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
DRV8871	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.

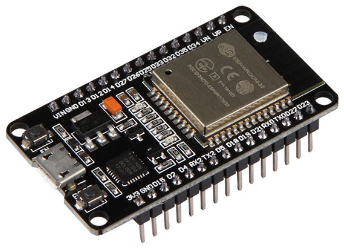

Figure 1.1: ESP32 development board

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

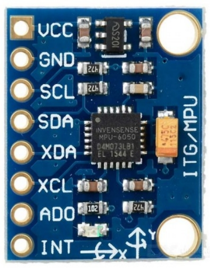

Figure 1.2: MPU6050 Inertial Measurement Unit (IMU) sensor

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 $250\,°/s$ 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 DRV8871

To control the DC motors, we use the DRV8871 motor driver (shown in Figure 1.4), which is capable of handling motor supply voltages from 6.5V to 45V. It's a double full-bridge driver, allowing for bidirectional control of the motors.

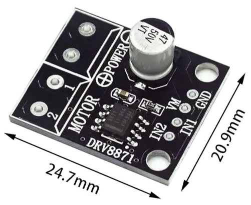

Figure 1.4: DRV8871 motor driver

It's capable of delivering up to 3.6A continuous current per channel, which is more than sufficient for our DC motors.

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

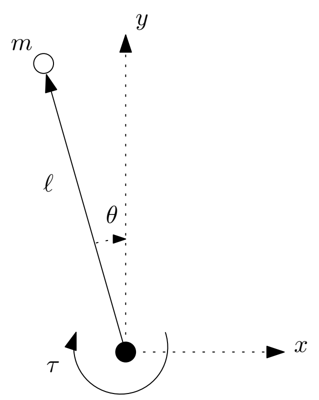

Figure 2.1: Inverse pendulum schematic representation

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 2.2 the complete block diagram is reported, where $G(s) = G_{pendulum}(s) \cdot G_{motor}(s) = \frac{\theta(s)}{v(s)}$.

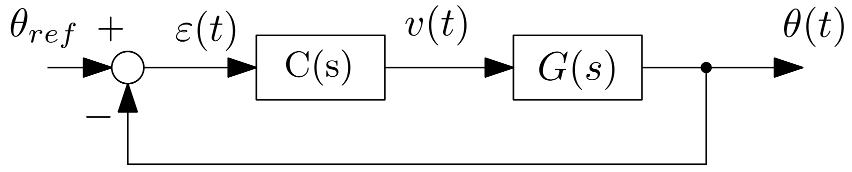

Figure 2.2: Complete block diagram of the self-balancing robot model

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:

```matlab
% 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);
```

Listing 2.1: MATLAB code for PID tuning

The obtained parameters are:

- $K_P = 18.60$
- $K_I = 1.15$
- $K_D = 0.065$
- $T_f = 10ms$

Chapter 3

Control

3.1 High Level System Description

3.2 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.

```c
typedef enum{
  STATE_INIT,
  STATE_MEASURE_ANGLE,
  STATE_COMPUTE_PID,
  STATE_DRIVE_MOTORS,
  STATE_RESTART,
  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. RESTART: used only at low level to take care of system critical conditions such as parameter overflow.
6. 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

3.3 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.

3.3.1 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:

$$t_{\text{tick}} = \frac{1}{f_{\text{timer}}} = \frac{\text{prescaler}}{f_{\text{CPU}}} = \frac{80}{80\,\text{MHz}} = 1\ \mu\text{s}. \tag{3.1}$$

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:

$$T_s = t_{\text{tick}} \times 1000 = 1\ \mu\text{s} \times 1000 = 1\ \text{ms}. \tag{3.2}$$

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

3.3.2 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 $\theta(t)$.

$$\theta(t) = \theta_0 + \int_0^t \omega(\tau)\, d\tau \tag{3.3}$$

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

$$\theta[k] = \theta[k-1] + \omega[k]\, T \tag{3.4}$$

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.

$$\theta[k] = \theta[k-1] + \omega[k]\, \Delta t_k \tag{3.5}$$

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

3.3.3 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.

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.

```matlab
%% 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, Tf %.2f [
    msec] \n', gi.Kp, gi.Ki, gi.Kd, gi.Tf*1e3);

[Nreg, Dreg] = tfdata(G_reg, 'v');
margin(G_reg*G_sys);
grid on;
```
Listing A.1: MATLAB code for PID tuning