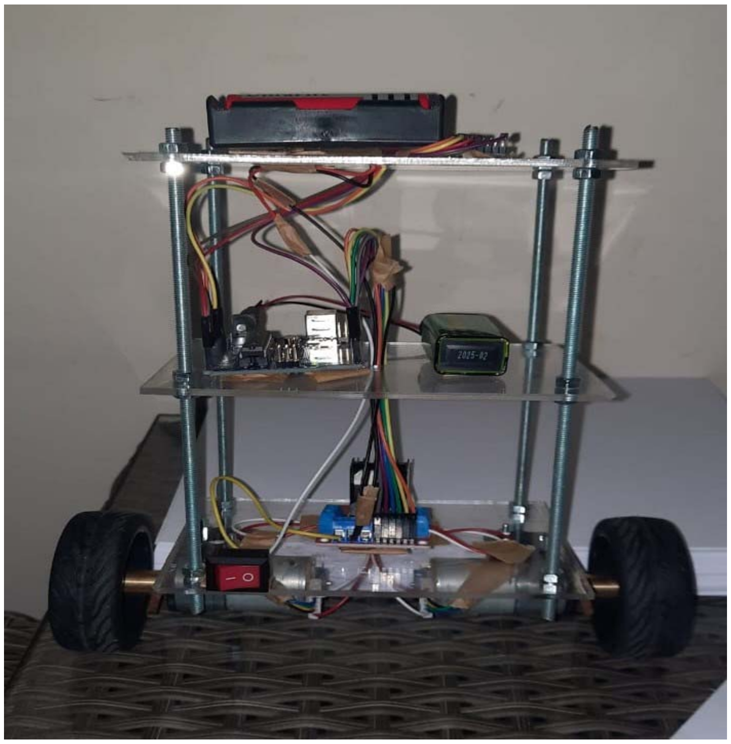
**<u>MCE410</u> - Mechatronics System Design I**

Section 34

**<u>Instructor:</u>** Dr.Bilal Komati

**<u>Final Project:</u>** Self-Balancing Robot



Joe Rahi # 202004912

8 December 2022

# Table of Contents

# Table of Figures

## Introduction:

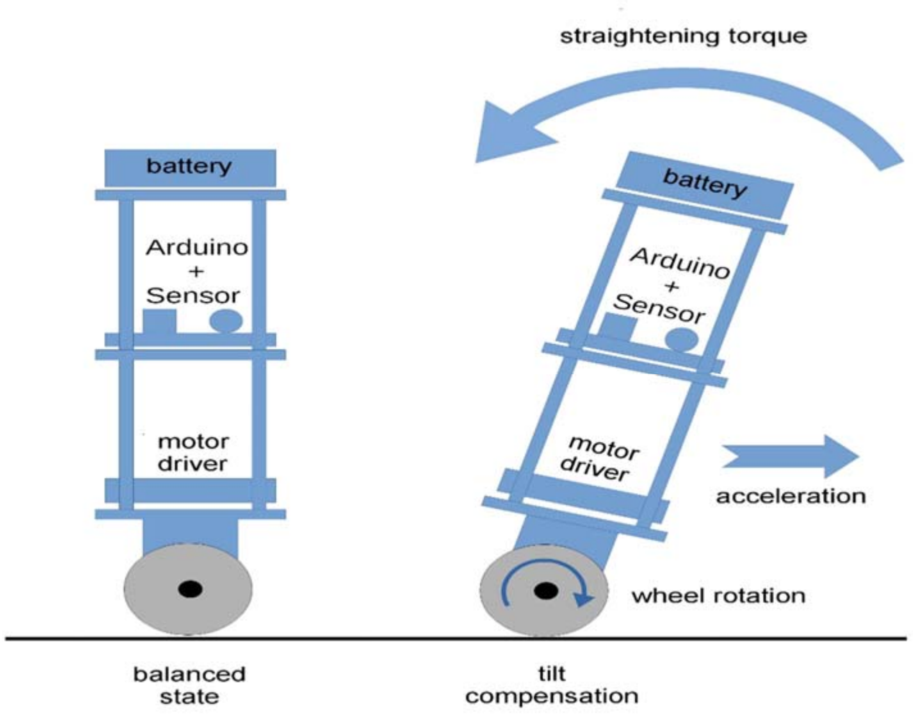


*Figure 1: Self balancing robot*

Self-balancing robot is based on the concept of inverted pendulum which itself explore the unstable dynamics system that makes it different from other robots. A two-wheel self-balancing robot is characterized by its ability to adjust itself with respect to changes in weight or position by the implementation of a closed loop algorithm. In order to maintain balance, a gyroscope and accelerometer in form of Industrial Measurement Unit (IMU) is used to deviate the angle of the robot and it is used to increase or decrease the speed of the motors with the goal of maintaining a constant angle. Its ability to turn on the spot and sustainable architecture increases its application in industries. Furthermore, it is essential for the robot to withstand external forces or unexpected disturbances. Some of the balancing applications include Segways, bipedal robots and space rockets. The focus of this project is to implement a Mechatronics Project that combines a Mechanical, electrical, and coding which in our case is the to balance and control a self-balancing robot.

## Source of Inspiration:

Much research has been done on this project and based on each project we combined the better and efficient design:

Links:

Awesome self-balancing robot - Mechatronics Exercises - Aalto University Wiki

Arduino SELF-BALANCING Robot - Arduino Project Hub

DIY Self Balancing Robot using Arduino (circuitdigest.com)

Building an Arduino-based self-balancing robot – Part 3 – Robotic Dreams (wordpress.com)

(48) How to Make Arduino Self Balancing Robot - YouTube

(48) Arduino MPU6050 GY521 6 Axis Accelerometer + Gyro (3D Simulation with Processing) - YouTube

The PID algorithm was used in most of the self-balancing robot projects that we found.

## Project Specification:

To implement this project many specifications should be taken into consideration:

- To derive a dynamic equation based on the theory of inverted pendulum.
- Form transfer function for the angle “psi” (the angle between the oblique line of the slip and the vertical line) deviation and position “x”
- Find a controller that can control these two conditions
- Set up requirements for the demonstrator
- Design a demonstrator that fulfils these essential requirements, investigate the boundaries of the control signal.

In our case these requirements were solved, that has simplified the process to the new specifications starting by the components needed such as two motors, a chassis, two wheels, a dual motor driver to control the spin direction of the motors, and Arduino board, DC power supply, a Gyro with accelerometer known as MPU, a switch button(optional) and jumper wires. It is more recommended to use a PCB board instead of wiring since wiring will affect the inertia position of the mechanical system design. An Arduino PID algorithm code is needed to control this project, taking into account the speed control PWM and the necessary constants to balance the robot.

## Design:

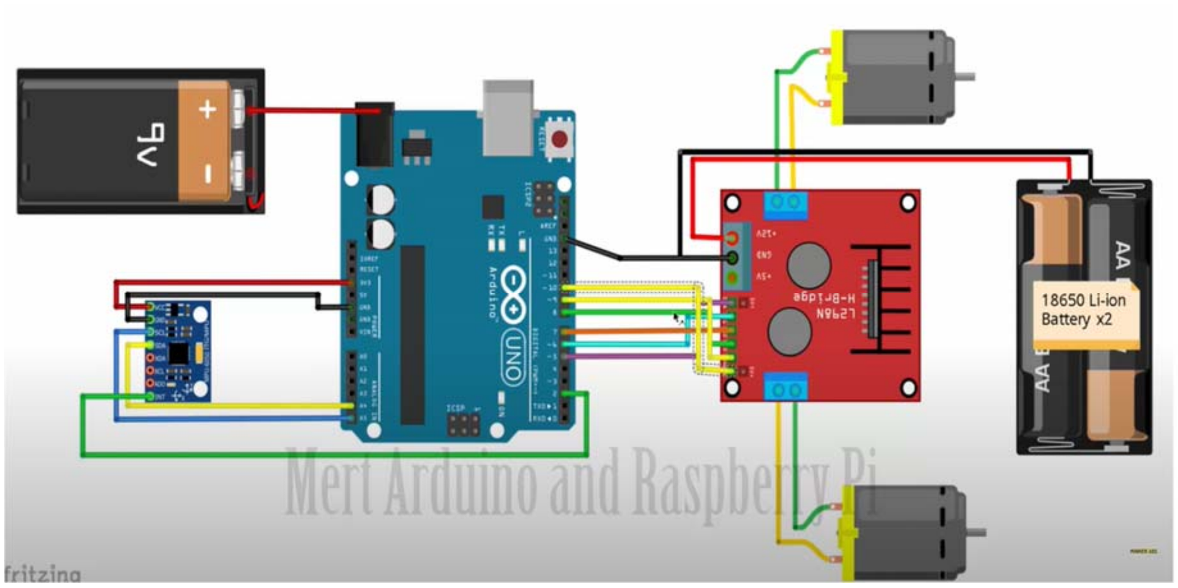


*Figure 2: Circuit Diagram of self-balancing robot*

This figure represents the circuit diagram that was implemented in our design, we could have connected the Arduino board to the 3.7 x 2 = 7.4V DC, but since after many researches we have found that Arduino board can be energized by 5V but in order for the board to be stable it is better if we connect it to a 9 to 12 V (Recommended Range) .

In addition, we have added a switch for this circuit to make it turn ON and OFF when needed. The switch was connected to the positive wire of the Li-ion Battery and to the input of the current to the H-bridge.

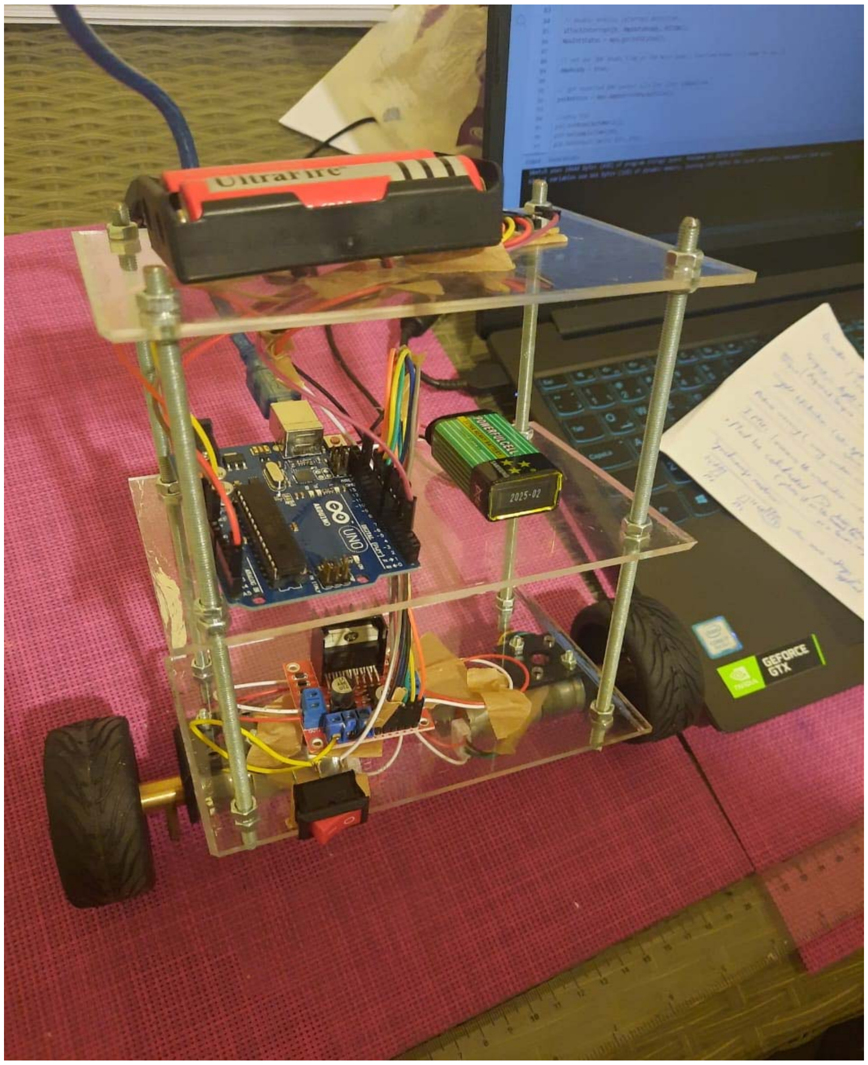


*Figure 3: Project wiring*

## Proposed Solution:

The design of the robot is case sensitive since we have to consider the weight distribution between the left and the right sides to create a weight balance, placing our components to exact center of the chassis will have a positive impact on the inertia position of the system and on our design. To clarify we have to design the mechanical part in a way that the inertia should be approximating the center of the system dimensions. Furthermore, these two factors of the design will lead to the motor selection, which using the following formula the torque can be calculated:

Torque = height x weight x gravity x sin (max tilt)

= 0.2m x2kgx9.81g/m^2 x sin(5) = 0.3419N.m

To be on the safe side:

Required Torque = torque x 2 which is the min of the motors torque

For our design Dc motors were selected since they have higher efficiency, they have a low energy consumption, and it does not heat fast when it is operating. In addition, we have to synchronize our motors since due to small manufacturing imperfections identical motors might have different velocities, if one of them is a bit slower we have to multiplying its input inside the Arduino sketch, so they can have the same rpm when the same voltage is applied. It is also a good application to reduce the wiring by using PCB boards which will reduce the complexity and can help to locate the problem in an easier way. The IMU is the component responsible for the perception in the self-balancing robot which measure the acceleration which can be used to calculate the inclination angle, for this case IMU must be calibrated using the Arduino calibration sketch, making sure that the MPU will not be disturbed by any movement to obtain a good calibration, and it should be well attached to the chassis because when the robot starts shacking during the demonstration, any change in the position if the sensor will lead to a catastrophic result.

## Hardware:

| **Component** | **quantity** |
|---|---|
| 6V 210 rpm encoder motor | 2 |
| wheels (1/10 RC Racing Wheels 62.5mm) | 2 |
| Arduino uno R3 | 1 |
| L298N Dual H-bridge Motor Driver Board | 1 |
| MPU 6050 3Axis Gyro with Accelerometer | 1 |
| 9V battery | 1 |
| 18650 Cell Box (x2) | 1 |
| 3.7 V rechargeable batteries | 2 |
| coupling (link wheel motor) | 2 |
| mounting bracket (link chassis-motor-coupling) | 2 |
| Jumper wires | 1 |
| Plexil Sheet (Laser cut) (15cmx10cmx3mm) | 3 |
| Distance Screws of 1 meter cut into (L=19.5cm, d=5mm) | 4 |
| Electrical Switch | 1 |

*Figure 4: Components:*

## Method to Control the System:

The control algorithm that was used to maintain the balance of the two-robot was the PID controller. The proportional (P), integral(I), and derivative(D) controller is well known as a three-term controller.

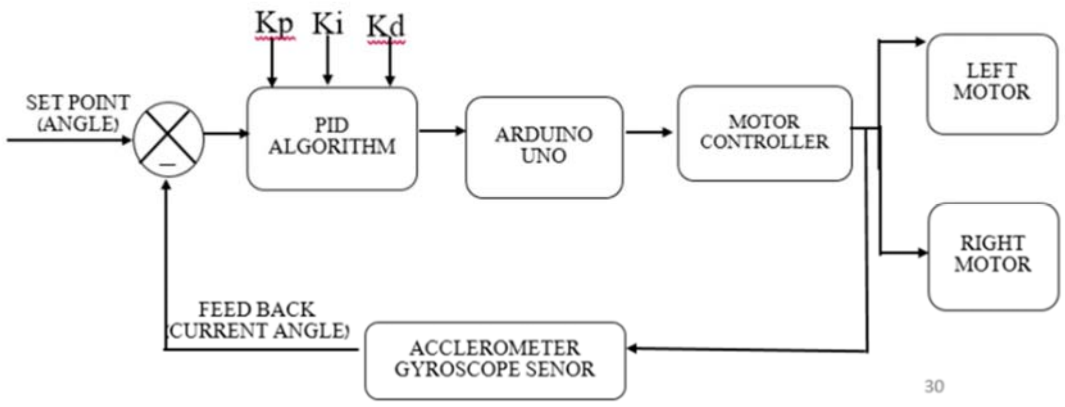


*Figure 5: Control Block Diagram*

The input to the controller is the error from the system. The Kp, Ki, and Kd are referred to as the proportional, the integral, and the derivative constants. The closed loop control system is also referred to give negative feedback. A negative feedback system works by monitoring the process output "assume Y" from a sensor. The measured process output gets subtracted from the reference set-point value to produce an error. The error is then fed into the PID controller, where the error gets managed in three ways. The error will be used on the PID controller to execute the proportional term, integral term for reduction of steady state errors, and the derivative term to handle overshoots.

The controller generates a signal "assume U" after the PID algorithm analyzes the error. The PID control signal then is fed into the process under control. The process under PID control is the two wheeled robot. The PID control signal will try to drive the process to the desired reference setpoint value. In the case of the two-wheel robot, the desired set-point value is the zero-degree vertical position.

The PID control algorithm can modelled mathematically to calculate the correction term:

**Correction** = Kp *error + Ki*$\int error$ + Kd* $\frac{d(error)}{dt}$

Where Kp, Ki, and Kd are the constants set experimentally.

Each of these terms has a particular and visual task described as follows:

- If only the first term had been used to calculate the correction, the robot would have acted similarly to the classical line following algorithm.
- The second term forces the robot to move toward the mean position faster.
- The third term particularly resists sudden changes in deviation.

Furthermore, the integral is the total error since it is the summation of all previous deviations, and the derivative is the difference between the current deviation and the previous deviation.

In addition to the control, Pitch, Roll, Yaw represents the moments about the X,Y,Z axes. In order to calibrate the MPU sensor to zero we were able to read the data using the code, then we have fixed the MPU sensor in a place where there is not motion, then we started analysing and approximating the offset of the gyroscope that we will update in the original code.

Furthermore, stabilized the speed (PWM) and controlled the spin directions of the motors. The value of the PWM signal and its direction was totally based on the analog values given by the sensor to Arduino. The greater the difference the more is the PWM. The direction of the PWM was determined by generating positive or negative using the code generated using the PID control mechanism and by visualizing the spin direction of the wheels.

## Procedure for the experiment:

In order to simplify the implementation of this Mechatronics project, we decided to break the project into 3 subprojects: Mechanical, Electrical, and Coding.

First, we started by the mechanical part where we first manufactured some mechanical parts and then built the mechanical chassis which in our case is 3 layers separated by a distance screw of 8cm, each layer of the chassis is fixed between 2 hex-nuts on the 4 sides to make the design safer. Then, we attached 2 mounting brackets symmetrically on the lower layer in order to screw the motors and wheel linked to a coupling connected to the shaft of the motor.

Then we continued our working on the electrical part, were we started to design our circuit diagram, then implemented the H-bridge on the upper face of the 1st layer chassis and we started connecting the wires from the motors to the H-bridge passing through the middle hole of the chassis. In addition, we brought our Arduino board and fixed it in the middle of the 2nd layer of the chassis with a 9V battery to provide it with an outside power supply, then we started connecting the H-Bridge to the Arduino board. Furthermore, the battery holder was placed on the top of the chassis layers in a balancing way in order to conserve the inertia of our system approximating equilibrium. The MPU6050 sensor was then placed on the top near the battery's holder of 3.7x2 = 7.4V, and then we completed the connection of our circuit by connecting the MPU sensor to the Arduino board.

Finally, after revising the implementation of the circuit we analyzed the code of the self-balancing robot and started by improving and matching the requirement of our environment (The calibration process is explained in other part).

## Results and Analysis:

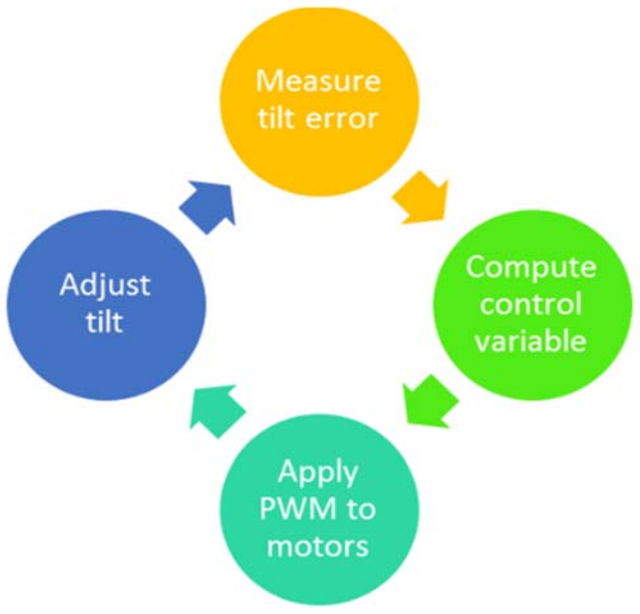
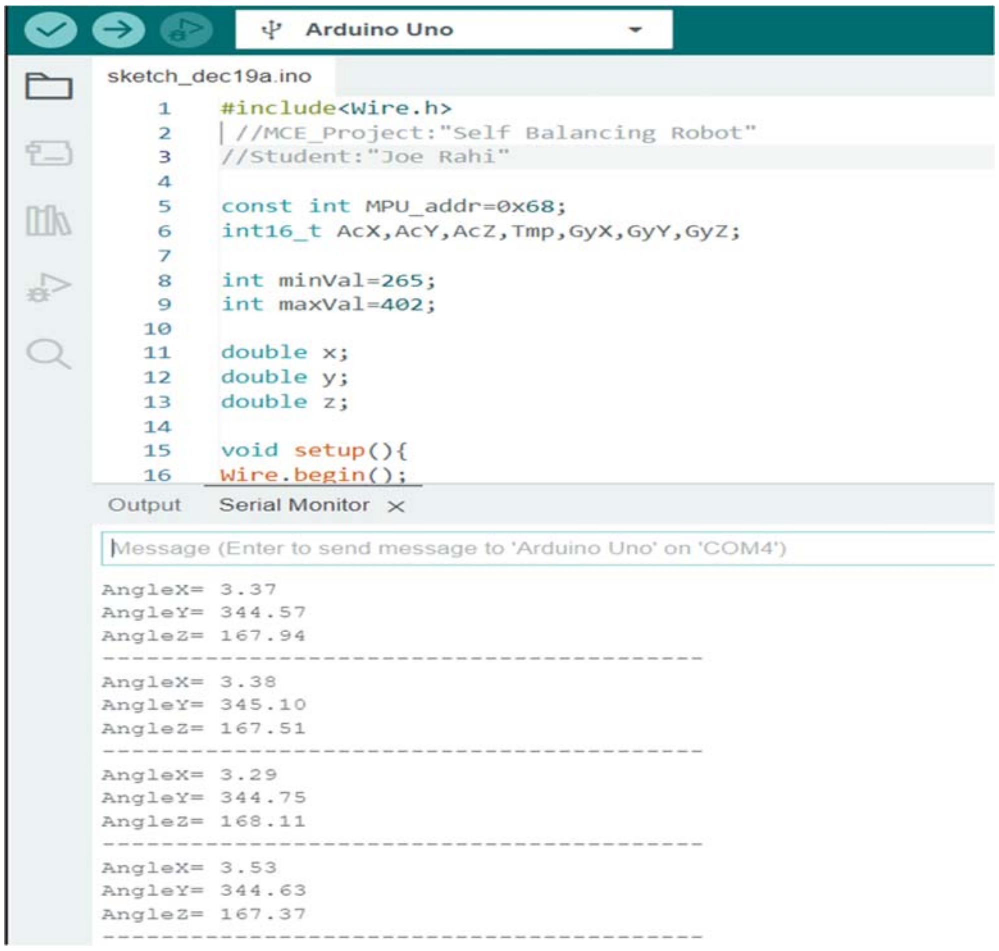


*Figure 6: Circular chain of the results*

The steps that lead us to the results were to compute the control variable after measuring the tilt, the apply the PWM to the motors to adjust the tils.

After calibrating the MPU sensor, we have fixed the robot in equilibrium the we have read the values of its 3D coordinates then we have updated these values in the original code and we also have controlled the speed of the motors using PWM, and after updated different times the values of Kp, Ki, and Kd, the robot were able to balance.



```
#include<Wire.h>
 //MCE_Project:"Self Balancing Robot"
//Student:"Joe Rahi"

const int MPU_addr=0x68;
int16_t AcX,AcY,AcZ,Tmp,GyX,GyY,GyZ;

int minVal=265;
int maxVal=402;

double x;
double y;
double z;

void setup(){
Wire.begin();
```

```
AngleX= 3.37
AngleY= 344.57
AngleZ= 167.94
-----------------------------------------
AngleX= 3.38
AngleY= 345.10
AngleZ= 167.51
-----------------------------------------
AngleX= 3.29
AngleY= 344.75
AngleZ= 168.11
-----------------------------------------
AngleX= 3.53
AngleY= 344.63
AngleZ= 167.37
-----------------------------------------
```

*Figure 7: measuring the MPU angles using Arduino*

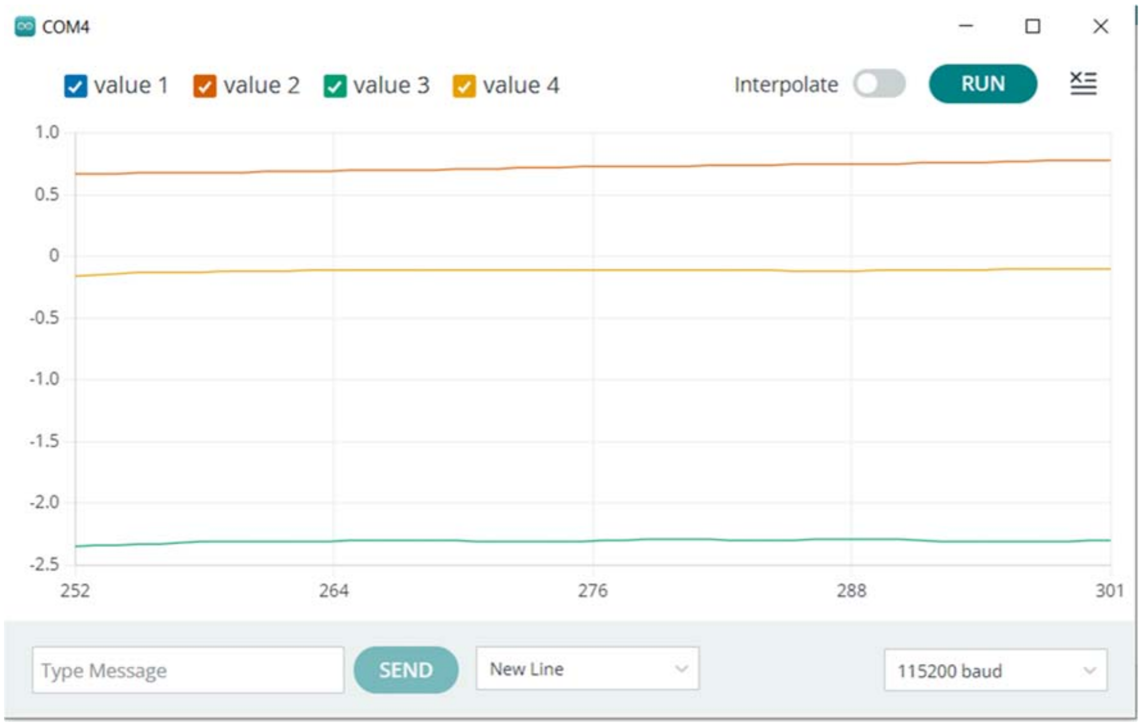


*Figure 8: Plotting angle values*

The plot represent the stable angle values, it seems that the angles are approximating constants when the robot is fixed, using these results to check the variation in angle when the offset of slip angle is 0.1 which at this value the robot will balance and will not exceed that angle.

After pushing the robot when he was in the balancing state, we can see that a pulse was energized in the angle measurement.

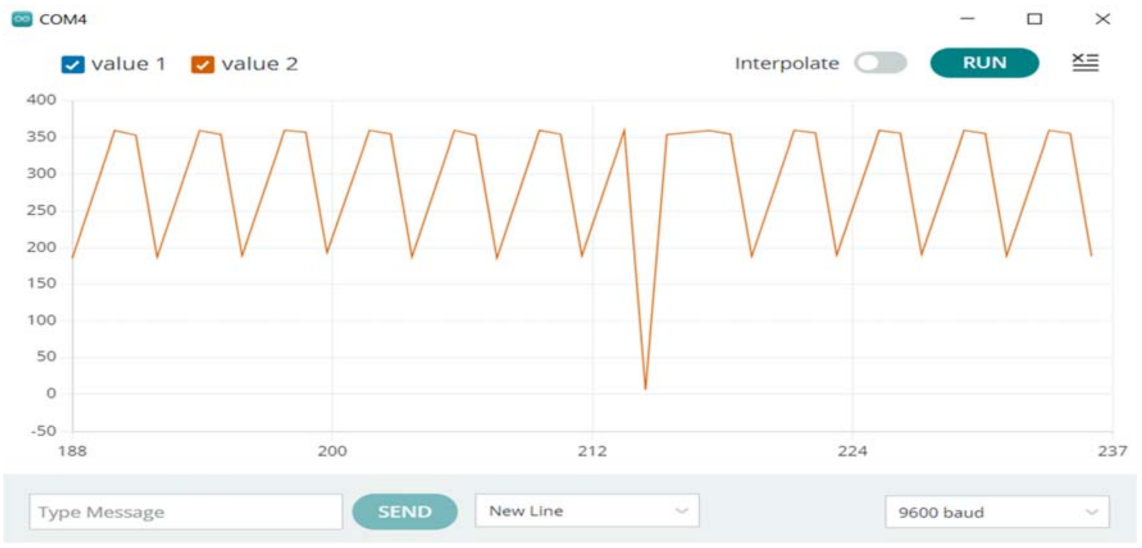


*Figure 9:Pulse angle*

## Conclusion:

To conclude, implementing a Self-Balancing Robot is a challenge since we have to make precautions in all the steps that should be made, any wrong changes either in the mechanical design or in the electrical design, specially motors will make the robot fall. We have faced many challenges of robot falling but after trying and studying different times the PID constant values with the motor constant speed, we have reached our target and made the robot balance itself.

Many improvements can be made but since I am alone in this project, I tried to do my best and to make this challenging robot balance successfully.

## Possible Amelioration:

The amelioration can always be possible in the MPU sensor calibration and in the PID constants to improve the equilibrium of the balancing robot since our design was based on the experimentation of the constants to balance the robot, but in industries and in difficult environments (space for example) this require the mathematical study of the system taking into account friction and challenges that may happen. Furthermore, the mechanical classification of the components can also be stabilized to maintain good inertia.

# Appendix:

Self-balancing robot original code:

(Note: Everything code is present in the files + Cup of water video Demo)

```
#include <PID_v1.h>
#include <LMotorController.h>
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"

#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
 #include "Wire.h"
#endif

#define MIN_ABS_SPEED 30

MPU6050 mpu;

// MPU control/status vars
bool dmpReady = false; // set true if DMP init was successful
uint8_t mpuIntStatus; // holds actual interrupt status byte from MPU
uint8_t devStatus; // return status after each device operation (0 = success,
!0 = error)
uint16_t packetSize; // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount; // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q; // [w, x, y, z] quaternion container
VectorFloat gravity; // [x, y, z] gravity vector
float ypr[3]; // [yaw, pitch, roll] yaw/pitch/roll container and gravity
vector

//PID
double originalSetpoint = 172.5;
double setpoint = originalSetpoint;
double movingAngleOffset = 0.1;
double input, output;

//adjust these values to fit your own design
double Kp =80;
double Kd = 2.2;
double Ki = 235;
PID pid(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);

double motorSpeedFactorLeft = 0.7;
double motorSpeedFactorRight = 0.5;

//MOTOR CONTROLLER
int ENA = 5;
int IN1 = 6;
int IN2 = 7;
int IN3 = 9;
int IN4 = 8;
int ENB = 10;
LMotorController motorController(ENA, IN1, IN2, ENB, IN3, IN4,
motorSpeedFactorLeft, motorSpeedFactorRight);

volatile bool mpuInterrupt = false; // indicates whether MPU interrupt pin has
gone high
void dmpDataReady()
{
 mpuInterrupt = true;
}


void setup()
{
 // join I2C bus (I2Cdev library doesn't do this automatically)
 #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
 Wire.begin();
 TWBR = 24; // 400kHz I2C clock (200kHz if CPU is 8MHz)
 #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
 Fastwire::setup(400, true);
 #endif

 mpu.initialize();

 devStatus = mpu.dmpInitialize();

 // supply your own gyro offsets here, scaled for min sensitivity
 mpu.setXGyroOffset(220);
 mpu.setYGyroOffset(76);
 mpu.setZGyroOffset(-85);
 mpu.setZAccelOffset(1788); // 1688 factory default for my test chip

 // make sure it worked (returns 0 if so)
 if (devStatus == 0)
 {
 // turn on the DMP, now that it's ready
 mpu.setDMPEnabled(true);

 // enable Arduino interrupt detection
 attachInterrupt(0, dmpDataReady, RISING);
 mpuIntStatus = mpu.getIntStatus();
 // set our DMP Ready flag so the main loop() function knows it's okay to use
it
 dmpReady = true;

 // get expected DMP packet size for later comparison
 packetSize = mpu.dmpGetFIFOPacketSize();

 //setup PID
 pid.SetMode(AUTOMATIC);
 pid.SetSampleTime(10);
 pid.SetOutputLimits(-255, 255);
 }
 else
 {
 // ERROR!
 // 1 = initial memory load failed
 // 2 = DMP configuration updates failed
 // (if it's going to break, usually the code will be 1)
 Serial.print(F("DMP Initialization failed (code "));
 Serial.print(devStatus);
 Serial.println(F(")"));
 }
}


void loop()
{
 // if programming failed, don't try to do anything
 if (!dmpReady) return;

 // wait for MPU interrupt or extra packet(s) available
 while (!mpuInterrupt && fifoCount < packetSize)
 {
 //no mpu data - performing PID calculations and output to motors
 pid.Compute();
 motorController.move(output, MIN_ABS_SPEED);

 }

 // reset interrupt flag and get INT_STATUS byte
 mpuInterrupt = false;
 mpuIntStatus = mpu.getIntStatus();

 // get current FIFO count
 fifoCount = mpu.getFIFOCount();

 // check for overflow (this should never happen unless our code is too
inefficient)
```

```
 if ((mpuIntStatus & 0x10) || fifoCount == 1024)
 {
 // reset so we can continue cleanly
 mpu.resetFIFO();
 Serial.println(F("FIFO overflow!"));

 // otherwise, check for DMP data ready interrupt (this should happen
frequently)
 }
 else if (mpuIntStatus & 0x02)
 {
 // wait for correct available data length, should be a VERY short wait
 while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

 // read a packet from FIFO
 mpu.getFIFOBytes(fifoBuffer, packetSize);

 // track FIFO count here in case there is > 1 packet available
 // (this lets us immediately read more without waiting for an interrupt)
 fifoCount -= packetSize;

 mpu.dmpGetQuaternion(&q, fifoBuffer);
 mpu.dmpGetGravity(&gravity, &q);
 mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
 input = ypr[1] * 180/M_PI + 180;
 }
}
```

Some Robot images (Demo Video in the Photo Files):

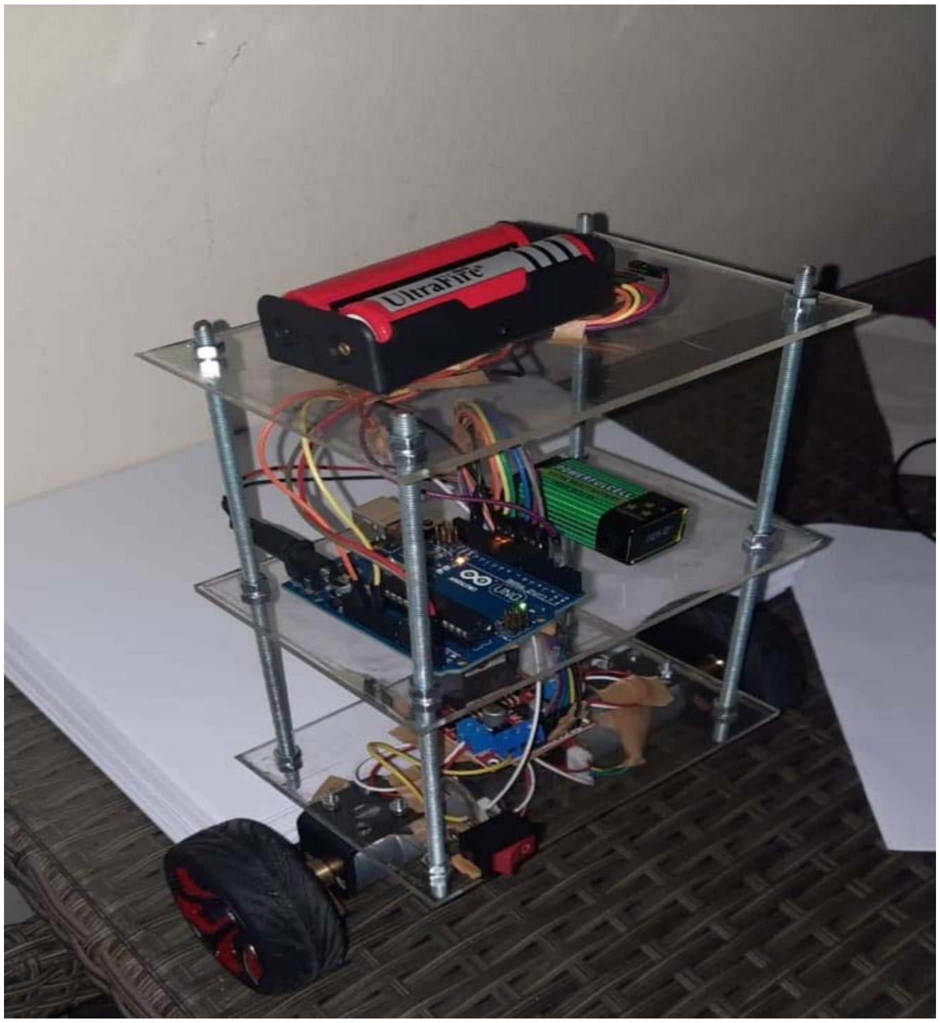


*Figure 10:Robot image*

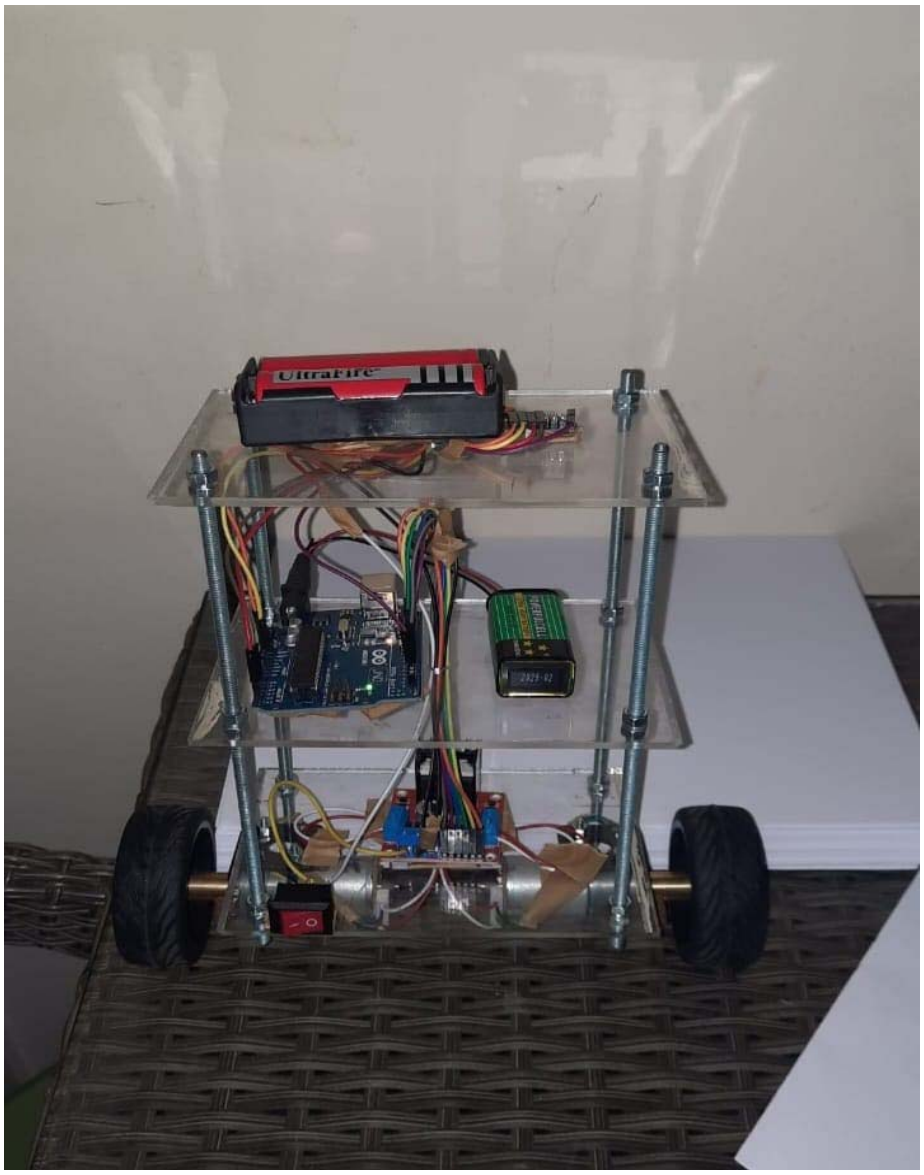


*Figure 11:Robot image*

## References (Datasheets):

[L298N Motor Driver.pdf (handsontec.com)]

[MPU-6050 Datasheet(PDF) - List of Unclassifed Manufacturers (alldatasheet.com)]

[A000066-datasheet.pdf (arduino.cc)]

[Metal DC Geared Motor w/Encoder - 6V 210RPM 10Kg.cm - DFRobot]