|  |  |
| --- | --- |
| A blue square with white text  Description automatically generated | **HOLY SPIRIT UNIVERSITY OF KASLIK**  **School of Engineering**  **Department of Electrical and Electronics Engineering**  **Department of Mechanical Engineering** |

Linear Control Systems Laboratory

GMC475/GEL477

Year 2023/2024

**Self-Balancing Robot Project**

*Presented to:*

Dr. Elias SALLOUM

*Presented by:*

Andrew DAGHER

Elie AL KOBERSSY

Samir DACCACHE

Karl FARAH

Elias Charbel SALAMEH

*10th of April 2024*

# Acknowledgment

Our sincere appreciation goes out to Dr. Elias Salloum for all his help and assistance during the Linear and Control Systems Lab course. His knowledge and support have been crucial to our comprehension and implementation of difficult control systems ideas. We also value his commitment to creating a stimulating and cooperative learning environment. We sincerely appreciate the friendship and cooperation of our fellow engineering students, which tremendously enhanced our experience. The goal of this project was to construct a self-balancing robot, and it would not have been achieved without the insights and combined efforts of all those involved.

Contents

[Acknowledgment 1](#_Toc167132283)

[Abstract 4](#_Toc167132284)

[Introduction 5](#_Toc167132285)

[Description 5](#_Toc167132286)

[Prototype Description 6](#_Toc167132287)

[Components and Dimensions: 6](#_Toc167132288)

[Assembly Process: 7](#_Toc167132289)

[Transfer function for the system: 9](#_Toc167132290)

[Self-Balancing Robot 3D Modeling Using Simscape 13](#_Toc167132291)

[Simulink Representation 13](#_Toc167132292)

[Cart Execution 14](#_Toc167132293)

[Implementation of Chassis 15](#_Toc167132294)

[Joints Implementation 16](#_Toc167132295)

[Control System 16](#_Toc167132296)

[Electrical Design and Circuitry 18](#_Toc167132297)

[Materials Used: 18](#_Toc167132298)

[The code 19](#_Toc167132299)

[References 23](#_Toc167132300)

List of Figures

[Figure 1: Prototype of the self-balancing robot 6](#_Toc167133087)

[Figure 2: DC Motor and Wheels Connection 7](#_Toc167133088)

[Figure 3: FBD of the Mechanical System 9](#_Toc167133089)

[Figure 4: Self-Balancing Robot Simulation on Simulink 11](#_Toc167133090)

[Figure 5: Response 11](#_Toc167133091)

[Figure 6: Steady-State Error 11](#_Toc167133092)

[Figure 7: Slow and Aggressive Autotune 11](#_Toc167133093)

[Figure 8: Less Aggressive Response 12](#_Toc167133094)

[Figure 9: Mid-Aggressive/Robust Response 12](#_Toc167133095)

[Figure 10: Inverted Pendulum 13](#_Toc167133096)

[Figure 11: Self-Balance Robot 3D Model 13](#_Toc167133097)

[Figure 12: Simulink Simulation 13](#_Toc167133098)

[Figure 13: Mechanism Configuration Block for Gravity Specification 14](#_Toc167133099)

[Figure 14: Right Wheel Subsystem 14](#_Toc167133100)

[Figure 15: Cart Subsystem 15](#_Toc167133101)

[Figure 16: Pillars Subsystem 15](#_Toc167133102)

[Figure 17: Chassis Subsystem 16](#_Toc167133103)

[Figure 18: PID Parameters 17](#_Toc167133104)

[Figure 19: Response for Specified PID Parameters 17](#_Toc167133105)

[Figure 20: Our Circuit 18](#_Toc167133106)

List of Tables

[Table 1: Routh Stability Criteria 10](#_Toc167132321)

# Abstract

The creation and application of a self-balancing robot for the Linear and Control Systems Lab course is described in depth in this report. The project's goal was to apply control systems theory to a real-world engineering problem. The robot was built with sensors, microcontrollers, and feedback control algorithms to enable it to maintain equilibrium on its own. To attain stability and responsiveness, the procedure required a great deal of experimentation and incremental design modifications. The project's successful completion highlights the value of practical experience in engineering education as well as the applications of control theory in real-world settings. The outcomes demonstrate the efficacy of our methodology and offer a strong basis for further improvements and robotics applications.

# Introduction

An immersive and practical approach to comprehending and putting control systems principles into practice was offered by the Linear and Control Systems Lab course. We worked with a range of tools and techniques during the semester, such as Arduino for programming and hardware interface, Simulink and Simscape for modeling and simulation, and actual prototype building. To close the gap between theoretical understanding and practical applications, these subjects were essential. Our main project was to design and build a self-balancing robot, which provided a great opportunity to apply and integrate the lessons we had learnt in class. This report outlines the main takeaways and difficulties we ran into while covering these wide-ranging subjects.

# Description

The self-balancing robot project was a complex project that called for a multidisciplinary strategy that included hardware assembly, coding, and simulation. Before putting the robot into real life, we were able to mimic its behavior and improve our control tactics by modeling its dynamics using Simulink and Simscape. These tools offered an interactive and visible way to see how the system responded to different disturbances and control inputs.

We next moved on to coding, using Arduino microcontrollers to put our control methods into practice. Because of its adaptability and simplicity of integration with sensors and actuators, the Arduino platform was selected. Our microcontroller was configured to receive data from an inertial measurement unit (IMU) sensor, which gave us real-time orientation feedback for the robot. For the control system to maintain balance and make accurate modifications, this data was essential.

Assembling the robot's structural elements—steel rods, plates, for example—, and integrating the electronic parts were necessary for building the prototype. The design was carefully considered in order to guarantee toughness and dependability. The Arduino microcontroller governed the IMU sensor, motor drivers, and power supply in the final build.

We overcame many obstacles during the project, such as adjusting the control parameters to produce steady balancing and getting around hardware constraints. Nonetheless, these difficulties were priceless teaching moments that strengthened our comprehension of control systems and their real-world applications. The research resulted in a self-balancing robot that functions properly, highlighting the efficient use of control theory and practical engineering knowledge.

# Prototype Description

The prototype for the self-balancing robot is designed with careful consideration of weight distribution and structural integrity. The overall weight of the prototype is 276 grams. This weight is distributed among several key components, including steel rods and plates, which form the core structure of the robot.

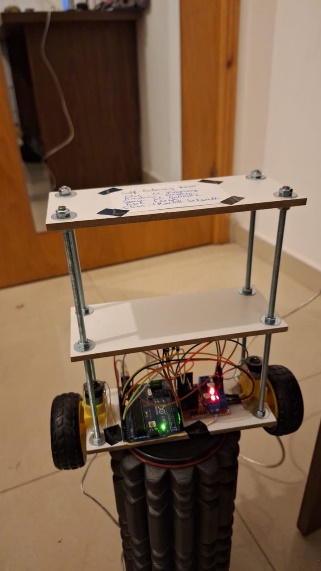


Figure 1: Prototype of the self-balancing robot

## Components and Dimensions:

* Steel Rods:
* Quantity: 4
* Dimensions: Length: 22 cm each, Diameter: 5 mm
* Material Properties: Density of Steel: 7850 kg/m³
* Weight: Each rod weighs 43 grams.

These steel rods are essential for the robot's framework, providing the necessary strength and rigidity while maintaining a lightweight structure.

* Plates:
* Dimensions: Length: 17 cm, Width: 7 cm, Thickness: 5 mm
* Weight: Each plate weighs 34.66667 grams.

The plates are likely used for mounting various components and ensuring the robot's balance. Their dimensions and weight are optimized to maintain the overall lightness of the prototype while providing sufficient surface area for component placement.

The detailed design of the self-balancing robot, including the precise weight and dimensions of its components, ensures that the prototype will have the necessary stability and functionality. This thoughtful balance of materials and weight distribution is crucial for achieving the desired performance in a self-balancing system.

Other key components that play a crucial role for the well-functioning of the self-balancing robot, include two DC motors clamped each on a wheel, which form the core structure of the robot.



Figure 2: DC Motor and Wheels Connection

Here is a breakdown of specific details and instructions for our self-balancing robot project:

* DC Motors:
* Voltage: 6 Volts
* Current: 60 mA
* RPM: 180 RPM
* Wheels:
* Diameter: 65 mm
* Material: Plastic
* Tread: Ribbed tread

## Assembly Process:

The DC motors and wheels came pre-assembled from the manufacturer that simplifies the assembly process significantly.

1. Check Pre-Assembled Components:

We inspected the pre-assembled DC motors and wheels to ensure they are securely attached and aligned properly.

1. Mount Motors onto Chassis:

* We positioned the pre-assembled motors onto the chassis in the desired locations and position.
* Using glue onto the pre-assembled components to fasten the motors onto the chassis, ensuring they are securely attached and aligned properly.
* Making sure the wheels are centered and firmly attached to the chassis assembly.

1. Connect Components:

* Connecting the pre-assembled motors to the control board using wires.
* Connect the control board to a power source and ensure proper voltage and polarity.
* Testing the motors to ensure they spin freely and in the correct direction.
* To ensure the connections are secure, we followed the manufacturer’s instructions for wiring.

1. Test and Calibration:

Before testing the robot, calibrate the motors and perform any necessary adjustments according to the manufacturer’s guidelines. Test the robot in a controlled environment to ensure proper functionality before deploying it for full operation.

Even though the motors and wheels are pre-assembled, it is still essential to double-check all connections and ensure everything is securely fastened to ensure stability and reliability and to prevent any issues during operation.

We considered adding a sensor and a control algorithm for self-balancing functionality.

# Transfer function for the system:

To be able to find the transfer function of the self-balancing robot, we must go to the mathematical calculations. The concept of the mechanical system is the same as for an inverted pendulum, when there is no balancing force, the pendulum will fall. In figure 1, the FBD of the self-balancing robot presented by an inverted pendulum with a mass on it, is shown.

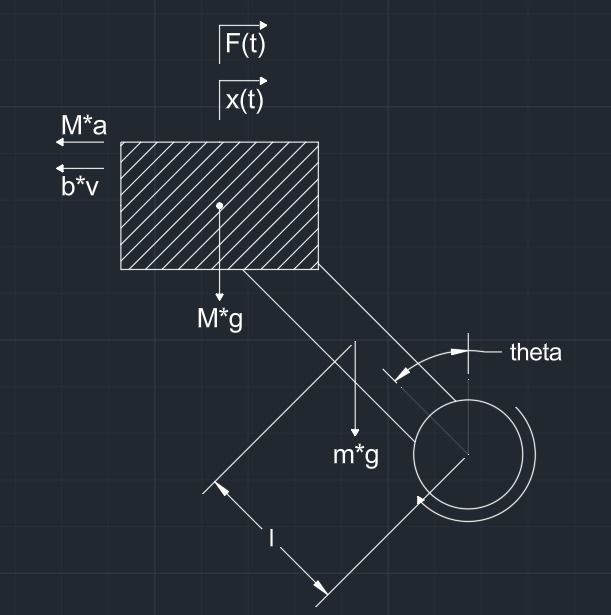


Figure 3: FBD of the Mechanical System

The equations of equilibrium from the 2nd law of Newton are applied. Now, it can be obtained the mathematical system modeling equation for self-balancing robot as follows:

(1)

(2)

(3)

With:

* M: Mass of the chassis
* m: mass of rods
* b: Viscous friction coefficient
* l: half distance of rod
* F: Force applied to the system
* g: Gravitational acceleration
* : Pendulum angle
* : Mass moment of inertia of pendulums

Assuming that , we get from equations 1 and 2 the EOM of the system which is:

Now, we apply the Laplace transformation to both the equations, and we find the ratio of output over the input which will be our transfer function.

With:

So, after replacing with the values we get:

A system's stability can be determined by studying where its poles are in relation to the field s; if the poles are to the left of the field, the system is stable. We thus employed the stability of Routh to determine the placement of the poles in a system. To find these poles we need a certain gain to this system which will give us between which values of this gain we will have a stable system. Introducing a gain K to the open loop transfer function we will have:

Applying the Routh stability criteria we get:

Table 1: Routh Stability Criteria

|  |  |  |
| --- | --- | --- |
|  | 1 | -61.44 |
|  | 0.317 |  |
|  |  |  |
|  |  |  |

So, for stability we need to have:

Thus, we get that:

The simulation of this process on Simulink is:

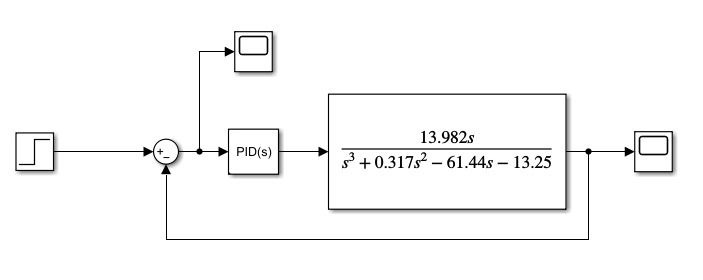


Figure 4: Self-Balancing Robot Simulation on Simulink

Using the autotune option on Simulink PID Parameters can be found as:

P: 5.815

I: 8.222

D: 0.534

For these parameters:

|  |  |
| --- | --- |
| Figure 5: Response | Figure 6: Steady-State Error |

This response is considered slow and aggressive as shown on the top of the figure below retrieved from the autotune option:

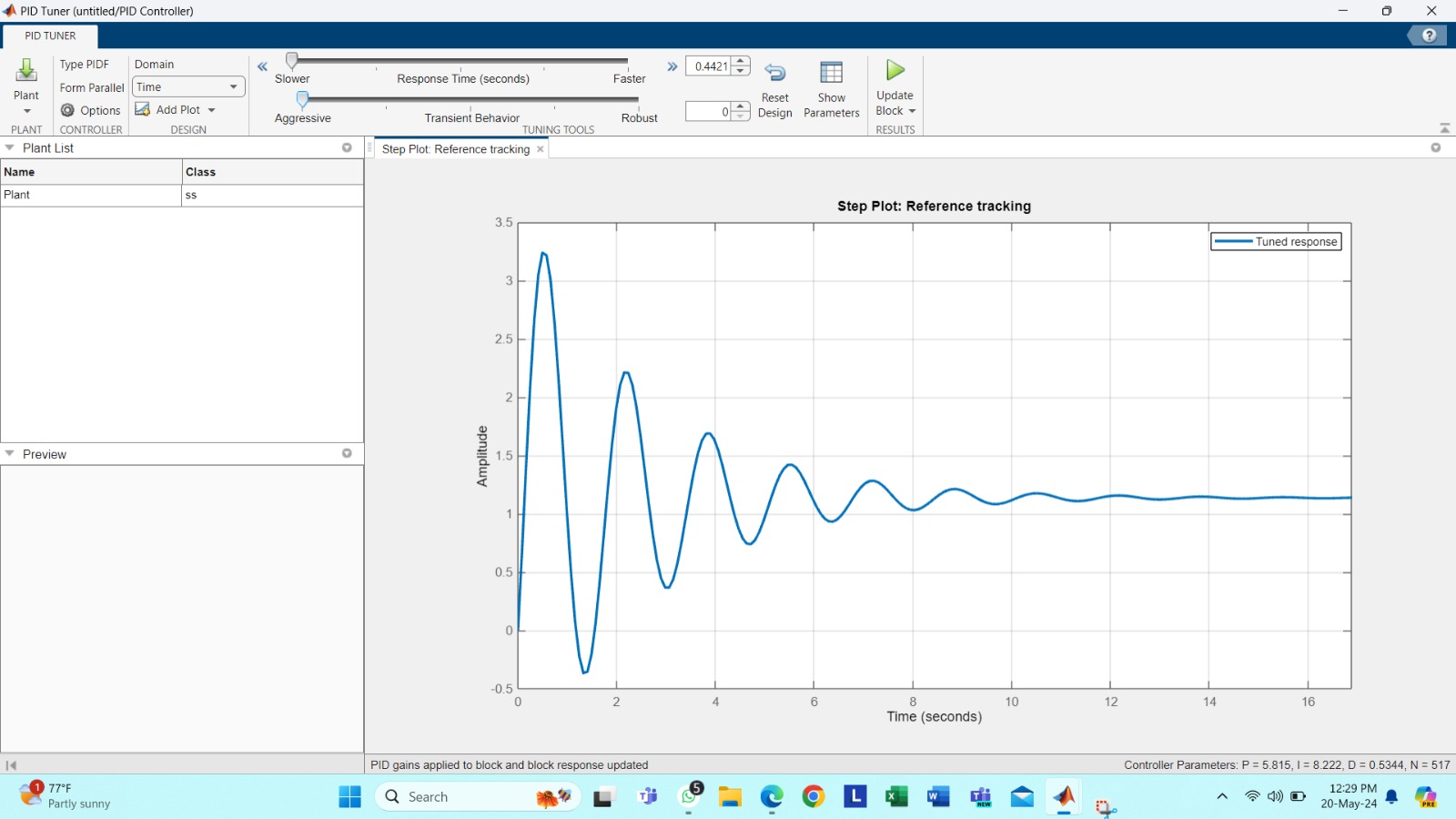


Figure 7: Slow and Aggressive Autotune

If less aggressive is needed, the following response is obtained:

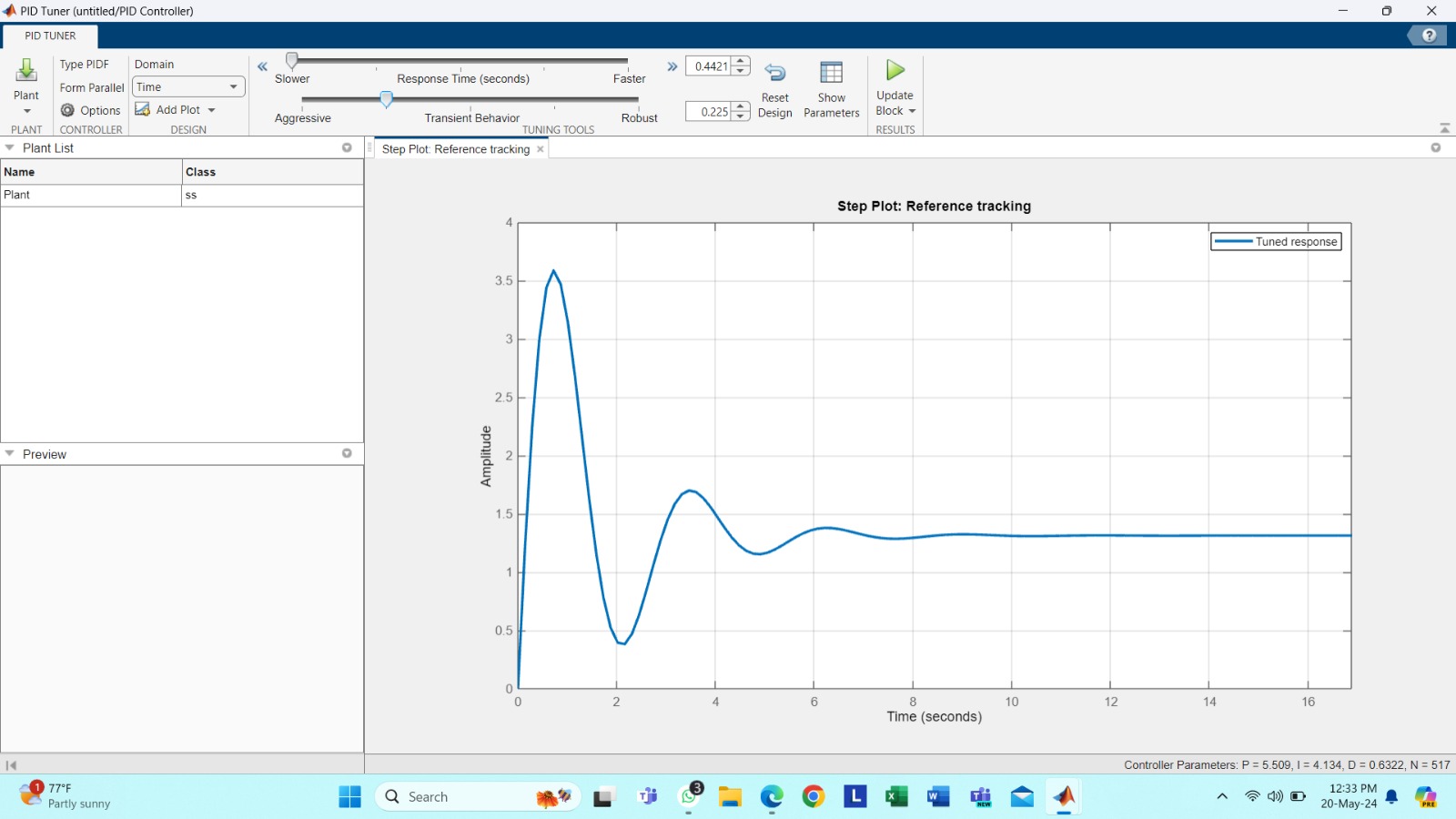


Figure 8: Less Aggressive Response

However, it can be seen that overshoot is increased with decreasing oscillations.

If response that is more robust is needed:

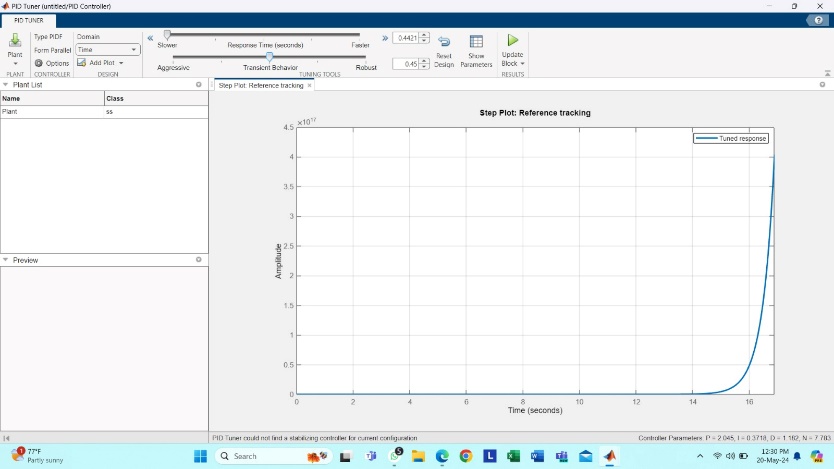


Figure 9: Mid-Aggressive/Robust Response

It can be seen that as the response becomes more robust, unstable system is reached.

Self-Balancing Robot 3D Modeling Using Simscape  
Principle

A 3D model of a 2-wheeled self-balancing robot was built using Simscape from Simulink to gain insights. A PID controller will be used to manage the system. This concept is based on the principles of an inverted pendulum, where the objective is to keep the mass in an upright position, requiring constant movement of the base governed by a control system.

If the base is replaced with two wheels and the rod and mass with a chassis, then we have constructed a self-balancing robot.

|  |  |
| --- | --- |
| Inverted Pendulum with Stationary Pivot Point. | Download Scientific Diagram  Figure 10: Inverted Pendulum | Figure 11: Self-Balance Robot 3D Model |

## Simulink Representation

The simulation of this robot is shown below:

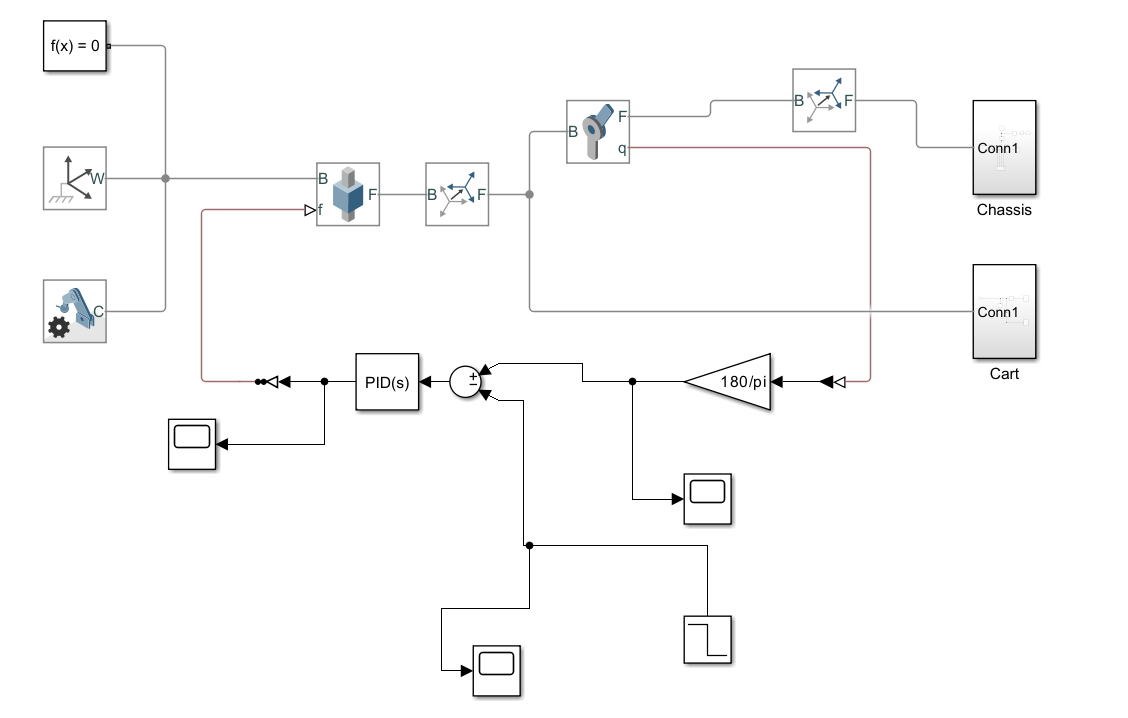


Figure 12: Simulink Simulation

The components listed on the left are essential for building a Simscape Multibody model:

* *Solver Configuration: Performs calculations.*
* *World Frame: Represents the ground of every mechanical model in Simulink and provides access to the world coordinate frame.*
* *Mechanism Configuration: Specifies a uniform gravity for the entire mechanism.*

*(Since x-axis is the vertical axis in this simulation, the gravity must be pointed in the -x direction with a norm of -9.8065 m/s2)*

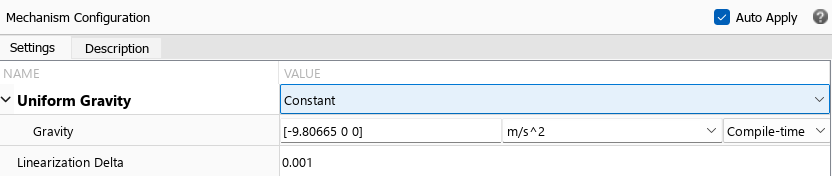
**

Figure 13: Mechanism Configuration Block for Gravity Specification

### Cart Execution

In the “Cart” subsystem, the wheels for our robot are represented by two cylindrical solids: one for the interior and one for the tire. The correct dimensions are specified in the properties of these cylinders, including radius, thickness, mass, and color.

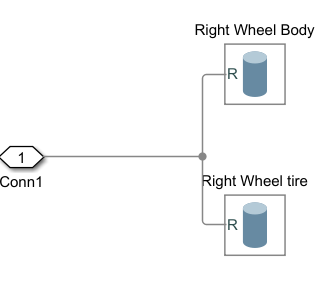


Figure 14: Right Wheel Subsystem

To complete our cart, a brick solid is used as a shaft, with the appropriate dimensions and weight inserted. A rigid transform is then used to translate each wheel to one end of the shaft by specifying a standard Z-axis translation marking the distance from the center.

* *Rigid Transform: Transforms one solid block with respect to another solid block, ensuring they move as a single unit during the simulation since their pose with respect to each other is constrained. This is useful for building compound bodies from simple solid blocks.*

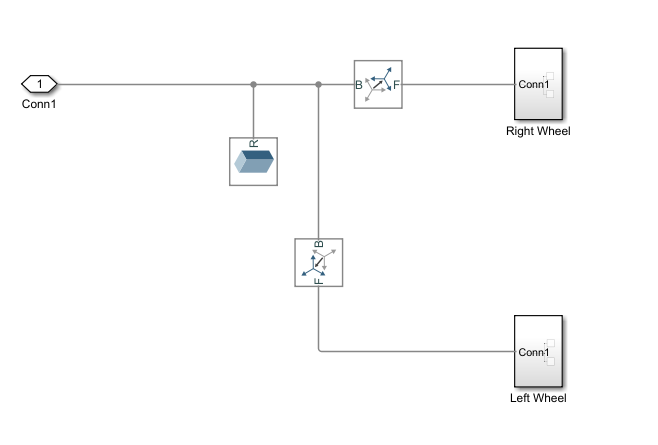


Figure 15: Cart Subsystem

### Implementation of Chassis

Making the upper body of the robot, which is present in the “Chassis” subsystem, consists of three rectangular planes attached together using four rods. The three rectangular planes are represented with brick solids as three shelves, specifying their respective properties (dimensions, weight, color). Four cylindrical blocks with appropriate properties are used as rods.

Z-axis translations are present between chassis plate 1 and chassis plate 2, and between chassis plate 1 and chassis plate 3, assigning the distance between each pair of plates with chassis plate 1 as the base frame. A similar procedure is used to attach the pillars, connecting each rod and specifying the location on the XYZ diagram using Cartesian translation in the properties to position them at each corner of the three plates. The “pillars” subsystem present in the “Chassis” is represented as follows:

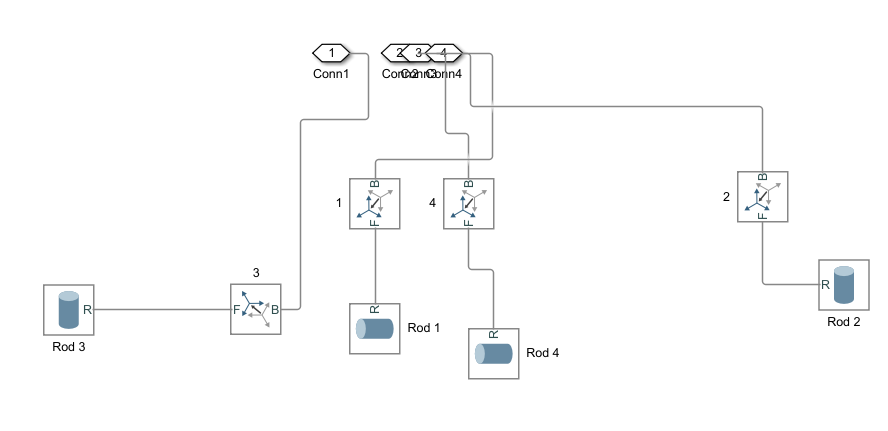


Figure 16: Pillars Subsystem

The entire process of assembling the pillars and plates is grouped into a subsystem called “Chassis.”

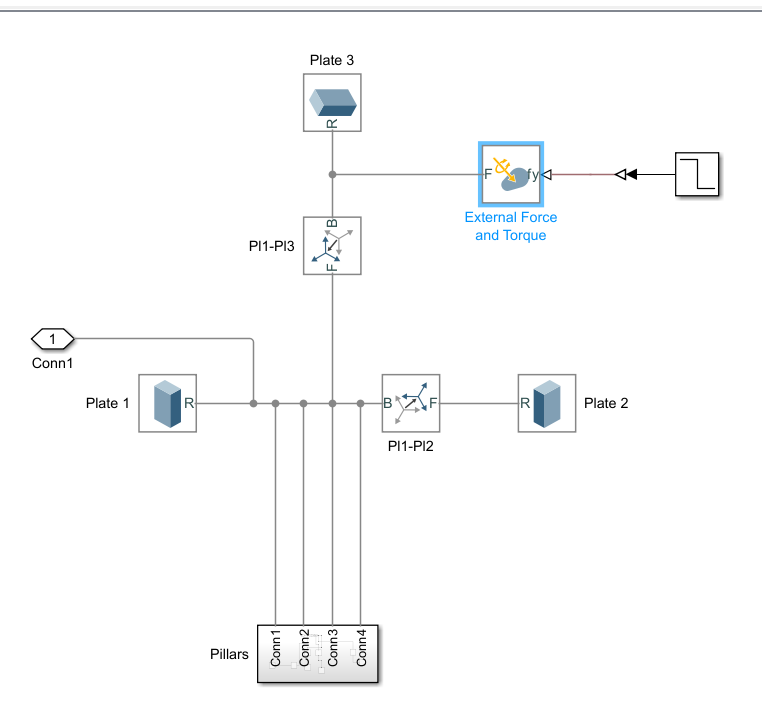


Figure 17: Chassis Subsystem

* *External Force and Torque: Actions applied on the robot used to induce falling and swinging, similar to a normal pendulum.*

The "Cart" and "Chassis" are connected by attaching the cart to the world frame and to the chassis through a rigid transform, performing a sequence of rotations. A 'Rotation Sequence' is applied in the rotation properties of the rigid body, with 90° rotations on the y-axis and 180° rotations on the x-axis.

### Joints Implementation

The chassis and the cart are connected by a revolute joint that provides a degree of freedom, allowing the chassis to swing like a pendulum.

A prismatic joint, interconnected with a rigid body between the cart and the world frame, is inserted to allow the robot to move back and forth, stabilizing the body in an upright position as controlled by the appropriate controller.

### Control System

All that is left to do is provide the correct input to the prismatic joint to achieve the desired behavior from this system. This is done by implementing a control loop that measures the inclination angle Θ, compares it with the desired output (the setpoint of 0°), and feeds the error to the PID controller, which then provides the appropriate force to stabilize the robot.

The inclination angle can be read from the position of the revolute joint in its properties. The force in the prismatic joint is set to be provided by input in actuation, with the motion automatically computed.

The measured angle is in radians; to convert it to degrees, a gain of is used. The measured angle is then compared to our setpoint via a summing point, and the error is fed to the PID controller block. In the Simulink-PS converter, the input signal is set to Newtons (N), and in the PS-Simulink converter, the output signal is in radians (rad). Scopes are used to observe the behavior of the input and output signals. Auto-tune is used to adjust the PID parameters to achieve the desired behavior.

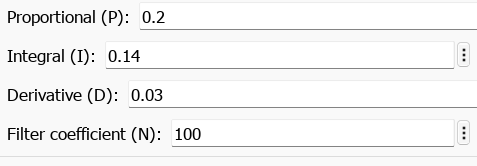


Figure 18: PID Parameters

The response to these parameters is presented below:

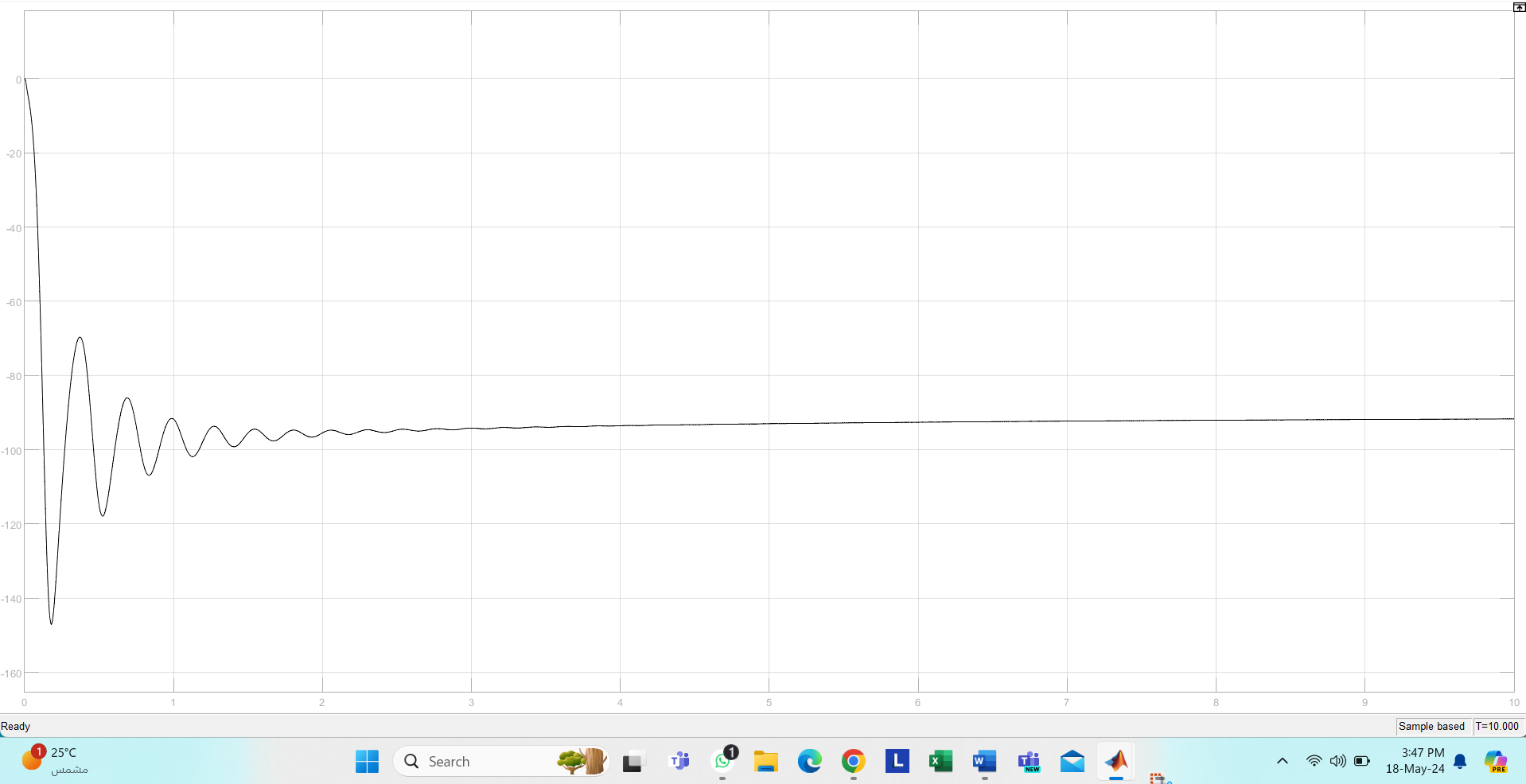


Figure 19: Response for Specified PID Parameters

# Electrical Design and Circuitry

Based on the system identification mentioned above, we must control our system. There are various ways to control the system. We can either use the MATLAB equivalent method that generates a code to the Arduino UNO board or we can use directly the Arduino 2.3.2 IDE.  
We opt for the latter method.

Using the C code in the IDE, we need to download multiple libraries to the environment that we are working in such as:

* I2C serial communication
* PID Controller library
* Motor Driver library

## Materials Used:

1. Arduino UNO
2. 1x L298N Motor Driver
3. 2x BO Motors
4. 2x BO tyres
5. MPU-6050
6. Test board
7. Wires

We aim to synchronize the 2 motors in this study since we are just controlling the balance of the motors on the inclination of the y-axis and z-axis.

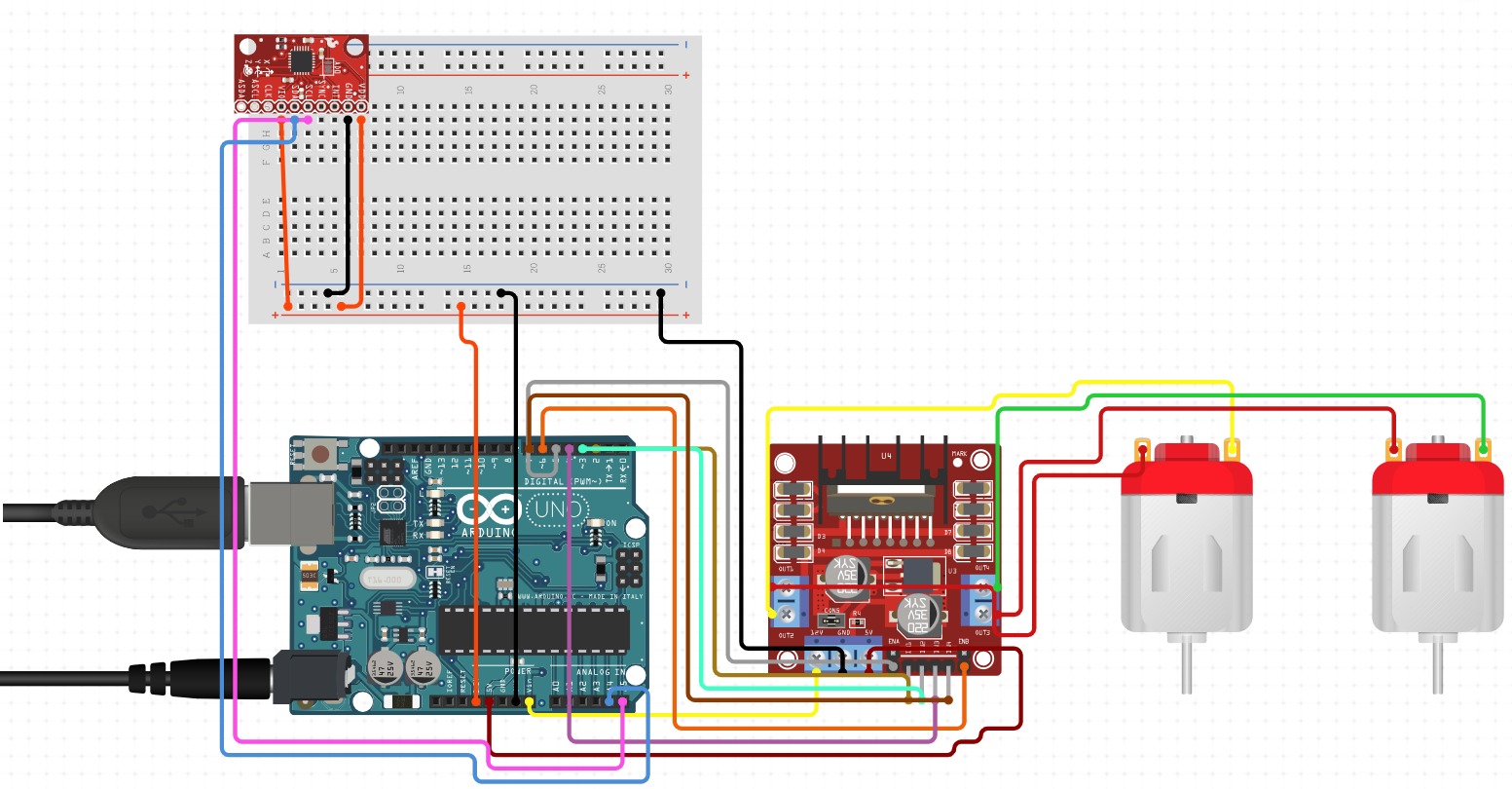


Figure 20: Our Circuit

## The code

#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

// MPU control

MPU6050 mpu;

bool dmpReady = false;

uint8\_t mpuIntStatus;

uint8\_t devStatus;

uint16\_t packetSize;

uint16\_t fifoCount;

uint8\_t fifoBuffer[64];

// orientation

Quaternion q;              // Quaternion container [w, x, y, z]

VectorFloat gravity;       // Gravity vector [x, y, z]

float ypr[3];              // Yaw/pitch/roll container and gravity vector

// PID

double originalSetpoint = 183;

double setpoint = originalSetpoint;

double movingAngleOffset = 0.1;

double input, output;

double Kp = 60;

double Kd = 2.2;

double Ki = 270;

PID pid(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);

double motorSpeedFactorLeft = 0.6;

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

  #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();

  mpu.setXGyroOffset(220);

  mpu.setYGyroOffset(76);

  mpu.setZGyroOffset(-85);

  mpu.setZAccelOffset(1788); // 1688 factory default for my test chip

  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();

    dmpReady = true;

    packetSize = mpu.dmpGetFIFOPacketSize();

    // PID setup

    pid.SetMode(AUTOMATIC);

    pid.SetSampleTime(10);

    pid.SetOutputLimits(-255, 255);

  } else {

    Serial.print(F("DMP Initialization failed (code "));

    Serial.print(devStatus);

    Serial.println(F(")"));

  }

}

void loop() {

  if (!dmpReady) return;

  while (!mpuInterrupt && fifoCount < packetSize) {

    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

  if ((mpuIntStatus & 0x10) || fifoCount == 1024) {

    // Reset

    mpu.resetFIFO();

    Serial.println(F("FIFO overflow!"));

  } else if (mpuIntStatus & 0x02) {

    // Wait for correct available data length

    while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

    // Read a packet from FIFO

    mpu.getFIFOBytes(fifoBuffer, packetSize);

    fifoCount -= packetSize;

    mpu.dmpGetQuaternion(&q, fifoBuffer);

    mpu.dmpGetGravity(&gravity, &q);

    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

    input = ypr[1] \* 180/M\_PI + 180; // Convert radians to degrees and normalize

  }

}

# References

1. *Circuit Design App for Makers- circuito.io*. (n.d.). <https://www.circuito.io/>
2. M. B. H. C. (2021, March 28). *applying an LQR command on a self-balancing robot in Simulink |2021|*. YouTube. <https://www.youtube.com/watch?v=SY8KtZ2hO9Q>
3. M. B. H. C. (2021, March 12). *self balancing robot | Simulink basics series*. YouTube. <https://www.youtube.com/watch?v=QtmVFlZi5T8>
4. A. S. (2021, March 2). *Arduino Self Balancing Robot with MPU6050*. YouTube. <https://www.youtube.com/watch?v=0KDyb7F0C_8>