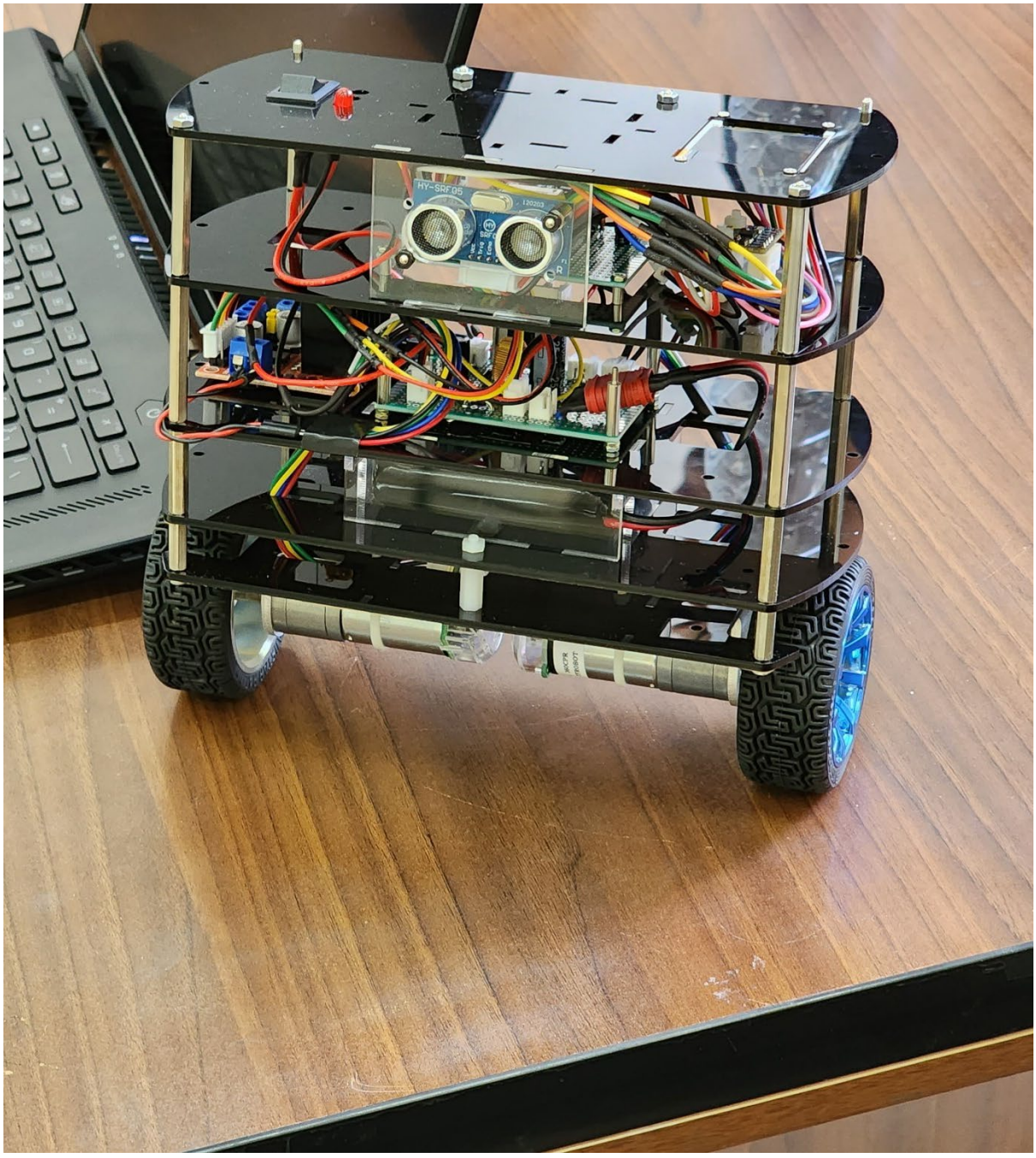
# Design and Construction of a Self-Balancing Robot

By Arian Hajizadeh, Erfan Riazati, Sorin Yousefnia



# Contents

## **1-** Introduction

Self-Balancing robots have become an exciting area of research and development in various technologies, and significant progress has been made in their construction in different countries. They are used in various industries, including construction, warehousing, medical applications, and assisting individuals with mobility impairments.

In this project, our aim is to build a simple type of balancing robot based on an inverted pendulum. The robot we are building can maintain balance, move, detect obstacles, and be remotely controlled using image processing. To achieve this, we are using the MPU-6050 inertial measurement sensor, HC-SRF05 ultrasonic distance measurement sensor, two JGA25 DC motors, L298 motor driver, three 18650 lithium-ion batteries with a current rating of 2200mA.H, and an ESP32 processor.

## **2-** Design

Based on our research and initial estimations, we made preliminary predictions about the robot and selected sensors that were proportional to the estimations and initial calculations. The initial design, shared previously, included the initial calculations and the reasons for selecting each component.

In the proposal and conceptual design section, we estimated the robot's capabilities and limitations and identified areas for improvement. After constructing and conducting a preliminary assessment, we found that the robot has achieved many of the proposed capabilities and even exceeded some expectations from the original design. The robot can maintain balance when tilted up to 20 degrees from the zero angle, despite its weight of around 1 kilogram. This indicates its ability to tolerate a payload beyond what was initially proposed. However, there are still opportunities for optimization in the secondary design.

Overall, the robot was built in accordance with the conceptual design diagram, but minor changes were made during the construction. For instance, we initially selected the L293 motor driver, but after testing the motor system, we realized, that it could not provide adequate power to the motors. As a result, we upgraded the driver.

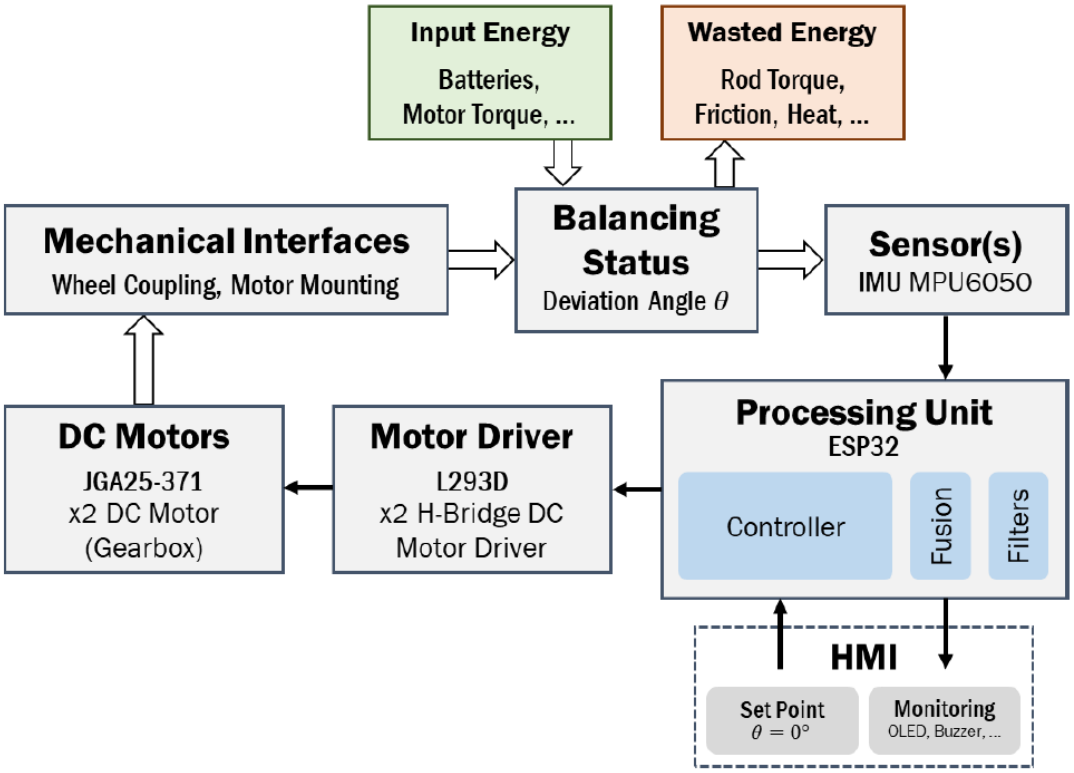



Figure2.1. Block diagram of the initial design

# 3- Simulation

## 3-1- Modeling

A physical model of a balancing robot can be accurately described by an inverted pendulum model. This model typically consists of a rigid pole attached to a frictionless joint, which is connected to a rigid cart that moves in one direction.

As shown in Figure 3.1, the system is modeled with an inverted pendulum, where the control input is a force F that moves the cart horizontally, and the system outputs are the angular position of the pendulum θ and the horizontal position of the cart x.

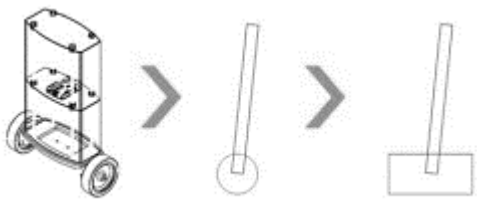


Figure3.1. Modeling the system with inverse pendulum

Using this model, we can analyze the dynamics of the system and design a control strategy to maintain balance. The inverted pendulum model is widely used in the field of robotics and control engineering due to its simplicity and effectiveness in describing the behavior of balancing systems.

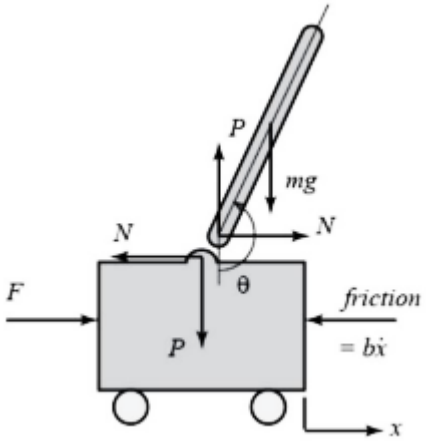



Figure3.2. Showing the parameters of the system on the model

The table of model parameters will be as follows:

Table3.1. Parameters of the system

| | | |
|---|---|---|
| $M$ | Weight of the Cart | 1000 g |
| $m$ | Weigh of the Pendulum | 200 g |
| $g$ | Gravity Acceleration | 9.8 $m/s^2$ |
| $l$ | Distance to CoM | 0.06 m |
| $b$ | Cart Friction Coefficient | 0.1 $N/m \cdot sec$ |
| $I$ | Pendulum Inertia | 0.006 $Kg \cdot m^2$ |
| $x$ | Position of the Cart | |
| $\theta$ | Pendulum Angle | |
| $F$ | Force Applied to the Cart | |

The pendulum and the cart each have one degree of freedom. We can write the differential equations for these degrees of freedom based on Newton's second law (ma = F) as follows:

$$\ddot{x} = \frac{1}{M}\sum_{cart} F_x = \frac{1}{M}(F - N - b\dot{x}) \quad (3.1)$$

$$\ddot{\theta} = \frac{1}{I}\sum_{pend} \tau = \frac{1}{I}(-Nlcos\theta * Plsin\theta) \quad (3.2)$$

$$m\ddot{x_p} = \sum_{pend} F_x = N \rightarrow N = m\ddot{x_p} \quad (3.3)$$

$$m\ddot{y_p} = \sum_{pend} F_y = P - mg \rightarrow P = m(\ddot{y_p} + g) \quad (3.4)$$

Here, M is the mass of the cart, I is the inertia moment of the pendulum, b is the coefficient of friction of the cart, N is the force exerted by the pendulum on the cart, and F is the force applied to the cart. The torque τ is the product of the force and the perpendicular distance from the pivot point to the force.

The forces N and P, which are the mutual effects between the cart and the pendulum, are obtained as follows:

$$x_p = x + lsin\theta \quad (3.5)$$

$$\dot{x}_p = \dot{x} + l\dot{\theta}cos\theta \quad (3.6)$$

$$\ddot{x}_p = \ddot{x} - l\dot{\theta}^2 sin\theta + l\ddot{\theta}cos\theta \quad (3.7)$$

$$y_p = -lcos\theta \quad (3.8)$$

$$\dot{y}_p = l\dot{\theta}sin\theta \quad (3.9)$$

$$\ddot{y}_p = l\dot{\theta}^2 cos\theta + l\ddot{\theta}sin\theta \quad (3.10)$$

$$N = m\left(\ddot{x} - l\dot{\theta}^2 sin\theta + l\ddot{\theta}os\theta\right) \quad (3.11)$$

$$P = m(l\dot{\theta}^2 cos\theta + l\ddot{\theta}sin\theta + g) \quad (3.12)$$

### 3-2- Controller

The PID controller has three coefficients, namely proportional (P), integral (I), and derivative (D), which are used to tune the controller's response and achieve the desired system behavior.

### 3-3- Simulation

In this project, two simulation methods were employed to model the system. The first method involved simulating the system using Simulink blocks, while the second method used Simscape blocks for simulation. The Simulink simulation relied on the equations mentioned in section 3.1, whereas the Simscape simulation only required the system parameters.

a) Simulink Simulation:

The Simulink simulation was implemented using a block diagram representation of the system, which is shown below:

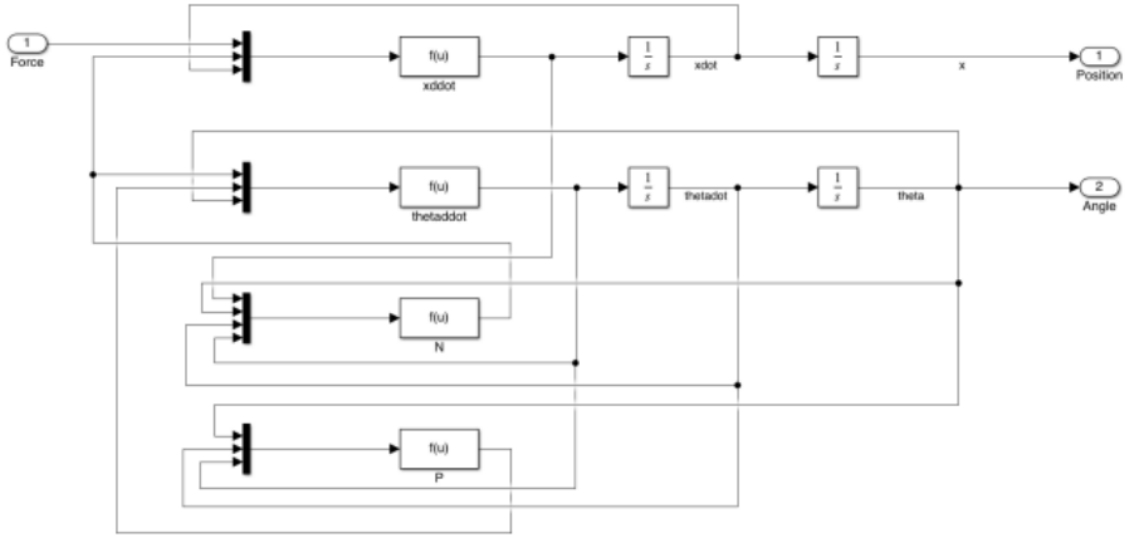



Figure3.3. Demonstration of the block diagram of the system

The block diagram of the system with a PID controller will be as follows:

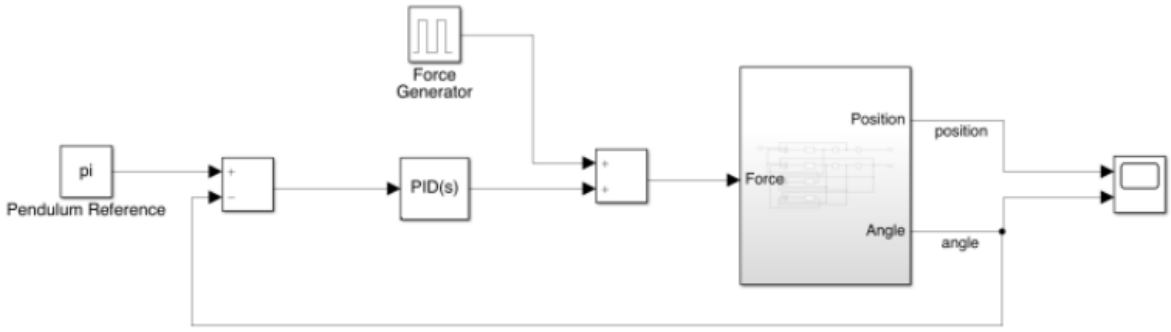



Figure3.4. Demonstration of the Controller

In the system model, a 1ns pulse is used as the input. The simulation file named "Simulink_Segway.slx" has been attached for reference.

b) Simscape Simulation:

One of the significant advantages of this simulation method is the real-time animation visualization of the model. The Simscape representation of the system is shown in the figure below:

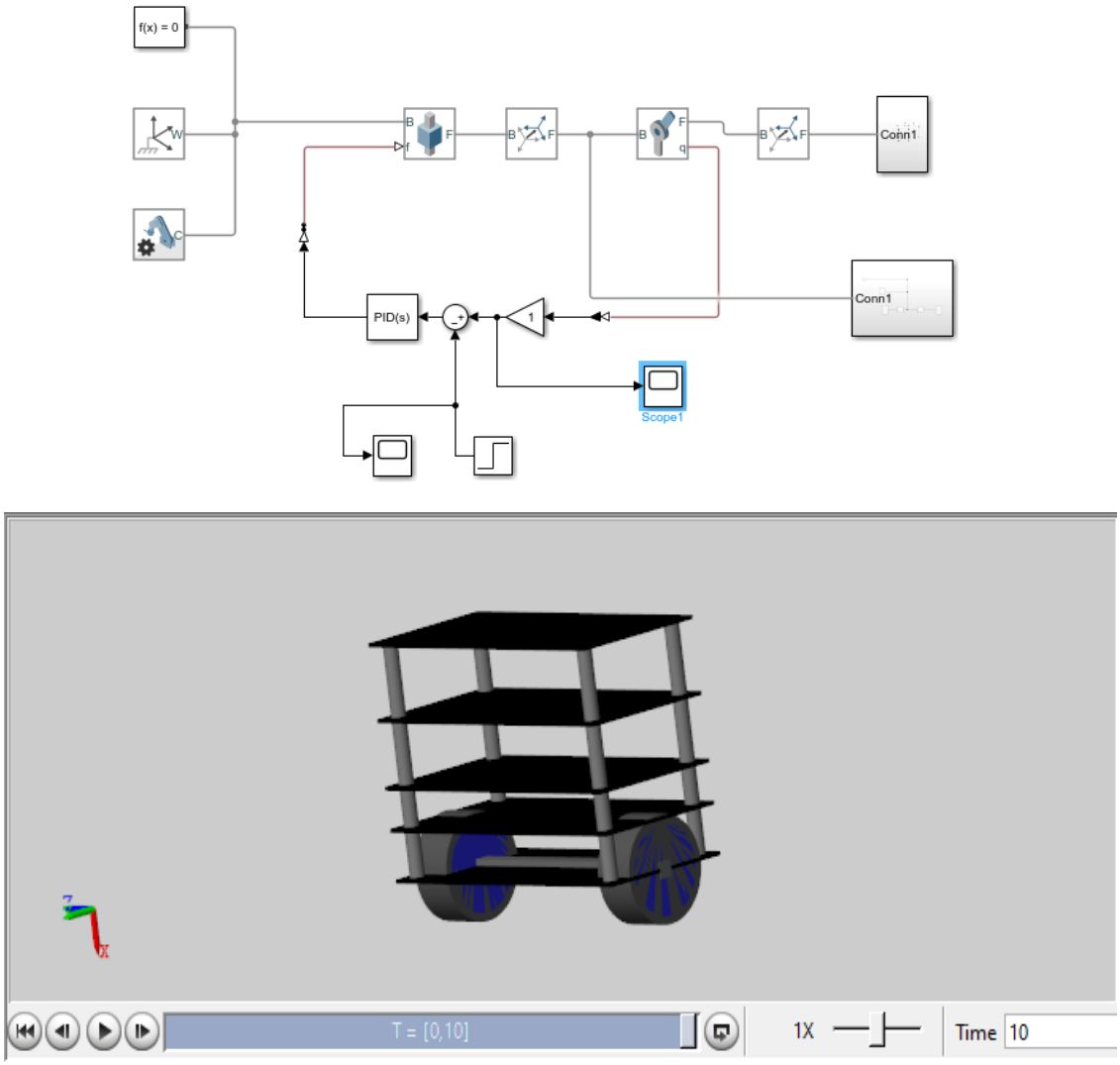



Figure3.5. Demonstration of the simulation in Simscape

## 3-4- Results

a) Simulation Results with Simulink Blocks:

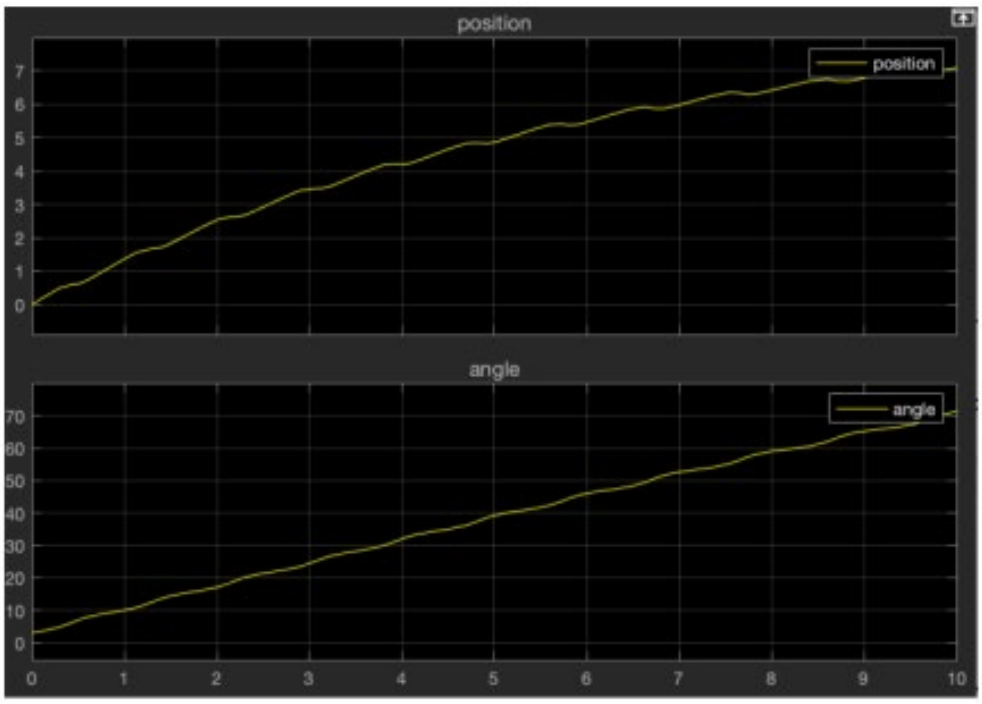



Fig3.6. Output of the system without controller

As shown in the graph, the system is unstable, so a PID controller is added to the system.

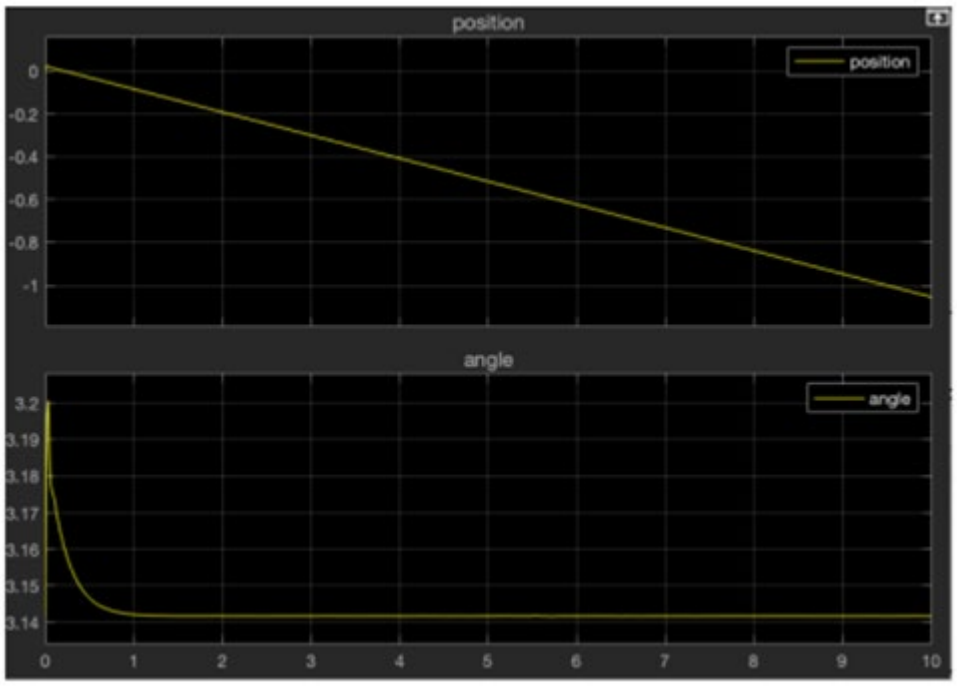



Figure3.7. Output of the system with PID controller

After implementing the PID controller, the pendulum angle changes instantly and returns to the desired position after a short time. It should be noted that the

value of π to which the angle has converged, as shown in Figure 2, indicates that the pendulum has not fallen and is standing upright.

b) Simulation Results with Simscape Blocks:

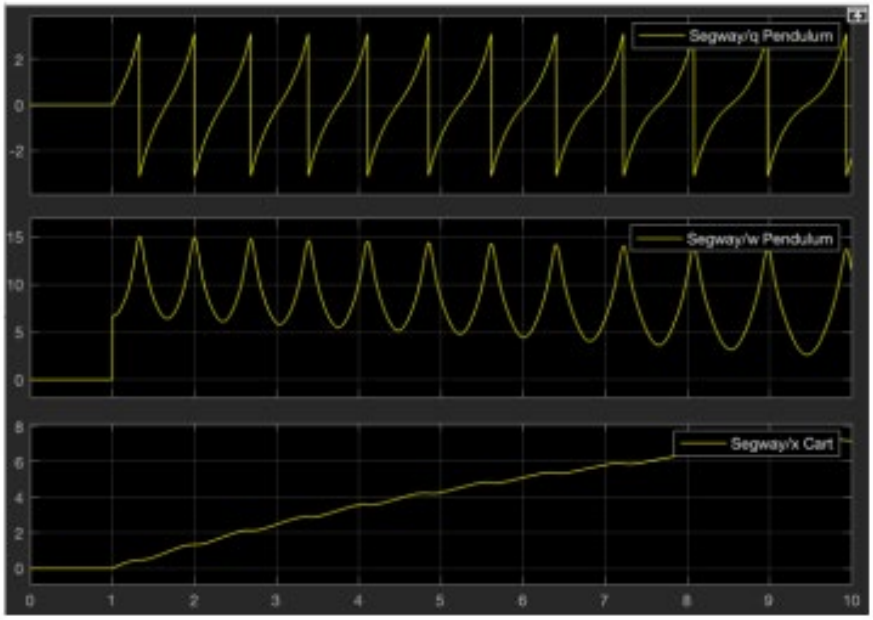



Figure3.8. System without controller

As shown in the animation, the pendulum falls without a controller (the motion is shown in the attached animation). Therefore, a PID controller is added to the system to maintain the balance of the pendulum.

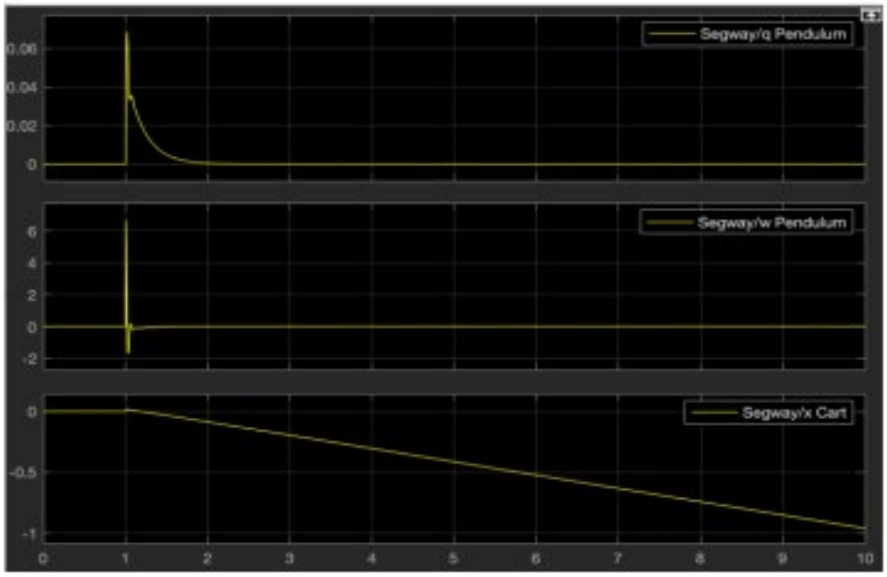



Figure3.9. System after adding PID controller

After implementing the PID controller, the pendulum angle changes instantaneously and returns to its equilibrium position in a very short time, as observed in the animation. (The motion of the system with the controller is shown in the second animation.)

## 4- Design of the Body and Mechanics

To simplify the construction process, the body and structure of the balance robot are designed in a modular manner. Specific electronic components are placed in each module, making the robot easy to assemble and maintain. One of the most important parameters in controlling a balance robot is its weight. To achieve a lightweight design, the body is made of Plexiglas material. The robot is ultimately constructed with a weight of less than one kilogram and a height of 18 centimeters. It's worth noting that the entire design process was done using SolidWorks software, which ensured accurate and efficient design.

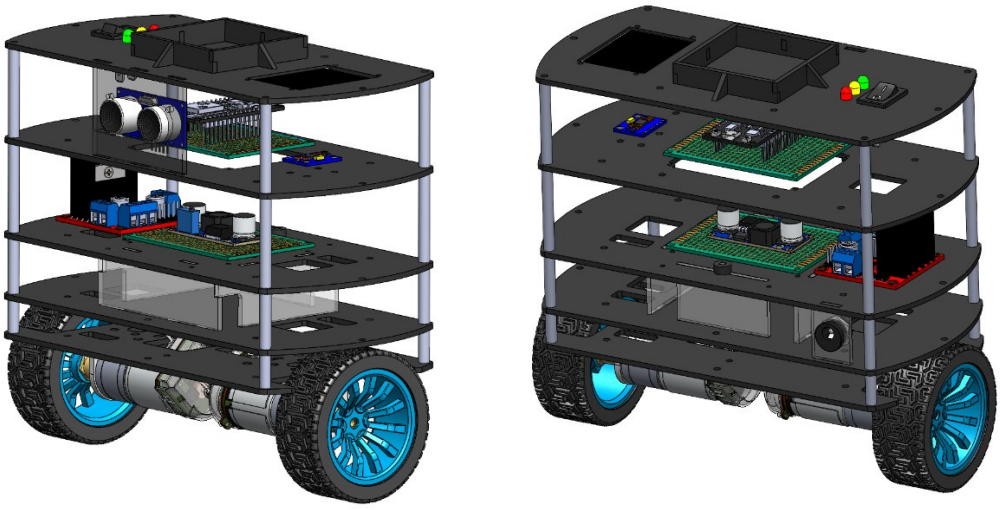


Figure4.1. Designed frame in SolidWorks

## 5- Hardware Design

### 5-1- Power System Design

In this stage, the main focus was to ensure proper power supply to the various components of the robot. As mentioned earlier, the robot is powered by three lithium-ion 18650 batteries with a current capacity of 2200 milliampere-hours. Each section of the robot requires a separate power supply. For instance, the processor requires a 5-volt output, and the motors require a 12-volt output.

To power the motors, we used the direct battery output. To ensure a safe and steady power supply to the processor, we designed and implemented an LM2576 5-volt regulator circuit on a perforated board. We also used appropriate connectors for each section to standardize the power supply. For example, we used a T-connector to connect the batteries to the board, and XH connectors were used to connect the power section to other parts of the robot for added safety against short circuits.

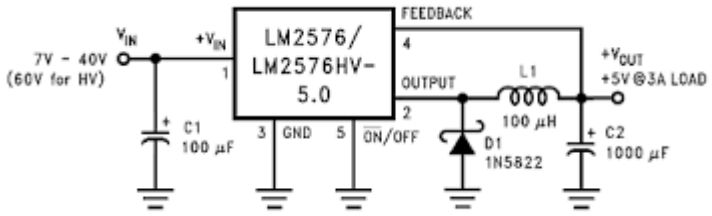



Figure5.1. Circuit Diagram of the Regulator

## 5-2- Motor Drivers

Due to the high current draw of the motors, driving them directly is not possible. Therefore, we used an L298 motor driver to control and provide power to the motors. The L298 motor driver has two H-bridge channels, each capable of supplying up to 2A of current. Each channel has two digital input pins that control the direction of rotation of the motors and an enable (EN) pin that controls the motor speed using PWM.

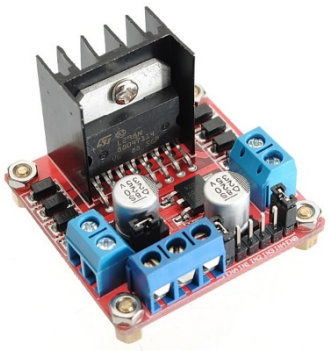


Figure5.2. L298 Motor Driver

## 5-3- Connections

For the initial setup, the IMU sensor is connected to the I2C bus, and the motor driver input is connected to two PWM pins and four digital GPIO pins. Other peripheral connections, such as the SPI output for the TFT display, the ADC input

for measuring the battery voltage, and digital output pins for status notifications (such as buzzers and LEDs), do not affect the balance robot's performance.

To operate the ultrasonic module, two timer pins are required for generating and receiving pulses. These connections are essential for the balance robot's optimal performance.

# 6- Software Design

## 6-1- Introduction to the tools

In the construction of this balance robot project, various tools and software were utilized, such as MATLAB, Python, Arduino, and Simulink. As mentioned earlier, the majority of the simulation work was performed using MATLAB and Simulink, which is not repeated here.

For programming the processor, we used the Arduino IDE 2 software. The advantage of using the second version of this software was the ability to draw graphs to visualize the sensor's performance. To program the robot, we first tested and verified the subsystems individually and adjusted their performance according to our needs. These subsystems include the motor section, IMU sensor, ultrasonic sensor, image processing section, and the Wi-Fi communication module. This approach helped us to ensure each subsystem's proper functionality before integrating them into the overall system.

## 6-2- IMU Sensor and Filters

One of the main challenges we faced during the software development was achieving reliable detection for the IMU section since it plays a crucial role in detecting any deviation and maintaining balance. To achieve reliable detection, we first studied the structure of the MPU-6050 and measured its response. We installed the IMU sensor on the robot in a way that any change in the robot's angle would result in a corresponding change in the ROLL angle of the sensor.

After the initial I2C protocol connections, we measured the raw output values of the sensor. This process involved several steps, including adjusting the settings of the MPU6050 sensor, such as the measurable angular velocity and acceleration, obtaining the sensor values using the data sheet and I2C addresses, and measuring the calibrated output values and their biases before entering the main control loop.

Another important aspect of setting up the IMU sensor is its reading frequency. In our robot, we set the frequency to 260Hz using a while loop and the micros()

function. The MPU6050 has a Rate Gyro sensor that detects changes in angular velocity and uses an integrator to calculate the static angle. This approach allowed us to achieve a reliable detection for the IMU section and maintain the robot's balance.

```
// IMU Sensor constants
const int MPU_ADDRESS = 0x68;
const int MPU_SMPLRT_DIV = 0x19;
const int MPU_CONFIG = 0x1A;
const int MPU_GYRO_CONFIG = 0x1B;
const int MPU_ACCEL_CONFIG = 0x1C;
const int MPU_ACCEL_XOUT_H = 0x3B;
const int MPU_TEMP_OUT_H = 0x41;
const int MPU_GYRO_XOUT_H = 0x43;

const int MPU_CALIBRATION_SAMPLES = 3000;
const double MPU_SCALE_FACTOR_GYRO = 65.5;
const double MPU_SCALE_FACTOR_ACCEL = 8192;

void mpu6050_init(void)
{
  // Activate the MPU-6050
  Wire.beginTransmission(MPU_ADDRESS);
  Wire.write(0x6B);
  Wire.write(0x00);
  Wire.endTransmission();

  // Configure the sample rate divider
  Wire.beginTransmission(MPU_ADDRESS);
  Wire.write(MPU_SMPLRT_DIV);
  Wire.write(0x00);
  Wire.endTransmission();

  // Configure the accelerometer
  Wire.beginTransmission(MPU_ADDRESS);
  Wire.write(MPU_ACCEL_CONFIG);
  Wire.write(0x08);   // 4g → 0x08
  Wire.endTransmission();

  // Configure the gyro
  Wire.beginTransmission(MPU_ADDRESS);
  Wire.write(MPU_GYRO_CONFIG);
  Wire.write(0x08);   // 500 deg/s → 0x08
  Wire.endTransmission();
}
```

```
void mpu6050_read(void) {
  Wire.beginTransmission(MPU_ADDRESS);
  Wire.write(MPU_ACCEL_XOUT_H);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU_ADDRESS, 14, true);

  acc_x = Wire.read() << 8 | Wire.read();
  acc_y = Wire.read() << 8 | Wire.read();
  acc_z = Wire.read() << 8 | Wire.read();
  temperature = Wire.read() << 8 | Wire.read();
```

```
  gyro_x = Wire.read() << 8 | Wire.read();
  gyro_y = Wire.read() << 8 | Wire.read();
  gyro_z = Wire.read() << 8 | Wire.read();
}

void gyro_calibration(void) {
  for (int cal_int = 0; cal_int < MPU_CALIBRATION_SAMPLES; cal_int++) {
    read_mpu_6050_data();
    gyro_x_cal += gyro_x;
    gyro_y_cal += gyro_y;
    gyro_z_cal += gyro_z;
    delay(3);
  }

  gyro_x_cal /= MPU_CALIBRATION_SAMPLES;
  gyro_y_cal /= MPU_CALIBRATION_SAMPLES;
  gyro_z_cal /= MPU_CALIBRATION_SAMPLES;
}
```

To improve the performance and increase the accuracy of momentary measurements, we implemented software filters. These filters prevent sudden changes and oscillations in the output. Initially, we applied a Complementary Filter to the gyroscope and accelerometer data. This filter combines high-frequency gyroscopic signals with low-frequency accelerometer signals to provide accurate and stable readings. By using this filter, we were able to reduce noise and improve the accuracy of our measurements, resulting in a more stable and reliable balance robot.

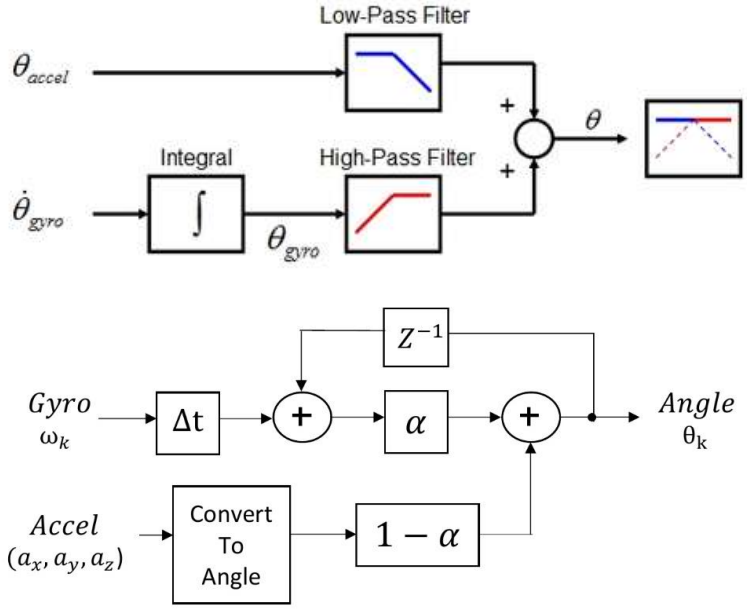



Fig6.1. Block Diagram of Complementary Filter

By knowing the sensor sampling frequency and the coefficient α, we can calculate the filter cutoff frequency. In our case, the sampling frequency is 260Hz,

and the time interval is approximately 0.004 seconds. We chose a value of α = 0.95 for our filter.

To calculate the filter cutoff frequency, we can use the following formula:

$$fc = \alpha / (2\pi * \Delta t) \quad (6.1)$$

where fc is the filter cutoff frequency, α is the filter coefficient, and Δt is the time interval between samples.

Plugging in our values, we get:

$$fc = 0.95 / (2\pi * 0.004) \approx 30 \text{ Hz} \quad (6.2)$$

Therefore, our filter cutoff frequency is approximately 30Hz. This means that any signal with a frequency higher than 30Hz will be attenuated by the filter. By setting the filter cutoff frequency appropriately, we can ensure that our filter provides accurate and stable readings, while also reducing noise and eliminating high-frequency oscillations.

$$\tau = \frac{T_s \cdot \alpha}{1-\alpha} \rightarrow \tau = \frac{0.004 \times 0.95}{0.05} = 0.076s \quad (6.3)$$

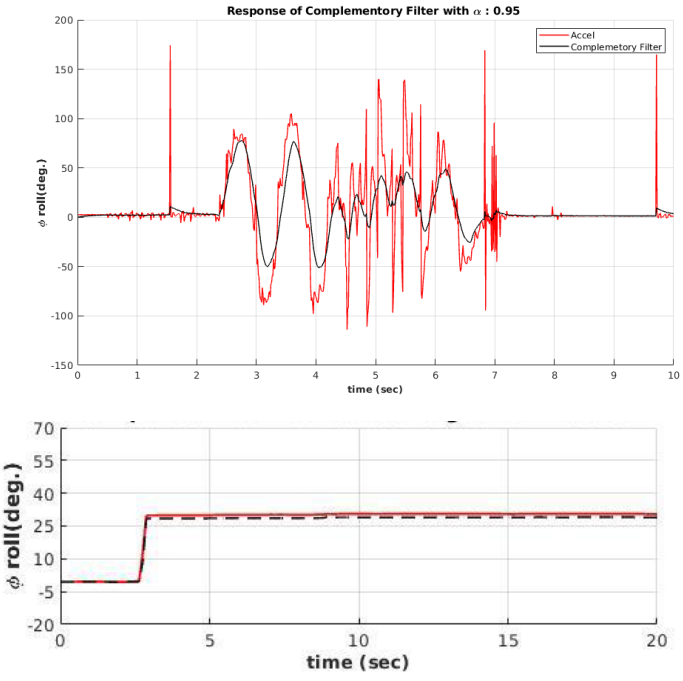



Figure6.2. Unfiltered and Filtered Results

```
void loop()
{
  dt = (micros() - loopTimer2) * 1e-6;
  loopTimer2 = micros();

  // Read raw accelerometer and gyroscope data
  mpu6050_read();

  // Subtract the offset calibration value
```

```
  gyro_x -= gyro_x_cal;
  gyro_y -= gyro_y_cal;
  gyro_z -= gyro_z_cal;

  // Convert to instantaneous degrees per second
  rotation_x = (double)gyro_x / MPU_SCALE_FACTOR_GYRO;
  rotation_y = (double)gyro_y / MPU_SCALE_FACTOR_GYRO;
  rotation_z = (double)gyro_z / MPU_SCALE_FACTOR_GYRO;

  // Convert to g-force
  accel_x = (double)acc_x / MPU_SCALE_FACTOR_ACCEL;
  accel_y = (double)acc_y / MPU_SCALE_FACTOR_ACCEL;
  accel_z = (double)acc_z / MPU_SCALE_FACTOR_ACCEL;

  // Calculate acceleration rate
  accelPitch = atan2(accel_y, accel_z) * RAD_TO_DEG;
  accelRoll = atan2(accel_x, accel_z) * RAD_TO_DEG;

  // Integrate over the gyro. rate
  gyroPitch += rotation_x * dt;
  gyroRoll -= rotation_y * dt;
  gyroYaw += rotation_z * dt;


    // Complementary Filter
  pitch = (FILTER_TAU) * (pitch + (rotation_x * dt)) + (1 - FILTER_TAU) * (accelPitch);
  roll = (FILTER_TAU) * (roll - (rotation_y * dt)) + (1 - FILTER_TAU) * (accelRoll);

  // Wait until the loopTimer reaches 3846us (260Hz) before next loop
  while (micros() - loopTimer <= 3846);
  loopTimer = micros();
}
```

Continuing on, to further improve the measurement accuracy, we also used a Kalman Filter. The Kalman Filter was used to compare and enhance the accuracy of the measurements. The output of the Kalman Filter with the settings we applied was more accurate and converged faster than the previous Complementary Filter.

The Kalman Filter is a mathematical algorithm that uses a series of measurements and predictions to estimate the state of a system. It is particularly useful for systems where measurements are subject to noise, and it can provide accurate and stable estimates even in the presence of significant noise.

By using the Kalman Filter, we were able to further improve the accuracy of our measurements and achieve a more stable and reliable balance robot. The combination of the Complementary Filter and the Kalman Filter allowed us to filter out noise and provide accurate readings, even in challenging conditions.

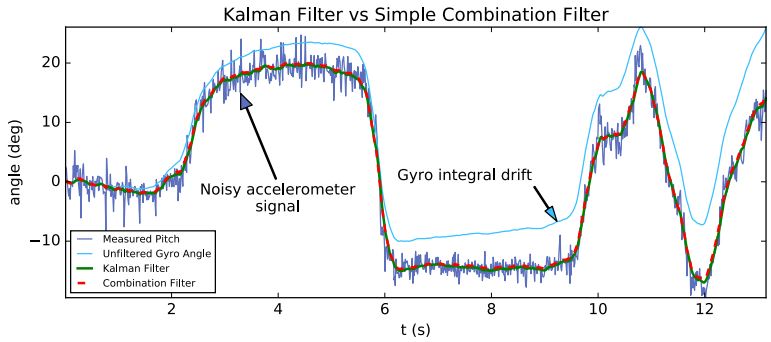



Figure6.4. Kalman vs no Kalman

```
// Kalman Filter Parameters
float Q_angle  =  0.01;
float Q_gyro   =  0.003;
float R_angle  =  0.01;
float P_00 = 0, P_01 = 0, P_10 = 0, P_11 = 0;
float K_0, K_1;
float y, S;

// Kalman Filter
P_00 +=  - dt * (P_10 + P_01) + Q_angle * dt;
P_01 +=  - dt * P_11;
P_10 +=  - dt * P_11;
P_11 +=  + Q_gyro * dt;

y = accelRoll - roll;
S = P_00 + R_angle;
K_0 = P_00 / S;
K_1 = P_10 / S;

roll +=  K_0 * y;
gyro_y_cal  +=  K_1 * y;
P_00 -= K_0 * P_00;
P_01 -= K_0 * P_01;
P_10 -= K_1 * P_00;
P_11 -= K_1 * P_01;
```

## 6-3- Motor Control

In the following section, the initial tests for the motors were conducted, using two different drivers. The first driver, L293D, was not a suitable option due to the motors' quick reverse rotation in the balancing robot, which caused excessive current draw. As a result, the L298 driver was selected and used for the project. During the final tests, an issue was identified with the processor's PinOut design. Specifically, some of the pins were only set for input, which was not reported in the reference datasheet. This issue was attributed to the processor being older than the models presented in the datasheet. Although this caused problems during the

initial motor tests, we were able to solve the issue through trial and error with other pins, and the problematic pins were marked for future reference.

In terms of software design, we required a comprehensive function to control the motors, allowing us to determine both the direction and speed of each motor simultaneously. It was also essential to ensure coordination between the inputs of this function and the output of the controller to optimize the code's efficiency and simplicity. Therefore, the function's input was designed to precisely follow the value of the controller output without any additional calculations or conditions. We understood that the controller's input corresponds to the robot's angle error with the Set Point value, which can be positive or negative depending on the robot's fall direction. Additionally, the controller's output is a function of this error and follows its sign. Consequently, we decided that the sign of the controller output would determine the motor's rotation direction, while its absolute value would determine the PWM speed. Since the goal of this function was to maintain the balance of the robot, the direction and speed of both motors were set to be the same.

```
/* Motor Functions */
void run_motor(int pwm_l, int pwm_r)
{
  if(pwm_r >= 0)
  {
    ledcWrite(pwm_channel_1, pwm_r);
    digitalWrite(R1, LOW);
    digitalWrite(R2, HIGH);
  }
  else
  {
    ledcWrite(pwm_channel_1, -1*pwm_r);
    digitalWrite(R1, HIGH);
    digitalWrite(R2, LOW);
  }

  if(pwm_l >= 0)
  {
    ledcWrite(pwm_channel_2, pwm_l);
    digitalWrite(L1, HIGH);
    digitalWrite(L2, LOW);
  }

  else
  {
    ledcWrite(pwm_channel_2, -1*pwm_l);
```

```
    digitalWrite(L1, LOW);
    digitalWrite(L2, HIGH);
  }
digitalWrite(L2, HIGH);
  }
```

### 6-4- Image Processing and Remote Control

The robot is powered by an ESP32 processor, which offers internet connectivity and is one of the most popular boards for Internet of Things applications due to its speed, affordability, and dual-core architecture. To detect certain hand gestures, a Python program was developed using image processing techniques. The program can detect hand movements using a laptop webcam and connect to the processor via WiFi to execute the relevant codes for our balancing robot, such as turning the LED on and off as a form of interaction. It is worth noting that the LED can be substituted with other applications. The code was written using the CV2, Mediapipe, and TensorFlow libraries, along with the weights of the hand gesture recognition model.

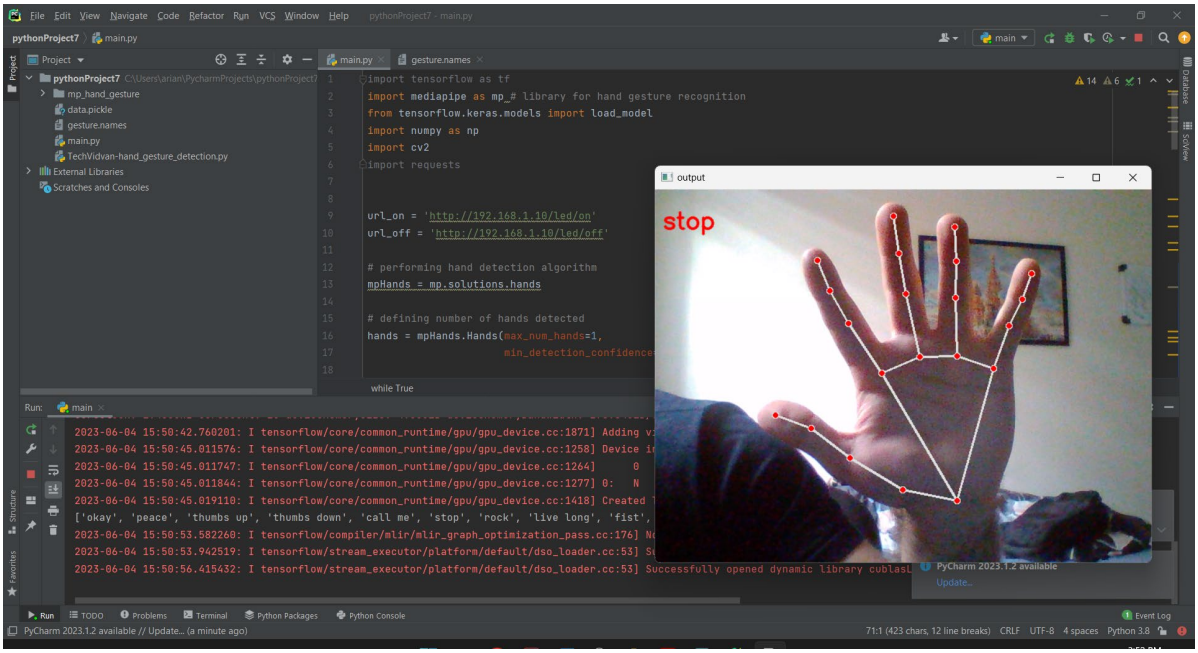



Figure6.5. Demonstration of the Function

## 7- Controller Design

After integrating the subsystems, a controller was required to maintain balance. Classic PID control methods were employed for this purpose. The robot's sampling and control frequency is 260Hz, and the reference input is set to zero to

maintain balance. In the robot control, classic PID control was used, with the filtered output of the IMU sensor as the feedback signal and a zero balance angle as the reference input. Initially, we attempted to use practical methods such as Ziegler-Nichols to determine the PID coefficients. However, due to the system's instability and the lack of access to the output and time stamps, this proved to be impossible. As a result, the coefficients had to be determined through simulation or real-time adjustment using serial communication. We started with a P controller and later added the I and D parameters to improve the system's performance.

```
if(abs(roll) < max_deg) {
  digitalWrite(LED_BLINK, LOW);

  // PID Implementation
  error = roll - setpoint;
  error = ((int)(error * 10 + .5) / 10.0);

  pid_p =  kp * error;
  pid_i += ki * (error * dt);
  pid_d =  kd * ((error - error_previous) / dt);

  PID = pid_p + pid_i + pid_d;

  PID_dir = PID / abs(PID);
  if (abs(roll) < 0.5)
    PID += PID_dir * 80;
  else if(abs(roll) <1)
    PID += PID_dir * 70;
  else if(abs(roll) < 1.5)
    PID += PID_dir * 60;

  if(abs(roll) <= 0.1)
    pid_i = 0;

  if(abs(PID) > max_pwm){
    PID /= abs(PID);
    PID *= max_pwm;
   }

  run_motor(PID, PID);
  error_previous = error;
}
else {
  // Serial.println("WARNING! MAXIMUM ANGLE OVERSHOOT");
  digitalWrite(LED_BLINK, HIGH);
  run_motor(0, 0);
}
```

Table7.1. Tuned PID Table

| | **P** | **I** | **D** |
|---|---|---|---|
| **#1** | 70 | 0 | 0 |

| | | | |
|---|---|---|---|
| **#2** | 70 | 1 | 0 |
| **#3** | 70 | 10 | 0 |
| **#4** | 70 | 5 | 0 |
| **#5** | 70 | 5 | 1 |
| **#6** | 70 | 5 | 0.1 |
| **#7** | 70 | 5 | 0.05 |
| **#8** | 68 | 5 | 0.05 |
| **#9** | *68* | *6.5* | *0.05* |

## 8- Summary and Conclusion

We designed and built a balancing robot, covering all the essential phases of conceptual design, main design, simulation, and software and hardware construction. Our prototype demonstrated multiple capabilities, such as hand gesture recognition, and we identified several areas for future improvement.

Throughout the project, our focus was to gain practical experience in designing and simulating the robot, programming the hardware, debugging the robot, standardizing subsystems, and solving other constructive and useful issues. We also aimed to learn about building a balancing robot from scratch.

This report provides a general overview of the project and the challenges we faced and overcame. We hope that this report can serve as a helpful guide for others interested in building their own balancing robot, as well as highlight the importance of practical experience and problem-solving skills in mechatronics.

Arian Hajizadeh – Erfan Riazati – Sorin Yousefnia