DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

COURSE OF MODELLING AND CONTROL OF ELECTRIC DRIVES

A self balancing robot using ESP32 and MPU6050

Lecturer
Prof. Bianchi Nicola

Students
Bano Massimo, Sperandio Nicola

ID Numbers
2139251-2156437

ACCADEMIC YEAR 2025-2026

Abstract

This projects aims to design and implement a self-balancing robot using an ESP32 microcontroller and an IMU (Inertial Measurement Unit) sensor. The digital controller is based on a Proportional-Integral-Derivatived (PID) algorithm, which is tuned in order to achieve stability in the equilibrium position. The chassis of the robot is 3D printed, and the robot is powered by two DC motors. Throughout this work, the theoretical concepts behind the design of a self-balancing robot are explored, including the mathematical modeling of the system, the design of the PID controller, and the implementation details. At the end of this project, in the appendix section, the code for MATLAB/Simulink modelling and the microcontroller code provided.

Contents

Chapter 1

Introduction

We first start by introducing the concept of a self-balancing robot, which is a type of two-wheels robot that can maintain its balance while standing upright: this is achieved by continuisly measure the pitch angle of the robot and consequently adjust the motor's speed to keep the robot balanced. The main components of the self-balancing robot are reported in Table 1.1.

Component	Description
ESP32	Microcontroller
MPU6050	Inertial Measurement Unit (IMU) sensor
DFRobot DC 6V	DC Motors
DRV8833	Motor Driver
Molicel P42A 3.6V 45A	Series of two batteries for power supply.

Table 1.1: Main components of the self-balancing robot

1.1 ESP32 Microcontroller

The ESP32 (reported in Figure 1.1) is a powerful microcontroller developed by Espressif Systems, widely used in IoT applications due to its built-in Wi-Fi and Bluetooth capabilities. It features a dual-core processor, ample memory, and various peripherals, making it suitable for real-time control tasks required in self-balancing robots.

We chose the ESP32 for our self-balancing robot project because of its processing power but moreover for its higher clock speed (240 MHz) compared to other microcontrollers like Arduino Uno (16 MHz) or Arduino Mega (16 MHz). This allows faster control loop execution, which is crucial for maintaining balance in real-time.

The microcontroller can be programmed using the Arduino IDE but we chooose for another IDE, called PlatformIO, which offers more advanced features and better project management capabilities. This IDE can be integrated into Visual Studion Code and so we can take trace of all the changes with Git version control system.

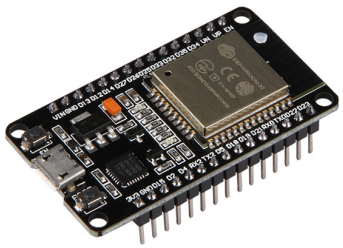

Figure 1.1: ESP32 development board

1.2 MPU6050

For the IMU sensor, we selected the MPU6050 (shown in Figure 1.2) which combines a 3-axis gyroscope and a 3-axis accelerometer. This sensor provides all the data through the I2C communication protocol. The MPU6050 range of measurement are reported in Table 1.2.

Sensor	Range of Measurement
Accelerometer	±2g, ±4g, ±8g, ±16g
Gyroscope	±250, ±500, ±1000, ±2000 °/s

Table 1.2: MPU6050 range of measurement

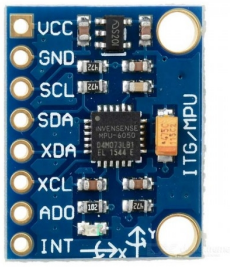

Figure 1.2: MPU6050 Inertial Measurement Unit (IMU) sensor

In our application we configure the accelerometer to a range of $\pm 4g$ for reasons that will be explained in Chapter ... and the gyroscope to a range of $250\,^{\circ}/s$ since we need high sensitivity and we don't expect higher angular velocities.

1.3 Motor Driver and DC Motors

1.3.1 DC motor

For the DC motor we selected the DFRobot DC 6V (shown in Figure 1.3), a motor that comes with a 120:1 gear ratio, providing high torque at low speeds, which is ideal for

balancing applications. The motor is powered by a 6V power supply and can draw a stall current of up to 1.2A.

Figure 1.3: DFRobot DC 6V motor with 120:1 gear ratio

1.3.2 DRV8833 Motor Driver

To control the DC motors, we use two DRV8833 motor drivers (shown in Figure 1.4), which are capable of handling motor supply voltages from 6.5V to 45V. They are double full-bridge drivers, allowing for bidirectional control of two motors each. However, each channel of each driver can only supply 2A, slightly less than what our motors need in steady state to achieve maximum torque. For this reason, we connected the two channels of each driver in parallel, to achieve a maximum capability of 4A per motor.

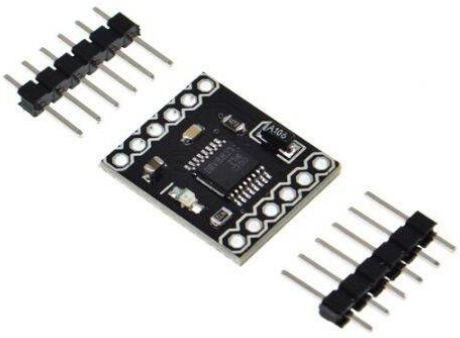

Figure 1.4: DRV8833 motor driver

Chapter 2

Modelling and simulation

Now that we described the hardware components of our self-balancing robot in Chapter 1, we can proceed to model the system and simulate it's behavior in MATLAB/Simulink environment.

2.1 Physical model

The best way to model a self-balancing robot is to treat it as a *inverse pendulum*, like the one reported in Figure 2.1

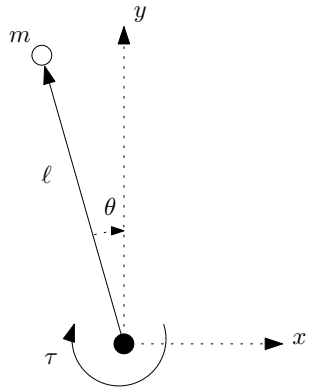

Figure 2.1: Inverse pendulum schematic representation

Where:

- m is the mass of the pendulum
- ℓ is the length of the pendulum
- θ is the angle of the pendulum with respect to the vertical axis
- τ is the torque applied to the base of the pendulum

The equations that govern the motion of the inverse pendulum can be derived using Newton's laws.

$$m\ell^2\ddot{\theta}(t) = mg\ell sin(\theta) + \tau(t) \tag{2.1}$$

Where g is the acceleration due to gravity and $\ddot{\theta}(t)$ is the angular acceleration of the pendulum. To linearize the equation 2.1, we can use the small angle approximation, which states that for small angles (in radians), $sin(\theta) \approx \theta$. This leads to the following linearized equation:

$$m\ell^2\ddot{\theta}(t) = mg\ell\theta + \tau(t) \tag{2.2}$$

From equation 2.2, we can derive the transfer function of the system by taking the Laplace transform, assuming zero initial conditions:

$$G_{pendulum}(s) = \frac{\theta(s)}{\tau(s)} = \frac{\frac{1}{m\ell^2}}{s^2 - \frac{g}{\ell}} \tag{2.3}$$

In equation 2.3 we can see that the system has two poles at $s = \pm\sqrt{\frac{g}{\ell}}$, indicating that the system is unstable, as one of the poles is in the right half of the s-plane.

2.2 DC motor modelling

To model the DC motors used in our project, we can use the following equations that describe the electrical and mechanical dynamics of a DC motor:

$$v(t) = L\frac{di(t)}{dt} + Ri(t) + K_\phi\omega(t) \tag{2.4}$$

By applying the Laplace transform to equation 2.4, and considering also the gear ratio $K_G = 120$ we obtain:

$$G_{motor}(s) = \frac{\tau(s)}{v(s)} = k_G \cdot \frac{K_\phi}{R + sL} \tag{2.5}$$

The obtained transfer function is a first order system.

2.3 Complete model

The complete model is the product of the two transfer functions obtained: we have to consider that we do not consider the *back EMF* of the motor in the transfer function: this is a simplification of the model. In Figure 3.2 the complete block diagram is reported, where $G(s) = G_{pendulum}(s) \cdot G_{motor}(s) = \frac{\theta(s)}{v(s)}$.

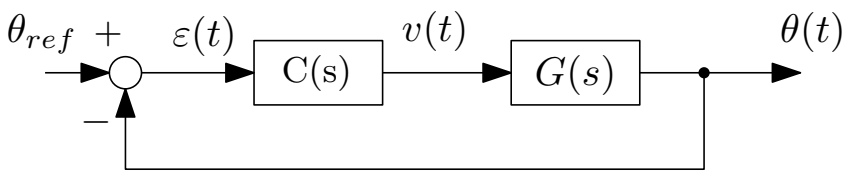

Figure 2.2: Complete block diagram of the self-balancing robot model

2.4 Controller design

Now that we have the complete transfer function of the system we can design the controller: we will use a *PID controller*. The PID controller has the following transfer function:

$$G_{PID}(s) = K_P + \frac{K_I}{s} + K_D s \tag{2.6}$$

Where K_P, K_I and K_D are the proportional, integral and derivative gains respectively. We want a controller such that the rise time of the closed loop system is around $50ms$ and the overshoot is less than 15%: rise time is choosen in order to have a fast response of the system, while overshoot is limited in order to avoid that the motors broke due to overvoltage. We choose to let an automatic tuning *MATLAB* function to find the best values for the PID gains:

```
% rise time
ts = 50e-3;
% overshoot choosen to be under a safe margin of maximum
   voltage of DC motors, that is 7.5V
M = 0.15;

% Crossover frequency and phase margin
fct = 2/(ts*2*pi);
pmt = 1.04 - 0.8*M;

% PID tuning options
opt = pidtuneOptions;
opt.PhaseMargin = 180*pmt/pi;

% PID controller design
gi = pidtune(G_sys, 'pid', 2*pi*fct, opt);
```

Listing 2.1: MATLAB code for PID tuning

The obtained parameters are:

- $K_P = 0.0361$
- $K_I = 0.1412$
- $K_D = 0.0023$

2.5 Simulation of the model

Now that we have the complete linearized model of the system, we can evaluate its behavior with a step response: we impose a step of 20°, simulating a small disturbance that makes the robot tilt of 20° from the vertical position. The step response is reported in Figure 2.3.

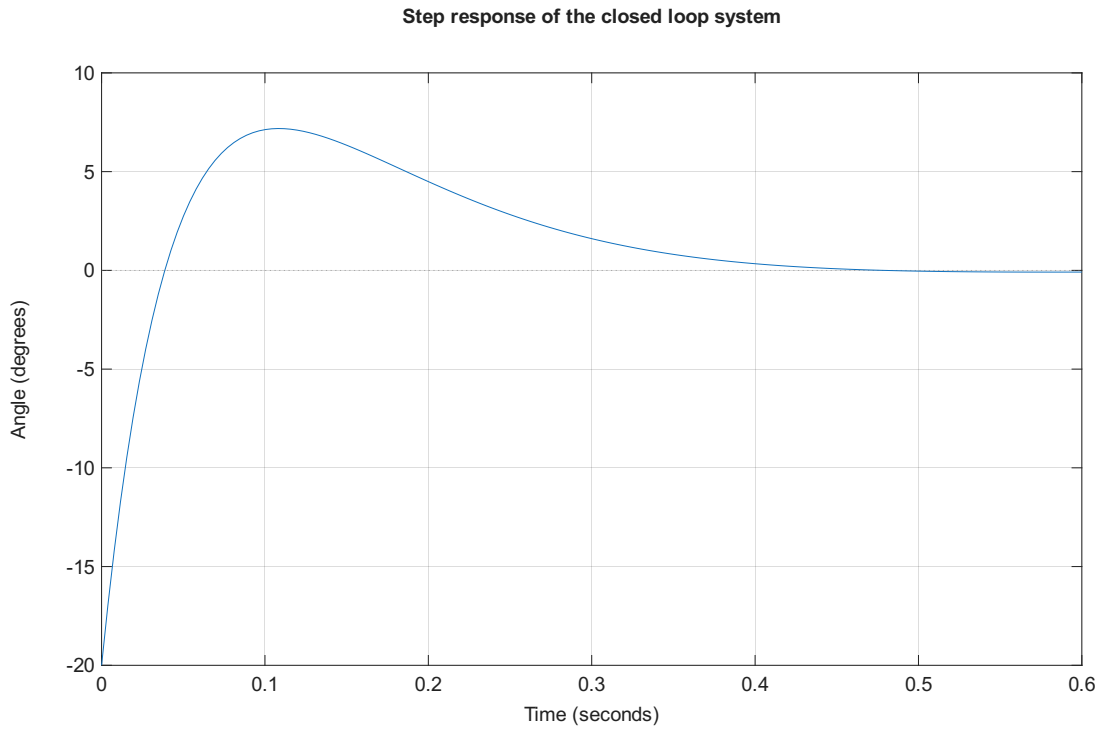

Figure 2.3: Step response of the closed loop system with PID controller

Chapter 3

Control

3.1 High Level System Description

The objective of the control is to stabilize the robot, returning its angle to the set reference. The microcontroller reads the angle via an inertial unit, the MPU6050 sensor, and outputs a PWM that feeds into the DC motor driver. Hence, the system is characterized by a singular position loop.

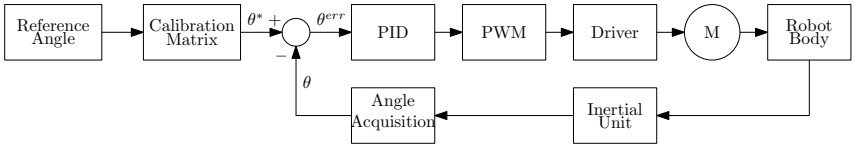

Figure 3.1: Simplified block diagram of the control structure

3.2 Angle Measurement

Retrieving the required angle from the inertial unit is not as simple as receiving the raw data. First of all, the output of the MPU6050 unit is composed of two three-valued vectors: (ax, ay, az) are the coordinates in space given by the accelerometer, while (gx, gy, gz) are the components of angular velocity given by the gyroscope.

3.2.1 Accelerometer Derived Angle

The accelerometer data can be utilized to determine the angle in respect to the initial position thanks to trigonometry, utilizing the arctangent function:

$$\theta_{acc} = \arctan\left(\frac{y_{acc}}{z_{acc}}\right) \cdot \frac{180}{\pi} \tag{3.1}$$

Initially, we utilized only this measurement, but it turned out it is too noisy and slow for accurate and fast transient reponse.

3.2.2 Gyroscope Data

The gyroscope outputs angular velocity $\omega(t)$ readings along the three rotational axes of the device. In our case, by integrating the speed of rotation along x, we obtain relative angle from the starting point.

$$\theta_{gyro}[k] = \theta_{gyro}[k-1] + \omega_{gyro}[k]\,\Delta t_s \tag{3.2}$$

The angle obtained from this method is fast, but imprecise. At the same time, it is gravely affected by low-frequency drift and, on its own, it is not suitable for our robot.

3.2.3 Complementary Filter

Instead of relying on either one of the two angle measurements, we took the best from each of them, by combining the low frequency response of the accelerometer and the high frequency dynamics of the gyroscope. We used the following complementary filter to mix the readings:

$$\theta[k] = \alpha\left(\theta[k-1] + \omega_{\text{gyro}}[k]\,\Delta t_{\text{s}}\right) + (1-\alpha)\,\theta_{\text{acc}}[k] \tag{3.3}$$

We found empirically that the α parameter must be very high, in the order of $0.90 - 0.99$, to guarantee the quick response from the gyroscope to be dominant. The output from this filter is both stable and clean at low frequency and fast for quick transients, providing already a strong derivative action that acts regardless of the PID regulator. Also for this reason, the system had to be tuned experimentally.

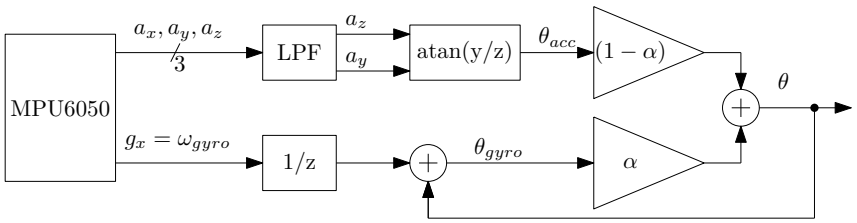

Figure 3.2: Block diagram of the angle measurement

3.3 Finite State Machine

The firmware makes use of a Finite State Machine to ensure the correct sequence of events. Towards the objective of clean and mantainable code we defined the RobotState type shown in 3.1. The definition of a custom type allows to easily recognize code blocks and inherently limits the admissible states to the ones explicitly stated in the enumeration.

```
typedef enum{
  STATE_INIT,
  STATE_MEASURE_ANGLE,
  STATE_COMPUTE_PID,
  STATE_DRIVE_MOTORS,
```

```
  STATE_CRASHED
}RobotState;
```

Listing 3.1: FSM definition

We used a "switch" control structure the main loop to cycle between states.

1. INIT: initialization state, utilized simply for the startup of the system
2. MEASURE_ANGLE: in this state, the robot has to start the angle measurement
3. COMPUTE_PID: the measured angle can be utilized to compute the PID output.
4. DRIVE_MOTORS: the computed PID value is taken and processes to be given as input to the PWM module driving the motors.
5. CRASHED: when the accelerometer reading exceeds a fixed treshold, the robot enters the "crashed" state, forcing it to stop driving the motors.

```
switch(currentState){
    case STATE_MEASURE_ANGLE:{

        angle=angleEstimation(angle);
        if(angle.acc>CRASH_TRESHOLD || angle.acc<-CRASH_TRESHOLD){
        currentState = STATE_CRASHED;
        }else{
        currentState = STATE_COMPUTE_PID;
        }
        digitalWrite(DEBUG_PIN_1, 1);
        break;
    }
    case STATE_COMPUTE_PID:{
        PIDresponse(angle.fusion);
        currentState = STATE_DRIVE_MOTORS;
        break;
    }
    case STATE_DRIVE_MOTORS:{
        motorControl(Output);
        digitalWrite(DEBUG_PIN_1, 0);
        buf[wptr] = {angle.acc, angle.gyro, angle.fusion, Output, angle.
    gyroRate}; // buffer for serial print
        wptr = (wptr + 1) % 512;
        currentState = STATE_MEASURE_ANGLE;
        break;
    }
    case STATE_CRASHED:{
        motorControl(0);
        PIDresponse(0);
        buf[wptr] = {angle.acc, angle.gyro, angle.fusion, Output, angle.
    gyroRate}; // buffer for serial print
        wptr = (wptr + 1) % 512;
        currentState = STATE_MEASURE_ANGLE;
        break;
    }
```

```
}
```

Listing 3.2: FSM Working Principle

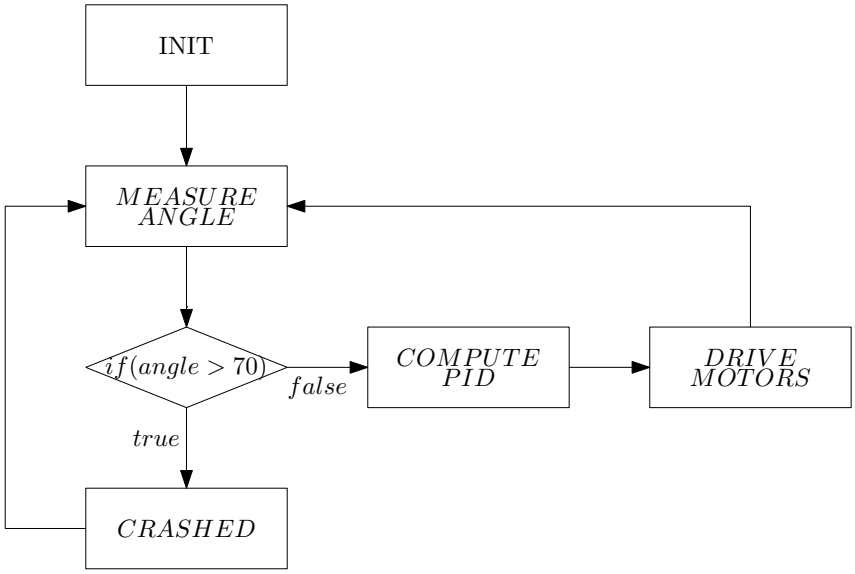

Figure 3.3: Block diagram of the state machine

3.4 Timing Constraints

In this project, as in any digital control system, it is crucial that the timing is regular and as precise as possible. In particular we have to pay attention to the integrative action of the PID computation and of the complementary filter.

During our first tests we did not pay much attention to these issues, which resulted in an uncontrollable system. With three simple precautions we have avoided issues regarding these timings.

3.4.1 Hardware Interrupt

We configured the interrupt in the setup sequence of the microcontroller:

```
Timer0_Cfg = timerBegin(0, 80, true); // 80e6/80 = 1e6 => 1 tick = 1 us
timerAttachInterrupt(Timer0_Cfg, &Timer0_ISR, true);
timerAlarmWrite(Timer0_Cfg, 1000, true); //1000*1us = 1ms
timerAlarmEnable(Timer0_Cfg);
```

Listing 3.3: Interrupt Setup

The first line initializes a hardware timer with a prescaler of 80. Since the microcontroller runs at 80 MHz, each timer tick corresponds to:

$$t_{\text{tick}} = \frac{1}{f_{\text{timer}}} = \frac{\text{prescaler}}{f_{\text{CPU}}} = \frac{80}{80\,\text{MHz}} = 1\ \mu\text{s}. \tag{3.4}$$

The second line attaches our interrupt service routine `Timer0_ISR` to this timer. The third line sets the alarm value to 1000 ticks, which corresponds to an interrupt period of:

$$T_s = t_{\text{tick}} \times 1000 = 1\ \mu\text{s} \times 1000 = 1\ \text{ms}. \tag{3.5}$$

Finally, the alarm is enabled, so the ISR is called every $T_s = 1$ ms. It coult be noted that this timing is not as fast as the microcontroller allows. Since the system's bottleneck is the minimum motor on-time, we have chosen $1ms$ interrupt period as is fast enough to provide accurate control while leaving a comfortable amount of time for non-critical tasks.

The interrupt service routine (ISR) is very simple as the Arduino core hides most of the options and low-level complexity from the user. The only action we perform in this routine is to set the global flag `tick`.

```
void IRAM_ATTR Timer0_ISR(){
  portENTER_CRITICAL_ISR(&timerMux);
  tick = true;
  portEXIT_CRITICAL_ISR(&timerMux);
}
```

Listing 3.4: Hardware ISR

In the main loop, we simply detect the value of the global flag `tick`. If this flag is set to `true`, it means the ISR has been called, and we can move on to our code. Otherwise, if the flag is not set, the software keeps running in the outer loop. Note that while the software is running, the microcontroller is not necessarily idle: if it was, we would be heavily bottlenecking our system and needlessly reducing our bandwidth. Instead, the microcontroller is simply receiving data from the MPU6050, computing PID or sending out data over serial.

```
void loop() {
  if(tick == true){
    portENTER_CRITICAL(&timerMux);
    tick = false;
    portEXIT_CRITICAL(&timerMux);
    ... // The following code is executed only when an interrupt happens
    }
    ... // The following code is executed free running in the loop
}
```

Listing 3.5: Tick Flag use in Loop

3.4.2 Adaptive Window Size

Let us consider the code block tasked with the estimation of angle from the gyroscope data. The gyroscope provides only angluar speed $\omega(t)$ which needs to be integrated to obtain the angle $\theta(t)$.

$$\theta(t) = \theta_0 + \int_0^t \omega(\tau)\, d\tau \tag{3.6}$$

Since we are in a digital domain, this integration is performed by summing discrete angular velocity samples at a fixed interval T:

$$\theta[k] = \theta[k-1] + \omega[k]\,T \tag{3.7}$$

Clearly, to achieve consistent results, the integration window must be fixed. This has been partially resolved thanks to the interrupt time base. However, small deviations can still occur. To compensate for these variations, the actual sampling interval Δt_k can be measured dynamically between consecutive samples.

$$\theta[k] = \theta[k-1] + \omega[k]\,\Delta t_k \tag{3.8}$$

In our code (3.6) we used the function "micros()" to obtain the absolute time of the current function call. Subtracting the value of the previous function call, stored in the global variable "imuFilter_lastCall", we can obtain the current Δt_k. We also introduced a simple initialization to ensure the expected sampling time $\Delta t_k = 1ms$ is applied to the first computation.

```
unsigned long imuFilter_now = micros();
  double imuFilter_dt=0;
  if(imuFilter_lastCall == 0){
    imuFilter_dt = 1e-3;
  }else{
    imuFilter_dt = (double)(imuFilter_now - imuFilter_lastCall) / 1e6;
  }
  imuFilter_lastCall = imuFilter_now;

// [deg] angle estimated from gyroscope rate integrated over time
  measAngle.gyro = previousAngle.gyro + gyroRate * imuFilter_dt;
```

Listing 3.6: Adaptive Window Size

3.4.3 Serial Output Buffering

Typically, in simple Arduino projects, it is possible to print complex strings over the serial interface for debugging or event logging purposes. In time-critical applications, however, serial communication can introduce significant timing disturbances due to its asynchronous operation.

In our system, it is crucial that the microcontroller can read sensor data and compute PID output within a fixed time window. We observed that, when a "Serial.print()" call is inserted into this loop, the execution time can no longer be considered deterministic, varying with a range of several milliseconds. Although serial transmission is handled asynchronously via interrupts, the function may block if the transmit buffer becomes full, leading to unpredictable delays within the control loop.

Our solution to this problem is a custom buffer. During each iteration of the control loop, the useful data is written in this buffer instead of being transmitted directly. At the end of the loop, when the microcontroller would otherwise be idle, the buffered data is trasmitted. In our case, this simple strategy is enough to preserve deterministic loop timing.

Chapter 4

Physical Implementation

4.0.1 Mechanical structure

Moving from the theoretical model and control design to the actual implementation. We designed the structure of the robot using the CAD Solidworks and produced it using a 3D printer. The main structure of the robot is made of PETG material, while the wheels are made of TPU to ensure better grip on the ground.

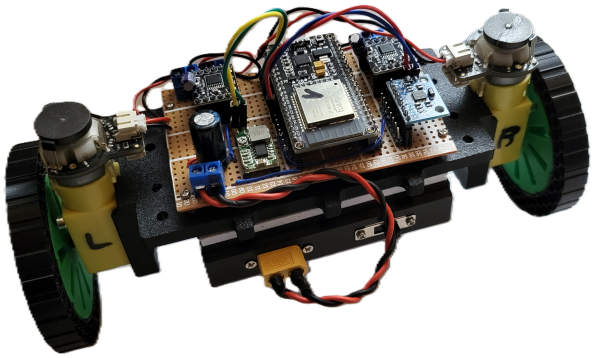

Figure 4.1: Robot

The complete structure is shown in Figure 4.1, while 4.2 shows different tire designs.

During some initial tests it was found that the limited weight of the robot was not enough to ensure it stayed well connected to the ground. To reduce the loss of traction on smooth surfaces, we had to change the tire structure. Unfortunately, changing material is not really feasible, as not many flexible filaments options are available other than 95A TPU. Instead, we changed the tire density, lowering sparse infill to 15% and eventually 8%. To further add flexibility, the shoulder of the tire had to be increased too, resulting in a larger diameter. Thanks to this simple solution, the tire deforms enough under torque to stay well connected to smooth tabletops.

4.0.2 PID adjustments

The PID controller parameters obtained in Chapter 2 are a good starting point for the real implementation, but due to the simplifications made in the model and the non-

Figure 4.2: Different tire geometries tried for the robot. Left to right for lower infill density and higher friction coefficient

idealities of the real components, it is necessary to adjust them experimentally. After several tests, the final parameters used in the firmware are:

- $K_P = 10$
- $K_I = 0.02$
- $K_D = 0.4$

These values ensure a good balancing performance, with a fast response to disturbances and minimal oscillations around the vertical position.

4.0.3 Other adjustments

Other parameters that needed to be adjust experimentally were for example the full-scale range of the accelerometer: we experinced that setting it to $\pm 4g$ ensured a good compromise between sensitivity and range of measurement, while setting it to $\pm 2g$ resulted in a high noise measurement.

In order to use the robot without the need of a computer connected via USB, we inserted a simple buck converter to power the ESP32 from the batteries.

Chapter 5

Conclusions

In this report, we presented the design and implementation of a self-balancing robot using an ESP32 microcontroller and an MPU6050 IMU sensor. We modelled the system as an inverted pendulum and designed a PID controller to maintain balance. The physical implementation involved selecting appropriate hardware components, designing a mechanical structure, and tuning the controller parameters experimentally. The final robot successfully demonstrated the ability to balance itself and respond to disturbances, validating our modelling and control approach. Future work could involve exploring more advanced control strategies or implementation of a wireless interface for remote monitoring and control.

In this project we were able to apply theoretical concepts learned in class but also to gain practical experience in analyzing and solving real-world engineering problems.

Appendix A

MATLAB code

In this appendix the main MATLAB code used for the modelling and controller design of the self-balancing robot is reported.

```
%% Self-balancing robot parameters

% Using stand-alone model of DC motor and inverse pendulum to
    derive transfer function
clc;
clearvars;
% Inverse pendulum data
% [Kg] total mass, considering also motors and battery
m = 0.1;
% [m/s^2] gravity acceleration
g = 9.81;
% [m] length of the pendulum
l = 0.08;
% [Kg*m^2] moment of inertia of inverse pendulum
J = m*l^2;

% DC motor parameters
% [ohm] measured resistance
R = 3.5;
% [H] estimated from comparison with similar motors
L = 50e-6;
% [V*s] torque costant (Va_max-Ra*Ia)*1/Omega_max
KePhi = 0.336;
% gear ratio
k_gear = 120;

%% Transfer functions
s = tf('s');
% between torque to angle of the inverse pendulum
G_torq_ang = (1/J) *(1/(s^2 - g/l));
```

```
figure('Name', 'Transfer function of the inverse pendulum');
bode(G_ang_torq);

% between voltage to torque of the DC motor
G_volt_torq = KePhi * k_gear/(R + L*s);
figure('Name', 'Transfer function of the DC motor');
bode(G_torq_volt);

% total transfer function
G_sys = G_volt_torq*G_torq_ang;

%% Requirements: tempo di salita e overshoot
% rise time
ts = 50e-3;
% overshoot choosen to be under a safe margin of maximum
    voltage of DC motors, that is 7.5V
M = 0.15;

% Crossover frequency and phase margin
fct = 2/(ts*2*pi);
pmt = 1.04 - 0.8*M;

% PID tuning options
opt = pidtuneOptions;
opt.PhaseMargin = 180*pmt/pi;

% PID controller design, with filter to reduce noise in
    derivative action
gi = pidtune(G_sys, 'pidf', 2*pi*fct, opt);

% Regulator transfer function
G_reg = gi.Kp + gi.Ki/s + gi.Kd*s/(1+s*gi.Tf);

fprintf('PID parameters: Kp %.4f, Ki %.4f, Kd %.4f\n', gi.Kp,
    gi.Ki, gi.Kd);

[Nreg, Dreg] = tfdata(G_reg, 'v');
margin(G_reg*G_sys);
grid on;

%% Step response of the closed loop system
G_cl = feedback(G_reg*G_sys, 1);
figure('Name', 'Step response of the closed loop system');
stepSetting = RespConfig('Amplitude', 20, 'Bias', -20);
step(G_cl, stepSetting);
```

```
72 title('Step response of the closed loop system');
73 grid on;
74 hold on;
75 ylabel("Angle (degrees)");
76 xlabel("Time");
77 fontsize(gca, 13, 'points');
```

Listing A.1: MATLAB code for PID tuning

Appendix B

ESP32 code

```
// Load necessary libraries
#include <Arduino.h>
#include <Wire.h>
#include <MPU6050.h>
#include <PID_v1.h>

// Define pin numbers
#define SDA_PIN 21
#define SCL_PIN 22

#define DEBUG_PIN_1 2
#define DEBUG_PIN_2 0

#define M_R_forward 18
#define M_R_backward 17
#define M_L_forward 4
#define M_L_backward 16

// Target angle in degrees
#define PID_REFERENCE 0
#define CRASH_TRESHOLD 60

#define CH_M_R_forward 0
#define CH_M_R_backward 1
#define CH_M_L_forward 2
#define CH_M_L_backward 3


#define ACC_FILTER_ALPHA 0.2 // LPF coefficient for
   ACCELEROMETER reading
#define FUSION_FILTER_ALPHA 0.98

```

```
32
33 // STATE MACHINE STATES
34 typedef enum{
35   STATE_INIT,
36   STATE_MEASURE_ANGLE,
37   STATE_COMPUTE_PID,
38   STATE_DRIVE_MOTORS,
39   STATE_RESTART,
40   STATE_CRASHED
41 }RobotState;
42
43 struct AccFilter{
44   double  ax_f;  // ax filtered value
45   double  ay_f;  // ay filtered value
46   double  az_f;  // az filtered value
47   double  alpha;  // filter coefficient
48 };
49
50 struct GyroFilter{
51   double  ax_f;  // ax filtered value
52   double  ay_f;  // ay filtered value
53   double  az_f;  // az filtered value
54   double  alpha;  // filter coefficient
55 };
56
57 AccFilter accFilter = {0,0,0,ACC_FILTER_ALPHA};
58
59 struct EstimatedAngle{
60   double  acc;     // [deg] accelerometer estimated angle
61   double  gyro;    // [deg] gyroscope estimated angle
62   double  fusion;  // [deg] estimated angle using sensor fusion
63   double  gyroRate; // [deg/s] gyro rate of change of the angle
64 };
65 EstimatedAngle angle = {0.0,0.0,0.0};
66
67 // SERIAL PRINTING
68 struct Sample {
69     double angleAcc;
70     double angleGyro;
71     double error;
72     double control;
73     double rateGyro;
74 };
75 Sample buf[512];
76 volatile int wptr = 0;
```

```
77 volatile int wptr_old = 0;
78
79 /* Define PID parameters
80  Input => Measured value
81  Output => Voltage applied to motor
82  Setpoint => Desired value of angle, stable at 0 degrees */
83  double Setpoint, Input, Output;
84
85 // PID tuning parameters
86
87 uint16_t sampleTime = 4; //sampletime in ms
88 double Kp=10, Ki=0.02, Kd=0.4; //08/12/2025 15:05
89 // Create an MPU6050 object
90 MPU6050 mpu;
91
92 // Create a PID controller object
93 PID controller(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);
94
95 // Variables for angle calculations
96 double pitchAcc, pitchGyro, rateGyro;
97
98 // Complementary filter
99 unsigned long imuFilter_lastCall=0;
100
101 // Complementary filter parameters
102 double imuFilter_alpha = 0.93; // Complementary filter
      coefficient -> higher value gives more weight to gyroscope
103
104 // PWM definition at 8 bit
105 // Impostazioni PWM
106 const int pins[] = {M_R_forward, M_R_backward, M_L_forward,
      M_L_backward};
107 const int channels[] = {CH_M_R_forward, CH_M_R_backward,
      CH_M_L_forward, CH_M_L_backward};
108 const int freq = 100;//19531;  // Frequenza massima per 12 bit
      (80 MHz / 4096)
109 const int res = 8;
110
111 hw_timer_t *Timer0_Cfg = NULL;
112 portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;
113 volatile bool tick = false;
114 uint16_t time_count = 0;
115
116 RobotState currentState = STATE_INIT;
117
```

```
118 EstimatedAngle angleEstimation(EstimatedAngle previousAngle);
119 void motorControl(int16_t pwm);
120 void PIDresponse(double angle_deg);
121
122 void IRAM_ATTR Timer0_ISR(){
123   portENTER_CRITICAL_ISR(&timerMux);
124   tick = true;
125   portEXIT_CRITICAL_ISR(&timerMux);
126 }
127
128
129 void setup() {
130
131   pinMode(SDA_PIN, INPUT);
132   pinMode(SCL_PIN, INPUT);
133   pinMode(M_R_forward, OUTPUT);
134   pinMode(M_R_backward, OUTPUT);
135   pinMode(M_L_forward, OUTPUT);
136   pinMode(M_L_backward, OUTPUT);
137   pinMode(DEBUG_PIN_1, OUTPUT);
138   pinMode(DEBUG_PIN_2, OUTPUT);
139
140   Serial.begin(115200);
141   Wire.begin();
142
143   Serial.println("MPU6050 Initialization...");
144   mpu.initialize();
145
146   if (!mpu.testConnection()) {
147     Serial.println("Unable to connect to MPU6050!");
148     while (1);
149   }
150   Serial.println("MPU6050 connected succesfully!");
151
152   mpu.setFullScaleGyroRange(MPU6050_GYRO_FS_250); // set gyro
      range to maximum 250 degrees (best resolution)
153   mpu.setFullScaleAccelRange(MPU6050_ACCEL_FS_4); // set
      accelerometer range to maximum 4g
154
155   mpu.CalibrateAccel(6);  // 6 samples for calibration
156   mpu.CalibrateGyro(6);
157
158   Serial.println("Initial Calibration Completed!");
159   mpu.PrintActiveOffsets();  // Mostra gli offset calcolati
160
```

```
161
162   // Imposta il setpoint iniziale
163   Setpoint = PID_REFERENCE; // Target angle in degrees
164   controller.SetMode(AUTOMATIC); // Attiva il PID controller
165
166   //calibrateGyroBias();
167   Serial.println("Regulator Setup Completed!");
168
169   // Configura i canali PWM
170   for (int i = 0; i < 4; i++) {
171     ledcSetup(channels[i], freq, res);
172     ledcAttachPin(pins[i], channels[i]);
173   }
174
175   Timer0_Cfg = timerBegin(0, 80, true); // 80e6/80 = 1e6 => 1
      tick = 1 us
176   timerAttachInterrupt(Timer0_Cfg, &Timer0_ISR, true);
177   timerAlarmWrite(Timer0_Cfg, 1000, true); //1000*1us = 1ms
178   timerAlarmEnable(Timer0_Cfg);
179
180   delay(1000);
181
182   currentState = STATE_MEASURE_ANGLE;
183
184 }
185
186 void loop() {
187   if(tick == true){
188     portENTER_CRITICAL(&timerMux);
189     tick = false;
190     portEXIT_CRITICAL(&timerMux);
191
192     //if(time_count == 10){
193       time_count = 0;
194       switch(currentState){
195         case STATE_MEASURE_ANGLE:{
196
197           angle=angleEstimation(angle);
198           if(angle.acc>CRASH_TRESHOLD || angle.acc<-
      CRASH_TRESHOLD){
199             currentState = STATE_CRASHED;
200           }else{
201             currentState = STATE_COMPUTE_PID;
202           }
203           digitalWrite(DEBUG_PIN_1, 1);
```

```
204           break;
205         }
206         case STATE_COMPUTE_PID:{
207           PIDresponse(angle.fusion);
208           currentState = STATE_DRIVE_MOTORS;
209           break;
210         }
211         case STATE_DRIVE_MOTORS:{
212           motorControl(Output);
213           digitalWrite(DEBUG_PIN_1, 0);
214           buf[wptr] = {angle.acc, angle.gyro, angle.fusion,
    Output, angle.gyroRate}; // buffer for serial print
215           wptr = (wptr + 1) % 512;
216           currentState = STATE_MEASURE_ANGLE;
217           break;
218         }
219         case STATE_CRASHED:{
220           motorControl(0);
221           PIDresponse(0);
222           buf[wptr] = {angle.acc, angle.gyro, angle.fusion,
    Output, angle.gyroRate}; // buffer for serial print
223           wptr = (wptr + 1) % 512;
224           //Serial.println("\nCRASHED!\n");
225           currentState = STATE_MEASURE_ANGLE;
226           break;
227         }
228       }
229     }
230     //Serial.println("Angle: " + String(angle) + " Pitch Acc "
    + String(pitchAcc) + " Pitch Gyro " + String(pitchGyro) + "
    PID Output "+ String(Output)+",");
231     //Serial.println(String(angle) +","+ String(pitchAcc) +","+
     String(pitchGyro) +","+ String(Output)+" ");
232
233     if((wptr != wptr_old) && (wptr%10==0)){
234       Serial.print((int16_t)(buf[wptr].angleAcc));
235       Serial.print(",");
236       Serial.print((int16_t)(buf[wptr].angleGyro));
237       Serial.print(",");
238       Serial.print((int16_t)(buf[wptr].error));
239       Serial.print(",");
240       Serial.print((int16_t)(buf[wptr].control));
241       Serial.print(",");
242       Serial.print((int16_t)(buf[wptr].rateGyro));
243       Serial.print("\n");
```

```
244         wptr_old = wptr;
245       }
246
247       //time_count++;
248     }
249 //}
250
251 // Function to estimate angle using complementary filter
252 EstimatedAngle angleEstimation(EstimatedAngle previousAngle){
253   int16_t ax=0, ay=0, az=0, gx=0, gy=0, gz=0;
254   double ax_d=0, ay_d=0, az_d=0;
255   double gyroRate = 0;
256
257   EstimatedAngle measAngle;
258   mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz); // Legge i dati
        grezzi da MPU6050
259
260   ax_d = (double)ax * 0.002394202; // (9.8/4096) for 8g
       acceleration range
261   ay_d = (double)ay * 0.002394202;
262   az_d = (double)az * 0.002394202;
263
264   accFilter.ax_f += accFilter.alpha * (ax_d - accFilter.ax_f);
265   accFilter.ay_f += accFilter.alpha * (ay_d - accFilter.ay_f);
266   accFilter.az_f += accFilter.alpha * (az_d - accFilter.az_f);
267
268   // Estimation pitch angle (y-axis) accelerometer
269   measAngle.acc = atan2(ay_d, az_d)*(360/PI); // angle
       estimated from accelerometer (FILTERED)
270
271   // Estimation pitch angle (y-axis) gyroscope
272   gyroRate = (double)gx * (1.0/131); // [deg/s]
273
274   // This block finds the elapsed time between the current and
       the previous function call to determine the integration
       interval
275   unsigned long imuFilter_now = micros();
276   double imuFilter_dt=0;
277   if(imuFilter_lastCall == 0){
278     imuFilter_dt = 1e-3;
279   }else{
280     imuFilter_dt = (double)(imuFilter_now - imuFilter_lastCall)
        / 1e6;
281   }
282   imuFilter_lastCall = imuFilter_now;
```

```
283
284   measAngle.gyro = previousAngle.gyro + gyroRate * imuFilter_dt
      ; // [deg] angle estimated from gyroscope rate integrated
      over time
285
286   // Complementary filter to combine accelerometer and
      gyroscope data
287   measAngle.fusion = FUSION_FILTER_ALPHA * (previousAngle.
      fusion + gyroRate * imuFilter_dt) + (1 - FUSION_FILTER_ALPHA
      ) * measAngle.acc;  //dt =0.03 max limit for oscillations
288   measAngle.gyroRate = gyroRate;
289   return measAngle;
290 }
291
292 // PID response function
293 void PIDresponse(double angle_deg){
294 // Input of PID will be the y-axis angle
295   Input = angle_deg; //* PI/180; // in radians
296   controller.Compute(); // Calcola il nuovo output del PID
297   controller.SetOutputLimits(-255, 255);//(-4095, 4095); //
      Limita l'output tra -4095 e 4095 (12 bit)
298   controller.SetSampleTime(sampleTime);
299 }
300
301  // Control motors based on PID output
302 void motorControl(int16_t pwm){
303
304   if (Output > 0) {
305     // Move forward
306     ledcWrite(CH_M_R_forward, pwm);//analogWrite(M_R_forward,
      Output);
307     ledcWrite(CH_M_R_backward, 0);
308     ledcWrite(CH_M_L_forward, pwm);
309     ledcWrite(CH_M_L_backward, 0);
310   }
311   else if (Output < 0) {
312     // Move backward
313     ledcWrite(CH_M_R_forward, 0);
314     ledcWrite(CH_M_R_backward, -pwm);
315     ledcWrite(CH_M_L_forward, 0);
316     ledcWrite(CH_M_L_backward, -pwm);
317   }
318   else {
319     // Stop
320     ledcWrite(CH_M_R_forward, 0);
```

```
321         ledcWrite(CH_M_R_backward, 0);
322         ledcWrite(CH_M_L_forward, 0);
323         ledcWrite(CH_M_L_backward, 0);
324     }
325 }
```

Listing B.1: Arduino code for ESP 32