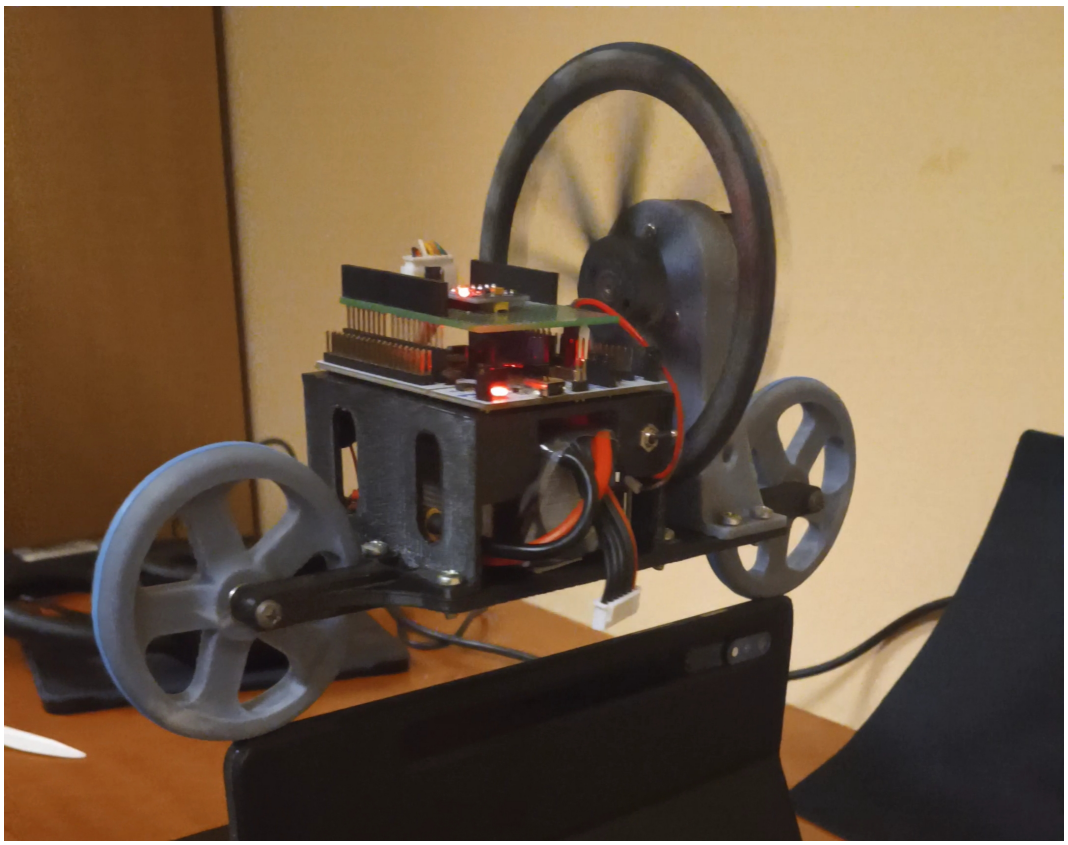
SELF-BALANCING BIKE

Embedded Control Course - A.Y. 2023/2024



Submitted by:
Group STmVZ

Group Members:
STASI Francesco, TONTI Davide, VATINNO Simona, ZITO Giovanni

Advisor: Prof. Eng. PhD. DE CICCO Luca

Contents

1 Introduction

This project involves the development of a self-balancing bicycle designed to remain stable in a stationary position on a flat surface. The balance is autonomously maintained through a reaction wheel powered by a motor, whose controlled movement counteracts the bicycle's tilt. For small tilt angles, the system can be approximated as an inverted pendulum.

The structure was created using 3D printing and then assembled, following a preliminary modeling phase in SolidWorks.

The control system relies on precise timing: A main timer manages a callback triggered at defined time intervals, while a second timer regulates the generation of the PWM signal needed to control the motor. An additional timer has been set up for reading the encoder data. As for the data from the IMU, it is acquired via the I2C protocol.

The control model is structured with two loops. The inner loop uses encoder data to implement closed-loop speed control, while the outer loop relies on IMU readings to adjust the position.

To achieve an accurate angle reading, sensor data fusion has been implemented through a filter that combines information from an MPU6050 IMU, equipped with an integrated accelerometer and gyroscope.

2 Selected Hardware

This chapter lists and briefly describes the hardware used for our scope.

2.1 STM32 NUCLEO-F446RE Overview

The STM32 NUCLEO-F446RE (fig.2.1) is an advanced microcontroller from the STM32 family, built around an ARM Cortex-M4 32-bit core. Operating at frequencies of up to 180 MHz and equipped with a floating-point unit (FPU), it is designed to handle applications requiring high computational performance. With extended support for Digital Signal Processing (DSP) instructions, this microcontroller is ideal for tasks such as motor control, signal processing, and other advanced embedded applications.

It features 512 KB of flash memory for stable program storage and 128 KB of SRAM for dynamic data management during runtime. Its high-performance clock system ensures precision and efficiency even under demanding workloads. The configurable GPIO ports allow seamless interfacing with external devices such as sensors and actuators, making it highly versatile for various applications.

The STM32 NUCLEO-F446RE integrates a comprehensive set of peripherals, including 12-bit ADC and DAC for analog signal management, advanced timers for precise motor control and PWM generation, and communication interfaces such as I^2C, SPI, USART, CAN, and USB 2.0 Full-Speed. Additionally, the DMA (Direct Memory Access) controller enables direct data transfers between memory and peripherals, reducing CPU load and improving overall performance.

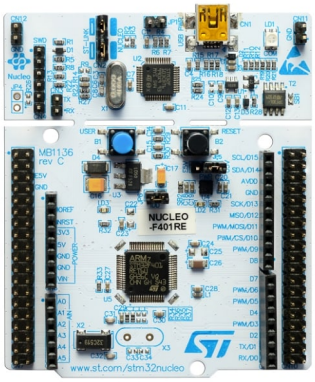


Figure 2.1: STM32 NUCLEO-F446RE

2.2 Motor

We choose the **Nidec H24** (fig.2.2), a brushless DC motor with 3 phases and 12 poles. It has a built-in electronic speed driver, that enables an easy control of the speed and direction of the motor, as well as its braking. In addition, it is quite small, compact and light, which is useful for the use case. Its contribution, along with the inertial wheel, guarantees an adequate output torque to balance the bike. It comes with a built-in rotary encoder, to decode its velocity and position.

Figure 2.2: Motor

When powered with 24V, the no-load current is 110mA and the speed is about 6100 rpm.

2.2.1 PWM Configuration

The PWM signal was generated with Timer 5 on Channel 1 in output mode. Its frequency was selected according to the specifications of the motor.

	PSC	ARR	f_{CLK}	f_{PWM}
TIM5	0	4199	84 *MHz*	20 *kHz*

Table 2.1: Configuration of timer 5

Where the frequency was obtained according to the following formula:

$$\frac{1}{f_{\text{PWM}}} = \frac{(1 + ARR) \cdot (1 + PSC)}{f_{\text{CLK}}} \tag{2.1}$$

2.3 Encoder

The motor itself is provided with a dual channel phase-tracking encoder (fig.2.3), which allowed us to discriminate the rotation direction and to determine the velocity of the shaft. It provides 100 PPR and 400 CPR.

Figure 2.3: Motor encoder

2.3.1 Encoder Configuration

The encoder was configured to work on Timer 1, where Channel 1 was associated to Channel A and Channel 2 to Channel B. We used the Quadrature Encoder Interface to generate the two square waves.

	PSC	ARR	f_{CLK}	f_{TIM}
TIM1	0	65535	84 *MHz*	1.3 *kHz*

Table 2.2: Configuration of the timer 1

The output frequency of an encoder is a function of the angular velocity of the motor shaft and of the number of divisions of the disk; it can be computed as:

$$\text{frequency} = \frac{RPM}{60} \cdot \text{Number of divisions of the disk} \tag{2.2}$$

Consequently, its maximum frequency is *10.2kHz* (period of *0.0984ms*), which a maximum velocity of 6100 RPM and 100 divisions of the disk.

2.4 Inertial Measurement Unit

The **MPU6050** (fig.2.4) sensor module is a comprehensive 6-axis motion tracking device. It integrates a 3-axis gyroscope, a 3-axis accelerometer, a Digital Motion Processor and an on-chip temperature sensor. The communication with microcontrollers is achieved via an I2C bus interface.
The gyroscope was **calibrated** by summing a series of 1000 readings and then computing their mean. Then, the result was subtracted from the raw readout, before the scaling.

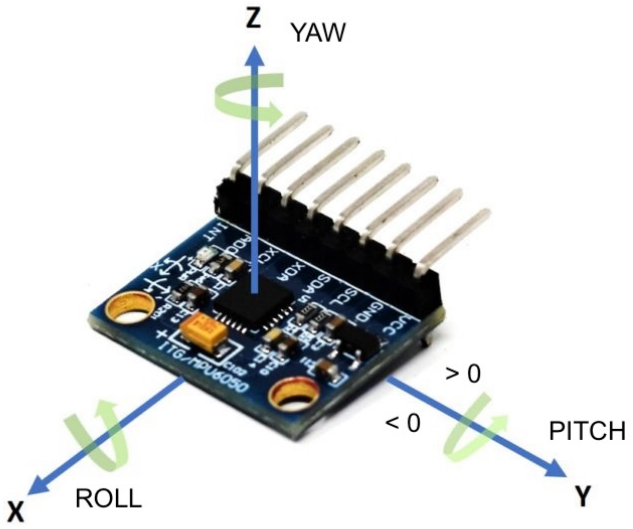


Figure 2.4: IMU MPU6050

The angles were computed using trigonometric formulas based on the accelerometer's readings:

$$\text{pitch} = \frac{180}{\pi} \cdot \text{atan2}\left(\text{accelX}, \sqrt{\text{accelY}^2 + \text{accelZ}^2}\right) \tag{2.3}$$

$$\text{roll} = \frac{180}{\pi} \cdot \text{atan2}\left(\text{accelY}, \sqrt{\text{accelX}^2 + \text{accelZ}^2}\right) \tag{2.4}$$

2.4.1 IMU Communication

We configured the I2C1 of the board in fast mode (i.e. with a frequency of 400 kHz) and with two GPIO pins associated to SDA and SCL configured as alternate function, that were connected to the sensor.

2.4.2 Sensor Fusion

Firstly, the **complementary filter** was employed to obtain the sensor fusion between accelerometer and gyroscope. More specifically, the gyroscope's readings have been high pass filtered, while the accelerometer's readouts were low pass filtered:

$$\text{currentAngle} = \alpha \cdot (\text{previousAngle} + \text{gyroAngle}) + (1 - \alpha) \cdot \text{accAngle} \tag{2.5}$$

$$\text{gyroAngle} = \text{gyroReading} \cdot T_{\text{s}} \tag{2.6}$$

where α was set to 0.98. The filter has been used to eliminate the tendency of drift of the gyroscope and to decrease the noise that comes from the accelerometer.

Then, we used the **Kalman filter** to smooth the variations and make the response faster, achieving a higher performance. As a matter of fact, the complementary filter offers a straightforward approach but involves a trade-off: you can achieve either a well-smoothed signal with notable delay and reduced amplitude or a signal with minimal delay and accurate amplitude tracking, though at the cost of considerable noise.
To address these limitations, the Kalman filter provides a more advanced alternative. It predicts the next value by leveraging knowledge of the system's model or the expected pattern of signal changes.
The following table shows the advantages of using the two filters:

Complementary Filter	**Kalman Filter**
Simplicity	Handling unreliable measurements
Speed of computation	Adaptability to system changes
Stable and reliable in real-worlds conditions	Ability to predict future system states
	Adaptability to complex dynamic models

Table 2.3: Filters advantages

The Kalman filter's predictive capabilities stem from its two-phase operation: **prediction** and **update**. During the prediction phase, the next state is estimated using the system's dynamics, as implemented in the following code:

Listing 2.1: Prediction phase

```
1 k->angle += DT_k * input; // State prediction based on input
2 k->uncertainty += PROCESS_NOISE; // Increase uncertainty due to model noise
```

Here, the angle is updated based on the input, scaled by the sampling interval `DT_k`. The uncertainty is increased to incorporate the process noise.
In the update phase, the Kalman filter refines this prediction using the new measurement, combining it with the prediction to minimize uncertainty. This is achieved through the computation of the Kalman gain and the correction of the prediction:

Listing 2.2: Update phase

```
k->kalmanGain = k->uncertainty / (k->uncertainty + MEASURE_NOISE); // Calculate Kalman gain
k->angle += k->kalmanGain * (measurement - k->angle); // Correct prediction using measurement
k->uncertainty = (1 - k->kalmanGain) * k->uncertainty; // Update uncertainty
```

The Kalman gain dynamically adjusts the weighting of the predicted state and the measurement, favoring the more reliable source. This iterative prediction process allows the Kalman filter to adapt to varying conditions, such as sensor noise levels or rapid changes in system dynamics. By continuously updating the state and considering uncertainty, it provides real-time performance improvements that are particularly valuable for applications like the sensor fusion of an IMU (MPU6050), where precise and stable data is crucial to guarantee the balancing.

2.5 Power Supply

Lithium-polymeric batteries (fig. 2.6) are rechargeable batteries that have about four times the energy of other kinds of accumulator. LiPo batteries are lighter and more flexible than other kinds of lithium-ion batteries because of their soft shells, allowing them to be used in mobile and other electronic devices. We used it along with its charger. It provides *22.2V* and *1300mAh*.

A **DC-DC buck converter** (fig.2.5) is a type of switch-mode power supply that steps down a higher DC input voltage (22.2V in our case) to a lower output voltage (i.e. 5V).

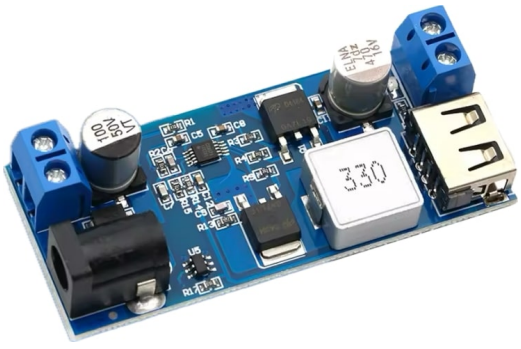


Figure 2.5: DC-DC buck converter

Figure 2.6: Lipo battery

3 Printed Parts

3.1 3D model design

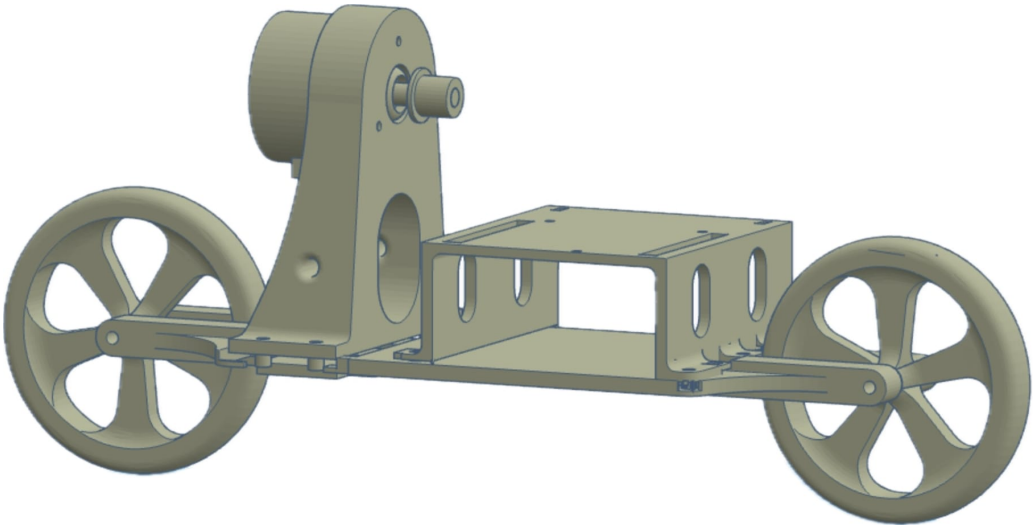


Figure 3.1: 3d model Solidworks

The 3D model (fig.3.1) was created using SolidWorks [2]. The structure had to be designed considering the weight and dimensions of the various components. Once the model was completed with all the parts, the software allowed for an accurate calculation of the structure's center of gravity.

Compared to similar projects, the use of a board like the STM32F446RE required a wider and taller overall structure. All parts was subsequently printed in PLA, ABS, or resin, and prepared for final assembly.

3.2 Motor Support

The support (fig.3.2) is meant to hold the motor in place and to group and organize the cables through a hole on its side.

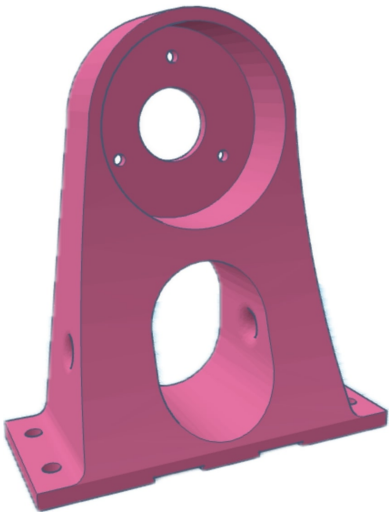


Figure 3.2: 3D printed bike

3.3 Wheels

The wheel (fig.3.3) is the load-bearing turn on which the bike will have to balance, with a bearing inside (fig.3.4) to accomodate the shaft.

Figure 3.3: Wheel

Figure 3.4: Bearing

Figure 3.5: Wheel and bearing

3.4 Frame Part

The frame (fig.3.6) it's the main weight-bearing structure, where all the components are placed.

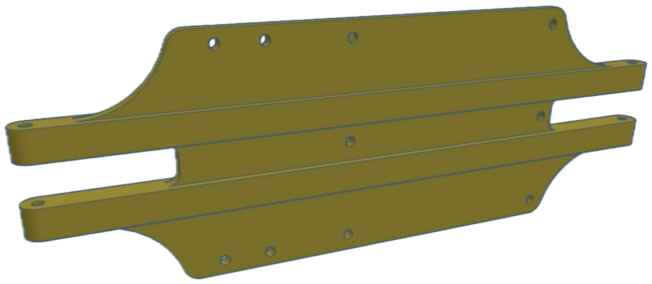


Figure 3.6: Frame part

3.5 Battery Case

A battery case (fig.3.7) supports the microcontroller and protects both the battery and the converter.

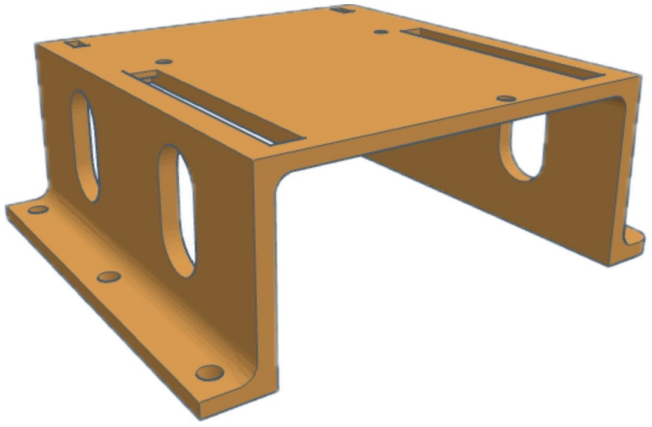


Figure 3.7: Battery case

3.6 Reaction Wheel

To ensure that the bicycle remains balanced, the reaction wheel (fig.3.8) must have sufficient momentum of inertia to counteract the torque generated by gravity acting on the bicycle. The equation for the moment of inertia of a body lying in two dimensions in the X-Y plane is given by:

$$I_Z = \sum_i m_i r_i^2 \tag{3.1}$$

Since the moment of inertia of a body is proportional to the square of the radial distance to the mass, the reaction wheel is designed so that most of its mass is concentrated at the outer edges. This increases the moment of inertia I_Z.

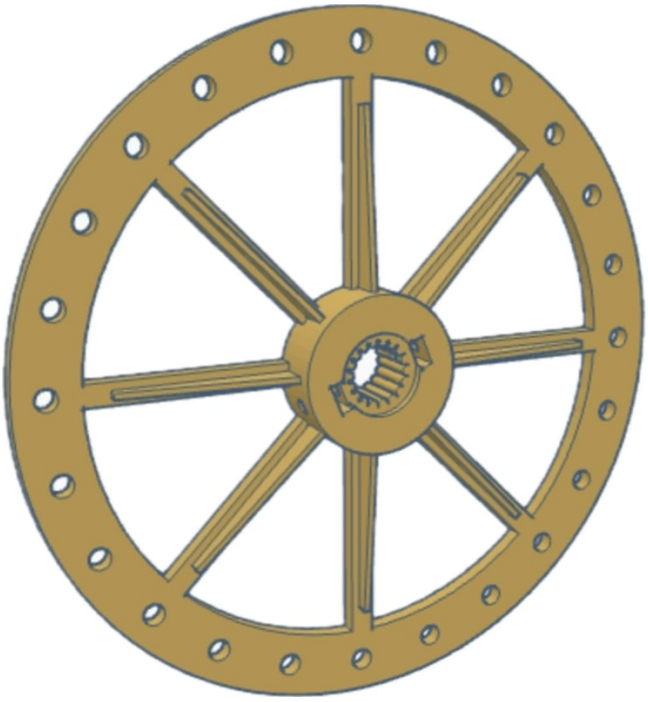


Figure 3.8: Reaction wheel

To achieve this weight distribution and allow for easy adjustments to the moment of inertia, a 3D-printed design was chosen, where the majority of the mass comes from a combination of M4 screws and nuts. This design provides flexibility in varying the moment of inertia by changing the position or number of screws and nuts.

4 Wirings

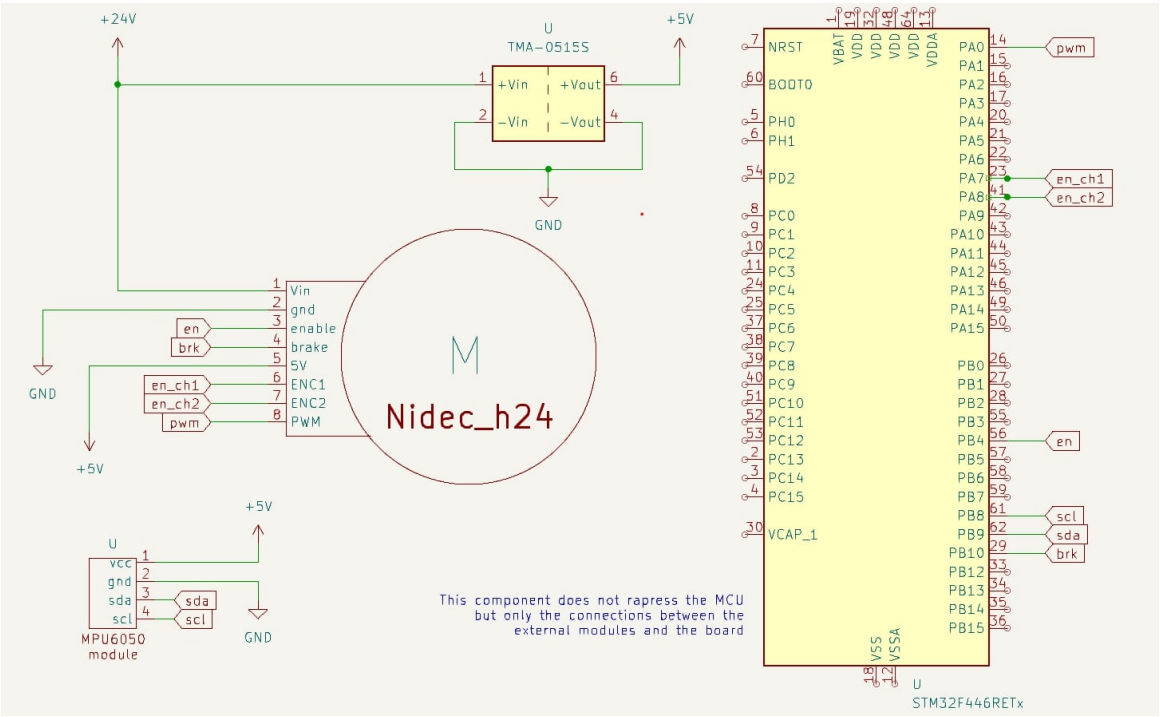


Figure 4.1: schematic

	Brake	PWM	Direction	ChA	ChB	SDA	SCL
Arduino	D5	A0	D6	D7	D8	D14	D15
Morpho	PB4	PA0	PB10	PA8	PA9	PB9	PB8

Table 4.1: Peripherals connection to the microcontroller

1	2	3	4	5	6	7	8
24V	GND	Brake	PWM	Direction	5V	ChA	ChB

Table 4.2: Motor pinout

Pin	Function
1	Motor supply voltage, 9V-24V.
2	Supply ground.
3	Brake: V_{high} = 2.0V to 5.0V = OFF, $V_{low} \leq$ 0.15V = ON (motor stop).
4	PWM = 20 kHz to 30 kHz; $V_{low} \leq$ 0.6V, V_{high} = 2.0V to 5.0V, duty cycle = 20% to 100%.
5	V_{high} = 2.0V to 5.0V or OPEN = Clockwise, $V_{low} \leq$ 0.15V = Counter-clockwise.
6	Standard = No connection. Encoder option = Logic supply, 5V ± 0.5V.
7	Standard = Open-drain tachometer, six pulses per revolution, I_C(max)=3.0mA. Encoder option = Channel A output: 90° phase tracking, 100 pulses per revolution, HIGH = 5V, LOW = 0V.
8	Standard = No connection. Encoder option = Channel B output: 90° phase tracking, 100 pulses per revolution, HIGH = 5V, LOW = 0V.

Table 4.3: Pinout and function description of the motor driver

A shield (fig.4.4) is used to solder components, ensuring more stable connections and enhancing modularity. This also makes the bicycle self-contained and streamlined, eliminating the clutter of wires and cables.

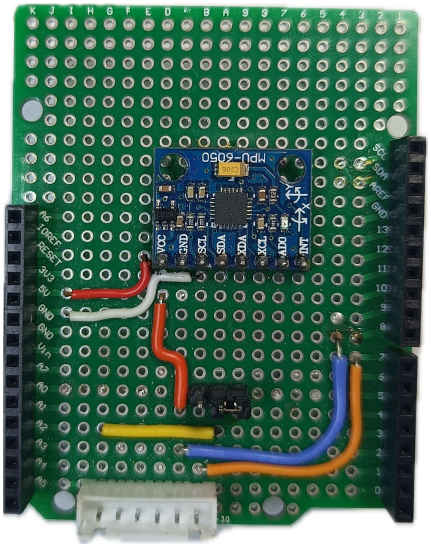


Figure 4.2: Shield: front

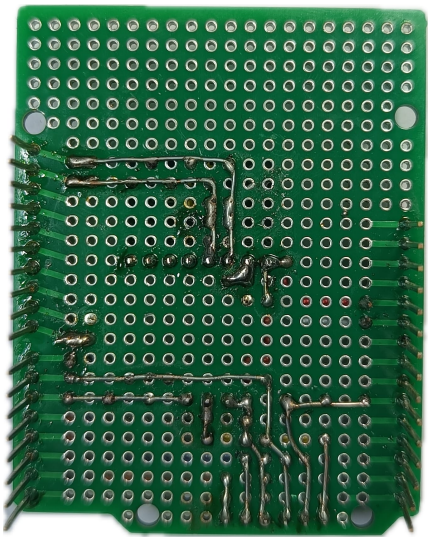


Figure 4.3: Shield: back

Figure 4.4: Shield

5 System's control

To **sample**, Timer 2 counts up so that an interrupt is generated each time its counter reaches the ARR value. The control logic is implemented in the timer's callback, which gets called by the interrupt's handler.

	PSC	ARR	f_{CLK}	f_s	T_s
TIM2	0	335999	84 *MHz*	250 *Hz*	4 *ms*

Table 5.1: Configuration of timer 2

The bicycle is an unstable system and tends to fall over very quickly. The control frequency depends on how fast the bicycle drops down and the effectiveness of the reaction wheel in correcting the fall. The control action needs to be fast relative to the system's dynamics. Lower frequencies may not respond quickly enough to disturbances, while very high frequencies could introduce noise.

Firstly, a simple position control was tested. Specifically, a single PID was used, and the IMU reading provided the feedback that closed the loop. This control proved ineffective as strong oscillations prevented the system from reaching equilibrium. In particular, the actuation variable (which generates the PWM) did not allow for direct control of the motor's speed.

Then, we decided to leverage the motor's integrated encoder by creating a cascade control system. With an inner loop handling the speed control of the motor, while an outer loop performed the position control for the plant. The innermost PID was tuned so that the controlled system could follow a specific setpoint (several tests were conducted, including the tracking of a sine-wave) (fig: 5.2).

Another PID was initially used for the outer control too, but this caused some issues. Although oscillations decreased with the introduction of a cascade control, the bike's balance could not be maintained. Moreover, it was complex to finely tune and synchronize the parameters of the two controllers.

When the reaction wheel rotates, it generates a torque, which is given by the product of the inertia of the reaction and its acceleration:

$$\tau = I \cdot \alpha \tag{5.1}$$

This acceleration generates the lateral force that moves the body, which will depend on the height of the center of mass with respect to the ground:

$$F_L = \frac{\tau}{h} \tag{5.2}$$

We concluded that the input setpoint speed, regardless of the tilt angle, should never be constant in order to generate a centripetal force that moves the bike. One technique could be to continuously increase (or decrease) the acceleration to reduce the error in the outer loop.

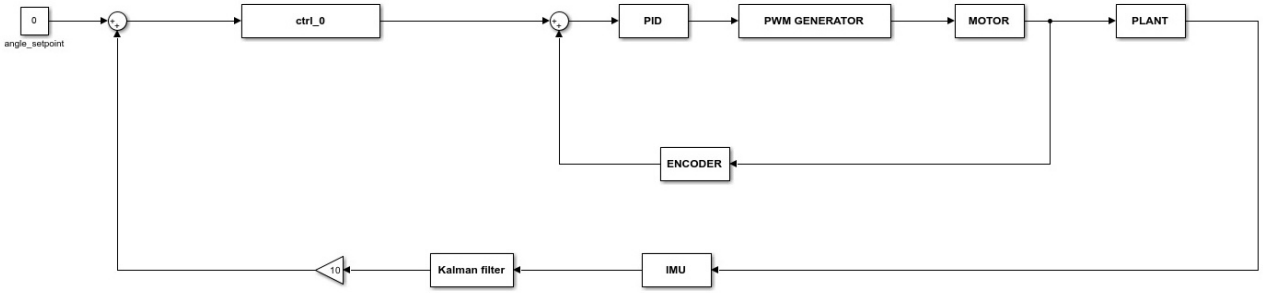


Figure 5.1: Control Loop

5.1 PID

The **PID** regulator has been chosen thanks to its capabilities to stabilize systems and to impose certain constraints:

- The **proportional** component decreases the rise time and reduces the steady-state error. However, it may cause overshoot in the response. It adjusts the proportionality based on how far the bike is tilted.
- The **integral** contribution shortens the rise time and eliminates long-term errors, but it increases the peak overshoot and prolongs the settling time.
- The **derivative** term reduces the overshoot and the settling time. This means the transient state of the system will be more dampened, controlling the rate of change in tilt.

Several tests (fig: 5.2) have been performed to finely tune the parameters and obtain a satisfying performance in terms of rapidity and stability, in the end we set up:

	K_p	K_i	K_d
PID coefficients	10	0.1	0.42

Table 5.2: PID constants after tuning

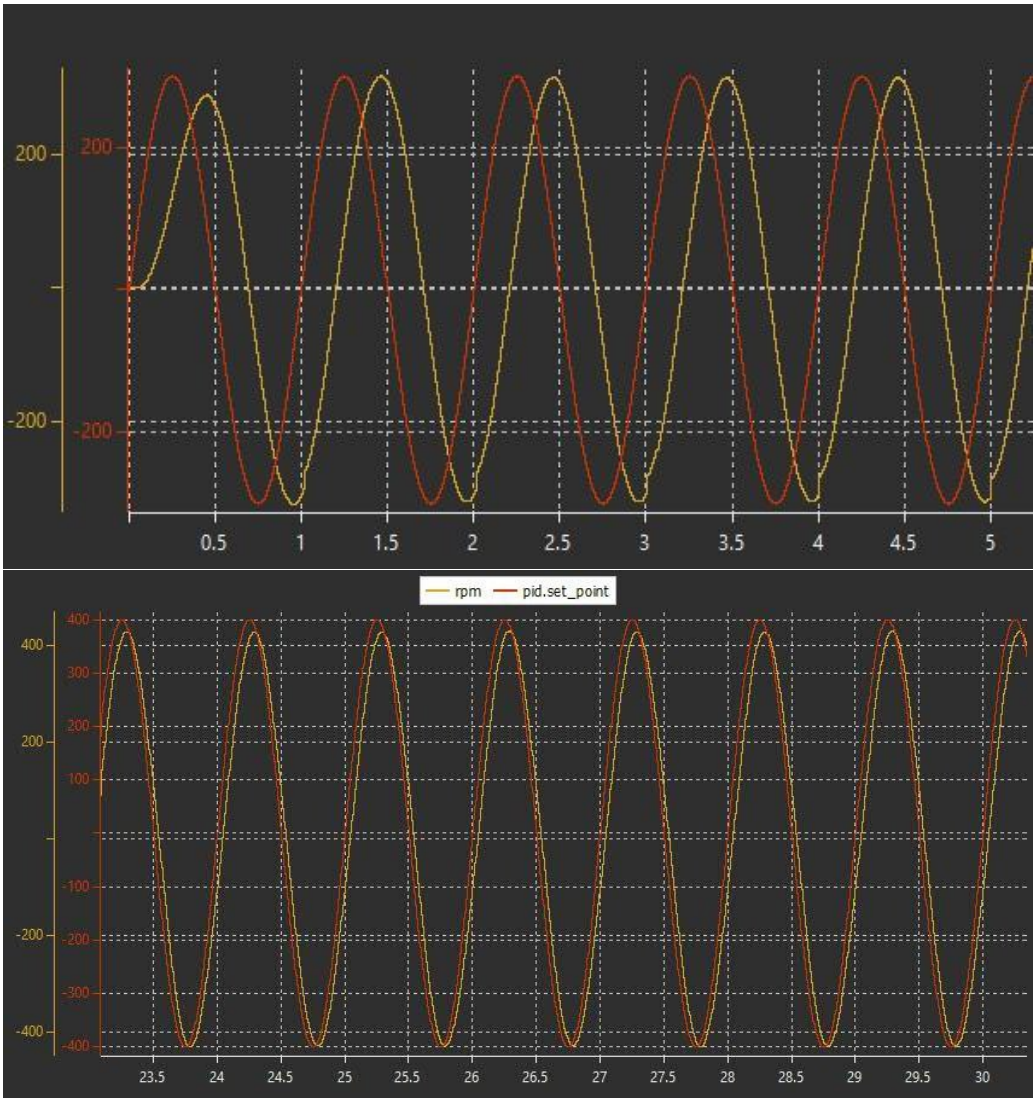


Figure 5.2: Pid test tuning

6 The most important parts of the Code

Code development and peripherals configuration are simplified with the STM32CubeIDE[3], an integrated development environment that in combination with the STM32CubeMX provides a user-friendly graphical interface for peripheral setup. Developers benefit from tools for real-time debugging and performance optimization. The software ecosystem includes HAL (Hardware Abstraction Layer) and LL (Low-Layer) libraries, along with example projects provided by STMicroelectronics, accelerating prototyping and development.

6.1 MPU6050 library

The IMU library has been developed in order to allow the return of data from the gyroscope and accelerometer; to do this, after an initial configuration phase, we just had to understand where the data was saved, create a data buffer and fill it with the raw data (using the I2C protocol and the HAL). The IMU was then calibrated by calculating the gyroscope bias, which was subtracted from the calculation of the readings.

Listing 6.1: All data reading MPU6050

```
mpu_data mpu6050_data() {
    uint8_t buffer[14];
    HAL_I2C_Mem_Read(&hi2c1, IMU_ADDR, ACCEL_XOUT_H_REG, I2C_MEMADD_SIZE_8BIT,
            buffer, 14, 100);
    mpu_data data;
    data.ax = (float) (int16_t) (buffer[0] << 8 | buffer[1]) / ACC_SCALE;
    data.ay = (float) (int16_t) (buffer[2] << 8 | buffer[3]) / ACC_SCALE;
    data.az = (float) (int16_t) (buffer[4] << 8 | buffer[5]) / ACC_SCALE;
    data.gx = (float) (int16_t) ((buffer[8] << 8 | buffer[9]) - gx_bias)
            / GYRO_SCALE;
    data.gy = (float) (int16_t) ((buffer[10] << 8 | buffer[11]) - gy_bias)
            / GYRO_SCALE;
    data.gz = (float) (int16_t) ((buffer[12] << 8 | buffer[13]) - gz_bias)
            / GYRO_SCALE;
    return data;
}
```

The code above highlights the structure used to collect the data (mpu_data) starting from the base address ACCEL_XOUT_H_REG of the MPU6050.

6.2 Nidec h24 motor library

The Nidec library allows the motor to be driven, in a clockwise or counterclockwise motion, depending on the modv input. We also specified a maximum value to limit the motor and a brake to activate or deactivate it. The timer was used to control the speed by comparing the out_ccr to the ARR value. The mapping for calculating the out_ccr is done with another specific

function within the library itself that allows the range of the input variable to be changed. The main library function is the following:

Listing 6.2: To make the motor turn based on the input

```
void nidec_h24_Move(float modv, float max_modv, uint8_t brk) {

    if (brk == 0) {
        // brk -> 0: brakes
        HAL_GPIO_WritePin(BRAKE_PORT, BRAKE_PIN, GPIO_PIN_RESET);
        TIM5->CCR1 = 0;
    }

    else {
        HAL_GPIO_WritePin(BRAKE_PORT, BRAKE_PIN, GPIO_PIN_SET);

        if (modv > 0) {
            // counter-clockwise
            HAL_GPIO_WritePin(DIRECTION_PORT, DIRECTION_PIN, GPIO_PIN_SET);
            out_ccr = map(modv, 0, max_modv, 3938, 0);
        } else {
            // clockwise
            HAL_GPIO_WritePin(DIRECTION_PORT, DIRECTION_PIN, GPIO_PIN_RESET);
            out_ccr = map(modv, 0, -max_modv, 3938, 0);
        }

        if (out_ccr > 0 && out_ccr < htim5.Instance->ARR)
            TIM5->CCR1 = out_ccr;
        else
            TIM5->CCR1 = 0;

        // Generate an update event to reload the value immediately
    }

}
```

6.3 Encoder library

The encoder is used for speed sensing which will be used for the inner PID control action.

Listing 6.3: Encoder Update

```
inline static void __encoder_update(encoder_t *e) {
    uint32_t cur_cnt;
    int32_t diff, cur_velocity;

    cur_cnt = e->tim->Instance->CNT;

    // Handle overflow and underflow
    if (__HAL_TIM_IS_TIM_COUNTING_DOWN(e->tim)) {
        if ((uint32_t)cur_cnt < (uint32_t)e->last_count) // underflow
            diff = (uint32_t)e->last_count - (uint32_t)cur_cnt;
        else
```

```
            diff = ((uint32_t)e->tim->Instance->ARR - (uint32_t)cur_cnt) + (uint32_t)e->
                last_count;
    } else {
        if (cur_cnt > e->last_count) // overflow
            diff = (uint32_t)e->last_count - (uint32_t)cur_cnt;
        else
            diff = ((uint32_t)e->tim->Instance->ARR - (uint32_t)e->last_count) + (uint32_t)
                cur_cnt;
    }

    // velocity in pulses per second
    if ((uint32_t)e->last_count == (uint32_t)cur_cnt)
        diff = 0;

    cur_velocity = (float) diff / DT_enc / (float) e->resolution;

    // Filtering velocity
    e->velocity_pps = BETA * e->velocity_pps + (1.0 - BETA) * cur_velocity;
    e->last_count = cur_cnt;
}
```

The main function of the implemented library reads the current timer count, calculates the difference from the previous count by handling overflow and underflow cases, and then calculates the rate in pulses per second.

6.4 Main control

In the main code, all the declarations of variables and structures have been placed; the functions related to peripherals, timers and so on, were initialized there too. The advantage of using timers is that we know precisely the time intervals of the various instructions as they are started within the callback function with jumps in the code if an interrupt is detected.
In our case the main callback is presented like this:

Listing 6.4: Period Elapsed Callback in the Main code

```
    if (htim->Instance == TIM2) {

        data = mpu6050_data();

        new_angle = -atan(data.ax / sqrt(data.ay * data.ay + data.az * data.az))
                * 180 / M_PI;

        kalmanUpdate(&filter, data.gy, new_angle);


        pitch_angle = filter.angle - angle_offset;

        distance = -10 * pitch_angle;
        distance_error = distance_setpoint - distance;

        if (distance_error < distance_setpoint) {
            distance_setpoint -= weight_balance * dt;
```

```
        } else {
            distance_setpoint += weight_balance * dt;
        }

        rpm = fabs(encoder_get_pps(&enc));

        if (pid_out < 0 && rpm > 700)
            distance_setpoint -= rpm_limit;

        if (pid_out > 0 && rpm > 700)
            distance_setpoint += rpm_limit;

        pid_set_setpoint(&pid, distance_setpoint);

        if (pitch_angle < -20 || pitch_angle > 20) {
            distance_setpoint = 0;
            nidec_h24_Move(0, 0, 0);
            pid_reset(&pid);
        } else {
            pid_out = pid_compute_control_action(&pid, distance);
            nidec_h24_Move(pid_out, 450, 1);
        }
    }
```

Firstly, the data are read from the IMU, then the Kalman filter gets updated. The position error with respect to the setpoint is calculated and then re-calibrated, the data from the encoder, whose frequency is much higher than the external position loop, is taken. Finally the PID is intervened to set a PWM reference for the motor.

7 Tests and Results

This section covers all the tests and results obtained during the design process. They are discussed in a qualitative manner, based on graphs from measuring instruments and the SWD Data Trace Timeline Graph tool from the STM32CubeIDE.

7.1 Complementary Filter Test

To assess the performance of the balancing algorithm, first of all we measured the tilt angle of the bike to ensure the readings were accurate.

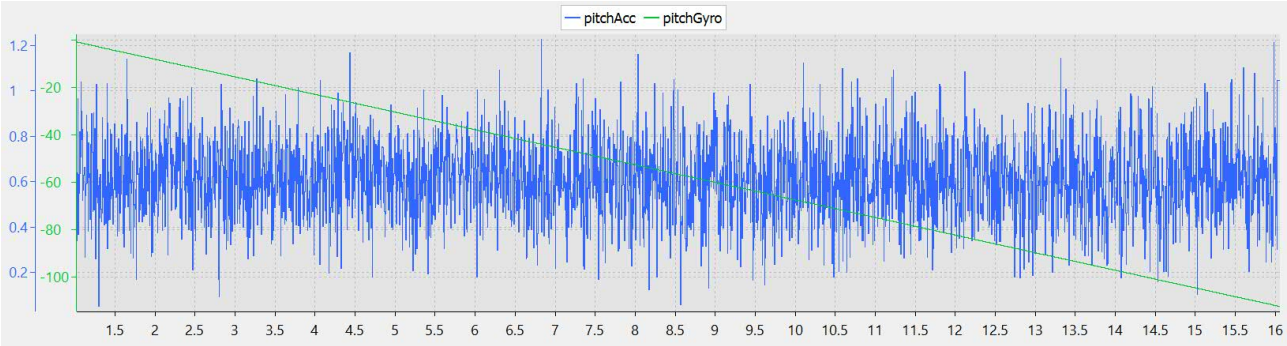


Figure 7.1: Raw pitch readings

As expected, the integration of the gyroscope contribution tends to increase; with rotation rate integration, the error on the angle gets higher, because it sums previous errors. We then processed this raw lean data (see fig.7.1) using a complementary filter to confirm the data's consistency and accuracy.

As shown in 7.2, 7.3 and 7.4, the filtered angle effectively smooths out fluctuations in the raw pitch data, removing sudden changes in the angle. This illustrates the effectiveness of our complementary filter in providing a stable readout.

Figure 7.2: Filtered pitch angle against raw reading 1

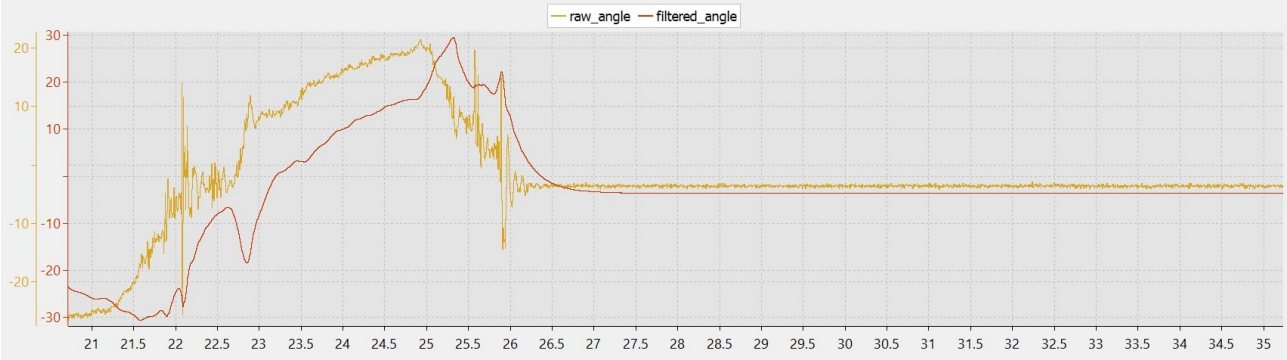


Figure 7.3: Filtered pitch angle against raw reading 2

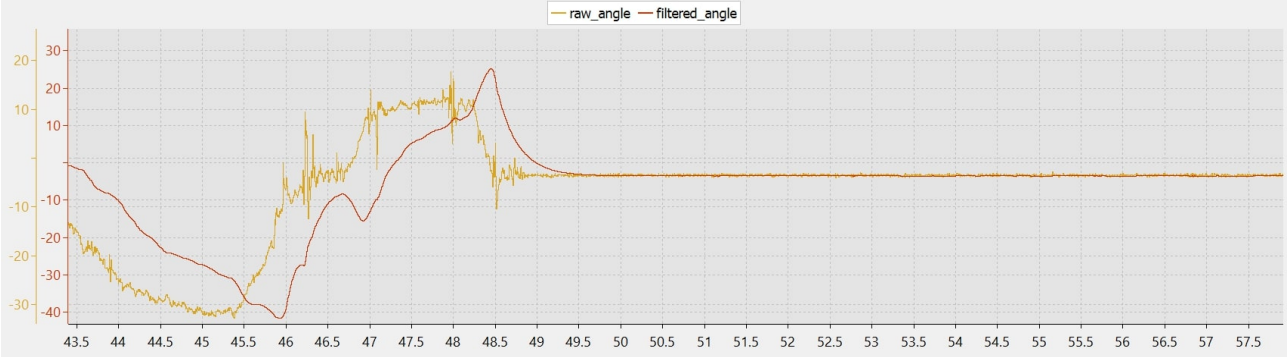


Figure 7.4: Filtered pitch angle against raw reading 3

7.2 Kalman Filter Comparison

As we discussed in the chapter 2.4.2, Kalman filter has allowed us to obtain better results and here is a brief comparison of the results obtained.

In the following figures 7.5 and 7.6, the yellow signal was the angle estimation obtaines through the Kalman filter, while the red one was derived with the complementary:

Figure 7.5: Kalman filter vs complementary filter 1

Figure 7.6: Kalman filter vs complementary filter 2

7.3 PWM Signal Verification

To verify the effective generation of the PWM signal, we utilized an oscilloscope. The measurement instrument confirmed that the signal maintains the specified frequency and pulse characteristics.

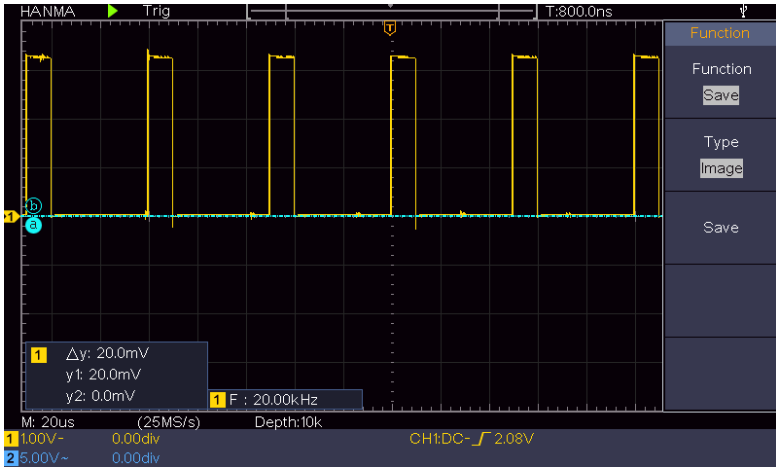


Figure 7.7: PWM signal with a 20% duty cycle

As shown in Figure 7.7, the signal exhibits a duty cycle of 20%. The voltage alternates between $3.33V$ and $0V$, consistent with expectations for this duty cycle.

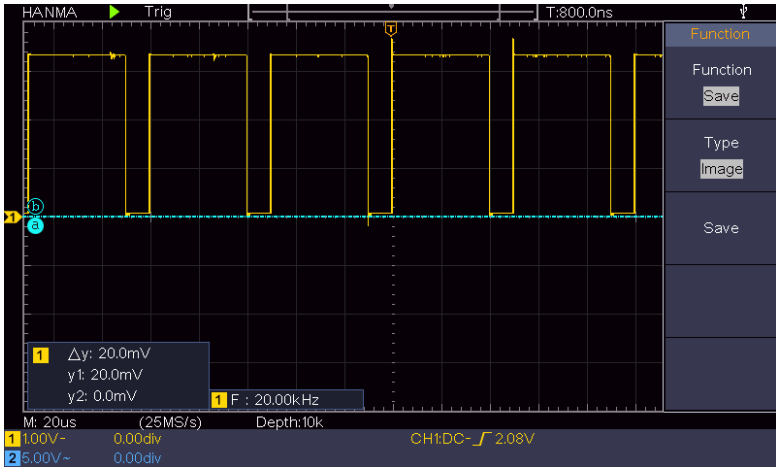


Figure 7.8: PWM signal with a 80% duty cycle

Similarly, Figure 7.8 shows the PWM signal at a 80% duty cycle. In each case, the oscilloscope measurements confirmed the expected behavior of the signal.

7.4 Encoder Signals Verification

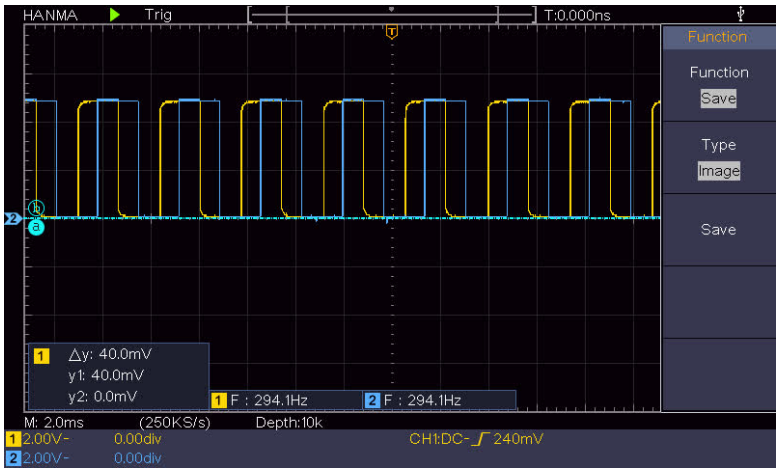


Figure 7.9: Encoder channels

Figure 7.9 displays in blue Channel A and in yellow Channel B. The signals are squared waveforms shifted of 90 degrees.

8 Conclusions and future developments

To conclude, the control system, based on two feedback loop, the inner one for speed and the outer one for position, ensures precise system management. The sensor fusion of data from an MPU6050 IMU, using a Kalman filter, provided accurate angle measurements, allowing for effective stabilization even under changing conditions.

The system demonstrated a remarkable ability to maintain balance despite external disturbances and small changes in tilt, showing the effectiveness of the implemented control. However, there could be some improvement and potential future developments to make the system even more complete.

A possible improvement to the project could involve enabling it to move through space while maintaining balance. This would require the addition of a servo motor for steering and an additional motor for propulsion. Furthermore, a remote control module would be necessary for distance control. These updates would entail modifying the structure.

It would also be interesting to test different control approaches, such as implementing a predictive control system, to make a comparison.

List of Figures

List of Tables

Bibliography

[1] *Github repository*. Last access date: 21 Nov 2024. URL: https://github.com/Phersax/Self-balancing-bike.

[2] *SolidWorks*. Last access date: 21 Nov 2024. URL: https://www.solidworks.com/.

[3] *STM32IDE*. Last access date: 21 Nov 2024. URL: https://www.st.com/en/development-tools/stm32cubeide.html.