sqrtf(accelData[0] * accelData[0] + accelData[2] * accelData[2])) * 180.0f / M_PI - OFFSET_THETA2);
    float tilt_gyro = tilt_prev - gyroData[0] * dt;
    *TiltAngle = ALPHA * tilt_gyro + (1.0f - ALPHA) * tilt_acc;  // complementary filter
    tilt_prev = *TiltAngle; // updates tilt_prev
    /* Read Pan Velocity (dps) */
    *PanVel = gyroData[1];
    /* Read Tilt Velocity (dps) */
    *TiltVel = gyroData[0];
}

/* Compute Pan and Tilt (deg) starting from a point (x,y,z) in space */
void InverseKinematics(float targetPoint[3], float *PanAngle, float *TiltAngle){
    float l1 = 106.0f; // link 1 (mm)
    float l2 = 15.9f; // link 2 (mm)

    float x = targetPoint[0];
    float y = targetPoint[1];
```

```
57      float z = targetPoint[2];
58      /* Non reachable points:
59       * z < l1
60       * ************************/
61
62      if (z < l1){
63          HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin); // toggle led2
64      } else {
65          if (x<0 && y<0){
66              *PanAngle = atan(y/x) * 180.0f / M_PI;
67              *TiltAngle = 180 - atan((z - l1) / (sqrt(pow(x,2) + pow(y,2))))
                    * 180.f / M_PI;
68          } else if (x<0 && y>=0){
69              *PanAngle = 180 + atan(y/x) * 180.0f / M_PI;
70              *TiltAngle = atan((z - l1) /( sqrt(pow(x,2) + pow(y,2)))) *
                    180.f / M_PI;
71          } else if (x>0 && y<0){
72              *PanAngle = 180 + atan(y/x) * 180.0f / M_PI;
73              *TiltAngle = 180.0 - atan((z - l1) / (sqrt(pow(x,2) + pow(y,2))
                    )) * 180.f / M_PI;
74          } else if (x>0 && y>=0){
75              *PanAngle = atan(y/x) * 180.0f / M_PI;
76              *TiltAngle = atan((z - l1) / (sqrt(pow(x,2) + pow(y,2)))) *
                    180.f / M_PI;
77          } else if (x == 0 && y>0){
78              *PanAngle = 90.0;
79              *TiltAngle = atan((z - l1) / (sqrt(pow(x,2) + pow(y,2)))) *
                    180.f / M_PI;
80          } else if (x == 0 && y<0){
81              *PanAngle = 90.0;
82              *TiltAngle = 180 - atan((z - l1) / (sqrt(pow(x,2) + pow(y,2))))
                    * 180.f / M_PI;
83          } else if (x == 0 && y == 0){
84              *PanAngle = 90.0f;
85              *TiltAngle = 90.0f;
86          }
87      }
88      if (*TiltAngle > 127){
89          *TiltAngle = 127; // physical limit of Tilt Angle
90      }
91  }
92
93  /* Initialize function for PID_Controller */
94  void PID_Init(PID_Controller *pid, float Kp, float Ki, float Kd, float
        outputMin, float outputMax){
95      pid->Kp = Kp;
96      pid->Ki = Ki;
97      pid->Kd = Kd;
98      pid->target = 0.0f;
99      pid->integral = 0.0f;
100     pid->derivative = 0.0f;
101     pid->prevError = 0.0f;
102     pid->dt = 0.005f; // dt of TIM6
103     pid->outputMin = outputMin; //250 //200
104     pid->outputMax = outputMax; //800 //900
105 }
```

```
106
107 void Motor1_Control(PID_Controller *position_pid, PID_Controller *
        velocity_pid, float feedbackPosition, float feedbackVelocity, float
        targetPosition, ControlState *state, float *errorPosition){
108     /* Position Controller (P) */
109     *errorPosition = targetPosition - feedbackPosition;
110     float targetVelocity = (position_pid->Kp * *errorPosition); // Output
            of position controller == target velocity
111     /* Velocity Controller (PI) */
112     float errorVelocity = targetVelocity - feedbackVelocity;
113     velocity_pid->integral += errorVelocity * velocity_pid->dt; // Integral
             Term
114     float output = (velocity_pid->Kp * errorVelocity) + (velocity_pid->Ki *
             velocity_pid->integral);
115     // Limit output
116     if (output > velocity_pid->outputMax) {
117         output = velocity_pid->outputMax;
118         // Anti-windup: limit integral term when output is clamped
119         velocity_pid->integral = (output - (velocity_pid->Kp *
                errorVelocity)) / velocity_pid->Ki;
120     } else if (output < velocity_pid->outputMin) {
121         output = velocity_pid->outputMin;
122         // Anti-windup: limit integral term when output is clamped
123         velocity_pid->integral = (output - (velocity_pid->Kp *
                errorVelocity)) / velocity_pid->Ki;
124     }
125     /* Actuate Motor 1 */
126     if (*errorPosition > TOLERANCE){
127         TIM2 -> CCR1 = output;
128         HAL_GPIO_WritePin(Mot1_Dir1_GPIO_Port, Mot1_Dir1_Pin,
                GPIO_PIN_RESET);
129         HAL_GPIO_WritePin(Mot1_Dir2_GPIO_Port, Mot1_Dir2_Pin, GPIO_PIN_SET)
                ;
130         *state = PAN_CONTROL;
131     } else if (*errorPosition < - TOLERANCE) { // turn motor in the
            opposite verse
132         TIM2 -> CCR1 = output;
133         HAL_GPIO_WritePin(Mot1_Dir1_GPIO_Port, Mot1_Dir1_Pin, GPIO_PIN_SET)
                ;
134         HAL_GPIO_WritePin(Mot1_Dir2_GPIO_Port, Mot1_Dir2_Pin,
                GPIO_PIN_RESET);
135         *state = PAN_CONTROL;
136     } else if (fabsf(*errorPosition) <= TOLERANCE) {
137         TIM2 -> CCR1 = 0; //stop motor
138         *state = TILT_CONTROL;
139     }
140 }
141
142 void Motor2_Control(PID_Controller *position_pid, PID_Controller *
        velocity_pid, float feedbackPosition, float feedbackVelocity, float
        targetPosition, ControlState *state, float *errorPosition){
143     /* Position Controller (P) */
144     *errorPosition = targetPosition - feedbackPosition;
145     float targetVelocity = (position_pid->Kp * *errorPosition);  // Output
            of position controller == target velocity
146     /* Velocity Controller (PI) */
```

```
147     float errorVelocity = targetVelocity - feedbackVelocity;
148     velocity_pid->integral += errorVelocity * velocity_pid->dt; // Integral
            Term
149     float output = (velocity_pid->Kp * errorVelocity) + (velocity_pid->Ki *
            velocity_pid->integral);
150     // Limit output
151     if (output > velocity_pid->outputMax) {
152         output = velocity_pid->outputMax;
153     } else if (output < velocity_pid->outputMin) {
154         output = velocity_pid->outputMin;
155     }
156     /* Actuate Motor 2 */
157     if (*errorPosition > TOLERANCE){
158         TIM2 -> CCR2 = output;
159         HAL_GPIO_WritePin(Mot2_Dir1_GPIO_Port, Mot2_Dir1_Pin, GPIO_PIN_SET)
                ;
160         HAL_GPIO_WritePin(Mot2_Dir2_GPIO_Port, Mot2_Dir2_Pin,
                GPIO_PIN_RESET);
161         *state = TILT_CONTROL;
162     } else if (*errorPosition < - TOLERANCE){ // turn motor in the opposite
            verse
163         TIM2 -> CCR2 = output;
164         HAL_GPIO_WritePin(Mot2_Dir1_GPIO_Port, Mot2_Dir1_Pin,
                GPIO_PIN_RESET);
165         HAL_GPIO_WritePin(Mot2_Dir2_GPIO_Port, Mot2_Dir2_Pin, GPIO_PIN_SET)
                ;
166         *state = TILT_CONTROL;
167     } else if (fabsf(*errorPosition) <= TOLERANCE) {
168         TIM2 -> CCR2 = 0; //stop motor
169         *state = SEQUENCE_DONE;
170     }
171 }
172
173 void configure_swo(void) {
174     CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;   // enable tracking
175     DBGMCU->CR |= DBGMCU_CR_TRACE_IOEN;               // enable clock of SWO
176     ITM->LAR = 0xC5ACCE55;                            // unlock ITM
177     ITM->TCR = 0x0001000D;                            // enable ITM
178     ITM->TER |= 1UL;                                  // enable channel 0
179 }
180
181 void send_data_to_swv(float PanAngle, float TiltAngle) {
182     char buffer[50];
183     snprintf(buffer, sizeof(buffer), "Pan:␣%.2f,␣Tilt:␣%.2f\n", PanAngle,
            TiltAngle);
184     for (char *ptr = buffer; *ptr != '\0'; ptr++) {
185         ITM_SendChar(*ptr);
186     }
187 }
```

Code 5: Main code.

```
1 /* USER CODE BEGIN Header */
2 /**
3   ******************************************************************************
4   * @file           : main.c
5   * @brief          : Main program body
```

```
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "i2c.h"
#include "tim.h"
#include "usart.h"
#include "gpio.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mpu9265.h"
#include "mpu9265_fun.h"
#include "motors.h"
#include "control_system.h"
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/

```

```c
/* USER CODE BEGIN PV */
PID_Controller PositionController_Motor1;
PID_Controller PositionController_Motor2;
PID_Controller VelocityController_Motor1;
PID_Controller VelocityController_Motor2;

ControlState state = PAN_CONTROL; // starting state

float targetPoint[3] = {-35.0f, -20.0f, 160.0f};     // xyz target point (in mm)
float panTarget, tiltTarget;                         // computed value of target angles (in deg)
float panAngle, tiltAngle;                           // current value of angles (in deg)
float panVelocity, tiltVelocity;                     // current value of velocity (in dps)

float errorPosition_Mot1, errorPosition_Mot2;        // position error (in mm)

/* USER CODE END PV */

/* Private function prototypes
    -----------------------------------------------*/
void SystemClock_Config(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code
    ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration
      ---------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

```

```
  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_I2C1_Init();
  MX_TIM2_Init();
  MX_TIM6_Init();
  /* USER CODE BEGIN 2 */
  /* Initialize IMU */
  MPU9265_Init();
  /* Initialize motors */
  Motor1_Init(360); // ccr = 360 -> duty_cycle = 0.08 and rotate in dir1
  Motor2_Init(540); // ccr = 540 -> duty_cycle = 0.12 and rotate in dir2 (
      cam backwards)
  /* Compute Inverse Kinematics */
  InverseKinematics(targetPoint, &panTarget, &tiltTarget);
  /* Initialize controllers */
  PID_Init(&PositionController_Motor1, 18.0, 0.0, 0.0, -INFINITY, INFINITY)
      ; // no limit for output
  PID_Init(&VelocityController_Motor1, 5.0, 4.0, 0.0, 0, 675);
  PID_Init(&PositionController_Motor2, 15.0, 0.0, 0.0, -INFINITY, INFINITY)
      ; // no limit for output
  PID_Init(&VelocityController_Motor2, 7.5, 3.5, 0.0, 0, 540);
  /* Configure SWO to transfer position data */
  configure_swo();
  /* Starting TIM6 Interrupt */
  HAL_TIM_Base_Start_IT(&htim6);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
    while (1)
  {
        /* Check IMU Sensor */
        //MPU9265_CheckStatus(&id, &status);

        /* Gyroscope Calibration */
        //MPU9265_CalibrateGyro(&gyroOffset);
        //break;

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
```

```
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 4;
  RCC_OscInitStruct.PLL.PLLN = 180;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Activate the Over-Drive mode
  */
  if (HAL_PWREx_EnableOverDrive() != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
  {
    Error_Handler();
  }
}

/* USER CODE BEGIN 4 */

/* Control Action each dt_TIM6 = 5 ms */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    if (htim->Instance == TIM6){
        ReadPanTilt(&panAngle, &tiltAngle, &panVelocity, &tiltVelocity);
```

```
211         send_data_to_swv(panAngle, tiltAngle);
212
213         __disable_irq();
214               switch (state){
215               case PAN_CONTROL:
216                 Motor1_Control(&PositionController_Motor1, &
                        VelocityController_Motor1, panAngle, panVelocity,
                        panTarget, &state, &errorPosition_Mot1);
217                 break;
218               case TILT_CONTROL:
219                 Motor2_Control(&PositionController_Motor2, &
                        VelocityController_Motor2, tiltAngle, tiltVelocity,
                        tiltTarget, &state, &errorPosition_Mot2);
220                 break;
221               case SEQUENCE_DONE:
222                 break;
223               }
224          __enable_irq();
225
226     }
227 }
228
229 /* USER CODE END 4 */
230
231 /**
232   * @brief  This function is executed in case of error occurrence.
233   * @retval None
234   */
235 void Error_Handler(void)
236 {
237   /* USER CODE BEGIN Error_Handler_Debug */
238   /* User can add his own implementation to report the HAL error return
          state */
239   __disable_irq();
240   while (1)
241   {
242   }
243   /* USER CODE END Error_Handler_Debug */
244 }
245
246 #ifdef  USE_FULL_ASSERT
247 /**
248   * @brief  Reports the name of the source file and the source line number
249   *         where the assert_param error has occurred.
250   * @param  file: pointer to the source file name
251   * @param  line: assert_param error line source number
252   * @retval None
253   */
254 void assert_failed(uint8_t *file, uint32_t line)
255 {
256   /* USER CODE BEGIN 6 */
257   /* User can add his own implementation to report the file name and line
          number,
258      ex: printf("Wrong parameters value: file %s on line %d\r\n", file,
          line) */
259   /* USER CODE END 6 */
```

```
260 }
261 #endif /* USE_FULL_ASSERT */
```